Міністерство освіти і науки України Національний університет «Києво-Могилянська академія»
Факультет інформатики Кафедра математики

Кваліфікаційна робота
освітній ступінь – бакалавр

на тему: «**Дослідження методів паралелізації математичних обчислень для GPU у бібліотеці скінченних елементів MoFEM**»

Виконав: студент 4-го року навчання
освітньої програми «Прикладна математика»,
спеціальності 113 Прикладна математика
Шевченко Богдан Юрійович

Керівник: Бублик В.В.

канд. фіз-мат. наук, доцент

Рецензент: ______________________________

Кваліфікаційна робота захищена
з оцінкою ______________________________

Секретар ЕК______________________________
(підпис)

«__________» _________________20___р.

Київ - 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики

ЗАТВЕРДЖУЮ

Зав. кафедри математики, доцент, канд. ф.-м. наук

____________ Чорней Р.К.

«____» _______________2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту 4-го курсу, факультету інформатики
Шевченка Богдана Юрійовича

Тема: Дослідження методів паралелізації математичних обчислень для GPU у бібліотеці скінченних елементів MoFEM
Зміст кваліфікаційної роботи:

1. Анотація
2. Перелік умовних скорочень
3. Вступ
4. Розділ 1: Теоретичні основи та постановка задачі
5. Розділ 2: Розробка GPU прискорення
6. Розділ 3: Експерименти та результати
7. Висновки
8. Список використаних джерел

Дата видачі «___» _________ 2025 р.

Керівник ________________
(підпис)

Завдання отримав ___________
(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Визначення теми	01.10.2024	
2	Дослідження предметної області	01.11.2024	
3	Пошук тематичної літератури	15.11.2024	
4	Ознайомлення зі знайденою літературою	01.02.2025	
5	Написання вступної частини	15.02.2025	
6	Написання теоретичної частини	15.03.2025	
7	Реалізація програм для порівняння	01.04.2025	
8	Аналіз отриманих результатів	15.04.2025	
9	Написання висновків	01.05.2025	
10	Підготовка презентації	10.05.2025	
11	Захист роботи	06.06.2025	

Студент Шевченко Богдан Юрійович

Керівник Бублик В.В.

“____” ________________

Зміст

Вступ 6

Розділ 1. Метод скінченних елементів(МСЕ)

1.1 Основи скінченних елементів на прикладі................................ 8

1.2 Зведення сильного формулювання до слабкого.................................. 12

1.3 Перехід до системи лінійних рівнянь.. 15

1.5 Змішані скінченні елементи та функціональні простори................................ 22

1.5 Гібридизація змішаних скінченних елементів... 27

1.6 Метод доповнення Шура та солвер Крилова GMRES........................ 31

1.7 Постановка Задачі... 37

Розділ 2. Програмна архітектура

2.1 Що таке MoFEM.. 38

2.2 Існуюча архітектура MoFEM.. 39

2.3 Існуючі алгоритми MoFEM... 43

2.4 Огляд обраних бібліотек для GPGPUs 46

2.5 Розроблена гетерогенна архітектура 48

2.6 Алгоритм множення матриці на вектор 55

2.7 Алгоритм інверсії матриці внутрішніх елементів 58

Розділ 3. Експерименти та результати

3.1 Тестування на маленькому сервері для розробки 61

3.2 Тестування на потужному сервері 64

3.3 Тестування на Archer2.. 67

3.4 Вплив дослідження... 70

Висновки 71

Лістинг коду 73

Список використаних джерел 74

Вступ

Актуальність

Моделювання процесів руйнування конструкційних матеріалів, зокрема утворення тріщин у графітових блоках ядерних реакторів, є критично важливим для гарантування безпечної експлуатації атомних електростанцій. Графітові блоки відіграють роль сповільнювача нейтронів у реакторах, і через тривале радіаційне опромінення та теплові навантаження в них можуть з'являтися тріщини. Надійне прогнозування таких процесів дозволяє своєчасно виявляти потенційно небезпечні зони й ухвалювати технічно обґрунтовані рішення щодо подальшої експлуатації реакторів. Найбільш ефективним чисельним методом, який використовується для таких симуляцій, є метод скінченних елементів (МСЕ).

Однак тривимірні моделі з великою кількістю ступенів вільності потребують розв'язання великомасштабних систем лінійних алгебраїчних рівнянь. Це, у свою чергу, створює значне навантаження на обчислювальні ресурси. У зв'язку з цим важливим напрямом сучасних досліджень є паралелізація обчислень та використання графічних процесорів (GPU), які дозволяють значно пришвидшити розв'язання задач завдяки своїй архітектурі масового паралелізму.

Мета і завдання дослідження.

Метою цієї роботи є підвищення продуктивності бібліотеки методу скінченних елементів MoFEM для задач моделювання тріщин у графітових блоках реакторів шляхом розробки та впровадження GPU-орієнтованих алгоритмів лінійної алгебри, та протестувати розроблений підхід на найбільшому суперкомп'ютері Британії.

Для досягнення цієї мети було поставлено такі завдання:

1. Проаналізувати сучасні методи прискорення обчислень методу скінченних елементів.
2. Визначити основні вузькі місця продуктивності у наявній реалізації MoFEM.

3. Розробити архітектуру обміну даними між CPU та GPU для виконання обчислень у MoFEM.
4. Реалізувати ефективні алгоритми на GPU для вузьких місць продуктивності.
5. Провести експериментальні дослідження продуктивності реалізованих алгоритмів на найбільшому суперкомп'ютері Великої Британії - Archer2 та середній і маленькій серверних системах.

Об'єкт дослідження — процес розв'язання великомасштабних систем рівнянь у задачах механіки руйнування графітових блоків атомних реакторів методом скінченних елементів.

Предмет дослідження — методи прискорення обчислень лінійної алгебри для бібліотеки MoFEM із використанням графічних процесорів.

Методи дослідження.
У роботі використано аналітичні методи (аналіз літератури з обчислювальної механіки та паралельних обчислень), методи чисельного моделювання (реалізація алгоритмів у бібліотеці MoFEM), а також експериментальні обчислення на найбільшому суперкомп'ютері Великої Британії - Archer2 та двох серверних системах із 12- і 32-ядерними CPU та GPU (Nvidia GTX 1050 Ti, Nvidia H100).

Наукова новизна одержаних результатів.

Уперше у Великій Британії і втретє в Європі реалізовано розширюване GPU-прискорене розв'язання гібридизованих змішаних скінченних елементів для задач структурної механіки. Запропоновано оригінальну архітектуру для ефективної взаємодії між CPU та GPU, а також розроблено прискорені реалізації основних обчислювальних етапів: множення матриці на вектор і обчислення доповнення Шура.

Практичне значення одержаних результатів.

Розроблені рішення будуть використані для моделювання тріщин у графітових блоках атомних реакторів компанії EDF Energy, а також компанією Rolls-Royce для моделювання тріщин в деталях їх механізмів.

Апробація результатів.

Розроблені методи та програмні рішення були апробовані шляхом порівняння з CPU-версіями у MoFEM на найбільшому суперкомп’ютері Великої Британії - Archer2, сервері середньої потужності, та сервері низької потужності і продемонстрували суттєве прискорення.

Публікації.

За результатами роботи підготовлено тези доповіді, які подано на конференцію студентських наукових робіт.

Структура роботи.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, і списку використаних джерел. У першому розділі викладено математичні основи методу скінченних елементів, зокрема гібридизованих змішаних формулювань. Другий розділ присвячено аналізу алгоритмів розв’язання систем рівнянь, включаючи методи Крилова та доповнення Шура. Третій розділ містить опис реалізації GPU-прискореного солвера у бібліотеці MoFEM. У четвертому розділі наведено результати експериментів, порівняння з CPU-реалізацією та аналіз ефективності запропонованого підходу.

Розділ 1: Метод скінченних елементів (МСЕ)

Метод скінченних елементів (МСЕ) є чисельним методом для розв'язання диференціальних рівнянь, що описують фізичні явища. Розглянемо конкретний приклад, який дозволить зрозуміти суть методу та вивести сильне формулювання задачі.

1.1 Основи скінченних елементів на прикладі

Розглянемо пружний стержень довжиною L, закріплений на лівому кінці ($x = 0$) та навантажений розподіленою силою $f(x)$ вздовж своєї довжини. Додатково, на правому кінці ($x = L$) прикладена зосереджена сила F.

Нехай $u(x)$ - функція переміщення точок стержня, E - модуль Юнга матеріалу, а $A(x)$ - площа поперечного перерізу стержня, яка може змінюватися вздовж його довжини.

Фізичний аналіз задачі

Для початку розглянемо невеликий елемент стержня довжиною Δx між точками x та $x + \Delta x$. На цей елемент діють:

- Внутрішня сила $N(x)$ на лівому кінці
- Внутрішня сила $N(x + \Delta x)$ на правому кінці
- Розподілена зовнішня сила $f(x) \cdot \Delta x$

За законом рівноваги сума всіх сил повинна дорівнювати нулю:

$$N(x) - N(x + \Delta x) + f(x) \cdot \Delta x = 0$$

Переносячи другий член у праву частину та ділячи на Δx, отримуємо:

$$\frac{N(x) - N(x + \Delta x)}{-\Delta x} = f(x)$$

При переході до границі $\Delta x \to 0$, отримуємо диференціальне рівняння:

$$\frac{dN(x)}{dx} + f(x) = 0$$

За законом Гука, внутрішня сила $N(x)$ пов'язана з деформацією стержня співвідношенням:

$$N(x) = E \cdot A(x) \cdot \varepsilon(x)$$

де $\varepsilon(x) = \frac{du(x)}{dx}$ - деформація стержня.

