Optimizacija medatomskega energijskega potenciala Lennard-Jones z diferencialno evolucijo na arhitekturi CUDA

Aleš Zamuda, Aleš Čep, Janez Brest

Inštitut za računalništvo, Fakulteta za elektrotehniko, računalništvo in informatiko,
Univerza v Mariboru, Smetanova ulica 17, 2000 Maribor, Slovenija
E-pošta:{ales.cep,ales.zamuda,janez.brest}@uni-mb.si

Interatomic Lennard-Jones Potential Energy Optimization using Differential Evolution on CUDA Architecture

In this article we present an algorithm for calculation of Lennard-Jones energy potential using optimization. For optimization we use an evolution algorithm jDE, which is now implemented using CUDA architecture to increase computational speed.

Using CUDA 4.0, 25 runs on 3 different numbers of function evaluations on the problem were assessed and computation time reported. Also, the computational time measured is compared to a similar algorithm running on CPU.

1 Uvod

V tem članku predstavljamo algoritem za izračun medatomskega energijskega potenciala Lennard-Jones, poznanega v bioinformatiki. Bioinformatika je znanstvena disciplina, ki povezuje računalništvo oziroma informatiko ter biologijo. Namen povezave teh področij je izkoristiti moč in hitrost računalnikov za izračun modelov različnih kompleksnih bioloških procesov in pojavov (strukture proteinov, iskanje izraznosti genov, iskanje zdravil). Bioinformatika nudi še veliko neizkoriščenih raziskovalnih potencialov. V tem članku smo se osredotočili na problem iskanja minimalne potencialne energije med atomoma za določanje strukture proteinov.

V naslednjem poglavju obravnavamo sorodna dela, opišemo značilnosti potenciala Lennard-Jones, tehnologijo CUDA ter algoritem jDE. V tretjem poglavju opišemo delovanje predlaganega algoritma, v četrtem sledijo rezultati in na koncu še povzetek ter literatura.

2 Sorodna dela

Potencial Lennard-Jones je za reševanje z evolucijskimi algoritmi posebej zanimiv, saj ima veliko lokalnih minimumov, v katerih klasični optimizacijski algoritmi pogosto obtičijo [3]. Wales in Doye sta izračunala potenciale za do 110 atomov [11]. Uporabila sta optimizacijski algoritem Monte Carlo s 5000 koraki in dobila nove najboljše vrednosti za število atomov N={69,78,107}.

Moloi in Ali sta predlagala algoritem DELP [8] (*The local search based DE algorithm for potential minimization*), ki temelji na algoritmu diferencialne evolucije (DE) [2,9]. Razlikuje se v ustvarjanju populacije za naslednjo generacijo ter v računanju novega vektorja iz obstoječe populacije. DELP se je sicer nekajkrat znašel v lokalnem minimumu, v ostalih primerih je izračunal vrednosti, ki so bile zelo blizu takrat znanim optimumom. Na primer, za 19 atomov je bila najboljša vrednost -72.66, DELP je našel vrednost -72.60.

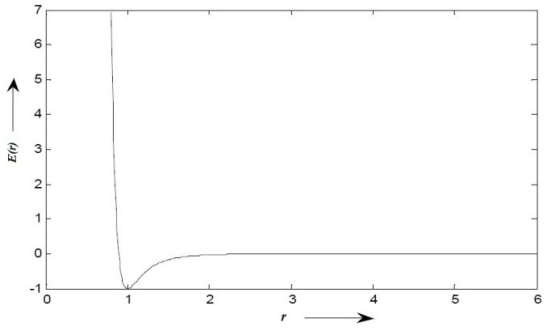
Algoritem DELP je splošno namenski, kar pomeni, da ga je možno uporabljati tudi za druge optimizacijske probleme, medtem ko je algoritem Walesa in Doyla napisan namensko za izračun potenciala Lennard-Jones. Zato so pridobljene vrednosti algoritma DELP toliko bolj zanesljive.

