Molekulový docking, simulace měkkých tkání

Jiří Filipovič

podzim 2009

Případové studie

V závěru předmětu se budeme zabývat studiemi konkrétního nasazení CUDA

- pro lepší představu, k čemu všemu se dají akcelerátory využít
- chápeme-li práci jiných, pomůže nám to v práci vlastních
  - rozdílné obory často sdílí mnoho společných principů
- pokusíme se o nabídku přesahující zkušenosti přednášejících
  - budou následovat zvané přednášky :-)

Molekulový docking

Problém ,,zadokování" (zapadnutí, zaskočení) jedné molekuly do druhé

- zpravidla dokujeme malou molekulu (ligand) do velké (receptor, většinou protein)
- hledáme stabilní komplex, kde je jedna molekula navázána na druhou
- zajímá nás, aby bylo navázání v *aktivním místě* receptoru
- tím modifikujeme vlastnosti receptoru (aktivace či inhibice)

Aplikace

- vývoj léků
- likvidace znečištění
- cokoliv těžící z možnosti upravovat vlastnosti proteinů

Molekulový docking z výpočetního hlediska

Můžeme uvažovat ,,tvar" molekul, nebo jejich silová pole

- my se budeme zabývat silovými poly

Molekuly na sebe působí silou, hledáme komplex s nejnižší potenciální energií

- můžeme na mřížce předpočítat silové působení receptoru
- následně můžeme hledat takové umístění ligandu, které má nejmenší energii vůči mřížce
- tím redukujeme časovou složitost z $\mathcal{O}(n \cdot m)$ na $\mathcal{O}(m)$ pro receptor o velikosti n a ligand o velikosti m atomů ($m << n$)

My se budeme zabývat předpočítáním silového pole.

Výpočet Coulombovského potenciálu

Potenciál v konkrétním bodě mřížky je dán vztahem

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0\epsilon(r_{ij})r_{ij}}$$

Kde $\epsilon(r_{ij})$ je dielektrikum závislé na vzdálenosti a r_{ij} je vzdálenost atomu od bodu mřížky.
Potenciál klesá s druhou mocninou vzdálenosti – to je relativně pomalu, často se tedy počítá pro každý bod mřížky potenciál vůči všem atomům receptoru.

CUDA implementace

Nejprve se budeme zabývat implementací s konstantním dielektrikem (tedy $\epsilon(r) = k$).

- John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry, Volume 28 Issue 16*, 2008.

Paralelizace

- každá buňka může být zpracovávána nezávisle na ostatních

Rychlostní omezení základního algoritmu

- 9 aritmetických operací na jeden atom
- informace o pozici buňky dány umístěním threadu
- informace o atomech v 16 bytech (4 floaty – pozice a náboj)
- při naivním pohledu jsme tedy omezeni rychlostí paměti

CUDA implementace

Omezení paměti

- každý thread potřebuje přečíst 4 floaty popisující právě zpracovaný atom
- v rámci warpu zpracovávají všechny thready současně *stejný atom* pro *různé buňky*
- údaje o atomech slouží *pouze ke čtení*
- ideální pro **paměť konstant**

CUDA implementace

Použijeme-li paměť konstant

- máme zajištěný cacheovaný přístup
  - nevadí, že nás zajímají pouze 4 (obecně nezarovnané) floaty
- data se z globální paměti čtou nejvýše jednou pro jeden warp
  - redukuje nároky na propustnost paměťi alespoň na 1/32
- počet atomů umístitelných do paměti konstant je omezen
  - pro více než 4096 atomů je třeba spouštět kernel vícekrát
  - doba spouštění je však pro takto dlouhý výpočet zanedbatelná

První kernel

```
loat curenergy = energygrid[outaddr];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
float coorz = gridspacing * zindex;
int atomid;
float energyval=0.0f;
for (atomid=0; atomid<numatoms; atomid++) {
  float dx = coorx - atominfo[atomid].x;
  float dy = coory - atominfo[atomid].y;
  float dz = coorz - atominfo[atomid].z;
  energyval += atominfo[atomid].w *
                rsqrtf(dx*dx + dy*dy + dz*dz);
}
energygrid[outaddr] = curenergy + energyval;
```

Co můžeme zrychlit?

