Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

Алгоритм множення розріджених матриць на графічному процесорі

Текстова частина до курсової роботи

за спеціальністю «Комп'ютерні науки»

Керівник курсової роботи
проф., д. ф.-м. н.
Малашонок Г. І.

_____

(Підпис)

"___" _____ 2022 року

Виконав студент 1 курсу

Магістерської програми

Комп'ютерні науки

Сичов В.С.

"___" _____ 2022 року

Київ 2022

Зміст

Анотація

У дані роботі було розглянуто реалізацію алгоритму множення розріджених матриць на графічному процесорі та на центральному процесорі. Порівняно їхню роботу та надано оцінку швидкодії множення в залежності від розміну досліджуваної матриці

Вступ

Наш світ тісно пов'язаний з обчисленнями, які необхідні виконуватися з тих чи різних причин. Цьому неабияк сприяє розвиток таких популярних технічних галузей, як штучний інтелект чи Big Data. Для якісної роботи вони потребують швидких та точних обчислень, які перетворюються на одні з найважливіх завдань нинішнього часу. Щоб прискорити обчислення, використовують графічні процесори з тисячами ядер. Зокрема, технологію CUDA, яку розробила компанія NVIDIA.

Саме цій технології присвячене дане дослідження, а саме: реалізації алгоритму множення розріджених матриць на графічному процесорі, який дозволяє вирішувать різноманітні задачі для використання штучного інтелекту

Розділ 1. Алгоритм множення матриць

1.1 Загальні відомості

Означення. Матрицею розміру $m \times n$ називається множина з mn елементів a_{ij}, які розміщені у вигляді прямокутної таблиці з m рядків та стовпців, m і n – розмірність матриці.

Означення: Матриця називається розрідженою, якщо більша частина елементів цієї матриці є нулями. Приклад розрідженої матриці A^R:

$$A^R = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{mn} \end{bmatrix}$$

Дана робота розглядає алгоритм множення розріджених матриць, тобто матриць, значна частина яких є нулями. Але перед тим, як почати розглядати алгоритм множення розріджених матриць, потрібно розібратися, як їх потрібно зберігати. Якщо розмірність матриці мала($4 \times 4,\ 10 \times 10$), то це питання не викликає труднощів: створив масив масивів і працюєш з ним. Проте при збільшення розмірності матриці, наприклад 512×512 виникають труднощі. Збереження такого обсягу даних не є оптимальним, особливо зважаючи на те, що більшість елементів – нулі. Крім того, прямолінійна обробка масиву масивів такого розміру призводить до підвищення складності алгоритму, і відповідно – до уповільнення роботи програми. Що робити в такому випадку? Змінювати спосіб збереження розрідженої матриці, а відповідно – і спосіб обробки і роботи з нею. В даній роботі використовується спосіб збереження розріджених матриць під назвою CSR, але для прикладу також буде інформацію про інші способи збереження розріджених матриць.

1.2 Способи збереження розріджених матриць

Як говорилося вище, спосіб збереження розрідженої матриці відіграє одну з ключових ролей у розробці алгоритму роботи з нею, тому розглянемо

поверхнево основні способи збереження розрідженої матриці та один(спосіб, який використовується в роботі) більш детально.

1) Словник по ключах(англ. Dictionary of Keys - DOK) – структура, яка будується на основі словника(асоціативного масиву, мапи)(Dictionary<Key<row,column>,Value<matrix value>>). Основний принцип полягає в побудові словника, який складається з ключа-пари типу ⟨рядочок, стовпчик⟩ а значення в словнику – це значення з матриці, яке знаходиться на перетині рядка і стовпчика;
2) Список списків(англ. List of Lists - LIL) – структура, яка будується на основі списку списків (List<List>) за наступним принципом: кожен елемент зовнішнього списку являється списком, який в свою чергу містить елементи типу ⟨стовпчик, значення⟩;
3) Список координат(англ. Coordinate list - COO) – простий тип зберігання. Будується на основі списку(List<row,column,value>).Основний принцип полягає в тому, що кожен елемент списку - це <рядок, колонка, значення > з матриці.
4) Стиснене зберігання стрічкою(англ. Compressed sparse row – CSR, Compressed row storage - CRS, Yale format - Єльський формат) – досить складний спосіб зберігання розрідженої матриці. Саме цей формат зберігання використовується в цій роботі. Принцип такого зберігання полягає в тому, що ми використовуємо не одну структуру даних, а декілька(три, якщо точніше), тобто ми розбиваємо розріджену матрицю $M^{n\times m}$ три наступних масиви:
   a) Масив значень(array[value]) – масив розміру N, який зберігає в собі не нульові значення елементів матриці, які були взяті підряд із першої не нульової стрічки, потім із другої ненульової стрічки, потім із третьої і так далі;
   b) Масив індексів стовпчиків(array[columns]) – масив розміру N, який зберігає в собі номери стовпчиків, що відповідають номерам стовпчиків елементів розрідженої матриці, які зберігаються в масиві значень;

c) Масив індексів рядків(array[rows]) – масив розміру N + 1(кількість рядків + 1). Для кожного індекса і зберігається кількість ненульових елементів в стрічках з першої до і – 1 стрічки включно. Також варто відмітити, що перший елемент масиву рядків завжди рівний нулю.[1]