Таким чином, внутрішня сила виражається як:

$$N(x) = E \cdot A(x) \cdot \frac{du(x)}{dx}$$

Підставляючи цей вираз у диференціальне рівняння, отримуємо:

$$\frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) + f(x) = 0$$

Граничні умови

Для повного опису задачі потрібно додати граничні умови:

$$u(0) = 0 \quad \text{(закріплений лівий кінець)}$$
$$E \cdot A(L) \cdot \frac{du(L)}{dx} = F \quad \text{(навантажений правий кінець)}$$

Сильне формулювання задачі

Отже, **сильне формулювання задачі** про деформацію одновимірного стержня має вигляд:

Знайти функцію $u(x)$, $x \in [0, L]$, таку що:

$$\frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) + f(x) = 0, \quad x \in (0, L)$$
$$u(0) = 0$$
$$E \cdot A(L) \cdot \frac{du(L)}{dx} = F$$

Це диференціальне рівняння другого порядку з граничними умовами називається сильним формулюванням, оскільки воно вимагає, щоб розв'язок $u(x)$ був двічі диференційовним на інтервалі $(0, L)$ і точно задовольняв рівняння в кожній точці цього інтервалу.

У разі постійних параметрів

Якщо модуль Юнга E та площа поперечного перерізу A постійні, то рівняння спрощується до:

$$EA\frac{d^2u(x)}{dx^2} + f(x) = 0$$

Для найпростішого випадку, коли зовнішнє навантаження відсутнє ($f(x) = 0$), отримуємо:

$$\frac{d^2u(x)}{dx^2} = 0$$

Це звичайне диференціальне рівняння, розв'язок якого має вигляд:

$$u(x) = C_1x + C_2$$

де C_1 і C_2 - константи, які визначаються з граничних умов.

Застосовуючи граничні умови:

$$u(0) = 0 \Rightarrow C_2 = 0$$
$$EA\frac{du(L)}{dx} = F \Rightarrow EA \cdot C_1 = F \Rightarrow C_1 = \frac{F}{EA}$$

Отже, розв'язок цієї найпростішої задачі має вигляд:

$$u(x) = \frac{F}{EA} \cdot x$$

Це лінійна функція, що описує видовження стержня під дією сили F на правому кінці.

Перехід до методу скінченних елементів

Для більш складних задач, де немає аналітичного розв’язку, ми використовуємо метод скінченних елементів. Ключовою ідеєю є перехід від сильного формулювання, яке вимагає, щоб розв’язок точно задовольняв диференціальне рівняння у кожній точці, до слабкого формулювання, яке вимагає виконання рівняння в інтегральному (варіаційному) сенсі.

Це дозволяє шукати наближений розв’язок у вигляді лінійної комбінації базисних функцій на дискретизованій області**. [4]**

1.2 Зведення сильного формулювання до слабкого

$$\frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) + f(x) = 0, \quad x \in (0, L) \quad (1)$$

$$u(0) = 0 \quad (2)$$

$$E \cdot A(L) \cdot \frac{du(L)}{dx} = F \quad (3)$$

Це формулювання називається "сильним", оскільки воно вимагає, щоб розв’язок $u(x)$ задовольняв диференціальне рівняння точно в кожній точці області $(0, L)$. Крім того, функція $u(x)$ повинна бути двічі неперервно диференційовною, тобто $u \in C^2(0, L)$, щоб вираз $\frac{d^2u}{dx^2}$ мав сенс.

Обмеження сильного формулювання

Сильне формулювання має кілька обмежень:

1. Воно вимагає високої гладкості розв’язку ($u \in C^2$), що не завжди можливо забезпечити для складних фізичних задач.
2. Аналітичні розв’язки можна знайти лише для простих задач з регулярною геометрією та однорідними властивостями матеріалів.
3. Складно враховувати розривні навантаження або властивості матеріалів.
4. Важко знайти розв’язки для задач із десятками змінних.

Щоб подолати ці обмеження, ми переходимо до слабкого формулювання, яке вимагає виконання диференціального рівняння не точно в кожній точці, а в інтегральному (усередненому) сенсі.

Вивід слабкого формулювання

Для переходу до слабкого формулювання ми використовуємо метод зважених нев’язок. Нехай $v(x)$ — довільна функція, яка задовольняє граничну умову $v(0) = 0$ і має достатню гладкість. Таку функцію називають пробною функцією або віртуальним переміщенням.

Помножимо вихідне диференціальне рівняння (1) на пробну функцію $v(x)$ та проінтегруємо по всій області:

$$\int_0^L \left[\frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) + f(x)\right] \cdot v(x)\, dx = 0 \qquad (4)$$

Розділимо інтеграл на дві частини:

$$\int_0^L \frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) \cdot v(x)\, dx + \int_0^L f\,(x) \cdot v(x)\, dx = 0 \qquad (5)$$

Тепер застосуємо інтегрування частинами до першого інтегралу:

$$\int_0^L \frac{d}{dx}\left(E \cdot A(x) \cdot \frac{du(x)}{dx}\right) \cdot v(x)\, dx =$$

$$E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot v(x)\Big|_0^L - \int_0^L E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot \frac{dv(x)}{dx}\, dx \qquad (6)$$

Розглянемо граничні члени. При $x = 0$ маємо $v(0) = 0$, тому перший член дорівнює:

$$E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot v(x)\Big|_0^L =$$

$$E \cdot A(L) \cdot \frac{du(L)}{dx} \cdot v(L) - E \cdot A(0) \cdot \frac{du(0)}{dx} \cdot v(0) = E \cdot A(L) \cdot \frac{du(L)}{dx} \cdot v(L) \quad (7)$$

Використовуючи граничну умову, отримуємо:

$$E \cdot A(L) \cdot \frac{du(L)}{dx} \cdot v(L) = F \cdot v(L) \qquad (8)$$

Підставляючи (6) та (8) у (5), отримуємо:

$$F \cdot v(L) - \int_0^L E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot \frac{dv(x)}{dx}\, dx + \int_0^L f\,(x) \cdot v(x)\, dx = 0 \qquad (9)$$

Переносячи члени, отримуємо слабке формулювання:

$$\int_0^L E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot \frac{dv(x)}{dx}\, dx = F \cdot v(L) + \int_0^L f\,(x) \cdot v(x)\, dx \qquad (10)$$

для всіх допустимих пробних функцій $v(x)$, таких що $v(0) = 0$.

Інтерпретація слабкого формулювання

Рівняння (10) можна інтерпретувати з точки зору віртуальної роботи:

- Ліва частина $\int_0^L E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot \frac{dv(x)}{dx}\, dx$ представляє віртуальну роботу внутрішніх сил (пружну енергію).
- Права частина $F \cdot v(L) + \int_0^L f\,(x) \cdot v(x)\, dx$ представляє віртуальну роботу зовнішніх сил.

Таким чином, слабке формулювання виражає принцип віртуальних переміщень: для рівноваги системи віртуальна робота внутрішніх сил повинна дорівнювати віртуальній роботі зовнішніх сил для всіх можливих віртуальних переміщень.

Переваги слабкого формулювання:

Слабке формулювання має кілька важливих переваг порівняно з сильним:

1. **Виконання рівняння "в середньому"**: Сильне формулювання вимагає точного виконання рівняння в кожній точці, тоді як слабке формулювання вимагає виконання рівняння лише в усередненому, інтегральному сенсі. Це також є "слабшою" вимогою.
2. **Природне включення граничних умов Неймана**: Граничні умови типу Неймана (як у нашому випадку умова $E \cdot A(L) \cdot \frac{du(L)}{dx} = F$) автоматично включаються у слабке формулювання та називаються природними граничними умовами. Граничні умови типу Діріхле (як $u(0) = 0$) накладаються на простір допустимих пробних функцій і називаються суттєвими.
3. **Просторові вимоги**: Слабке формулювання дозволяє шукати розв’язок у "більшому" функціональному просторі, який включає функції з меншою гладкістю.
4. **Симетричність**: Слабке формулювання часто приводить до симетричних білінійних форм, що спрощує числовий аналіз і розв’язання. **[5]**

1.3 Перехід до системи лінійних рівнянь

Ми отримали слабке формулювання задачі про деформацію одновимірного стержня у вигляді:

$$a(u, v) = l(v) \quad \forall v \in V$$

де

$$a(u, v) = \int_0^L E \cdot A(x) \cdot \frac{du(x)}{dx} \cdot \frac{dv(x)}{dx}\, dx$$
$$l(v) = F \cdot v(L) + \int_0^L f\,(x) \cdot v(x)\, dx$$
$$V = \{v \in H^1(0, L): v(0) = 0\}$$

Тепер ми переходимо до ключового етапу методу скінченних елементів – дискретизації задачі та її зведення до системи лінійних алгебраїчних рівнянь, яку можна розв’язати чисельно.

Дискретизація області та скінченновимірна апроксимація

Основна ідея МСЕ полягає в переході від безперервного простору функцій V до скінченновимірного підпростору $V_h \subset V$. Для цього ми розбиваємо область $[0, L]$ на n скінченних елементів:

$$0 = x_0 < x_1 < x_2 < \cdots < x_n = L$$

Кожен елемент e визначається двома сусідніми вузлами x_{e-1} та x_e, і має довжину $h_e = x_e - x_{e-1}$. У найпростішому випадку всі елементи мають однакову довжину $h = L/n$, але в загальному випадку розмір елементів може бути різним, що дозволяє створювати більш дрібну сітку в областях, де очікується складніша поведінка розв’язку.

Базисні функції та ступені свободи

Для побудови скінченновимірного підпростору V_h ми вводимо набір базисних функцій $\{\phi_1(x), \phi_2(x), \dots, \phi_n(x)\}$, які задовольняють певні властивості. У

найпростішому (лінійному) випадку кожна базисна функція $\phi_i(x)$ має такі характеристики:

1. $\phi_i(x)$ є кусково-лінійною функцією.
2. $\phi_i(x_j) = \delta_{ij} = \begin{cases} 1, & \text{якщо } i = j \\ 0, & \text{якщо } i \neq j \end{cases}$ (властивість Кронекера).
3. $\phi_i(x)$ дорівнює нулю за межами елементів, що містять вузол x_i.