Dělá paralelní kód nějakou redundantní práci?

- každý thread počítá pro každý atom druhé mocniny vzdáleností ve smyslu os x, y, z

Dokážeme se této redundance zbavit?

- počet buněk roste kubicky vzhledem k jemnosti mřížky
- kvadratický růst je pro škálování dostatečný
- spustíme-li kernel pro každý plát mřížky, můžeme jednu složku vzdálenosti předpočítat
- předpočítání může být provedeno na CPU (vypočtená data se užijí mnohokrát, výkon není kritický)

Co ještě můžeme zrychlit?

Redundance výpočtu vzdálenosti v jednotlivých prostorových složkách lze dále omezit **unrollingem**

- každý thread bude zpracovávat více buněk v jednom řádku
- výpočet vzdálenosti ve smyslu jedné osy použijeme pro více buněk
- režije for cyklu se sníží
- zvýší se však počet použitých registrů
- zhorší se šlálování algoritmu

Výsledný výkon na GeForce 8800GTX 35.5 GEvals

- odpovídá 291 GFlops
- cca 40× rychlejší než CPU implementace

Další síly

Kromě elektrostatiky mezi molekulami působí další síly

- van der Waalsovy
- vodíkové
- desolvatační

Všechny rychle klesají s rostoucí vzdáleností

- je možné provézt ořez vzdálených atomů
- méně triviální na GPU
- nižší časová složitost, lze provádět na CPU

Akcelerace programu AutoGrid

AutoGrid

- součást balíku AutoDock (velmi používaný dokovací software)
- původní kód pouze CPU (navíc ne příliš optimální)
- počítá se všemi výše zmíněnými silami

Akcelerace AutoGridu – program FastGrid

- Marek Olšák. Refaktorizace kódu a implementace urychlujících algoritmů do programu AutoGrid. *Fakulta informatiky Masarykovy university*. 2009.
- Marek Olšák, Jiří Filipovič, Martin prokop. FastGrid – The Accelerated AutoGrid Potential Maps Generation for Molecular Docking. *MEMICS Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. 2009 (accepted).

FastGrid

Výpočet ekvivalentní původnímu AutoGridu

- nezavádí žádné nové aproximace
- výsledky se mohou mírně lišit kvůli jiné implementaci floating-point operací v dnešních GPU

Hybridní CPU/GPU výpočet

- výpočet elektrostatiky prováděn na GPU
- ostatní síly jsou počítány na CPU jádrech

Dielektrikum závislé na vzdálenosti

AutoGrid používá volitelně dielektrikum závislé na vzdálenosti

- CPU kód předpočítává dielektrikum do tabulky
- je nutno rozšířit implementaci prezentovanou ve [Stone07]

Hodnotu dielektrika závislého na vzdálenosti lze

- vypočítat na místě pro každý atom v každém vlákně
- přečíst z globální paměti
- přečíst z paměti konstant
- přečíst z texturové paměti

Dielektrikum závislé na vzdálenosti

Přímý výpočet

- nemusíme čekat na paměť
- musíme udělat více práce

Předpočítaná tabulka v globální paměti

- méně výpočtů ve vlákně
- nekoalescentní přístup
- adresace vyžaduje celočíselné dělení

Předpočítaná tabulka v paměti konstant

- cacheovaný přístup
- přes jistou lokalitu nečteme vždy stejná data
- o paměť konstant se dělí atomy s předpočítanou tabulkou
- adresace vyžaduje celočíselné dělení

Dielektrikum závislé na vzdálenosti

Předpočítaná tabulka v texturové paměti

- cacheovaný přístup
- odpadá nutnost celočíselného dělení
- neomezuje počet atomů zpracovávaný v jednom volání kernelu
- latence může být bottleneck

Co je nejvýhodnější?