Розглянемо приклад зберігання матриці у форматі CRS:

Нехай є матриця $S = \begin{bmatrix} 5 & 6 & 0 \\ 0 & 7 & 0 \\ 0 & 1 & 8 \end{bmatrix}$, тоді

Масив значень = {5, 6, 7, 1, 8};

Масив індексів стовпчиків = {0, 1, 1, 1, 2},

Масив індексів рядків = {0, 2, 3, 5}

Приклад коду нижче показує, як можна працювати з матрицею в такому форматі.

```
/*
crsm looks like
{
 int n,
 int m,
 int nnz,
 double aval[],
 double aicol[],
 double airow[]
 }
*/
// b += Av
void smdv(const crsm *A, double *b, const double *v)
{

    for(int row = 0; row < n; ++row)
    {
        for(int i = A->airow[row]; i < A->airow[row+1]; ++i)
        {
            b[row] += A->aval[i] * v[A->aicol[i]];
        }
    }
}
```

Після того, як ми розглянули способи збереження розріджених матриць, потрібно розібратися, як їх можна перемножити, проте спершу розглянемо, як множаться не розріджені матриці.[2]

1.3 Множення не розріджених матриць

Нехай дано дві прямокутні матриці A та B над полем комплексних чисел, такі, що розмірність матриці A = $m \times n$, а розмірність матриці B = $n \times q$,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1q} \\ b_{21} & b_{22} & \cdots & b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nq} \end{bmatrix}$$

тоді добутком матриці A на матрицю B , буде така матриця C розмірності $m \times q$

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mq} \end{bmatrix}$$

де кожен елемент матриці C буде визначатися, як:

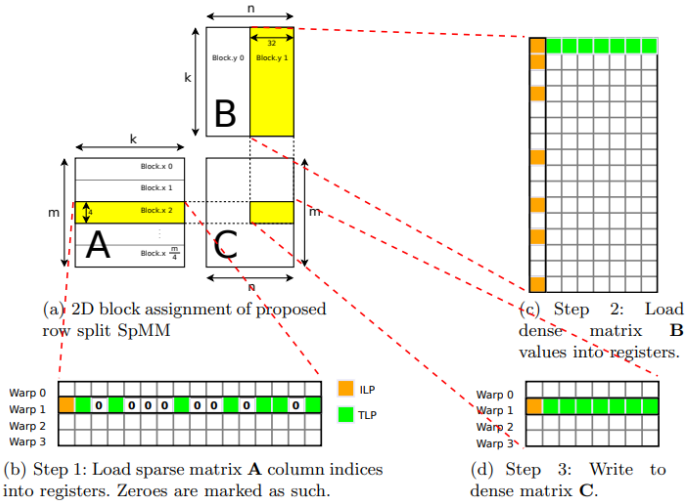
$c_{ij} = \sum_{r=1}^{1} a_{ir} b_{rj}$ $(i = 1,2, \dots m; j = 1,2, \dots q)$, тобто добуток матриць A та B буде складатися з усіх можливих комбінацій скалярних добутків векторів(рядків) матриці A на всі вектори(стовпчики) матриці B. Крім того, варто розуміти, що не можна помножити будь які дві матриці. Обов'язковою умовою множення матриць є те, що кількість стовпців у першому множнику дорівнює кількості рядків у другому множнику. Також операція множення матриць не є комутативною, тобто $A \times B \neq B \times A$(точніше, так може бути, але не завжди). Після того, як ми ознайомилися з множенням звичайних матриць, час перейти до множення розріджених матриць.[5]

1.4 Множення розріджених матриць

У бібліотеці CUSPARSE є дві основні функції множення розріджених матриць : csrmm та csrmm2. Вони обидві реалізовують різні алгоритми множення матриць. Про ці лгоритми ми іпоговоримо.

1) Перший алгоритм називається алгоритмом множення розрідженої матриці з розбиттям рядків(Row-splitting SpMM). Основний принцип полягає у призначенні кожного рядка різному потоці. Даний алгоритм множення розрідженої матриці є одним з найбільш поширених і найбільш простих алгоритмів множення розріджених матриць на графічному процесорі. Проте даний підхід має свої певні недоліки, але дана робота не про них.

Типове розділення рядків – це лише крайній лівий стовпець матриці B із оранжевими клітинками, заміненими зеленими, як показано на рисунку нижче.



(a) 2D block assignment of proposed row split SpMM

(b) Step 1: Load sparse matrix **A** column indices into registers. Zeroes are marked as such.

(c) Step 2: Load dense matrix **B** values into registers.

(d) Step 3: Write to dense matrix **C**.