2.1 Potencial Lennard – Jones

Potencial Lennard-Jones [5] računa energijski potencial med pari atomov v grozdu in je sestavljen iz dveh delov, odbojnega in privlačnega:

$$E = \left(\frac{A}{r^{12}}\right) - \left(\frac{B}{r^{6}}\right),$$

kjer prvi člen predstavlja odbojni del, ki opisuje odboj atomov in se zelo poveča, ko sta si ta preblizu skupaj. Drugi člen predstavlja privlačni del potenciala, ki z razdaljo hitro pojema (Slika 1).



Slika 1: Sprememba potenciala v odvisnosti od razdalje [3].

Potrebno število izračunov potenciala določimo v našem algoritmu, pri čemer je *N* število atomov:

$$sp = \frac{N*(N-3)}{2}.$$

Problem [3], ki ga rešuje predlagani algoritem, je definiran za *N* atomov, kjer je vsak predstavljen v prostoru s tremi kartezijskimi koordinatami:

$$p_i = \{x_i, y_i, z_i\}, \quad i = 1, 2, \ldots, N,$$

kjer so vse koordinate iz množice realnih števil.

Potencial Lennard-Jones za nabor atomov izračunamo:

$$E = \sum_{i<j}\left(r_{ij}^{-12} - 2 * r_{ij}^{-6}\right) = \sum_{i=1}^{N-1}\sum_{j=i+1}^{N}\left(r_{ij}^{-12} - 2 * r_{ij}^{-6}\right),$$

kjer vidimo, da je najmanjša vrednost energije med atomoma -1 pri razdalji $r = 1$; če sta atoma bližje, začne energija skokovito naraščati.

Dimenzijo problema so v [3] delno zmanjšali s tem, da so prvi atom postavili v središče koordinatnega sistema (vse tri koordinate postavimo na nič), drugega so položili na os x ($x_2 \in [0,4]$), tretjega so pritrdili na ravnino $z = 0$ ($x_3 \in [0,4]$, $y_3 \in [0,\pi]$). Za ostale atome dobimo koordinate naključno iz intervala:

$$\left[-4 - \frac{1}{4}\left\lfloor\frac{i-4}{3}\right\rfloor, 4 + \frac{1}{4}\left\lfloor\frac{i-4}{3}\right\rfloor\right],$$

nato dobljene vrednosti zaokrožimo na absolutno gledano najbližje celo število. Vrednosti i je zaporedno število atoma.

2.2 CUDA

CUDA (*Compute Unified Device Architecture*) [4,6] je arhitektura, ki jo je razvilo podjetje NVIDIA in omogoča programiranje za grafične procesne enote (GPE). Izvajanje na njej je lahko veliko hitrejše kot na centralni procesni enoti (CPE), saj GPE omogoča vzporedno izvajanje v veliko večjem obsegu, kot to omogočajo današnji procesorji osebnih računalnikov. V tem trenutku zmorejo najboljši procesorji (CPE) 8 jeder (16 navideznih niti), medtem ko je nekaj vsakdanjega, da ima GPE več kot 128 jeder in vsako lahko izvaja 512 navideznih niti. To pomeni, da imamo na CPE na voljo 16 niti, na GPE pa 50000 niti.

CUDA deluje po principu *gostitelja* in *naprave*, kjer *gostitelja* predstavlja CPE, na kateri se izvaja program in *napravo* GPE, na kateri se paralelno izvajajo *ščepci*, ki so pravzaprav njene funkcije. Program se začne izvajati na CPE, od koder se nato kličejo ščepci na GPE; ko se ti izvedejo, spet prevzame nadzor CPE in ta cikel se nadaljuje do konca izvajanja programa (algoritem 1).