- globální paměť můžeme rovnou vyloučit
- zbytek implementujeme a otestujeme

Dielektrikum závislé na vzdálenosti

Výběr vhodného kernelu záleží na velikosti problému

- velké mřížky
  - paměť textur
  - zřejmě hraje roli lepší adresace a méně spouštění kernelu
- malé mřížky
  - přímý výpočet
  - horší lokalita na větších buňkách, málo buněk zřejmě nedokáže plně saturovat cache

Výkon na dostatečně velkých mřížkách je okolo 24.4 GEvals na GeForce GTX 280 (pro konstantní dielektrikum lze dosáhnout 54.8 GEvals).

Malé mřížky

Pro malé mřížky algoritmus špatně škáluje

- je optimální vzdát se některých optimalizací :-)
- kernel nemusí být unrollován
- můžeme pracovat současně na celé mřížce namísto plátků

Hybridní algoritmus

Původní AutoGrid

```
for all points of the grid do
  for all atoms of molecule do
    compute electrostatic potential
    if distance between point and atom < cutoff value then
      compute desolvation potential
      compute van der Waals potential
      compute hydrogen bonds potential
    end if
  end for
end for
```

Rozhodnutí, zda se budou počítat i síly ořezávané pro větší vzdálenost se provádělo na základě eukleidovské vzdálenosti získané při výpočtu Coulombovského potenciálu. Pro vodíkové vazby bylo zapotřebí nelézt nejbližší vodík.

Hybridní algoritmus

Vzdálenost buňky od atomu nyní počítáme na GPU

- redundantní výpočet na CPU by zvýšil časovou složitost CPU kódu
- využít data z GPU by znamenalo nechat GPU budovat seznam blízkých atomů pro každou buňku a tento seznam kopírovat přes PCI-E
- využijeme tedy lepší datovou strukturu

Blízké atomy

- hrubá detekce blízkých atomů pomocí regulární mřížky, tu vybudujeme a naplníme při startu
- blízké vodíky budeme hledat pomocí k-NN

Využití typické konfigurace

- často je přítomno více CPU jader nežli GPU procesorů
- cyklus běžící na CPU paralelizujeme přes plátky mřížky

Výkon

S rostoucí velikostí mřížky roste zátěž na GPU

- teoreticky jsme omezeni rychlostí GPU implementace (24.4 GEvals a 54.8 GEvals)
- pro smysluplně velké problémy je CPU u konstantního dielektrika bottleneck
  - ne příliš významný
  - řešením je přenést více práce na GPU

Na smysluplně velkých problémech překonáme implementaci AutoGridu cca 300× až 400×.

Simulace měkkých tkání

Zajímá nás, jak se chovají měkké tkáně

- jejich uchycení, deformace
- interakce s nástroji
- řezání, trhání
- interakce s fluidní dynamikou
- ...

K čemu to?

- aplikace v medicíně

Z výpočetního hlediska

- tkáň diskretizujeme, nalezneme fyzikální popis materiálu, ze kterého je složena
- chování simulujeme pomocí *metody konečných prvků* (FEM)
- z výpočetního hlediska
  - pro realistickou simulaci je třeba počítat nelineární FEM
  - iterativně skládáme a řešíme systém rovnic
    - je jich velmi mnoho (jedna rovnice pro stupeň volnosti)
    - lze řešit derivačními i nederivačními metodami

Výpočetní náročnost

Obecně dva velmi náročné úkoly

- složit systém rovnic
- vyřešit systém rovnic

Oba vhodné pro GPU

- řešením se nebudeme zavývat (publikovány práce o multigrid metodách, konjugovaných gradientech, faktorizaci matic)
- skládání bylo více na okraji zájmu
  - méně obecná implementace
  - u derivačních metod může být složitá implementace

Skládání rovnic

Pro každý element potřebujeme vyjádřit stiffness matici

- aplikace několika algebraických operací na malé datové struktury

Každý element přispívá do soustavy rovnic, kterou je třeba vyřešit

- hodnoty pro vrcholy, které elementy sdílí, je třeba sečíst
- více možností (atomické operace, barvení grafu)

Existující práce

Akcelerace skládání pro konkrétní FEMy

- několik kernelů implementuje skládání konkrétního FEMu
- velmi rychlé, může být komplikované v případě velkého množství zpravcovávaných dat na element
- málo flexibilní – změna FEMu vyžaduje reimplementaci
- několik publikovaných implementací, mapování element-thread

Automatické generování invokace kernelů

- implementace kernelů pro třídu FEMů
- ve vyšším programovacím jazyku lze popsat FEM, tento kód je skompilován do kódu volajícího příslušné kernely

Co děláme my?