2) Наступний алгоритм являється алгоритмом множення розріджених матриць на основі злиття(Merge-based SpMM). Суть алгоритму на основі злиття полягає в явному і рівномірному розподіленні ненульових значень елементів між паралельними процесорами. Сам процес реалізується таким чином, що виконання виконується на основі двофазового розкладання: на першому етапі (PartitionSpmm) алгоритм злиття розподіляє роботу між потоками таким чином, що T роботи призначається кожному потоку, і на основі цього розподілу виводить початкові індекси кожного елементу. Після виконання відповідної координації, робота виконується на другому етапі. Виходячи з цієї реалізації, в алгоритмі реалізовуються наступні дизайнерські підходи: підхід в доступі до пам'яті, використання регістрів та накладні витрати на доступ до пам'яті. Наступний приклад псевдокоду показує реалізацію даного алгоритму:

Input: Sparse matrix in CSR A ∈ $R^{m\times k}$ and dense matrix B ∈ $R^{k\times n}$

Output: C ∈ $R^{m\times n}$ such that C ← AB.

procedure SpmmMerge(A, B)

//Phase 1: Divide work and run binary-search

```
limits[] ← PartitionSpmm(A, blockDim.x)
for each CTA i in parallel do  //Phase 2. Do computation
    num rows ← limits[i + 1] − limits[i]

    // Read A and store to shared memory
    shared.csr ← GlobalToShared(A.row ptr + limits[i], num rows)
    end ← min(blockDim.x, A.nnz - blockIdx.x × blockDim.x)
    if row ind < end then
        col ind ← A.col ind[row ind] //Read A if matrix not finished
        valA ← A.values[row ind]
    else
        col ind ← 0
        valA ← 0 13
    end if
    for each thread j in parallel do
        for j = 0, 1, . . . , 31 do
            new ind[j] ← Broadcast(col_ind, j)
            new val[j] ← Broadcast(valA, j)
            valB[j] ← B[col_ind][j] × new val[j]
        end for
    end for
    terms ← PrepareSpmm(shared.csr)
    carryout[i] ← ReduceToGlobalSpmm(C, valB, valB)
end for
```

FixCarryout(C, limits, carryout)

return C

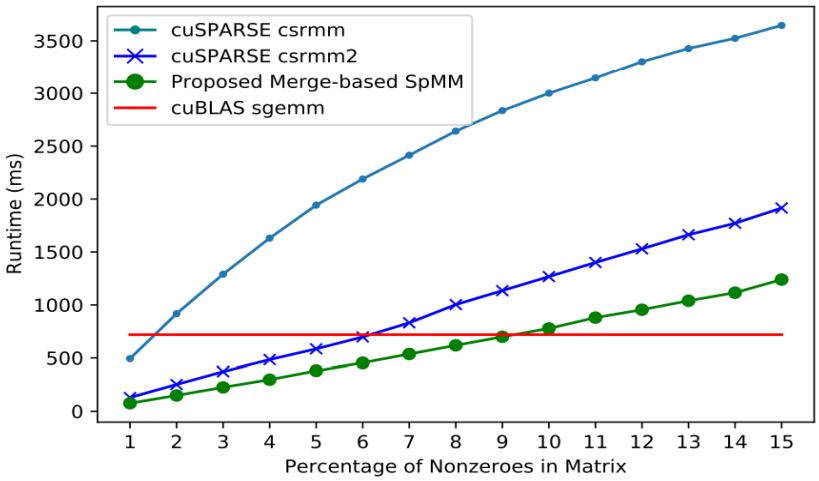
end procedure

Таблиця нижче демонструє відмінності між цими двома алгоритмами:

	Множення розріджених матриць	
Операція	Розділення рядків	Злиття
Зчитати A.col ind і A.val	1	T(1)
Зчитати B	$0 < L \leq 32$	32T(32)
Записати C	1	32T(32)
Використання регістрів	64	64T(64)
Накладні витрати на доступ до пам'яті	0	$\frac{B.ncols \times A.nnz}{B \times T}$

Ця таблиця показує кількість незалежних інструкцій на кожен потік у графічному процесорі для множення розріджених матриц зі значенням за замовчуванням, показаним у дужках, а також використання регістра та додаткову кількість доступів до пам’яті відповідно до алгоритму розбиття рядків. T — кількість робочих елементів на потік (як правило, вказується як параметр налаштування алгоритму). L – це число ненульових модулів 32 у рядку A, яке ми обчислюємо. B – розмір CTA(compute thread array). Типові значення для T в алгоритмі множення розріджених матриць становлять 1 , тоді як типове значення для B дорівнює 128. T не можна встановити довільно високим, оскільки високе використання регістра призводить до меншої зайнятості. A.nnz — кількість ненульових елементів у розрідженій матриці A. B.ncols — кількість стовпців матриці B.

Також, для порівняння, розглянемо графік , який показує функціональну залежність між часом виконання обробки алгоритму множення розріджених матриць та кількістю не нульових елементів у розрідженій матриці.