Явний вигляд лінійних базисних функцій для внутрішніх вузлів ($i = 1,2,\dots,n-1$):

$$\phi_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{якщо } x \in [x_{i-1}, x_i] \\ \dfrac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{якщо } x \in [x_i, x_{i+1}] \\ 0, & \text{в іншому випадку} \end{cases}$$

Для граничного вузла $x_n = L$:

$$\phi_n(x) = \begin{cases} \dfrac{x - x_{n-1}}{x_n - x_{n-1}}, & \text{якщо } x \in [x_{n-1}, x_n] \\ 0, & \text{в іншому випадку} \end{cases}$$

Зауважимо, що базисна функція для $x_0 = 0$ не включена, оскільки ми розглядаємо простір функцій з нульовою граничною умовою $u(0) = 0$.

Тепер будь-яку функцію $u_h \in V_h$ можна представити як лінійну комбінацію базисних функцій:

$$u_h(x) = \sum_{i=1}^{n} U_i \, \phi_i(x) \qquad (11)$$

де коефіцієнти U_i називаються **ступенями свободи** і представляють значення функції u_h у вузлах: $U_i = u_h(x_i)$.

Поняття ступенів свободи є центральним у методі скінченних елементів. Це фізичні величини (переміщення, повороти, температури тощо), які визначають стан системи у вузлах дискретизованої моделі. Кількість та тип ступенів свободи

залежать від типу задачі та **порядку апроксимації(скільки елементів ми використовуємо)**.

Формування системи лінійних алгебраїчних рівнянь

Тепер ми можемо перейти від слабкого формулювання до системи лінійних рівнянь. Ми шукаємо наближений розв'язок $u_h \in V_h$, який задовольняє:

$$a(u_h, v_h) = l(v_h) \quad \forall v_h \in V_h \qquad (12)$$

Оскільки u_h має вигляд (11), а базисні функції ϕ_j утворюють базис простору V_h, достатньо перевірити виконання рівняння (12) для $v_h = \phi_j$, $j = 1,2,\dots,n$:

$$a\left(\sum_{i=1}^{n} U_i \, \phi_i, \phi_j\right) = l(\phi_j) \quad j = 1,2,\dots,n$$

Використовуючи лінійність форми $a(u, v)$ по першому аргументу, отримуємо:

$$\sum_{i=1}^{n} U_i \, a(\phi_i, \phi_j) = l(\phi_j) \quad j = 1,2,\dots,n$$

Це система n лінійних алгебраїчних рівнянь відносно n невідомих U_i. Її можна записати в матричній формі:

$$\mathbf{KU} = \mathbf{F}$$

де:

- **K** – матриця жорсткості розміром $n \times n$ з елементами $K_{ji} = a(\phi_i, \phi_j)$.
- **U** – вектор невідомих ступенів свободи розміром n: $\mathbf{U} = [U_1, U_2, \dots, U_n]^T$.
- **F** – вектор навантажень розміром n з елементами $F_j = l(\phi_j)$.

Для нашого одновимірного прикладу елементи матриці жорсткості обчислюються як:

$$K_{ji} = a\big(\phi_i, \phi_j\big) = \int_0^L E \cdot A(x) \cdot \frac{d\phi_i(x)}{dx} \cdot \frac{d\phi_j(x)}{dx} \, dx \qquad (13)$$

а елементи вектора навантажень:

$$F_j = l(\phi_j) = F \cdot \phi_j(L) + \int_0^L f\,(x) \cdot \phi_j(x)\, dx$$

Зауважимо, що $\phi_j(L) = \begin{cases} 1, & \text{якщо } j = n \\ 0, & \text{якщо } j \neq n \end{cases}$, тому $F \cdot \phi_j(L) = F$ для $j = n$ і 0 для інших j.

Обчислення елементів матриці жорсткості

Розглянемо детальніше обчислення елементів матриці жорсткості . Оскільки базисні функції ϕ_i та ϕ_j є ненульовими лише на обмеженій кількості елементів, інтеграл (13) можна розбити на суму інтегралів по окремих елементах.

Для лінійних базисних функцій перекриваються лише сусідні функції, тому ненульові елементи матриці **K** отримуємо лише для $|i - j| \leq 1$. Це призводить до трьохдіагональної структури матриці:

$$\mathbf{K} = \begin{bmatrix} K_{11} & K_{12} & 0 & 0 & \dots & 0 \\ K_{21} & K_{22} & K_{23} & 0 & \dots & 0 \\ 0 & K_{32} & K_{33} & K_{34} & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & K_{n-1,n-2} & K_{n-1,n-1} & K_{n-1,n} \\ 0 & 0 & \dots & 0 & K_{n,n-1} & K_{n,n} \end{bmatrix}$$

У випадку елементів однакової довжини $h = L/n$ з постійними властивостями матеріалу (E та A є константами), елементи матриці мають вигляд:

$$K_{ii} = \frac{EA}{h} \cdot 2 \quad \text{для } i = 1,2, \dots, n-1$$

$$K_{nn} = \frac{EA}{h}$$

$$K_{i,i+1} = K_{i+1,i} = -\frac{EA}{h} \quad \text{для } i = 1,2, \dots, n-1$$

Це пов’язано з тим, що похідні лінійних базисних функцій є кусково-постійними:

$$\frac{d\phi_i(x)}{dx} = \begin{cases} \frac{1}{x_i - x_{i-1}} = \frac{1}{h}, & \text{якщо } x \in [x_{i-1}, x_i] \\ \frac{-1}{x_{i+1} - x_i} = \frac{-1}{h}, & \text{якщо } x \in [x_i, x_{i+1}] \\ 0, & \text{в іншому випадку} \end{cases}$$

Модульна структура обчислень: локальні та глобальні матриці

На практиці обчислення глобальної матриці жорсткості $\mathbf{K}$ та вектора навантажень $\mathbf{F}$ виконується за модульним принципом. Спочатку розглядаються окремі скінченні елементи, для яких формуються локальні матриці жорсткості та вектори навантажень, а потім ці локальні компоненти збираються (складаються) у глобальну систему.

Розглянемо елемент e з вузлами i та $i+1$. Локальна матриця жорсткості $\mathbf{K}^{(e)}$ для цього елемента має розмір 2×2 і визначається як:

$$K_{11}^{(e)} = \int_{x_i}^{x_{i+1}} E\,A \left(\frac{d\phi_i^{(e)}}{dx}\right)^2 dx = \frac{EA}{h_e}$$

$$K_{22}^{(e)} = \int_{x_i}^{x_{i+1}} E\,A \left(\frac{d\phi_{i+1}^{(e)}}{dx}\right)^2 dx = \frac{EA}{h_e}$$

$$K_{12}^{(e)} = K_{21}^{(e)} = \int_{x_i}^{x_{i+1}} E\,A \frac{d\phi_i^{(e)}}{dx} \frac{d\phi_{i+1}^{(e)}}{dx} dx = -\frac{EA}{h_e}$$

де $\phi_i^{(e)}$ та $\phi_{i+1}^{(e)}$ є локальними базисними функціями для елемента e, а $h_e = x_{i+1} - x_i$ – довжина елемента.

Таким чином, локальна матриця жорсткості для елемента e має вигляд:

$$\mathbf{K}^{(e)} = \frac{EA}{h_e}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Локальний вектор навантажень $\mathbf{F}^{(e)}$ також має розмір 2 і визначається як:

$$F_1^{(e)} = \int_{x_i}^{x_{i+1}} f(x)\phi_i^{(e)}(x)dx$$
$$F_2^{(e)} = \int_{x_i}^{x_{i+1}} f(x)\phi_{i+1}^{(e)}(x)dx$$

У випадку постійного навантаження $f(x) = f_0$ на елементі, отримуємо:

$$\mathbf{F}^{(e)} = \frac{f_0 h_e}{2}\begin{bmatrix}1\\1\end{bmatrix}$$

Для врахування зосередженої сили F на правому кінці стержня, ми додаємо цю силу до локального вектора навантажень останнього елемента (для вузла n).

Процес «збирання» глобальної матриці жорсткості та вектора навантажень полягає в додаванні відповідних елементів локальних матриць та векторів до глобальних. Для цього використовується матриця інцидентності, яка показує відповідність між локальними та глобальними номерами ступенів свободи.

Структура матриці жорсткості та її властивості

Матриця жорсткості **K**, отримана в методі скінченних елементів, має низку важливих властивостей:

1. **Симетричність**: $K_{ij} = K_{ji}$ для всіх i, j. Це випливає з симетричності білінійної форми $a(u, v) = a(v, u)$.
2. **Додатна визначеність** (для більшості фізичних задач): $\mathbf{x}^T\mathbf{K}\mathbf{x} > 0$ для всіх ненульових векторів **x**. Це пов'язано з фізичною інтерпретацією виразу $a(u, u)$ як пружної енергії, яка є додатною для будь-яких ненульових переміщень.
3. **Розрідженість**: Більшість елементів матриці дорівнюють нулю. Ненульові елементи відповідають взаємодії між вузлами, з'єднаними спільними елементами.
4. **Блочна структура** (для векторних задач): Якщо кожен вузол має кілька ступенів свободи (наприклад, компоненти переміщення по різних осях), матриця має блочну структуру.

Розв’язання системи лінійних рівнянь

Після формування глобальної системи лінійних рівнянь $\mathbf{KU} = \mathbf{F}$, її необхідно розв’язати для знаходження вектора невідомих ступенів свободи $\mathbf{U}$.

Для невеликих систем (до кількох тисяч невідомих) можна використовувати прямі методи, такі як метод Гаусса або LU-розкладання. Однак для великих систем, які виникають у реальних інженерних задачах (мільйони і більше невідомих), ефективнішими є ітераційні методи, такі як метод спряжених градієнтів чи **метод узагальнених мінімальних нев’язок (GMRES)**.