Implementace větších operací

- nelze provézt v rámci threadu kvůli omezeným zdrojům
- stále ne dost složité pro mapování element-blok
- zavedení medium-grained operací

Flexibilní implementace

- namísto kernelů řešících specifické části skládání implementujeme kernely pro hromadné aritmetické operace
- takovéto operace mohou být použity k popisu libovolného skládání rovnic
- operace jsou však omezeny rychlostí globální paměti – je zapotřebí je spojovat

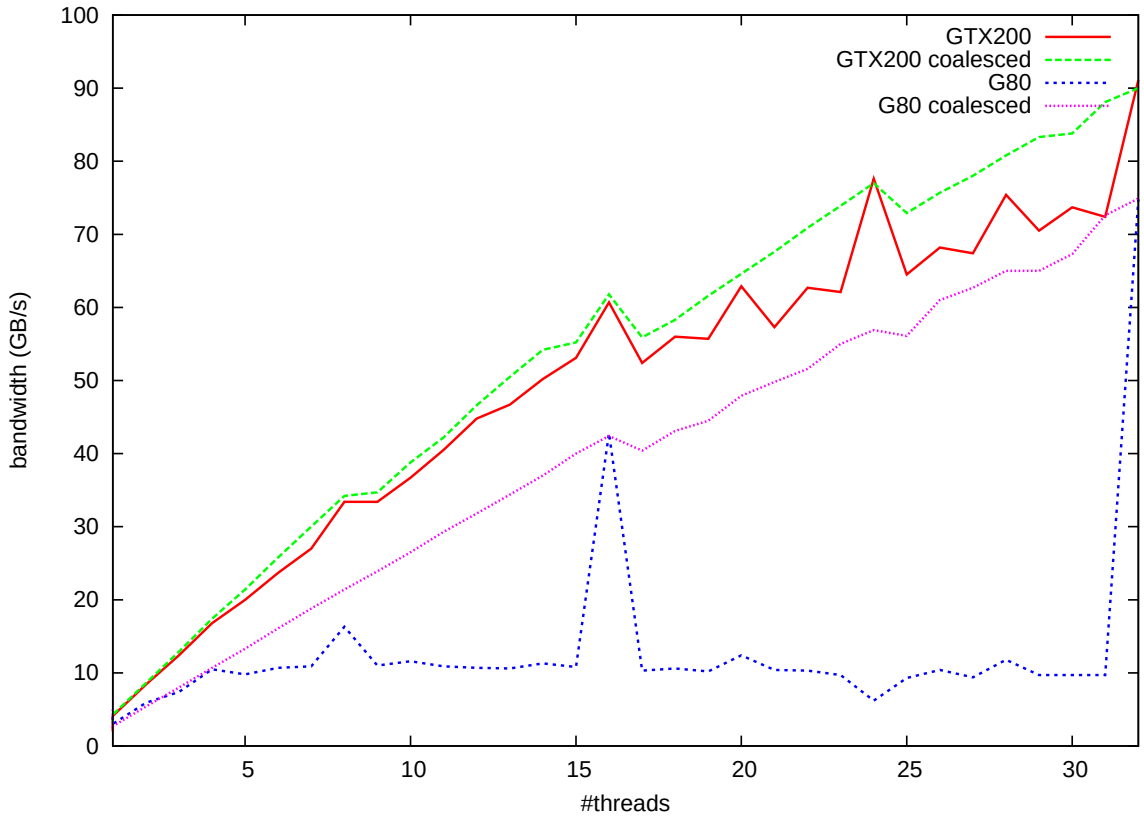
Medium-grained operace

Pracují s daty příliš malými pro blok, ale příliš velkými pro thread

- pokud operace spojíme, problém se ještě zvětší (dělí se o prostředky)
- jeden blok bude zpracovávat více operací, každou operaci bude zpracovávat více threadů
  - tím jsme schopni dosáhnout lepšího využití prostředků dostupných multiprocesoru
  - složitější implementace, složitější optimalizace přístupu do sdílené paměti
- pro operace dekomponovatelné na počet threadů blízkých násobku warpu lze nechat část threadů nevyužitých
  - může vézt na jednodušší a rychlejší implementaci
  - na zařízeních s CC < 1.2 vyžaduje složitější uložení dat

