

Паралельні та розподілені обчислення
ЛАБОРАТОРНА РОБОТА №8

**Тема: «Реалізація методу Якобі на базі архітектури паралельних
обчислень OpenCL».**

Виконала:
Ст. Пелещак Вероніка
ПМІ-35с

Тема: Реалізація методу Якобі на базі архітектури паралельних обчислень OpenCL.

Завдання: Реалізувати метод Якобі для розв’язування СЛАР трьома способами:

- послідовно на CPU;
- паралельно на CPU із використанням потоків (std::thread);
- на GPU із використанням OpenCL.

Порівняти час виконання для різних розмірностей матриць. Обчислити прискорення та ефективність паралельних реалізацій. Зробити висновки про доцільність використання GPU.

Теоретичні відомості:

Метод Якобі — ітераційний метод розв’язування системи рівнянь.

$$Ax = b,$$

де A — квадратна матриця коефіцієнтів, x — вектор невідомих, b — вектор вільних членів. На кожній ітерації k нове наближення обчислюється за формулою:

$$x_i^{(k+1)} = 1 / A_{ii} (b_i - \sum A_{ij} x_j^{(k)}), \text{ де } j \neq i.$$

Обчислення для кожного x_i не залежать одне від одного, тому метод зручно розпаралелювати. Щоб забезпечити збіжність, матриця A повинна бути **діагонально домінантною**, тобто:

$$|A_{ii}| > \sum |A_{ij}|, \text{ де } j \neq i.$$

Після достатньої кількості ітерацій метод сходиться до точного розв’язку.

Опис програми:

1. Бібліотеки

- `<iostream>`, `<vector>`, `<random>`, `<chrono>`, `<cmath>`, `<iomanip>`, `<thread>` — стандартні бібліотеки C++.
- `<CL/cl.h>` або `<OpenCL/opencl.h>` — для роботи з OpenCL API.
- OpenCL використовується для доступу до GPU через C API (створення контексту, черги, буферів тощо).

2. Функції програми

- **check()**
 - Службова функція для перевірки помилок OpenCL.
 - Якщо `err != CL_SUCCESS`, програма завершується з повідомленням про помилку.
- **genDiagDomA()**
 - Генерує квадратну матрицю розміру $N \times N$, яка **діагонально домінантна**.
 - Кожен елемент генерується випадково, а головна діагональ збільшується так, щоб гарантувати збіжність.
 - Повертає `std::vector<float>`.
- **genVec()**
 - Генерує вектор `b` розміру `N` з випадковими значеннями в діапазоні `[-1, 1]`.

- **residual()**

- Обчислює **нев'язку** $\|b-A\|_{\infty}$.
- Чим менше це значення, тим ближчий розв'язок до істинного.
- Використовується для перевірки точності результатів.

- **jacobiCPU()** (послідовний метод)

- Звичайна реалізація Якобі на CPU без потоків.
- На кожній ітерації для кожного i рахується сума по всіх $j \neq i$, а потім нове значення $x_new[i]$.
- Вектори x і x_new міняються місцями наприкінці кожної ітерації.
- Виконується max_iter разів.

- **jacobiCPU_parallel()** (паралельний метод)

- Та сама логіка, але обчислення для різних i діляться між кількома потоками (`std::thread`).
- Кожен потік обробляє свій діапазон індексів $[start, end)$.
- Наприкінці ітерації всі потоки синхронізуються (`join()`), і вектори міняються місцями.
- Забезпечує прискорення завдяки використанню кількох ядер процесора.

3. OpenCL ядро (KERNEL_JACOBI)

```
__kernel void jacobi_step(  
    const int N,  
    __global const float* A,  
    __global const float* b,  
    __global const float* x,  
    __global float* x_new)
```

- Це код, який виконується **на GPU**.
- Кожен потік GPU обчислює один елемент $x_new[i]$.
- `get_global_id(0)` визначає номер потоку (індекс i).
- Виконується обчислення так само, як і в CPU-варіанті:

$$x_{new}[i] = (b[i] - \sum A[i * N + j] * x[j]) / A[i * N + i], \text{ де } j \neq i.$$

- Таким чином, весь вектор x_new обчислюється одночасно багатьма потоками GPU.

- **main()**

- Створює розміри матриць: $N = 765, 1500, 2450$.
- Ініціалізує OpenCL: знаходить платформу, пристрій (GPU або CPU), створює контекст, чергу, компілює програму.
- Для кожного N :

1. Генерує матрицю A і вектор b .

2. Виконує три обчислення:
 - CPU послідовно;
 - CPU паралельно (8 потоків);
 - GPU через OpenCL.
3. Обчислює часи, прискорення та ефективність.
4. Перевіряється точність результатів.
5. Виводить усі результати у зручному форматі.

4. OpenCL API функції:

- `clGetPlatformIDs`, `clGetDeviceIDs` — пошук пристроїв.
- `clCreateContext`, `clCreateCommandQueue` — створення середовища виконання.
- `clCreateProgramWithSource`, `clBuildProgram`, `clCreateKernel` — компіляція OpenCL коду.
- `clCreateBuffer`, `clSetKernelArg`, `clEnqueueNDRangeKernel`, `clEnqueueReadBuffer` — передача даних та запуск кернелів.
- `clFinish` — синхронізація.

Результати:

```
/Users/veronikapelesak/Desktop/practice/parallel_comp/lab8/
=====

+++++++ Matrix N = 765 ++++++
CPU sequential time: 11375.7340 ms
CPU parallel time: 3471.7870 ms
OpenCL GPU time: 691.7060 ms
-----
Speedup (seq → par): 3.2766
Speedup (seq → GPU): 16.4459
Speedup (par → GPU): 5.0192
-----
Efficiency (CPU par): 0.4096
GPU Throughput: 5.9147 GFLOP/s

Residuals: seq=0.0000, par=0.0000, gpu=0.0000

+++++++ Matrix N = 1500 ++++++
CPU sequential time: 44836.4470 ms
CPU parallel time: 12854.5350 ms
OpenCL GPU time: 1214.0370 ms
-----
Speedup (seq → par): 3.4880
Speedup (seq → GPU): 36.9317
Speedup (par → GPU): 10.5883
-----
Efficiency (CPU par): 0.4360
GPU Throughput: 12.9646 GFLOP/s

Residuals: seq=0.0000, par=0.0000, gpu=0.0000

+++++++ Matrix N = 2450 ++++++
CPU sequential time: 120063.1040 ms
CPU parallel time: 35609.0210 ms
OpenCL GPU time: 2664.1040 ms
-----
Speedup (seq → par): 3.3717
Speedup (seq → GPU): 45.0670
Speedup (par → GPU): 13.3662
-----
Efficiency (CPU par): 0.4215
GPU Throughput: 15.7653 GFLOP/s

Residuals: seq=0.0000, par=0.0000, gpu=0.0000

=====

Process finished with exit code 0
```

Висновок: У лабораторній роботі було реалізовано та протестовано метод Якобі для розв'язування систем лінійних алгебраїчних рівнянь трьома способами. Порівняння показало, що:

1. GPU через OpenCL забезпечує **найвищу швидкість**.
2. Багатопотокова версія на CPU показує хороше, але обмежене прискорення через спільний кеш і синхронізацію.
3. Метод Якобі добре масштабується і легко переноситься між архітектурами.
4. OpenCL дозволяє реалізувати універсальне рішення, сумісне як із GPU, так і з CPU, що робить його ефективним вибором для паралельних чисельних обчислень.