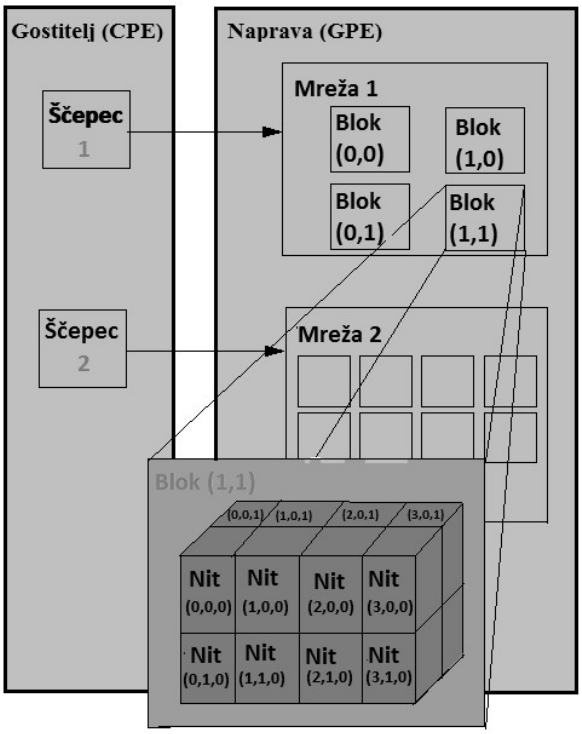
CUDA pozna dve stopnji paralelizacije: bloke in niti. Največje število za oboje je odvisno od grafične kartice. Največje število niti, ki jih lahko izvaja en blok, je pri večini novejših kartic 512. Število blokov in niti pomeni, koliko paralelnih procesov se lahko sočasno izvaja.

Algoritem 1: CUDA – primer programa.

```
1  #include <cuda.h>
2  int main(void){
3      // deklariramo spremenljivko za GPE
4      int *args;
5      // rezerviramo prostor na GPE za spremenljivko args
6      cudaMalloc((void **)&args, sizeof(int));
7
8      dim3 d1(2,2,1);
9      dim3 d2(4,2,2);
10     // klic ščepca na GPE
11     scepecNaGPE1 <<< d1, d2 >>> (args);
12     // ob zaključku ščepca se nadaljuje izvajanje na CPE
13     int a = 4 + 3;
14
15     dim d3(16,16,1);
16     // klic drugega ščepca na GPE
17     scepecNaGPE2 <<< d1, d3 >>> (args);
18
19     // sproščanje pomnilnika na napravi
20     cudaFree(args);
21     return 0;
22 }
```

Klic funkcije na GPE je prikazan v algoritmu 1 (vrstici 11 in 17). Sintaksa se rahlo razlikuje od tiste v C/C++. Najprej podamo ime klicane funkcije, v lomljenih oklepajih sledita vrednosti, ki predstavljata število blokov in niti klicanega ščepca. Za njima podamo še vhodne parametre ščepca.

Na sliki 2 je ščepec 1, pognan s parametri (2,2,1) ter (4,2,2), kar pomeni, da se bo sočasno izvajalo 64 niti.



Slika 2: Hierarhija blokov in niti znotraj mreže [6].

Pri klicu ščepca se vsakemu bloku in niti določi svoj identifikator, s katerim lahko programer določi njuno posebno obnašanje.

2.3 jDE

Diferencialna evolucija (DE) [2,9,13] je algoritem, ki se uspešno uporablja za globalno optimizacijo realno kodiranih numeričnih funkcij. Sestavljen je iz glavne evolucijske zanke, v kateri z evolucijskimi operatorji mutacije, križanja in selekcije postopno in vzporedno izboljšuje približek iskane rešitve. Evolucijski operatorji vplivajo na vsak primerek x_i, $\forall i \in [0, NP]$ v populaciji rešitev, iz katerih se zgradi nova populacija za naslednjo generacijo. Eno kreiranje novega osebka imenujemo iteracija, skupno število ovrednotenj posameznikov pa označimo s FEs. V vsaki iteraciji operator mutacije izračuna mutiran vektor $\boldsymbol{v}_{i,G+1}$:

$$v_{i,G+1} = x_{r1,G} + F * (x_{r2,G} - x_{r3,G}),$$

kjer so r_1, r_2, $r_3 \in 1, 2, \ldots, NP$ paroma in od i različni indeksi primerkov iz populacije v generaciji G, $i \in 1, 2, \ldots, NP$ in $F \in [0, 2]$. F označuje faktor ojačanja.
Po mutaciji dobljeni mutiran vektor $v_{i,G+1}$ križamo s ciljnim vektorjem $x_{i,G}$ in tako dobimo poskusni vektor $u_{i,G+1}$. Binarni operator križanja v algoritmu DE zapišemo kot:

$$u_{i,j,G+1} = \begin{cases} v_{i,j,G+1} & rand(0,1) \leq CR \text{ ali } j = j_{rand} \\ x_{i,j,G} & \text{sicer} \end{cases},$$

kjer $j \in [1,D]$ označuje j-ti iskalni parameter v prostoru z D dimenzijami, funkcija `rand(0, 1)` $\in [0, 1]$ označuje vzorčenje uniformno (psevdo) naključno porazdeljenega naključnega števila in j_{rand} izbira uniformno naključen indeks iskalnega parametra, ki ga vedno izmenjamo (da bi s tem preprečili izdelavo enakih posameznikov). CR označuje krmilni parameter stopnje križanja (povzeto po [13]).

Zadnji korak osnovnega algoritma je operacija selekcije, v kateri uporabimo ocenitveno funkcijo ter določimo vektor za naslednjo generacijo:

$$\vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G+1} & f(\vec{u}_{i,G+1}) \leq f(\vec{x}_{i,G+1}) \\ \vec{x}_{i,G} & \text{sicer} \end{cases},$$

kjer je f ocenitvena funkcija in je odvisna od problema, ki ga rešujemo. Pri vsakem klicu ocenitvene funkcije povečamo števec FEs.

jDE [1,13] je izboljšava algoritma DE, ki so ga uvedli Brest in sodelavci. Od DE se razlikuje po tem, da sta F in CR uvedeta za vsak vektor posebej in se kasneje v zanki spreminjata:

$$F_{i,G+1} = \begin{cases} F_l + rand(0,1) * F_u & rand(0,1) \leq \tau_1 \\ F_{i,G} & \text{sicer} \end{cases},$$

$$CR_{i,G+1} = \begin{cases} rand(0,1) & rand(0,1) \leq \tau_2 \\ CR_{i,G} & \text{sicer} \end{cases}.$$

Vrednosti τ_1, τ_2, F_l in F_U so konstantne vrednosti in so določene kot 0,1, 0,1, 0,1, 0,9.

3 Opis delovanja programa

Za optimizacijo smo uporabili evolucijski algoritem jDE, kjer smo kot pogoj za končanje izvajanja določili število klicev ocenitvene funkcije (FEs). Število atomov (N) smo omejili na 10, kar pomeni, da imamo v vsakem vektorju 30 lastnosti, izmed katerih je optimiranih koordinat le 27. Na koncu vsakega vektorja so dodane tri vrednosti, F, CR in izračunani potencial. Število vektorjev v posamezni generaciji (NP) smo določili na 100.

Arhitektura CUDA ščepce izvaja na procesorjih GPE in funkcije, ki jih izvaja CPE (predvsem funkcija `rand()`, ki je nujno potrebna pri evolucijskem algoritmu), na GPE ne delujejo. Zato smo z algoritmom 2 implementirali funkcijo `randomGPE()`, ki na GPE generira psevdonaključna števila po podobnem algoritmu kot funkcija `rand()` na CPE [7, 10].