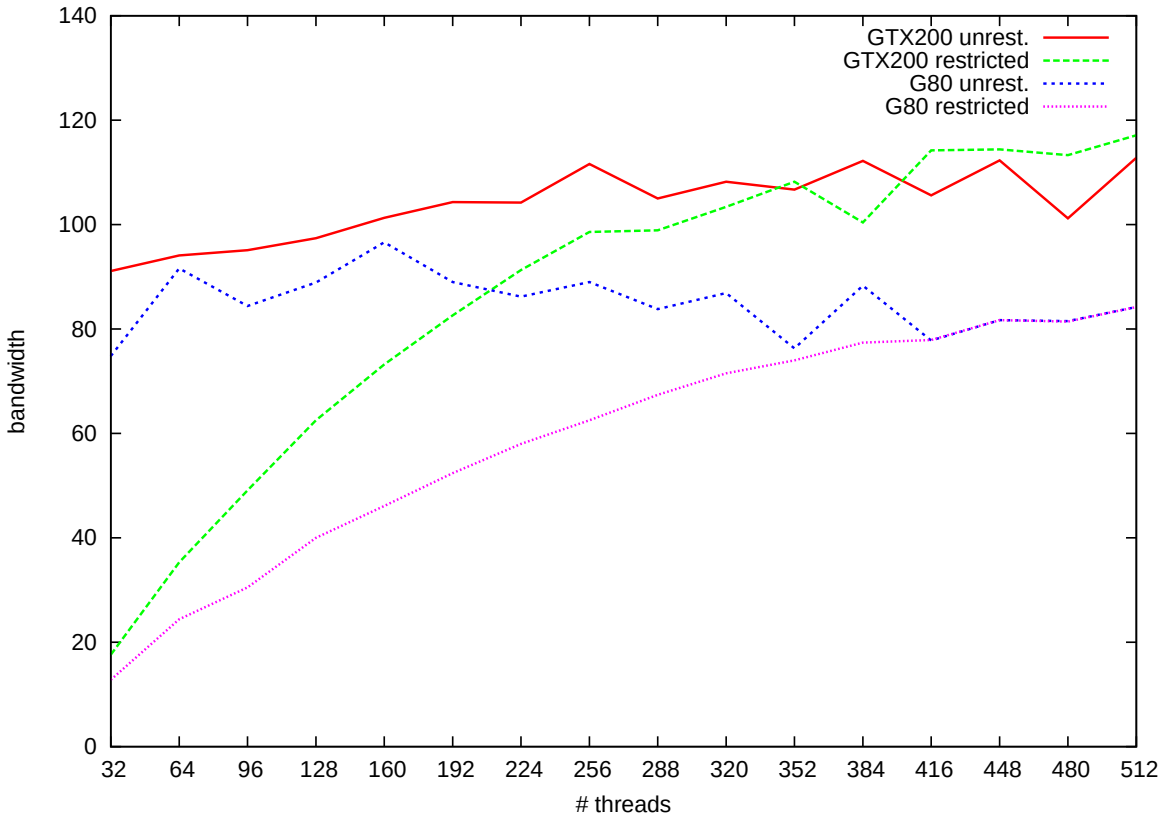
Medium-grained operace

V rozhodnutí, jak operace mapovat na thready, nám může pomoci podrobnější benchmarkování karty.