Як показано на рисунку, час виконання алгоритму множення розріджених матриць збільшується зі збільшенням кількості ненульових елементів, проте швидкість зростання часу виконання різна. У алгоритмі розбиттям рядків час виконання в залежності від ненульових елементів зростає швидше, ніж у алгоритмі злиттям.[5]

Розділ 2. Опис CUDA

2.1 Загальні відомості

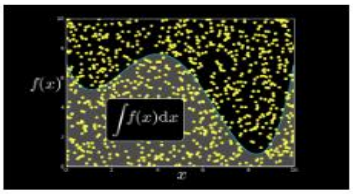
CUDA(англ. *Compute Unified Device Architecture*) — технологія з відкритим вихідним кодом, розроблена компанією NVIDIA для паралельних обчислень використанням їхніх графічних процесорів(відеокарт). Реалізація програмно-апаратної архітектури дозволяє значно підвищити обчислювальні можливості графічних процесорів при обробці великих масивів даних. Підхід в реалізації полягає у включенні власних підпрограм, написаних мовою С, у команди, які запускаються на графічному процесорі. Самі команди реалізовані на діалекті мови С. Архітектура платформи дозволяє розробнику власноруч використовувати обчислювальні потужності відеокарти, зокрема керувати виділенням і доступом до пам'яті.

Перша версія CUDA SDK була випущена 15 лютого 2007 року. Основою CUDA API є розширена мова С. Оскільки звичайний компілятор не може обробляти команди ,написані цим діалектом, в основі платформи CUDA є свій компілятор, створений компанією NVIDIA, NVCC. В основі даного компілятора лежить компілятор Open64.

Структура технології включає реалізацію різних бібліотек для вирішення математичних задач різних областей. Ось основні математичні бібліотеки CUDA:

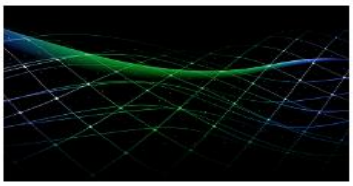
1. cuRAND – бібліотека для генерації випадкових чисел з використанням GPU;
2. CUDA Math Library - стандартні математичні функції з використанням GPU;
3. cuFFT – функції, реалізовані для обрахунків швидкого перетворення Фур'є. Реалізація дозволяє підвищити швидкість обрахунків до 10 разів швидше, ніж при реалізації з використанням CPU.
4. cuBLAS – бібліотека функцій для операцій базової лінійної алгебри;

5. cuSPARSE – бібліотека реалізації операцій базової лінійної алгебри, оптимізована під роботу з розрідженими матрицями;
6. AmgX – бібліотека алгоритмів симуляції та неструктурованих методів обробки даних;
7. cuTENSOR – бібліотека тензорних обрахунків;
8. cuSOLVER – бібліотека з алгоритмами для роботи з щільними та розрідженими марицями.

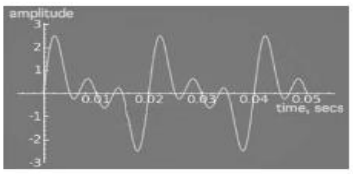


cuRAND

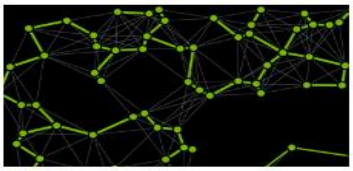
NPP



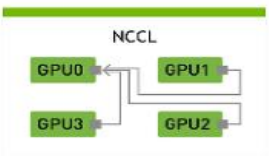
Math Library



cuFFT



nvGRAPH



NCCL

2.2 Переваги GPU для паралельних обрахунків

Використання графічних процесорів для паралельних обрахунків дозволяє використовувати більшу пропускну здатність пам'яті та інструкцій в порівнянні з центральним процесором, за умови наявності такого ж рівня ціни та потужності. Таким чином, використання великою кількістю програм цих пропускних і потужносних можливостей дозволяє підвищити швидкість і продуктивність виконання програми. Крім того, програмування на GPU забезпечує гнучкі можливості для розрозбки, в порівнянні, з тим самим, FPGA(англ. *Field-Programmable Gate Array*). Різниця у швидкострі роботи

обрахунків на GPU та CPU полягає також у відмінності їхнього застосування. Якщо взяти до уваги центральний процесор(CPU), то він в більшій мірі призначений для виконання послідовних обрахунків використанням сотень потоків, тим часом, як графічний процесор(GPU) призначений для паралельних обрахунків тисяч потоків і слабко себе показує при послідовних операціях. Оскільки GPU розроблений для паралельних обрахунків, то його конструкція і архітектура використовує більшу кількість транзисторів для обробки даних, та перевагу контролю потоку та створенню кешу даних. Нижче наведена схема розподілу обчислювальних потужностей центрального процесора(CPU) та графічного процесора(GPU):

2.3 Архітектура низького рівня

При розробці програм з використанням CUDA, архітектура програми поділяється на чотири базові компоненти: ядро, блок, потік та сітка.