Важливою перевагою системи рівнянь, яка виникає в методі скінченних елементів, є її розрідженість. Як ми бачили, для лінійних елементів матриця жорсткості є трьохдіагональною, тобто більшість її елементів дорівнюють нулю. Ця властивість зберігається і для елементів вищого порядку та для багатовимірних задач, хоча структура розрідженості стає складнішою. *Насправді, це залежить від функціонального простору, який ми розглядаємо і описано в наступному підрозді.*

Для зберігання розріджених матриць використовуються спеціальні формати, такі як CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) або COO (Coordinate). **[8][5]**

1.4 Змішані скінченні елементи та функціональні простори

До цього ми розглянули класичний підхід до методу скінченних елементів, де шукаємо наближений розв'язок в одному функціональному просторі – просторі Соболєва $H^1(\Omega)$. Цей підхід добре працює для багатьох задач, але для деяких класів проблем виникають труднощі, пов'язані з точністю апроксимації окремих фізичних величин або з чисельними нестабільностями. Змішані скінченні елементи дозволяють подолати ці обмеження, вводячи декілька невідомих функцій, які належать до різних функціональних просторів і мають різні фізичні інтерпретації.

Мотивація для змішаних формулювань

Розглянемо задачу теорії пружності в двовимірному випадку. У класичному формулюванні переміщень ми шукаємо вектор переміщень $\mathbf{u} = [u_1, u_2]^T$, який належить до простору $[\mathcal{H}^1(\Omega)]^2$, де Ω – область, яку займає тіло. Напруження обчислюються як похідні від переміщень:

$$\boldsymbol{\sigma} = \mathbf{C}: \boldsymbol{\varepsilon}(\mathbf{u}) = \mathbf{C}: \nabla^s \mathbf{u}$$

де $\nabla^s \mathbf{u} = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)$ – симетричний градієнт (тензор деформацій), а $\mathbf{C}$ – тензор пружності четвертого рангу.

Проблема полягає в тому, що напруження, обчислені таким чином, мають на один порядок нижчу точність порівняно з переміщеннями. Це пов'язано з тим, що для обчислення напружень потрібно диференціювати переміщення, що знижує порядок апроксимації. Якщо переміщення апроксимуються поліномами степеня p, то напруження апроксимуються поліномами степеня $p - 1$.

Функціональні простори Соболєва та їх фізична інтерпретація

Для розуміння змішаних скінченних елементів потрібно спочатку розглянути різні функціональні простори, в яких можуть бути визначені наші невідомі функції. Кожен з цих просторів має свою математичну структуру та фізичну інтерпретацію.

Простір $L^2(\Omega)$

Простір $L^2(\Omega)$ складається з функцій u, квадрат яких є інтегровним на області Ω:

$$L^2(\Omega) = \left\{ u: \Omega \to \mathbb{R} \mid \| u \|^2_{L^2(\Omega)} = \int_\Omega u^2 \, d\Omega < \infty \right\}$$

Норма в цьому просторі визначається як:

$$\| u \|_{L^2(\Omega)} = \left(\int_\Omega u^2 \, d\Omega \right)^{1/2}$$

з відповідним скалярним добутком:

$$(u, v)_{L^2(\Omega)} = \int_\Omega u \, v \, d\Omega$$

Важливо зауважити, що функції з $L^2(\Omega)$ не обов'язково є неперервними – вони можуть мати розриви, але їх квадрат повинен бути інтегровним.

Простір $H^1(\Omega)$ (простір Соболєва першого порядку)

Простір $H^1(\Omega)$ є підпростором $L^2(\Omega)$ і складається з функцій, які разом зі своїми слабкими похідними першого порядку належать до $L^2(\Omega)$:

$$H^1(\Omega) = \left\{ u \in L^2(\Omega) \mid \frac{\partial u}{\partial x_i} \in L^2(\Omega),\ i = 1,2, \dots, d \right\}$$

де d – розмірність простору (1, 2 або 3).

Норма в просторі $H^1(\Omega)$ визначається як:

$$\| u \|^2_{H^1(\Omega)} = \| u \|^2_{L^2(\Omega)} + \| \nabla u \|^2_{L^2(\Omega)} = \int_\Omega (u^2 + |\nabla u|^2) \, d\Omega$$

Скалярний добуток у $H^1(\Omega)$:

$$(u, v)_{H^1(\Omega)} = (u, v)_{L^2(\Omega)} + (\nabla u, \nabla v)_{L^2(\Omega)} = \int_\Omega (uv + \nabla u \cdot \nabla v) \, d\Omega$$

Важливою властивістю простору $H^1(\Omega)$ є можливість визначення слідів функцій на границі області. Для функцій з $H^1(\Omega)$ можна коректно визначити граничні умови Діріхле.

Простір $H(\mathrm{div}, \Omega)$

Простір $H(\mathrm{div}, \Omega)$ складається з векторних функцій $\mathbf{v} = [v_1, v_2, \dots, v_d]^T$, які разом зі своєю дивергенцією належать до $L^2(\Omega)$:

$$H(\mathrm{div}, \Omega) = \{ \mathbf{v} \in [L^2(\Omega)]^d \mid \nabla \cdot \mathbf{v} \in L^2(\Omega) \}$$

де $\nabla \cdot \mathbf{v} = \sum_{i=1}^{d} \frac{\partial v_i}{\partial x_i}$ – дивергенція векторного поля.

Норма в цьому просторі:

$$\| \mathbf{v} \|_{H(\mathrm{div},\Omega)}^2 = \| \mathbf{v} \|_{[L^2(\Omega)]^d}^2 + \| \nabla \cdot \mathbf{v} \|_{L^2(\Omega)}^2 = \int_\Omega (|\mathbf{v}|^2 + (\nabla \cdot \mathbf{v})^2)\, d\Omega$$

Скалярний добуток:

$$(\mathbf{u}, \mathbf{v})_{H(\mathrm{div},\Omega)} = (\mathbf{u}, \mathbf{v})_{[L^2(\Omega)]^d} + (\nabla \cdot \mathbf{u}, \nabla \cdot \mathbf{v})_{L^2(\Omega)}$$

Важливою властивістю простору $H(\mathrm{div}, \Omega)$ є можливість визначення нормальної компоненти векторного поля на границі області. Це дозволяє коректно формулювати граничні умови Неймана для потокових змінних.

Простір $\boldsymbol{H}(\mathbf{curl}, \boldsymbol{\Omega})$

Простір $H(\mathrm{curl}, \Omega)$ складається з векторних функцій, які разом зі своїм ротором (curl) належать до $L^2(\Omega)$:

$$H(\mathrm{curl}, \Omega) = \{\mathbf{v} \in [L^2(\Omega)]^d \mid \nabla \times \mathbf{v} \in [L^2(\Omega)]^{d'}\}$$

де $d' = 1$ для двовимірних задач та $d' = 3$ для тривимірних задач.

У двовимірному випадку ротор скалярний:

$$\nabla \times \mathbf{v} = \frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}$$

У тривимірному випадку ротор векторний:

$$\nabla \times \mathbf{v} = \begin{bmatrix} \frac{\partial v_3}{\partial x_2} - \frac{\partial v_2}{\partial x_3} \\ \frac{\partial v_1}{\partial x_3} - \frac{\partial v_3}{\partial x_1} \\ \frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2} \end{bmatrix}$$

Норма в просторі $H(\mathrm{curl}, \Omega)$:

$$\| \mathbf{v} \|_{H(\mathrm{curl},\Omega)}^2 = \| \mathbf{v} \|_{[L^2(\Omega)]^d}^2 + \| \nabla \times \mathbf{v} \|_{[L^2(\Omega)]^{d'}}^2$$

Важливою властивістю простору $H(\mathrm{curl}, \Omega)$ є можливість визначення тангенціальної компоненти векторного поля на границі області.

Взаємозв'язок між функціональними просторами

Різні функціональні простори пов'язані між собою через диференціальні оператори, що утворює так званий комплекс де Рама:

$$\mathbb{R} \xrightarrow{\mathrm{grad}} H^1(\Omega) \xrightarrow{\nabla} H(\mathrm{curl}, \Omega) \xrightarrow{\nabla\times} H(\mathrm{div}, \Omega) \xrightarrow{\nabla\cdot} L^2(\Omega) \to 0$$

Цей комплекс має властивість точності (exactness): образ кожного оператора є ядром наступного. Наприклад: - $\nabla \times (\nabla u) = 0$ для будь-якої скалярної функції $u \in H^1(\Omega)$ - $\nabla \cdot (\nabla \times \mathbf{v}) = 0$ для будь-якого векторного поля $\mathbf{v} \in H(\mathrm{curl}, \Omega)$

Ця властивість є фундаментальною для побудови стабільних змішаних скінченно-елементних просторів.

Загальна формулювання змішаного методу скінченних елементів

Розглянемо загальну постановку змішаної задачі. Нехай ми маємо дві невідомі функції u та σ, які належать до різних функціональних просторів U та Σ відповідно. Змішане варіаційне формулювання має вигляд:

Знайти $(u, \sigma) \in U \times \Sigma$ такі, що:

$$a(\sigma, \tau) + b(\tau, u) = \langle f_1, \tau \rangle \quad \forall \tau \in \Sigma$$
$$b(\sigma, v) - c(u, v) = \langle f_2, v \rangle \quad \forall v \in U$$

де: - $a(\cdot,\cdot)$ – білінійна форма на $\Sigma \times \Sigma$ (зазвичай пов'язана з конститутивними співвідношеннями) - $b(\cdot,\cdot)$ – білінійна форма на $\Sigma \times U$ (пов'язана з кінематичними співвідношеннями) - $c(\cdot,\cdot)$ – білінійна форма на $U \times U$ (зазвичай пов'язана з додатковими рівняннями або стабілізацією) - $\langle f_1, \cdot \rangle$ та $\langle f_2, \cdot \rangle$ – лінійні функціонали, що представляють зовнішні навантаження

У матричному записі змішана система має вигляд:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & -\mathbf{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{\sigma} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}$$

де матриці $\mathbf{A}$, $\mathbf{B}$ та $\mathbf{C}$ відповідають дискретизованим білінійним формам $a(\cdot,\cdot)$, $b(\cdot,\cdot)$ та $c(\cdot,\cdot)$.

Такі системи називаються седловими (saddle-point) системами через специфічну структуру матриці, яка не є додатно визначеною. Розв'язання таких систем потребує спеціальних чисельних методів.