Algoritem 2: Izračunanje psevdonaključnih števil.

```
1  //funkcija, ki nastavi začetno seme
2  __global__ void nastaviRandom(unsigned long *cas)
3  {
4      int id = threadIdx.x + blockIdx.x * blockDim.x;
5      randomC[id] = *cas * ((unsigned long)id+1) * 30;
6  }
7
8  //funkcija, ki vrne random vrednost med 0 in 1
9  __device__ float randomGPE()
10 {
11     int id = threadIdx.x + blockIdx.x * blockDim.x;
12
13     while(id > 256 * 512)
14     {
15         id = id - threadIdx.x;
16     }
17     randomC[id] = randomC[id] * 1103515245 + 12345;
18     return (((unsigned)randomC[id]/65535)%32768)/(float)32767;
19 }
```

Predlagano implementacijo optimizacije medatomskega energijskega potenciala Lennard-Jones z diferencialno evolucijo na arhitekturi CUDA sestavljajo koraki:

1. Najprej inicializiramo koordinate atomov. V ta namen kličemo ščepec, v katerem vsaka nit postavi eno koordinato atoma.
2. Izračunamo potenciale za začetno populacijo, ki smo jo inicializirali.
3. Preidemo v evolucijsko zanko, ki se zaključi, ko je število FEs večje ali enako maksimalnemu FEs.
4. Kličemo ščepec, kjer se izvedeta operaciji mutacije in križanja, novi vektorji se shranijo v poskusno populacijo.
5. Izračunamo potencial za poskusno populacijo.
6. Naredimo selekcijo in zamenjamo vektorje, kadar je potencial vektorja iz poskusne populacije boljši od pripadajočega vektorja iz trenutne populacije.
7. Vrnemo se na korak 3.
8. Ko zaključimo z zanko, kličemo ščepec, ki poišče najboljšo ocenitev v generaciji.

Časovno najbolj zahteven del algoritma je računanje potencialov za posamezne vektorje. Izkoristili smo maksimalno število niti, ki jih lahko zaženemo v enem bloku. Označimo L kot število lastnosti za en atom, M kot faktor pakiranja, S pa kot število blokov po preoblikovanju. Tedaj lahko zapišemo, da smo osnovno polje vektorjev velikost $L \cdot NP$ pretvorili v polje vektorjev velikosti $(L \cdot M) \cdot S$:

$$L\,NP \leq (L\,M)\,S,$$

kjer je število S manjše kot NP, saj se vektorji shranijo en za drugim znotraj enega bloka, dokler ni število lastnosti večje ali enako številu možnih niti, sicer se vektor shrani v novo

vrstico. Ščepec za izračun potenciala smo klicali s parametroma (*S,M*). S tem smo ustvarili toliko blokov, kolikor je v novem polju vrstic ter toliko niti na vsak blok, kolikor je v posamezni vrstici zloženih vektorjev. Tako vsaka nit izračuna potencial za svoj vektor.

4 Rezultati

V tabeli 1 so izpisani rezultati, ki smo jih dobili z opisano implementacijo. Kot vidimo, implementiran program uspešno deluje za optimizacijo dane funkcije [3]. Čas izvajanja implementacije smo primerjali s časom izvajanja algoritma aDE [12], ki ima računsko kompleksnost enako kot algoritem jDE. Tabela 2 prikazuje čase izvajanja implementacije algoritma aDE in naše implementacije jDE ter pohitritev, ki smo jo dosegli z našim pristopom. Programa sta bila zagnana 25 krat, prikazane vrednosti so statistično povprečje vseh zagonov. Čas označuje skupen čas zagonov. Kot vidimo iz tabele 2, je naš pristop v vseh primerih hitrejši od pristopa aDE, ki opravlja izračunavanje na CPE s pomočjo priloženih Linux ovojnic za okolje MATLAB [3]. Naš pristop je skupaj več kot 70 krat hitrejši kot pristop aDE.

Naš program smo zaganjali na grafični kartici GeForce 8800GTS ter procesorju Intel Core 2 CPE (2,13GHz). Programsko okolje, v katerem smo razvijali algoritem, je Visual Studio 2010 in CUDA 4.0.