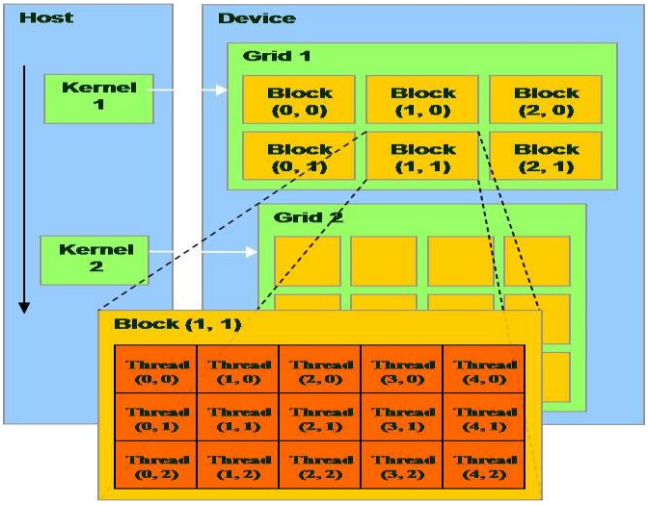
Сітка (grid) – контейнер для збереження в собі всіх даних, які необхідно обробляти. На вершині ієрархії потоків даної архітектури знаходить сітка. В свою чергу, сітка може містити декілька потоків та блоків. Розмірність сітки може варіюватися в залежності від ситуації, тобто кількість вимірів може бути одновимірною, двовимірною або тривимірною. Розмірність сітки залежить від розмірності даних або складності обрахунків, які необхідно виконати. Для прикладу, розмірність сітки для матриці буде рівна двом.

Кожна сітка може складатися з декілька блоків. Також існує ліміт на кількість блоків, який обмежено для кожної розмірності сітки і залежить лише від конкретного GPU, який використовується для обробки даних. Розмірність блоків залежить від розмірності сітки, тобто при розмірності сітки 2, розмірність блоків буде теж 2.

В свою чергу, кожен блок містити в собі кілька потоків. Обмеження по максимальній кількості потоків залежить від характеристик обладнання, що використовується для обрахунків, зазвичай максимальна кількість потоків дорівнює 2^10, або 1024 потоки. Як і в сітки та блоків, розмірність потоків залежить від розмірності блока, тобто розмірність потоків у блоці розмірності 2 буде рівна теж 2. Важливим зауваженням є те, що кожен блок повинен визначатися як незалежний елемент від іншого блоку. Така особливість пов'язана з тим, що код програми для виконання всередині блоку може виконуватися не на одному пристрої, а на декількох пристроях, які можуть не спілкуватися один з одним або витрачати на взаємодію багато часу та обчислювальних можливостей. Також існує два способи виконання коду всередині блоку: паралельне виконання та послідовне.[1]

Ядром (kernel) - це функція, яку виконує лише одним потік CUDA. Виконання ядра відбувається паралельно на різних потоках. Виконання кожного такого потоку має свій власний ідентифікатор, доступ до якого може бути отриманий всередині самого ядра.

Нижче на малюнку, приведено приклад архітектури CUDA:



Крім того, також можна відмітити, що платформа для розробки CUDA не надає функціональних інструментів для синхронізації на низькому рівні, тобто рівні сітки для більшості графічних процесорів. Даний підхід у наданні функціональних інструментів доступний лише на деяких нових графічних процесорах. Також підхід до всіх розрахунків варто планувати, виходячи з того, що будь-які комунікації між блоками відсутні.

Причина того, що для програмування на платформі CUDA відсутні інструменти для міжблочної синхронізації полягає в тому, що блоки, які виконують обрахунки, виконуються не одночасно. Графічний процесор, який виконує обрахунки, сам ставить блоки у чергу і при цьому може виконувати обрахунки у випадковому порядку, згідно з наявними ресурсами на відеокарті : ступеню завантаженості та кількості вільної пам'яті. В ситуаціях, коли один блок намагається очікувати результат виконання іншого блоку, програма може почати гальмувати або і взагалі зависнути(deadlock), так як немає ніяких гарантій, що блок, якого очікують, взагалі завершить своє виконання або виконається до того моменту, як завершить виконання даного блоку. Проте комунікація між блоками все ж таки може бути допущена. __syncthreads() – функція, яка може під час виконання у заданій точці програмі синхронізувати всі потоки всередині блоку. По факту, даний підхід – це єдиний надійний і безпечний спосіб синхронізації виконання обчислень без прямої необхідності їхнього завершення.[1]

2.4 Опис програмної моделі

Технологія CUDA (компілятор nvcc.exe) реалізовує ряд додаткових діалектів мови C, які необхідні для написання коду для графічних процесорів:

1. Класифікатори змінних, які служать для вказівки типу використовуваної пам'яті графічного процесора;
2. Класифікатори функцій, які показують, як і звідки буду виконуватися функції для обрахунків на графічному процесорі ;
3. Класифікатори, які класифікують ядра графічного процесора;
4. Додаткові типи використовуваних змінних;
5. Вбудовані змінні, які ідентифікують потоки, блоки та інші параметри при виконанні коду в ядрі графічного процесора.

Специфікатор функцій показують, яким чином і з якого місця можуть бути викликані функції:

- __global__ - виконання функція відбувається на графічному процесорі, викликається функція центральному процесорі;
- __host__ - виконання функції відбувається на центральному процесорі, виклик функції відбувається також на центральному процесорі;
- __device__ - виконання функції відбувається на графічному процесорі, виклик відбувається також на графічному процесорі.