LBB умова

Для існування та єдиності розв'язку змішаної задачі, а також для стабільності дискретизованої задачі, необхідно виконання inf-sup condition (інф-суп), також відомої як умова Ладиженської-Бабушки-Брецці (LBB).

Умова інф-суп для неперервної задачі формулюється як:

$$\inf_{0 \neq v \in U} \sup_{0 \neq \tau \in \Sigma} \frac{b(\tau, v)}{\| \tau \|_\Sigma \| v \|_U} \geq \beta > 0$$

де β – додатна константа, незалежна від параметрів дискретизації.

Порушення умови інф-суп призводить до нестабільності чисельного розв'язку, яка може проявлятися у вигляді: - Осциляцій розв'язку (checker-board modes) - Втрати збіжності при зменшенні h - Погіршення обумовленості системи рівнянь

Стабілізація змішаних методів

Коли пара скінченно-елементних просторів не задовольняє умову інф-суп, виникає необхідність у стабілізації.

Least-Squares стабілізація

Цей підхід полягає в додаванні до варіаційного формулювання стабілізуючих, які мінімізують нев’язку в сенсі найменших квадратів:

$$a_h(\sigma_h, \tau_h) + b_h(\tau_h, u_h) + S_h(\sigma_h, \tau_h) = \langle f_1, \tau_h \rangle$$
$$b_h(\sigma_h, v_h) - c_h(u_h, v_h) + T_h(\sigma_h, v_h) = \langle f_2, v_h \rangle$$

де S_h та T_h – стабілізуючі білінійні форми вигляду:

$$S_h(\sigma_h, \tau_h) = \sum_{T \in \mathcal{T}_h} \alpha_T \int_T (\mathcal{L}_1 \sigma_h + \mathcal{B}_1 u_h - f_1) \cdot (\mathcal{L}_1 \tau_h)\, dx$$
$$T_h(\sigma_h, v_h) = \sum_{T \in \mathcal{T}_h} \alpha_T \int_T (\mathcal{L}_2 \sigma_h + \mathcal{B}_2 u_h - f_2) \cdot (\mathcal{B}_2 v_h)\, dx$$

де $\mathcal{L}_1, \mathcal{L}_2, \mathcal{B}_1, \mathcal{B}_2$ – диференціальні оператори з вихідної системи рівнянь, $\alpha_T > 0$ – параметри стабілізації, які залежать від локального розміру елемента.

Обчислювальні аспекти змішаних елементів

З обчислювальної точки зору змішані скінченні елементи хоч і мають вищу точність та адаптивність, але мають помітно вищі вимоги по обчисленнях і пам'яті у порівнянні з використанням суцільного функціонального простру.

Структура матриць у змішаних методах

Уваги заслуговує структура матриць, що виникають при дискретизації змішаних задач. Типова система має седлову структуру:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{B}_1^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{B}_2^T \\ \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{g} \end{bmatrix}$$

де блоки $\mathbf{A}_{ij}$ відповідають взаємодії між змінними одного типу, блоки $\mathbf{B}_i$ – зв’язку між різними типами змінних, а $\mathbf{C}$ – додатковим рівнянням (стабілізації або фізичним обмеженням). **[3] [15]**

1.5 Гібридизація змішаних скінченних елементів

Гібридизація є потужним методом зменшення обчислювальної складності змішаних скінченних-елементів. Основна ідея полягає у введенні додаткових невідомих на границях елементів та статичній конденсації (виключенні) внутрішньоелементних ступенів свободи, що призводить до значного зменшення розмірності глобальної системи лінійних рівнянь.

Математичні основи гібридизації

Розглянемо стандартне змішане варіаційне формулювання: знайти $(u_h, \boldsymbol{\sigma}_h) \in U_h \times \Sigma_h$ такі, що

$$a(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) + b(\boldsymbol{\tau}_h, u_h) = l_1(\boldsymbol{\tau}_h) \quad \forall \boldsymbol{\tau}_h \in \Sigma_h$$
$$b(\boldsymbol{\sigma}_h, v_h) + c(u_h, v_h) = l_2(v_h) \quad \forall v_h \in U_h$$

У гібридизованому формулюванні ми замінюємо глобальні простори U_h та Σ_h на простори функцій, які можуть бути розривними на межелементних границях. Неперервність розв'язку забезпечується через введення множників Лагранжа λ_h, визначених на скелеті сітки $\mathcal{E}_h$ (множині всіх граней елементів).

Нехай $\mathcal{T}_h$ – розбиття області Ω на скінченні елементи, а $\mathcal{E}_h$ – множина всіх граней (ребер у 2D, граней у 3D) елементів. Для кожного елемента $T \in \mathcal{T}_h$ визначимо локальні простори:

$$U_h(T) = \{v \in L^2(T): v|_T \in P_k(T)\}$$
$$\Sigma_h(T) = \{\boldsymbol{\tau} \in [L^2(T)]^d: \boldsymbol{\tau}|_T \in RT_k(T)\}$$

де $P_k(T)$ та $RT_k(T)$ – локальні поліноміальні простори на елементі T.

Простір множників Лагранжа визначається на скелеті сітки:

$$\Lambda_h(\mathcal{E}_h) = \{\mu \in L^2(\mathcal{E}_h): \mu|_e \in P_k(e) \text{ для всіх } e \in \mathcal{E}_h\}$$

Гібридизоване варіаційне формулювання

Гібридизоване варіаційне формулювання полягає у знаходженні $(u_h, \boldsymbol{\sigma}_h, \lambda_h)$ із простору $\prod_{T\in\mathcal{T}_h} U_h\,(T) \times \prod_{T\in\mathcal{T}_h} \Sigma_h\,(T) \times \Lambda_h(\mathcal{E}_h)$ таких, що для всіх елементів $T \in \mathcal{T}_h$:

$$a_T(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) + b_T(\boldsymbol{\tau}_h, u_h) = l_{1,T}(\boldsymbol{\tau}_h) + \langle \lambda_h, \boldsymbol{\tau}_h \cdot \mathbf{n}_T \rangle_{\partial T} \quad \forall \boldsymbol{\tau}_h \in \Sigma_h(T) \qquad (18)$$
$$b_T(\boldsymbol{\sigma}_h, v_h) + c_T(u_h, v_h) = l_{2,T}(v_h) \quad \forall v_h \in U_h(T) \qquad (19)$$
$$\langle \boldsymbol{\sigma}_h \cdot \mathbf{n}_T, \mu_h \rangle_{\partial T} = 0 \quad \forall \mu_h \in \Lambda_h(\partial T) \qquad (20)$$

де ∂T позначає границю елемента T, $\mathbf{n}_T$ – зовнішню нормаль до ∂T, а $\langle \cdot,\cdot \rangle_{\partial T}$ – інтеграл по границі елемента. Локальні білінійні форми a_T, b_T, c_T та лінійні функціонали $l_{1,T}$, $l_{2,T}$ є обмеженнями відповідних глобальних форм на елемент T.

Третє рівняння(20) забезпечує неперервність нормальної компоненти потоку $\boldsymbol{\sigma}_h$ через межелементні границі, що є дискретним аналогом умови неперервності розв'язку.

Статична конденсація та локальні солвери

Ключовою перевагою гібридизованого формулювання є можливість розв'язання перших двох рівнянь(18-19) незалежно на кожному елементі для фіксованих значень λ_h на границі. Це дозволяє виразити локальні невідомі $(u_h|_T, \boldsymbol{\sigma}_h|_T)$ як лінійні функції граничних невідомих $\lambda_h|_{\partial T}$.

Для кожного елемента T можна записати локальну систему рівнянь у матричному вигляді:

$$\begin{bmatrix} \mathbf{A}_T & \mathbf{B}_T^T \\ \mathbf{B}_T & \mathbf{C}_T \end{bmatrix} \begin{bmatrix} \boldsymbol{\sigma}_T \\ \mathbf{u}_T \end{bmatrix} = \begin{bmatrix} \mathbf{f}_{1,T} + \mathbf{D}_T \boldsymbol{\lambda}_{\partial T} \\ \mathbf{f}_{2,T} \end{bmatrix}$$

де $\boldsymbol{\sigma}_T$ та $\mathbf{u}_T$ – вектори локальних ступенів свободи на елементі T, $\boldsymbol{\lambda}_{\partial T}$ – вектор граничних невідомих на ∂T, а $\mathbf{D}_T$ – матриця, що описує вплив граничних невідомих на локальну систему.

Припускаючи, що локальна матриця

$$\mathbf{K}_T = \begin{bmatrix} \mathbf{A}_T & \mathbf{B}_T^T \\ \mathbf{B}_T & \mathbf{C}_T \end{bmatrix}$$

є оберненою, можемо виразити локальні невідомі:

$$\begin{bmatrix} \boldsymbol{\sigma}_T \\ \mathbf{u}_T \end{bmatrix} = \mathbf{K}_T^{-1} \begin{bmatrix} \mathbf{f}_{1,T} + \mathbf{D}_T \boldsymbol{\lambda}_{\partial T} \\ \mathbf{f}_{2,T} \end{bmatrix} = \mathbf{K}_T^{-1} \begin{bmatrix} \mathbf{f}_{1,T} \\ \mathbf{f}_{2,T} \end{bmatrix} + \mathbf{K}_T^{-1} \begin{bmatrix} \mathbf{D}_T \\ \mathbf{0} \end{bmatrix} \boldsymbol{\lambda}_{\partial T}$$

Позначимо

$$\begin{bmatrix} \boldsymbol{\sigma}_T^{(0)} \\ \mathbf{u}_T^{(0)} \end{bmatrix} = \mathbf{K}_T^{-1} \begin{bmatrix} \mathbf{f}_{1,T} \\ \mathbf{f}_{2,T} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{R}_T \\ \mathbf{S}_T \end{bmatrix} = \mathbf{K}_T^{-1} \begin{bmatrix} \mathbf{D}_T \\ \mathbf{0} \end{bmatrix}$$

Тоді локальні невідомі виражаються як:

$$\boldsymbol{\sigma}_T = \boldsymbol{\sigma}_T^{(0)} + \mathbf{R}_T \boldsymbol{\lambda}_{\partial T} \quad (21)$$

$$\mathbf{u}_T = \mathbf{u}_T^{(0)} + \mathbf{S}_T \boldsymbol{\lambda}_{\partial T} \quad (22)$$

Формування глобальної системи для граничних невідомих

Підставляючи ці вирази у третє рівняння(20), отримуємо систему рівнянь відносно граничних невідомих λ_h. Для кожної грані $e \in \mathcal{E}_h$ це рівняння має вигляд:

$$\sum_{T:e\in\partial T} \langle (\boldsymbol{\sigma}_T^{(0)} + \mathbf{R}_T \boldsymbol{\lambda}_{\partial T}) \cdot \mathbf{n}_T, \mu_h \rangle_e = 0 \quad \forall \mu_h \in \Lambda_h(e)$$

де сума береться по всіх елементах T, що мають грань e на своїй границі.

