Molekulový docking, mapování funkcí

Jiří Filipovič

podzim 2010

Případové studie

Ve druhé polovině 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 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 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, Jiří Filipovič, Martin prokop. FastGrid – The Accelerated AutoGrid Potential Maps Generation for Molecular Docking. *Computing and Informatics*. (to appear)
- 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 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×.

Definice map

map$(f, L_1, .., L_n)$ aplikuje n-ární funkci $f(l_1, .., l_n)$ na seznam(y) vstupních elementů $L_1, .., L_n$, $n \geq 1$.

- f nazýváme *mapovaná funkce*
- jednotlivé instance f jsou nezávislé – triviální paralelizace
- avšak potenciálně problematická implementace f

Naše aplikace

- per-element výpočty v metodě konečných prvků

Paralelní granularita

Paralelní granularita

- samotný map nám pro velké problémy poskytuje dostatečný stupeň paralelismu
- co když jsou ale nároky na on-chip paměťové zdroje funkce f příliš vysoké?
  - paměťové nároky f lze snížit paralelizací
- je-li f paralelní, může ji provádět blok threadů
  - to ale vyžaduje paralelizovatelnost na dostatečný počet vláken

Středně-zrnný paralelismus

Problematické mapované funkce

- nelze efektivně řešit v jednom threadu (příliš mnoho on-chip zdrojů)
- nelze efektivně řešit v bloku (příliš málo paralelizovatelné)

Středně-zrnná implementace

- navrhli jsme zavézt úroveň paralelismu mezi thready a bloky
- f je paralelizována na více threadů, ale více instancí f je počítáno v bloku
  - jednoduché pravidlo
  - složitější implementace

Přístup do sdílené paměti

Přístup do sdílené paměti je v případě středně-zrnných implementací složitější

- broadcast v rámci funkce není broadcast v rámci half-warpu (problém odpadá u Fermi)
- konflikty bank mohou vznikat mezi funkcemi
- složité vyčíslení stupně konfliktu

Zamezení konfliktů bank mezi funkcemi

- padding mezi datovými elementy
- prokládané uložení datových elementů

Násobení malých čtvercových matic

1 thread na 1 element výsledné matice, více matic v bloku

- všechny thready zpracovávající řádek C čtou stejnou hodnotu z A
- všechny thready zpracovávající sloupec C čtou stejnou hodnotu z B
- pro matice velikosti nedělitelné 16 požadavek na více broadcastů
  - problém pro c.c. 1.x
- konflikty mezi instancemi funkcí nenastávají

Násobení malých čtvercových matic

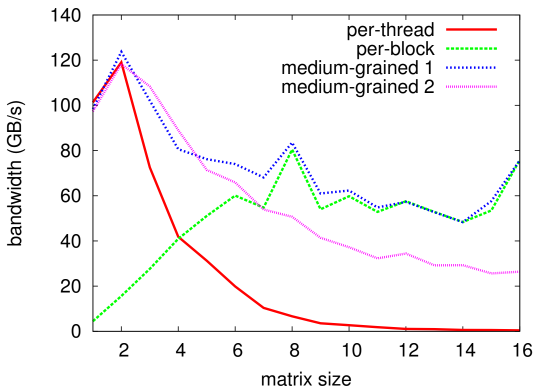
1 thread na řádek výsledné matice, více matic v bloku

- thready zpracovávající paralelně sloupec C čtou sloupec z A
- thready zpracovávající paralelně sloupec C čtou stejnou hodnotu z B
- při paddingu A a C u matic sudé velikosti žádné konflikty bank uvnitř funkcí
- konflikty mezi funkcemi u c.c. 1.x
  - násobné broadcasty z B

1 thread na sloupec výsledné matice, více matic v bloku

- stejný problém s násobnými broadcasty
- navíc obtížnější vyrovnání-se s konflikty čtení z B

Násobení matic



Někdy lze optimalizovat specifické případy

Násobení 4×4 matic

- lze obejít násobné broadcasty
  - při 4 threadech na funkci začneme násobit vektory dávající element (i, j) od pozice $j \mod 4$, respektive $i \mod 4$ počítá-li jeden thread sloupec výsledné matice
  - při 16 threadech na funkci začneme násobit vektory dávající element (i, j) od pozice $(i + j) \mod 4$
- nefunguje obecně

Granularita kernelů

Jak implementovat komplexnější problémy řešené pomocí map

- jednoduché mapované funkce jsou znovupoužitelné
- mapování komplexní funkce lze pak řešit jako sérii mapování jednoduchých funkcí

A je to efektivní?

- kernel by měl dělat ideálně dost práce na to, aby výpočet překryl paměťové operace
- u malých datových elementů problém s nízkým poměrem aritmetických operací k paměťovým

Takže všechno do jednoho kernelu?

- rozdílné paměťové nároky a paralelizovatelnost v rozdílných částech výpočtu degradují výkon

Hledání efektivní granularity

Hledání efektivního rozdělení práce do kernelů je složité

- nelze plně předem odhadnout
- manuální experimentování velmi složité
  - náchylné na chyby
  - příliš mnoho možností

Dekompozice-fúze

- problém dekomponujeme do jednoduchých základních operací
- základní operace implementujeme jako samostatné kernely
- jednotlivé kernely následně můžeme fúzovat do větších

Dekompozice-fúze

Výhody a omezení

- implementované funkce jsou znovupoužitelné
- menší náchylnost k chybám (fúze je relativně mechanická)
- balancování bohatosti implementací funkcí a dosažitelného výkonu
- stále velké množství rozdělení problému do kernelů

Automatizace

- fúze se tvoří docela mechanicky – automatizovatelná úloha

Aplikace ve FEM

Generátor fúzí

Základní schéma generátoru fúzí

- problém zadefinován jako sekvence volání funkcí v jednoduchém imperativním jazyce
- kód se přeloží do DAG zachycující předávání dat mezi funkcemi
- nad grafem je vygenerován prostor fúzí a jejich implementací (linearizace, alokace paměti a použité varianty funkcí)
- je provedena predikce výkonu každé implementace fůze
- je vygenerován kód fůzí a jejich volání
  - s nejlepším odhadem výkonu
  - pro několik nadějných s automatickým benchmarkingem

Ukázka odhadu výkonu

Odhad výkonu a naměřené časy pro rovnici

$$X = ||A \cdot B \cdot v||_2 \cdot (C \cdot D + C) \quad (1)$$

fusions configuration	predicted	measured	granularity
$\{ \ ||A \cdot B \cdot v||_2 \ \} \cdot \{ \ (C \cdot D + C) \ \}$	8.683 ns	8.077 ns	{3 3 1} {5 5 5}
$|| \ \{ \ A \cdot B \cdot v \ \} \ ||_2 \cdot \{ \ (C \cdot D + C) \ \}$	9.165 ns	8.343 ns	{3 3} 1 {5 5 5}
$||A \cdot B \cdot v||_2 \cdot \{ \ (C \cdot D + C) \ \}$	9.454 ns	8.967 ns	3 1 1 {5 5 5}
$\{ \ ||A \cdot B \cdot v||_2 \cdot (C \cdot D + C) \ \}$	9.869 ns	9.263 ns	{3 3 1 5 5 5}
no fusion	14.751 ns	13.663 ns	3 1 1 5 5 5