Класифікатори для запуску ядра служать, як опис для кількості блоків, потоків і пам'яті, яку потрібно виділити при роботі функцій на графічному процесорі. Синтаксис команди для запуску ядра виглядає наступним чином:

myKernelFunc <<< gridSize, blockSize, sharedMemSize, cudaStream >>> (float * param1, float * param2),

де

- myKernelFunc – функція ядра, яка має специфікатор __global__;
- gridSize - розмір сітки блоків, яку виділено для розрахунків;
- blockSize - розмір блоку, виділеного для розрахунків у блоці;

- sharedMemSize – розмір додаткової видуленої пам'яті, яка виділяється при запуску ядра;
- cudaStream – змінна, яка задає потік, в якому буде проведений виклик функції;

Проте, певні змінні для виклику ядра можна опускати, для прикладу, cudaStream і sharedMemSize.[1]

Крім того, ще є вбудовані змінні:

- blockDim - розмір блоку. Зберігає в собі розмір блоку, виділеного при поточному виклик ядра;
- gridDim - розмірність гріда, тип змінної - dim3. Зберігає розмір гріда, виділеного при поточному виклику ядра;
- blockIdx - індекс поточного блоку, який виконується в даний момент на графічному процесорі, має тип uint3;
- warpSize - розмірність warp, має тип int;
- threadIdx - індекс потоку, що виконується в даний момент на графічному процесорі, має тип uint3. [1]

2.5 Приклад програмного коду

Наступний зразок коду, використовуючи вбудовану змінну threadIdx, множить два вектори A і B розміром N і зберігає результат у векторі C. Тут кожен з N потоків, що виконують VecMult(), виконує одне парне множення.

```
// визначення ядра
__global__ void VecMult(float* A, float* B, float* C)
{
	int i = threadIdx.x;
	C[i] = A[i] + B[i];
}

int main()
{
	...

	// виклик ядра з N потоками
	VecMult <<<1, N>>>(A, B, C);
```

...
}[1]

2.6 Приклад застосування

Всі електронні пристрої, де ми працюємо з графікою або нам необхідно обчислювати велику кількість характеристик з плаваючою крапкою. Для прикладу, буде наведено галузі, де активно використовуються обрахункові потужності графічних процесорів:

- Data science та аналітичні дані;
- Big data;
- Оборонна промисловість;
- Медицина;
- Фінанси;
- Кібербезпека;
- Кіберспорт;
- Game development;
- Deep learning та machine learning ;
- Моделювання.

2.7 JCUDA

Для реалізації алгоритму множення в даній роботі було реалізовано за допомогою бібліотеки JCUDA – обгортки бібліотеки CUDA, яка використовує мову Java, запускається за допомогою JVM(Java virtual machine) та має Java подібний синтаксис. Дана бібліотека має відкритий вихідний код і розповсюджується за безкоштовною ліцензією. Також тут реалізовано найпопулярніші бібліотеки CUDA: jCuSPARSE, jCuSolver, jCuFFT, jCuBLAS та інші.

Розділ 3. Практична частина

Для порівняння роботи було написано два алгоритми множення розріджених матриць: за допомогою графічного процесора та за допомогою центрального процесора.

3.1 Реалізація алгоритму на центральному процесорі

Реалізація алгоритму була зроблена на мові C# з використання об'єктно-орієнтованого підходу. Було створено клас SparseMatrix, який зберігає розріджену матрицю в форматі CSR. Клас з приватними властивостями та конструктор мають наступний вигляд.

```
public class SparseMatrix
   {
       private  const int MAX = 1024;
       private int[][] data;
       private int row, col;
       private int len;

       public SparseMatrix(int r, int c)
       {
           row = r;
           col = c;
           len = 0;
           data = new int[MAX][];
           for (int i = 0; i < MAX; i++)
           {
               data[i] = new int[3];
           };
       }
}
```

Метод Insert виконує вставку даних в матрицю. В якості параметрів, він приймає рядок, стовпчик та значення елемента, який треба вставити. Метод нічого не повертає.

```
public void Insert(int r, int c, int val)
{
    if (r > row || c > col)
    {
        Console.WriteLine("Wrong entry");
    }
    else
    {
        data[len][0] = r;
        data[len][1] = c;
        data[len][2] = val;
        len++;
    }
}
```

Метод Transpose потрібен для множення матриці. Він транспонує матрицю. В якості вхідних параметрів, він нічого не приймає. Повертає транспоновану матрицю.

```
public SparseMatrix Transpose()
    {
        SparseMatrix result = new SparseMatrix(col, row);
        result.len = len;
        int[] count = new int[col + 1];

        for (int i = 1; i <= col; i++)
            count[i] = 0;

        for (int i = 0; i < len; i++)
            count[data[i][1]]++;

        int[] index = new int[col + 1];
        index[0] = 0;

        for (int i = 1; i <= col; i++)
            index[i] = index[i - 1] + count[i - 1];

        for (int i = 0; i < len; i++)
        {
            int rpos = index[data[i][1]]++;
            result.data[rpos][0] = data[i][1];
            result.data[rpos][1] = data[i][0];
            result.data[rpos][2] = data[i][2];
        }
        return result;
    }
```