Це призводить до глобальної системи лінійних рівнянь відносно граничних невідомих:

$$\mathbf{M}\boldsymbol{\lambda} = \mathbf{g}$$

де $\mathbf{M}$ – глобальна матриця граничних взаємодій, $\boldsymbol{\lambda}$ – глобальний вектор граничних невідомих, а $\mathbf{g}$ – глобальний вектор правої частини.

Матриця $\mathbf{M}$ формується шляхом збирання локальних внесків від кожного елемента:

$$\mathbf{M} = \sum_{T\in\mathcal{T}_h} \mathbf{A}_T^{bc}\, \mathbf{R}_T$$

де $\mathbf{A}_T^{bc}$ – матриця, що пов'язує локальні граничні ступені свободи елемента T з глобальними граничними ступенями свободи.

Алгоритм розв'язання гібридизованої системи

Повний алгоритм розв'язання гібридизованої змішаної задачі складається з наступних етапів:

Етап 1: Попередні обчислення (локальні солвери)

Для кожного елемента $T \in \mathcal{T}_h$:

Формується локальна матриця $\mathbf{K}_T$ та вектори правої частини

Обчислюється LU-розкладання (або інша факторизація) матриці $\mathbf{K}_T$

Обчислюються локальні розв'язки $(\boldsymbol{\sigma}_T^{(0)}, \mathbf{u}_T^{(0)})$ для однорідних граничних умов

Обчислюються матриці впливу граничних умов $(\mathbf{R}_T, \mathbf{S}_T)$

Етап 2: Формування глобальної системи

Збираються локальні внески у глобальну матрицю $\mathbf{M}$ та вектор $\mathbf{g}$

Застосовуються граничні умови Діріхле до системи $\mathbf{M}\boldsymbol{\lambda} = \mathbf{g}$

Етап 3: Розв'язання глобальної системи

Розв'язується система $\mathbf{M}\boldsymbol{\lambda} = \mathbf{g}$ для знаходження граничних невідомих $\boldsymbol{\lambda}$

Етап 4: Відновлення локальних розв'язків

Для кожного елемента $T \in \mathcal{T}_h$:

Використовуючи формули (21)-(22), обчислюються локальні невідомі $(\boldsymbol{\sigma}_T, \mathbf{u}_T)$**[2][6][7]**

Гібридизація надає кілька суттєвих обчислювальних переваг:

Зменшення розмірності глобальної системи

Як уже згадувалося, розмірність системи значно менша за розмірність початкової змішаної системи. Це призводить до: - Зменшення витрат пам'яті для зберігання глобальної матриці - Прискорення ітераційних солверів через меншу кількість невідомих - Можливості використання прямих методів для середньорозмірних задач

Природний паралелізм

Локальні розв'язки на етапах 1 та 4 алгоритму можуть бути обчислені незалежно для кожного елемента, що забезпечує ідеальний паралелізм. Це особливо важливо для GPU-обчислень, де можна ефективно розподілити обчислення між тисячами потоків.

Локальність пам'яті

Обчислення на кожному елементі використовують лише локальні дані, що забезпечує ефективне використання кешу та зменшує потреби в міжпроцесорному обміні даними в розподілених обчисленнях.

Можливість адаптивного згущення

Гібридизація спрощує реалізацію алгоритмів адаптивного згущення сітки, оскільки зміни в локальній структурі елементів впливають лише на локальні матриці та не вимагають повної перебудови глобальної системи.

Ефективне множення матриці на вектор

Формування матриці $\mathbf{M}$ та правої частини $\mathbf{g}$ включає багато операцій множення невеликих густих матриць на вектори, які можуть бути ефективно виконані на GPU з використанням оптимізованих BLAS-операцій.

Структурована розрідженість

Розрідженість матриці $\mathbf{M}$ має більш регулярну структуру порівняно з початковою змішаною системою, що полегшує ефективну реалізацію операцій розрідженої лінійної алгебри на GPU.

Застосування в бібліотеці MoFEM

Бібліотека MoFEM (Mesh-Oriented Finite Element Method), розробка якої розглядається у даній роботі активно використовує гібридизовані змішані скінченні елементи.

1.6 Метод доповнення Шура та солвер Крилова GMRES для розв'язання гібридизованих систем

У попередньому підрозділі ми показали, як гібридизація змішаних скінченних елементів призводить до системи лінійних рівнянь відносно граничних невідомих у вигляді $\mathbf{M}\boldsymbol{\lambda} = \mathbf{g}$, де матриця $\mathbf{M}$ є доповненням Шура початкової змішаної системи. У цьому підрозділі детально розглянемо математичні та обчислювальні аспекти методу доповнення Шура та ефективні алгоритми для розв'язання отриманих систем рівнянь, зокрема метод узагальнених мінімальних нев'язок (GMRES).

Математичні основи доповнення Шура

Доповнення Шура є фундаментальною концепцією лінійної алгебри, яка дозволяє ефективно розв'язувати блочні системи лінійних рівнянь через редукцію розмірності. Розглянемо загальну блочну систему:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}$$

де $\mathbf{A} \in \mathbb{R}^{n_1 \times n_1}$, $\mathbf{B} \in \mathbb{R}^{n_1 \times n_2}$, $\mathbf{C} \in \mathbb{R}^{n_2 \times n_1}$, $\mathbf{D} \in \mathbb{R}^{n_2 \times n_2}$ – блочні матриці, а $\mathbf{x}_1 \in \mathbb{R}^{n_1}$, $\mathbf{x}_2 \in \mathbb{R}^{n_2}$, $\mathbf{f}_1 \in \mathbb{R}^{n_1}$, $\mathbf{f}_2 \in \mathbb{R}^{n_2}$ – відповідні векторні блоки.

Припустимо, що матриця $\mathbf{A}$ є неособливою (має обернену). Тоді з першого рівняння системи можемо виразити $\mathbf{x}_1$:

$$\mathbf{A}\mathbf{x}_1 + \mathbf{B}\mathbf{x}_2 = \mathbf{f}_1 \Rightarrow \mathbf{x}_1 = \mathbf{A}^{-1}(\mathbf{f}_1 - \mathbf{B}\mathbf{x}_2)$$

Підставляючи цей вираз у друге рівняння системи:

$$\mathbf{C}\mathbf{A}^{-1}(\mathbf{f}_1 - \mathbf{B}\mathbf{x}_2) + \mathbf{D}\mathbf{x}_2 = \mathbf{f}_2$$

Розкриваючи дужки та перегруповуючи терми:

$$\mathbf{C}\mathbf{A}^{-1}\mathbf{f}_1 - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\mathbf{x}_2 + \mathbf{D}\mathbf{x}_2 = \mathbf{f}_2$$

$$(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})\mathbf{x}_2 = \mathbf{f}_2 - \mathbf{C}\mathbf{A}^{-1}\mathbf{f}_1$$

Матриця $\mathbf{S} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ називається доповненням Шура матриці $\mathbf{D}$ відносно блоку $\mathbf{A}$. Права частина редукованої системи має вигляд $\tilde{\mathbf{f}}_2 = \mathbf{f}_2 - \mathbf{C}\mathbf{A}^{-1}\mathbf{f}_1$.

Таким чином, розв'язання початкової системи зводиться до послідовного виконання наступних кроків:

Крок 1: Розв'язання системи доповнення Шура

$$\mathbf{S}\mathbf{x}_2 = \tilde{\mathbf{f}}_2$$

Крок 2: Відновлення першого блоку невідомих

$$\mathbf{x}_1 = \mathbf{A}^{-1}(\mathbf{f}_1 - \mathbf{B}\mathbf{x}_2)$$

Властивості доповнення Шура

Доповнення Шура має кілька важливих математичних властивостей, які роблять його привабливим для чисельних обчислень:

Збереження невиродженості

Якщо початкова блочна матриця у є невиродженою і блок $\mathbf{A}$ також є невиродженим, то доповнення Шура $\mathbf{S}$ також є невиродженим. Це гарантує існування та єдиність розв'язку системи.

Математично це можна довести через детермінант блочної матриці:

$$\det \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \det(\mathbf{A}) \cdot \det(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}) = \det(\mathbf{A}) \cdot \det(\mathbf{S})$$

Якщо детермінант блочної матриці ненульовий і $\det(\mathbf{A}) \neq 0$, то $\det(\mathbf{S}) \neq 0$.

Збереження симетрії

Якщо початкова блочна матриця симетрична і має структуру:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{D} \end{bmatrix}$$

то доповнення Шура також є симетричним:

$$\mathbf{S} = \mathbf{D} - \mathbf{B}\mathbf{A}^{-1}\mathbf{B}^T$$

Це легко перевірити:

$$\mathbf{S}^T = (\mathbf{D} - \mathbf{B}\mathbf{A}^{-1}\mathbf{B}^T)^T = \mathbf{D}^T - (\mathbf{B}^T)^T(\mathbf{A}^{-1})^T\mathbf{B}^T = \mathbf{D} - \mathbf{B}\mathbf{A}^{-1}\mathbf{B}^T = \mathbf{S}$$

бо $\mathbf{D}^T = \mathbf{D}$ (симетричність $\mathbf{D}$) та $(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1} = \mathbf{A}^{-1}$ (симетричність $\mathbf{A}$).

