# Molekulový docking, povodňové mapy

Jiří Filipovič

podzim 2014

# Případové studie

Ve zbývající části 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 vlastní
  - rozdílné obory často sdílí mnoho společných principů
- pokusíme se o nabídku co nejpestřejších témat :-)

# 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 výpočtu potenciální energie 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 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

```
float 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 provést ořez vzdálených atomů
- složitější 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 efektivní)
- počítá se všemi výše zmíněnými silami

Akcelerace AutoGridu – program FastGrid

- Marek Olšák, Jiří Filipovič, Martin Prokop. FastGrid – The Accelerated AutoGrid Potential Maps Generation for Molecular Docking. *Computing and Informatics, Volume 29, Issue 6+*. 2010.
- 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.

# 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 [Stone08]

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 vhodné 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×.

# Zadání

Umělý problém, projekt PV197, rok 2010

- zajímavé možnosti formulace algoritmu

Máme dánu výškovou mapu země, zdroj a výšku přítoku vody. Úkolem je zjistit, kam voda doteče.

- zdroj vody je neomezený a země nepropustná
- jde tedy o nalezení souvislého regionu, kde je země dostatečně nízká aby do ní natekla voda

# Naivní algoritmus

Zrejmě nejjednodušší verze

- vytvoříme thread pro každý bod mapy
- thread periodicky kontroluje své okolí, je-li v něm voda, a je-li bod spravovaný threadem níže než přítok, bod se zaplaví
- výpočet iteruje dokud vedou jednotlivé iterace k šíření vody

Silně neefektivní

- uvažujme triviálně celou mapu pod úrovní přítoku
- v každé iteraci provedeme $n^2$ operací, potřebujeme $2n$ iterací
- složitost je tedy $\mathcal{O}(n^3)$, složitost sekvenčního algoritmu $\mathcal{O}(n^2)$

# Plovoucí čára

Jak rozšířit více vody v jedné iteraci?

- vytvoříme plovoucí čáru, každý její bod může být zpracováván paralelně
- čára prochází obrázek v horizontálním i vertikálním směru ,,dopředu" i ,,dozadu"
- narazí-li na zdroj vody, šíří vodu dokud prochází body pod úrovní přítoku
- průchody do všech směrů provádíme iterativně, dokud rozšiřují vodu

# Plovoucí čára

Jak jsme na tom s efektivitou?

- v jedné iteraci provedeme $n^2$ kroků a jsme schopni zaplavit až $n^2$ políček
- situaci komplikují složité povodňové mapy (kde se voda šíří koryty)
- minimálně musíme provést 2 iterace po 4 průchodech čáry, v praxi více
- složitost je tedy $\mathcal{O}(i \cdot n^2)$, kde $i$ je počet iterací

Atak 250Mpix/s hranice (2. etapa)

- horizontální čára přistupuje do globální paměti nekoalescentně
- při čtení přes sdílenou paměť se dostaneme nad hranici 250Mpix/s

# Blokový přístup

Problém plovoucí čáry

- procházíme mnohokrát území, které jsme již zpracovali
- data nelze rozumně cacheovat, každý průchod znovu čte globální paměť

# Blokový přístup

Problém plovoucí čáry

- procházíme mnohokrát území, které jsme již zpracovali
- data nelze rozumně cacheovat, každý průchod znovu čte globální paměť

Zpracovávejme tedy jen tak velká data, která se vlezou do sdílené paměti

- můžeme zpracovat více průchodů čáry bez mezipřístupů do globální paměti
- pokud nějaký blok nemá v předešlé iteraci nově zaplavené sousedy, nemusí v aktuální iteraci nic provádět
- potřebujeme ale více iterací
- často dochází k redukci paralelismu

# Blokový přístup

Lepší efektivita

- z pohledu přístupů do globální paměti obvykle téměř nereplikujeme práci
- více průchodů čáry děláme až nad sdílenou pamětí (rychlejší, méně průchodů než při globálním zaplavování)

Triková optimalizace

- specifické zacházení s bloky kompletně pod/nad úrovní přítoku

# Dělení do regionů

Šíření vody je v realitě sekvenční (voda musí ,,dotéct")

- lze se tedy v algoritmu zbavit sekvenčního ,,rozlévání"?

# Dělení do regionů

Šíření vody je v realitě sekvenční (voda musí ,,dotéct")

- lze se tedy v algoritmu zbavit sekvenčního ,,rozlévání"?

Paralelně lze označit souvislé regiony pod úrovní přítoku vody

- nevíme ale, do kterých regionů se voda dostane
- je třeba zjistit, které regiony jsou spojeny s regionem do kterého přitéká voda

Sekvenční šíření vody se nám tak proměnilo v problém spojení regionů

- logaritmicky mnoho kroků spojování vzhledem k délce toku vody

# Vyhodnocení

Co studenti odevzdali

- sešla se široká paleta implementací různých přístupů (zde prezentovány spíše hlavní směry)
- zajímavé triky urychlující kód

Výkony implementací

- nejpomalejší a nejrychlejší implementace se liší o 4 řády
  - především díky rozdílným přístupům
- optimalizace pro CUDA architekturu a rozličné triky urychlující výpočet způsobují značný rozptyl výkonu v jednotlivých přístupech

# Vyhodnocení