Метод Multiply виконує множення матриць. В якості вхідного параметра він отримує матрицю, на яку потрібно помножити поточну матрицю. Повертає метод результат множення двох матриць, тобто екземпляр класу SparseMatrix.

```
public SparseMatrix Multiply(SparseMatrix b)
    {
        if (col != b.row)
        {
            Console.WriteLine("Can't multiply, Invalid dimensions");
            return null;
        }
        b = b.Transpose();
        int apos, bpos;
        SparseMatrix result = new SparseMatrix(row, b.row);
        for (apos = 0; apos < len;)
        {
            int r = data[apos][0];

            for (bpos = 0; bpos < b.len;)
            {
                int c = b.data[bpos][0];
                int tempa = apos;
                int tempb = bpos;
                int sum = 0;
                while (tempa < len && data[tempa][0] == r &&
                       tempb < b.len && b.data[tempb][0] == c)
                {
                    if (data[tempa][1] < b.data[tempb][1])
                        tempa++;
                    else if (data[tempa][1] > b.data[tempb][1])
                        tempb++;
                    else
                        sum += data[tempa++][2] *
                             b.data[tempb++][2];
                }
                if (sum != 0)
                    result.Insert(r, c, sum);

                while (bpos < b.len &&
                       b.data[bpos][0] == c)

                    bpos++;
            }
            while (apos < len && data[apos][0] == r)
                apos++;
        }
        return result;
    }
```

Метод Print розроблений чисто для зручності і візуалізації матриць, у форматі CSR.

```
public void Print()
    {
        Console.Write(String.Concat("\nDimension: ", row.ToString(), "x", col.ToString()));
        Console.Write("\nSparse Matrix: \nRow\tColumn\tValue\n");

        for (int i = 0; i < len; i++)
        {
            Console.WriteLine(string.Concat(data[i][0].ToString(), "\t ", data[i][1].ToString(), "\t ",
                                                                    data[i][2].ToString()));
        }
    }
```

Тепер наведемо приклад використання класу SparseMatrix.

SparseMatrix a = new SparseMatrix(512, 512);

SparseMatrix b = new SparseMatrix(512, 512);

a.Insert(4, 2, 10);

a.Insert(9, 5, 12);

a.Insert(98, 87, 5);

a.Insert(4, 1, 15);

a.Insert(4, 2, 12);

a.Insert(145, 445, 18);

b.Insert(1, 45, 80);

b.Insert(2, 43, 23);

b.Insert(98, 3, 92);

b.Insert(4, 67, 20);

b.Insert(4, 2, 25);

a.Multiply(b).Print();

3.2 Реалізація алгоритму на графічному процесорі

Алгоритм реалізовано на мові Java.

Спочатку іде ініціалізація змінних, які потрібні для подальшої роботи алгоритму і самі матриці, над якими виконується множення.

```
int A_NUM_ROWS = 4;
int A_num_rows = 4;
int A_num_cols = 4;
int A_nnz      = 9;
int B_num_rows = 4;
int B_num_cols = 4;
int B_nnz      = 9;
int   hA_csrOffsets[] = { 0, 3, 4, 7, 9 };
int   hA_columns[]    = { 0, 2, 3, 1, 0, 2, 3, 1, 3 };
float hA_values[]     = { 1.0f, 2.0f, 3.0f, 4.0f, 5.0f,
                          6.0f, 7.0f, 8.0f, 9.0f };
int   hB_csrOffsets[] = { 0, 2, 4, 7, 8 };
int   hB_columns[]    = { 0, 3, 1, 3, 0, 1, 2, 1 };
float hB_values[]     = { 1.0f, 2.0f, 3.0f, 4.0f, 5.0f,
                          6.0f, 7.0f, 8.0f };
int   hC_csrOffsets[] = { 0, 4, 6, 10, 12 };
int   hC_columns[]    = { 0, 1, 2, 3, 1, 3, 0, 1, 2, 3, 1, 3 };
float hC_values[]     = { 11.0f, 36.0f, 14.0f, 2.0f,  12.0f,
                          16.0f, 35.0f, 92.0f, 42.0f, 10.0f,
                                               96.0f, 32.0f };
```

Наступним етапом реалізовується ініціалізація обгорток С-подібних указників – Pointer. Дані обгортки потрібні для звернення до пам’яті графічного процесора, так як сама мова Java не надає можливості для прямої роботи з пам’яттю.[3]

```
Pointer dA_csrOffsets = new Pointer();
Pointer dA_columns = new Pointer();
Pointer dB_csrOffsets = new Pointer();
Pointer dB_columns = new Pointer();
Pointer dC_csrOffsets = new Pointer();
Pointer dC_columns = new Pointer();
Pointer dA_values = new Pointer();
Pointer dB_values = new Pointer();
Pointer dC_values = new Pointer();
```