Збереження додатної визначеності

Це є найважливішою властивістю для ефективності чисельних методів. Якщо початкова блочна матриця додатно визначена і має симетричну структуру, то доповнення Шура також є додатно визначеним.

Доведення базується на тому, що для будь-якого ненульового вектора $\mathbf{y} \in \mathbb{R}^{n_2}$ можна знайти вектор $\mathbf{z} \in \mathbb{R}^{n_1}$ такий, що $\mathbf{z} = -\mathbf{A}^{-1}\mathbf{B}^T\mathbf{y}$. Тоді:

$$\mathbf{y}^T\mathbf{S}\mathbf{y} = \mathbf{y}^T(\mathbf{D} - \mathbf{B}\mathbf{A}^{-1}\mathbf{B}^T)\mathbf{y} = \begin{bmatrix} \mathbf{z} \\ \mathbf{y} \end{bmatrix}^T \begin{bmatrix} \mathbf{A} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \mathbf{y} \end{bmatrix} > 0$$

оскільки блочна матриця додатно визначена.

Спектральні властивості

Власні значення доповнення Шура тісно пов’язані з власними значеннями початкової блочної матриці. Якщо $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_{n_1+n_2}$ – власні значення блочної матриці, а $\mu_1 \leq \mu_2 \leq \cdots \leq \mu_{n_2}$ – власні значення доповнення Шура, то виконуються нерівності:

$$\lambda_{n_1+1} \leq \mu_1 \leq \mu_2 \leq \cdots \leq \mu_{n_2} \leq \lambda_{n_1+n_2}$$

Це означає, що власні значення доповнення Шура "вкладені" у спектр початкової матриці, що має важливі наслідки для збіжності ітераційних методів.

Обчислювальні аспекти доповнення Шура

Хоча математично доповнення Шура визначається через явне обертання $\mathbf{A}^{-1}$, на практиці безпосереднє обчислення $\mathbf{S} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ рідко є оптимальним підходом з кількох причин:

Обчислювальна складність

Пряме обчислення оберненої матриці $\mathbf{A}^{-1}$ має обчислювальну складність $O(n_1^3)$, що може бути неприйнятно високим для великих n_1. Крім того, якщо матриця $\mathbf{A}$ розріджена, то $\mathbf{A}^{-1}$ зазвичай є густою, що призводить до значних витрат пам’яті.

Числова стабільність

Обчислення оберненої матриці є числово нестабільною операцією, особливо для погано обумовлених матриць. Помилки округлення можуть значно вплинути на точність результату.

Неявне обчислення доповнення Шура

На практиці доповнення Шура зазвичай не обчислюється явно. Замість цього використовуються ітераційні методи, які потребують лише операцій матрично-векторного множення з доповненням Шура. Для вектора $\mathbf{y}$ добуток $\mathbf{S}\mathbf{y}$

обчислюється через послідовність операцій:

$$\mathbf{w}_1 = \mathbf{B}^T\mathbf{y}$$
$$\mathbf{A}\mathbf{w}_2 = \mathbf{w}_1$$
$$\mathbf{w}_3 = \mathbf{C}\mathbf{w}_2$$
$$\mathbf{S}\mathbf{y} = \mathbf{D}\mathbf{y} - \mathbf{w}_3$$

Найбільш витратною операцією у цій послідовності є розв'язання системи $\mathbf{A}\mathbf{w}_2 = \mathbf{w}_1$. Для ефективності цей розв'язок виконується за допомогою попередньо обчисленої факторизації матриці $\mathbf{A}$ (наприклад, LU- або Choleski-розкладання). **[9][10]**

Метод Крилова

Для розв'язання системи лінійних рівнянь $\mathbf{S}\boldsymbol{\lambda} = \tilde{\mathbf{g}}$ з доповненням Шура ефективними є ітераційні методи, що базуються на підпросторах Крилова. Ці методи особливо ефективні для великих розріджених систем, коли пряме обертання матриці є неможливим через обчислювальні обмеження.

Визначення підпростору Крилова

Для матриці $\mathbf{S} \in \mathbb{R}^{n\times n}$ та початкового вектора $\mathbf{v}_0 \in \mathbb{R}^n$ підпростір Крилова порядку m визначається як:

$$\mathcal{K}_m(\mathbf{S}, \mathbf{v}_0) = \text{span}\{\mathbf{v}_0, \mathbf{S}\mathbf{v}_0, \mathbf{S}^2\mathbf{v}_0, \dots, \mathbf{S}^{m-1}\mathbf{v}_0\}$$

Цей підпростір містить всі лінійні комбінації векторів $\mathbf{v}_0, \mathbf{S}\mathbf{v}_0, \mathbf{S}^2\mathbf{v}_0, \dots, \mathbf{S}^{m-1}\mathbf{v}_0$.

Основна ідея методів підпросторів Крилова полягає у пошуку наближеного розв'язку $\boldsymbol{\lambda}_m$ системи $\mathbf{S}\boldsymbol{\lambda} = \tilde{\mathbf{g}}$ у підпросторі Крилова:

$$\boldsymbol{\lambda}_m \in \boldsymbol{\lambda}_0 + \mathcal{K}_m(\mathbf{S}, \mathbf{r}_0)$$

де $\boldsymbol{\lambda}_0$ – початкове наближення, а $\mathbf{r}_0 = \tilde{\mathbf{g}} - \mathbf{S}\boldsymbol{\lambda}_0$ – початкова нев'язка.

GMRES (Generalized Minimal Residual method) є одним із найбільш надійних та широко використовуваних методів підпросторів Крилова для розв'язання несиметричних систем лінійних рівнянь. Метод був запропонований Y. Saad та M. Schultz у 1986 році і базується на мінімізації евклідової норми нев'язки в підпросторі Крилова.

Математичні основи GMRES

Дамо точну математичну формулювання методу GMRES. Для системи $\mathbf{S}\boldsymbol{\lambda} = \tilde{\mathbf{g}}$ з початковим наближенням $\boldsymbol{\lambda}_0$ та початковою нев'язкою $\mathbf{r}_0 = \tilde{\mathbf{g}} - \mathbf{S}\boldsymbol{\lambda}_0$, метод GMRES на кроці m знаходить наближення $\boldsymbol{\lambda}_m$, що мінімізує норму нев'язки:

$$\boldsymbol{\lambda}_m = \boldsymbol{\lambda}_0 + \arg\min_{\mathbf{y}\in\mathcal{K}_m(\mathbf{S},\mathbf{r}_0)} \| \tilde{\mathbf{g}} - \mathbf{S}(\boldsymbol{\lambda}_0 + \mathbf{y}) \|_2 = \boldsymbol{\lambda}_0 + \arg\min_{\mathbf{y}\in\mathcal{K}_m(\mathbf{S},\mathbf{r}_0)} \| \mathbf{r}_0 - \mathbf{S}\mathbf{y} \|_2$$

Це означає, що серед усіх можливих наближень у афінному підпросторі $\boldsymbol{\lambda}_0 + \mathcal{K}_m(\mathbf{S}, \mathbf{r}_0)$, GMRES вибирає те, яке дає найменшу норму нев'язки.

Процес Арнольді та ортогональна основа

Для ефективної реалізації GMRES використовується процес Арнольді для побудови ортонормованої основи підпростору Крилова. Процес Арнольді генерує послідовність ортонормованих векторів $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m$ таких, що:

$$\text{span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\} = \mathcal{K}_k(\mathbf{S}, \mathbf{r}_0) \quad \text{для } k = 1,2, \dots, m$$

Алгоритм процесу Арнольді:

$\mathbf{v}_1 = \mathbf{r}_0/\| \mathbf{r}_0 \|_2$ $\mathbf{w} = \mathbf{S}\mathbf{v}_j$ $h_{i,j} = \mathbf{v}_i^T\mathbf{w}$ $\mathbf{w} = \mathbf{w} - h_{i,j}\mathbf{v}_i$ $h_{j+1,j} = \| \mathbf{w} \|_2$ **break** $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$

Процес Арнольді дає важливе співвідношення:

$$\mathbf{S}\mathbf{V}_m = \mathbf{V}_{m+1}\mathbf{H}_m$$

де $\mathbf{V}_m = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m]$ – матриця ортонормованих векторів, а $\mathbf{H}_m$ – $(m+1) \times m$ верхня матриця Хессенберга:

$$\mathbf{H}_m = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,m} \\ h_{2,1} & h_{2,2} & \cdots & h_{2,m} \\ 0 & h_{3,2} & \cdots & h_{3,m} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & h_{m,m-1} & h_{m,m} \\ 0 & \cdots & 0 & h_{m+1,m} \end{bmatrix}$$

Мінімізація нев'язки

Використовуючи ортонормовану основу $\mathbf{V}_m$, будь-який вектор з підпростору Крилова може бути записаний як $\mathbf{V}_m\mathbf{z}$ для деякого $\mathbf{z} \in \mathbb{R}^m$. Тоді задача мінімізації перетворюється на:

$$\min_{\mathbf{z}\in\mathbb{R}^m} \| \mathbf{r}_0 - \mathbf{S}\mathbf{V}_m\mathbf{z} \|_2$$

Використовуючи співвідношення Арнольді та той факт, що $\mathbf{r}_0 = \| \mathbf{r}_0 \|_2 \mathbf{v}_1 = \| \mathbf{r}_0 \|_2 \mathbf{V}_{m+1}\mathbf{e}_1$ (де $\mathbf{e}_1$ – перший стандартний базисний вектор), отримуємо:

$$\| \mathbf{r}_0 - \mathbf{S}\mathbf{V}_m\mathbf{z} \|_2 = \| \mathbf{V}_{m+1}(\| \mathbf{r}_0 \|_2 \mathbf{e}_1 - \mathbf{H}_m\mathbf{z}) \|_2 = \|\| \mathbf{r}_0 \|_2 \mathbf{e}_1 - \mathbf{H}_m\mathbf{z} \|_2$$

де останній крок використовує ортонормованість матриці $\mathbf{V}_{m+1}$.