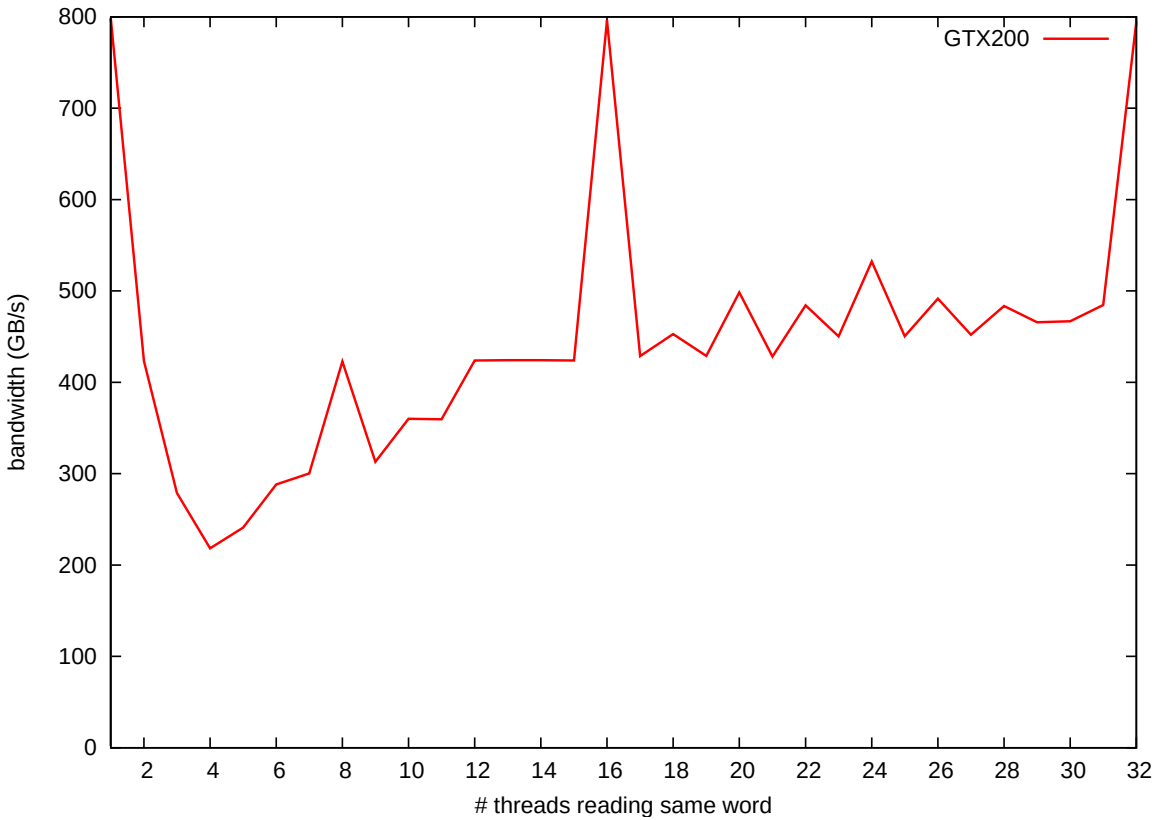
Obrázek: Bandwidth pro různý počet aktivních threadů ve warpu.

Medium-grained operace



Obrázek: Bandwidth pro různý počet threadů v bloku.

Medium-grained operace



Obrázek: Rychlost sdílené paměti pro různý počet threadů přistupujících ke stejné hodnotě.

Skládání operací

Tím, že rozbijeme problém skládání na jednotlivé aritmetické operace, výrazně snížíme flop-to-word ratio

- implementace ve třech krocích
  - načteme data z GMEM do SMEM
  - provedeme výpočet nad SMEM
  - uložíme výsledek do GMEM
- lze je tedy relativně jednoduše spojit
  - výpočty více operací voláme ve stejném kernelu
  - data si mění přes SMEM namísto pomalejší GMEM
- spojování má však své meze
  - omezené zdroje
  - rozdílná granularita

Skládání operací

Složitý úkol

- v současné době máme několik ručně seskládaných variant pro ověření konceptu
- problém vybrání operací ke složení je obecně velmi složitý
  - nejasný výběr operací ke složení
  - obtížné balancování množství spojených operací vs. redukovaný paralelismus
  - obtížně odhadnutelný efekt synchronizace
- zavedení metrik odhadujících výkon a automatického generování skládání
  - předmět současné a hlavně budoucí práce

Výkon prvotní implementace