Tabela 1: Pridobljeni rezultati pri različnem številu klicev ocenitvene funkcije.

FEs	Čas [s]	Najboljši	Medialni	Najslabši	Povprečje	Standardni odklon
5E+04	4,8	-9,4	-6,07	-3,32	-6,26	1,33
1E+05	9,6	-10,46	-8,43	-5,84	-8,06	1,3
1,5E+05	14,3	-12,34	-9,27	-6,29	-9,19	1,67

Tabela 2: Časovna primerjava in pohitritev.

FEs	Čas [s]		Pohitritev
	aDE	jDE - CUDA	
5E+04	356,2	4,8	72,69
1E+05	713,5	9,6	73,56
1,5E+05	1073,8	14,3	74,57
Skupno	2143,5	28,7	73,91

5 Zaključek

Predstavili smo algoritem jDE za arhitekturo CUDA, ki paralelno optimira medatomski energijski potencial Lennard-Jones. Prikazali smo rezultate in primerjali čase izvajanja naše implementacije in implementacije algoritma aDE [12]. Časovno gledano je naš pristop več kot 70 krat hitrejši v primerjavi z izvajanjem na CPE z okoljem MATLAB.

V prihodnje bi veljalo potencial Lennard-Jones optimirati na arhitekturi CUDA z znanimi izboljšavami algoritma jDE, s čimer bi izboljšali izračunane potenciale pri podobnih pohitritvah. Implementirani algoritem bi lahko uporabili tudi za reševanje drugih optimizacijskih problemov.

Literatura

[1] J. Brest, S. Greiner, B. Bošković, M. Mernik in V. Žumer, Self-Adapting Control Parameters in Differentialtial Evolution: A Comparative Study on Numerical Benchmark Problems, IEEE Transactions on Evolutionary Computation, 646-657, 2006.

[2] S. Das in P. N. Suganthan, Differential Evolution: A Survey of the State-of-the-art, IEEE Transactions on Evolutionary Computation 15, št. 1, 4-31, 2011.

[3] S. Das in P. N. Suganthan, Problem Definitions and Evaluation Criteria for CEC 2011 Competition on Testing Evolutionary Algorithms on Real World Optimization Problems, 4-6, 2010.

[4] T. Dobravec in P. Bulić, Strojni in programski vidiki arhitekture CUDA, Elektrotehniški vestnik 77, št. 5, 267-272, 2010.

[5] M.R. Hoare, Structure and dynamics of simple microclusters, Advances in Chemical Physics 40, 49-135, 1979.

[6] W.M. Hwu in D. Kirk, CUDA Programming Model. V: osnutek za knjigo CUDA textbook, 2008.

[7] S. Loosemore, R.M. Stallman, R. McGrath, A. Oram in U. Drepper, The GNU C Library Reference Manual, 531-532, 2007.

[8] N.P. Moloi in M.M. Ali, An Itarative Global Optimization Algorithm for Potential Energy Minimization, Computational Optimization and Applications, 119-132, 2003.

[9] R. Storn in K. Price, A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, 341-359, 1997.

[10] The GNU C Library, http://www.gnu.org/software/libtool/manual/libc.

[11] D.J. Wales in J. P. K. Doye, Global Optimization by Basin-Hopping and the Lowest Energy Structure of Lennard-Jones Cluster Containing up to 110 Atoms, J. Phys. Chem, 5011-5116, 1998.

[12] A. Zamuda, Diferencialna evolucija realnih industrijskih izzivov CEC 2011, Zbornik dvajsete mednarodne Elektrotehniške in računalniške konference ERK 2011, 2011.

[13] A. Zamuda in J. Brest, Večkriterijska rekonstrukcija numerično kodiranih proceduralnih modelov dreves z diferencialno evolucijo, Zbornik devetnajste mednarodne Elektrotehniške in računalniške konference ERK 2010, 155-158, 2010.