Проект 1: имплементација на паралелен алгоритам за ЛУ декомпозиција на матрица

Димитриј Мијоски

Универзитет Св. Кирил и Методиј

Факултет за Информатички науки и Компјутерско Инженерство

111132, [mijsoki.dimitrij@students.finki.ukim.mk](mailto:mijsoki.dimitrij@students.finki.ukim.mk)

Abstract

In this project we present an implementation in CUDA of parallel algorithm for LU matrix decomposition. Further, we benchmark it, also we benchmark an implementation for CPU, and at the end we compare both version by their execution times.

Keywords

LU decomposition, linear algebra, Gaussian elimination, cuda

Резиме

Во овој проект претставена е имплементација во КУДА на паралелен алгоритам за ЛУ-декомпозиција на матрица. Понатаму измерено е време на извршување, измерено е и време на имплементација за ЦПУ и на крајот дадена е споредба на двете верзии според времињата на извршување.

Keywords

ЛУ-декомпозиција, линеарна алгебра, гаусова елиминација, КУДА

# Вовед

За почеток ќе кажеме што е ЛУ декомпозиција на матрица на практичен начин. За теоретски опис најдобро е да се види книга за линеарна алгебра (Gilbert Strang, Introduction to Linear Algebra). ЛУ-декомозција ја дели една матрица на две, L и U така што производот на матрици L\*U=А. L е долно тријаголна матрица, додека U е горно тријаголна. Двете матрици можеме да ги добиеме ако направиме Гаусова елиминација на A така што U е матрицата што се добива со Гаусовиот метод, додека во L ги чуваме коефициентите со кои сме елиминирале, па така целата информација за А е зачувана. Ќе прикажеме со цртеж еден чекор од елиминацијата.

Во погорниов израз ни е прикажан еден чекот од елиминацијата. Ние

1. го земаме првиот ред;
2. го множиме со коефициенти;
3. и го одземаме од останатите редови под него така што првите елементи да станат нули.

Во ни е матрицата после елиминација со првата редица, а во првата колона од ни се коефициентите со кои сме ја множеле првата редица за да елиминираме. Во **втор чекор** со **втората редица** ќе ги **елиминираме сите под неа** така што и вторите елементи да дојдат 0. Во L ќе запишуваме во втората колона.

Испишаниот алгоритам работи само со квадратни и матрици. ЛУ може да се прави и врз неквадратни иако најчесто се користи со квадратни. За неквадратни матрици алгортимот е скоро ист и промените се тривијални – матрицата Л останува квадратна, само У се проширува или се стеснува да има исти димензии со А. Дополнително, алгоритамот не може да се справи ако е потребна преместување (пермутации) на редици. Сепак, кога е потребно преместување, веќе немаме обична декомпозиција туку имаме P\*A=L\*U каде P e производ од елементарни пермутациони матрици. За оваа декомпозиција алгоритамот би бил посложен, посебно паралелната верзија.

# Паралелен алгоритам

Прво да забележиме дека кога од една редица одземаме погорна, тоа можеме да го направиме целосно паралелно – тоа е класична векторска операција. Претходно треба да знаеме само една работа, коефициентот со кој множиме. Да забележиме и дека елиминацијата на една редица со погорна и на друга редица со истата погорна се независни операции, што значи ако елиминираме со редица, паралелно можеме да ги елиминираме сите под неа ако ги имаме пресметано соодветните коефициенти. Со изразите погоре ни се прикажани баш првите два временски чекора. Па паралелниот алгоритам оди вака.

Значи имаме временски чекори. Работната комплексност е истата како серискиот алгоритам, таа е . Ова значи дека на теоретска машина со бесконечен број на процесори временската комплексност кај серискиот алгоритам од ја кратиме на – ова значи постапката е многу погодна за паралелизирање.

Во прилог со документот е дадена основна имплементација. Овде ќе ги дадеме само функциите за ЛУ на ГПУ.

template <typename T>

\_\_global\_\_ void lu1(T \* L, T \* U, int n, int i)

{

//vo ovoj cekor presmetivame koef so koj kje mnozime gorna redica i kje gi eliminrame redicite pod nea

//idx ni e redica pocnuvanjki od A[i], i ni e kolona

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

int redica = idx + i;

if (redica < n) {

L[redica\*n + i] = U[redica\*n + i] / U[i\*n + i];

}

}

template <typename T>

\_\_global\_\_ void lu2(T \* L, T \* U, int n, int i)