Після створення поінтерів, необхідно розмістити наявні дані в пам’яті: зробити allocate.

```
cudaMalloc(dA_csrOffsets, (A_num_rows + 1) * Sizeof.INT);
cudaMalloc(dA_columns, A_nnz * Sizeof.INT);
cudaMalloc(dA_values,  A_nnz * Sizeof.FLOAT);
cudaMalloc(dB_csrOffsets, (B_num_rows + 1) * Sizeof.INT);
cudaMalloc(dB_columns, B_nnz * Sizeof.INT);
cudaMalloc(dB_values,  B_nnz * Sizeof.FLOAT);
cudaMalloc(dC_csrOffsets, (A_num_rows + 1) * Sizeof.INT);
```

Наступний крок – це створення матриці у форматі CSR засобами JCUD.

```
cusparseCreateCsr(matA, A_num_rows, A_num_cols, A_nnz,
                  dA_csrOffsets, dA_columns, dA_values,
                  CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I,
                  CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
cusparseCreateCsr(matB, B_num_rows, B_num_cols, B_nnz,
                  dB_csrOffsets, dB_columns, dB_values,
                  CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I,
                  CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
cusparseCreateCsr(matC, A_num_rows, B_num_cols, 0,
                  null, null, null,
                  CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I,
                  CUSPARSE_INDEX_BASE_ZERO, CUDA_R_32F);
```

Тепер відбувається виклик функції для множення матриць.

```
cusparseSpMM_bufferSize(handle, opA, opB, pAlpha, matA, dense_descr,
                        pBeta, dense_descr, computeType,
                        CUSPARSE_MM_ALG_DEFAULT, aBufferSizeSPMM);
```

Наступним кроком потрібно знищити дескриптори, які використовувалися для роботи з матрицями.

```
cusparseDestroySpMat(matA);
cusparseDestroySpMat(matB);
cusparseDestroySpMat(matC);
cusparseDestroy(handle);
```

В кінці роботи програми виконується деалокація всіх конструкцій, використаних в програмі.

```
cudaFree(dBuffer1);
cudaFree(dBuffer2);
cudaFree(dA_csrOffsets);
cudaFree(dA_columns);
cudaFree(dA_values);
cudaFree(dB_csrOffsets);
cudaFree(dB_columns);
cudaFree(dB_values);
cudaFree(dC_csrOffsets);
cudaFree(dC_columns);
cudaFree(dC_values);
```

Розділ 4. Результати експериментів

Експерименти були проведені на графічному процесорі та на центральному процесорі. Основне, на що зверталась увага – це швидкість множення матриць.

Параметри графічного процесора:

NVIDIA GeForce GTX 1650 TI

Графічний процесор – TU117

Кількість ядер CUDA – 1024

Об’єм пам’яті – 4 гігабайти

Тип пам’яті – GDDR6

Архітектура – Turing

Тактова частота – 1350 МГц

Дана таблиця показує експерименти з множення матриць. Розмірність матриць варіюється від 4 до 1024, тобто степені двійки. При матриці, розмірності 256 × 256 алгоритм на центральному процесорі не виконався. Вилетіло OutOfMemoryException

Розмір матриці	4	8	16	32	64	128	256	512	1024
Множення на GPU	1 мс	2 мс	6 мс	26 мс	89 мс	5 с	14 с	27 с	48 с
Множення на CPU	1 мс	3мс	11мс	76мс	520мс	10 с	-	-	-

З таблиці видно, що множення на графічному процесорі дійсно має переваги перед множенням на центральному процесорі.

Висновки

Було розглянуто та досліджено алгоритм множення розріджених матриць на графічному процесорі. Після цього було реалізовані множення розріджених матриць двома способами: за допомогою множення на графічному процесорі з використанням інструментів CUDA та бібліотеки JCUDA та з використанням центрального процесора за допомогою мови C# та платформи .NET.

Проведено дослідження швидкості виконання множення розріджених матриць на графічному процесорі, за допомогою JCUDA та на центральному процесорі з використанням мови C#. Крім того, було проведено дослідження залежності розміру матриці та часу виконання множення. Алгоритм множення матриць на графічному процесорі виглядав доволі непогано, так сказати – працювати можна.

Список використаної літератури

1. Документація NVIDIA CUDA. [Електронний ресурс]. – режим доступу: https://developer.nvidia.com/cuda-zone.
2. Опис проекту JCuda. [Електронний ресурс]. – режим доступу: http://jcuda.org/.
3. S. Lahabar and P. J. Narayanan, "Singular value decomposition on GPU using CUDA," 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-10, doi: 10.1109/IPDPS.2009.5161058. [Електронний ресурс]. – режим доступу: https://ieeexplore.ieee.org/document/5161058/citations?tabFilter=papers#citations.
4. Документація бібліотеки CUSPARSE. [Електронний ресурс]. – режим доступу: https://docs.nvidia.com/cuda/cusparse/index.html.
5. Carl Yang , Audin Buluc and John D. Owens, “Design Principles for Sparse Matrix Multiplication on the GPU”