{

//ovde idx ni e globalen indeks na niskata i globalen indeks vo matricata (row major) pocnuvajki od U[i+1][i]

int idx = blockDim.x \* blockIdx.x + threadIdx.x;

if (idx >= (n-i-1)\*(n-i)) {

return;

}

//(n - i) sirina na redica sto ja obrabotivame. delime so sorinata i gi dodavame pomestuvanjeata t.e. offsetite

int redica = idx / (n - i) + i + 1;

int kolona = idx % (n - i) + i;

U[redica \* n + kolona] -= L[redica \* n + i] \* U[i \* n + kolona];

}

template <typename T>

void lu(T \* d\_L, T \* d\_U, int n, const int NUMTHREAD = 128)

{

cudaMemset(d\_L, 0, sizeof(T)\*n\*n);

for (int i = 0; i < n-1; i++) {

int lu1blocks = (n-i + NUMTHREAD - 1) / NUMTHREAD;

int lu2blocks = ((n-i-1)\*(n-i) + NUMTHREAD - 1) / NUMTHREAD;

lu1<T> << <lu1blocks, NUMTHREAD >> >(d\_L, d\_U, n, i);

lu2<T> << <lu2blocks, NUMTHREAD >> >(d\_L, d\_U, n, i);

}

lu1<T> << <1, 1 >> >(d\_L, d\_U, n, n-1);

}

Изворен код 1. Функциите за пресметување ЛУ на ГПУ. На почеток А влегува во параметарот за У.

Алгоритмот нема никакви специјални оптимизации. Секако може да се оптимизира пристапот до меморија на повеќе начини, но тоа не е барање на проектов. Се бара само основна и точна имплементација на паралелен алгоритам. Во целосниот изворен код се наоѓаат и функциите со кои беше проверувана точноста додека го изработував алгоритмот. Тоа беше правено вака. За најдени L и U, наоѓаме A’=L\*U и A’ ја споредуваме со А. Ако броевите се исти или приближно исти (се наоѓа најголема разлика), алгоритмот е во ред. Нумеричка нестабилност се јавуваа заради заокружувањата (работиме со float). Декомпозицијата PA=LU може да се направи нумерички стабилна ако избираме подобри „пивоти“ односно редици со кои елиминираме.

# Мерења

Освен верзија за ГПУ, напишана е еквивалентка верзија за ЦПУ. Потоа од овие две верзии имаме 4 групи на експерименти и тоа:

1. ЛУ на ЦПУ со компајлирање без оптимизации (Debug)
2. ЛУ на ЦПУ со компајлирање со оптимизации (Release)
3. ЛУ на ГПУ со компајлирање без оптимизации (Debug)
4. ЛУ на ГПУ со компајлирање со оптимизации (Release)

Во секоја група за еден експеримент сметаме едно извршување на ЛУ врз матрица со големина Н. Беа направени експерименти со различни Н и за секое Н експериментот беше повторуван најмалку 3 пати (за големите матрици), а најмногу 125000 пати (за малите матрици). Повторуваме за да испегламе шум при мерење со тоа што од сите повторувања на еден ист експеримент пресметуваме просечно време. Добиени се следниве резултати.

Слика 1. Времиња на извршување на првите екперименти.

Ќе забележиме дека кај извршните датотеки за дебагирање јасно е забрзувањето на ГПУ. Сепак кај оптимизраните извршни датотеки времињата на ЦПУ и ГПУ се слични. Ако добро гледаме, ЦПУ е побрзо за матрици до кај 1400, и таму некаде графиците се прекрстуваат. Заради ова направен беше уште еден обид со низа екперименти, овој пат минималниот број на повторувања беше 100, а максималниот 100000.

Слика 2. Времиња на извршување на втор обид на експерименти.

Тука веќе е појасно дека кај поголеми матрици графичката е подобра. Со регресиона анализа можеме и прилично прецизно да добиеме модел за времињата на извршување за непознати Н.

Ќе дадеме и графикон на брзина за двата обиди, и тоа брзина на меморијата (bandwidth) во мерката гигабајти во секунда. Таа мерка ја вадиме така што броиме колку читања и запашувања имаме во матриците во функцијата lu1 и во lu2. Бројот е пресметан вака.

Бројот на бајти е бројот на зборови помножен со 4. Пропусниот опсег е количина на мемориски трансфери во бајти поделен со времето во секунди.

Слика 3. Брзини на првиот обид експерименти.

Слика 4. Брзини на вториот обид експерименти.

# Заклучок

Во овој проект беше осмислен паралелен алгоритам за ЛУ, напишана основна имплементација во КУДА, измерена и споредена со ЦПУ.