Таким чином, задача мінімізації зводиться до розв'язання задачі найменших квадратів:

$$\min_{\mathbf{z}\in\mathbb{R}^m} \|\| \mathbf{r}_0 \|_2 \mathbf{e}_1 - \mathbf{H}_m\mathbf{z} \|_2$$

Ця задача має значно меншу розмірність ($m \ll n$) і може бути ефективно розв'язана за допомогою QR-розкладання матриці $\mathbf{H}_m$.

Алгоритм GMRES

Повний алгоритм GMRES можна сформулювати наступним чином:

1. Вхід: S, $\tilde{\mathrm{g}}$, λ_0, максимальна розмірність підпростору m_{max}, допуск ϵ $\mathrm{r}_0 = \tilde{\mathrm{g}} - \mathrm{S}\lambda_0$ $\beta = \| \mathrm{r}_0 \|_2$ λ_0 $\mathrm{v}_1 = \mathrm{r}_0/\beta$

2.Виконати один крок процесу Арнольді для отримання v_{j+1} та $h_{i,j}$, $i = 1, \dots, j+1$

3.Розв'язати задачу найменших квадратів: $\min_{\mathrm{z}} \| \beta\mathrm{e}_1 - \mathrm{H}_j\mathrm{z} \|_2$

4.Обчислити нев'язку: $\rho_j = \| \beta\mathrm{e}_1 - \mathrm{H}_j\mathrm{z}_j \|_2$ $\lambda_j = \lambda_0 + \mathrm{V}_j\mathrm{z}_j$ λ_j

5.Якщо збіжності не досягнуто за m_{max} ітерацій, перезапустити або збільшити m_{max}

Чому GMRES працює: теоретичне обґрунтування

Фундаментальна причина ефективності GMRES базується на важливій теоремі про мінімальний многочлен матриці.

Твердження (мінімальний многочлен). Будь-яка матриця **S** задовольняє певний мінімальний многочлен:

$$q(t) = t^m + \alpha_{m-1}t^{m-1} + \cdots + \alpha_1 t + \alpha_0$$

де $m \leq n$, такий що

$$q(\mathbf{S}) = \mathbf{S}^m + \alpha_{m-1}\mathbf{S}^{m-1} + \cdots + \alpha_1\mathbf{S} + \alpha_0\mathbf{I} = \mathbf{0}$$

Якщо $\mathbf{S}$ є невиродженою (оберненою), то $\alpha_0 \neq 0$, і ми можемо записати:

$$\mathbf{S}^{-1} = -\frac{1}{\alpha_0}(\mathbf{S}^{m-1} + \alpha_{m-1}\mathbf{S}^{m-2} + \cdots + \alpha_1\mathbf{I})$$

Тепер, застосовуючи $\mathbf{S}^{-1}$ до вектора $\tilde{\mathbf{g}}$ у нашій системі $\mathbf{S}\boldsymbol{\lambda} = \tilde{\mathbf{g}}$, отримуємо:

$$\boldsymbol{\lambda}^\star = \mathbf{S}^{-1}\tilde{\mathbf{g}} \in \mathcal{K}_m(\mathbf{S}, \tilde{\mathbf{g}})$$

Ключовий висновок: Точний розв'язок $\boldsymbol{\lambda}^\star$ завжди належить підпростору Крилова розмірності m, де m – степінь мінімального многочлена матриці $\mathbf{S}$.

Оскільки GMRES знаходить оптимальний розв'язок у підпросторі Крилова в сенсі мінімізації нев'язки, це означає:

Теоретична гарантія збіжності: GMRES теоретично знайде точний розв'язок за не більше ніж m ітерацій.

Оптимальність: На кожній ітерації GMRES дає найкращий можливий розв'язок серед усіх методів, що працюють у тому ж підпросторі Крилова.

Цей результат пояснює, чому GMRES є таким надійним методом: він має теоретичну гарантію скінченної збіжності, а на практиці часто збігається значно швидше за теоретичну границю.

Оцінки збіжності GMRES

Існує кілька теоретичних оцінок збіжності GMRES. Найбільш загальна оцінка пов'язана з полінімами мінімального відхилення:

$$\frac{\| \mathbf{r}_m \|_2}{\| \mathbf{r}_0 \|_2} \leq \min_{p \in \mathcal{P}_m, p(0)=1} \max_{\lambda \in \sigma(\mathbf{S})} |p(\lambda)|$$

де $\mathcal{P}_m$ – множина поліномів степеня не більше m, а $\sigma(\mathbf{S})$ – спектр матриці $\mathbf{S}$.

Ця оцінка показує, що швидкість збіжності залежить від того, наскільки добре можна апроксимувати функцію $f(\lambda) = 1$ поліномом степеня m на спектрі матриці, з умовою, що поліном дорівнює 1 в точці $\lambda = 0$.

Обчислювальні аспекти GMRES для доповнення Шура

Найважливішою операцією в GMRES є множення матриці доповнення Шура на вектор. Як було показано раніше, для вектора $\mathbf{y}$ добуток $\mathbf{S}\mathbf{y}$ обчислюється через послідовність:

$$\mathbf{w}_1 = \mathbf{B}^T\mathbf{y}$$

$$\mathbf{A}\mathbf{w}_2 = \mathbf{w}_1$$

$$\mathbf{w}_3 = \mathbf{B}\mathbf{w}_2$$

$$\mathbf{S}\mathbf{y} = -\mathbf{w}_3$$

Наступний крок потребує розв'язання системи з блочно-діагональною матрицею $\mathbf{A}$, що зводиться до паралельного розв'язання локальних систем на кожному елементі.

Критерії зупинки

Для систем з доповненням Шура важливо правильно вибрати критерій зупинки. Стандартний критерій:

$$\frac{\| \mathbf{r}_j \|_2}{\| \mathbf{r}_0 \|_2} < \epsilon$$

може бути надміру консервативним, оскільки нев'язка вимірюється в термінах граничних змінних, а похибка в первинних змінних (переміщеннях, напруженнях) може бути значно більшою.

Чому GMRES ефективний саме для доповнення Шура

Існує кілька специфічних причин, чому GMRES особливо добре підходить для розв'язання систем з доповненням Шура в контексті гібридизованих змішаних скінченних елементів:

Спектральні властивості доповнення Шура

Як було показано раніше, власні значення доповнення Шура "вкладені" у спектр початкової блочної матриці. Для задач, що виникають з дискретизації еліптичних рівнянь (таких як задачі механіки деформівного твердого тіла), це означає, що доповнення Шура часто має добре кластеризований спектр, що забезпечує швидку збіжність GMRES.

Ефект регуляризації

Процес формування доповнення Шура діє як природний регуляризатор, зменшуючи числову нестабільність початкової змішаної системи. Це особливо важливо для седлових систем, які можуть мати погано обумовлену структуру.

Зменшення розмірності

Менший розмір системи доповнення Шура дозволяє використовувати більші підпростори Крилова в GMRES без критичних витрат пам'яті. Це збільшує ймовірність швидкої збіжності.**[17]**

1.7 Постановка Задачі

Перехід від одновимірних до дво- та тривимірних задач суттєво збільшує кількість ступенів свободи та складність системи рівнянь. Наприклад, у тривимірній задачі теорії пружності кожен вузол має три ступені свободи (компоненти переміщення по трьох осях), а кількість вузлів зростає пропорційно кубу характерного розміру дискретизації.

Для ілюстрації: якщо одновимірна модель має n елементів, то відповідна тривимірна модель із такою ж роздільною здатністю матиме порядку n^3 елементів. Якщо $n = 100$ (що є досить скромним розміром), то тривимірна модель матиме мільйон елементів і порядку трьох мільйонів ступенів свободи.

Використання змішаних скінченних елементів робить ситуацію ще гіршою.

Таке різке зростання розмірності задачі призводить до необхідності використання високопродуктивних обчислень, включаючи:

- Паралельні алгоритми для розподілу обчислень між багатьма процесорами (CPU).
- Графічні процесори (GPU) для прискорення обчислювально інтенсивних операцій.
- Спеціалізовані бібліотеки лінійної алгебри, оптимізовані для розріджених матриць.
- Методи декомпозиції області для розподіленої пам’яті та кластерних обчислень.

У тривимірних задачах з мільйонами ступенів свободи операції лінійної алгебри стають основним вузьким місцем обчислень.

Ця робота присвячена розробці масштабованого GPU-пришвидшеного

1. множення матриці жорсткості на вектор для GMRES

2. обчислення оберненої від розрідженої матриці внутрішніх елементі

Розділ 2 – Програмна архітектура

2.1 Що таке MoFEM

MoFEM (Mesh Oriented Finite Element Method) є сучасною бібліотекою скінченних елементів, яка кардинально відрізняється від традиційних елемент-центричних підходів. На відміну від класичних реалізацій методу скінченних елементів, де тип елемента визначає апроксимаційний простір та базис, MoFEM використовує інноваційну архітектуру, що дозволяє ефективно вирішувати складні багатодоменні, багатомасштабні та багатофізичні задачі. **[11]**

Фундаментальні принципи архітектури

Основна ідея MoFEM полягає у відокремленні структур даних для апроксимації польових змінних від конкретного типу скінченного елемента. Скінченний елемент у MoFEM конструюється за допомогою набору нижчерозмірних сутностей (entities), на яких визначаються апроксимаційні поля. Це дозволяє довільно змішувати різні апроксимаційні простори (H^1, $H(\text{curl})$, $H(\text{div})$, L^2) у межах одного скінченого елемента.

Апроксимаційний простір визначає суміжність степенів свободи на сутностях, при цьому кількість степенів свободи на сутності не залежить від апроксимаційного базису (принаймні для поліноміальних базисів). Базис на сутності є слідом базису на елементі, і навпаки – базис на сутності поширюється на елемент.

Екосистема програмного забезпечення

MoFEM інтегрується з провідними науковими обчислювальними інструментами, формуючи потужну екосистему.

Структура скінченного елемента

Скінченний елемент у MoFEM складається з підсутностей: вузлів, ребер, граней та об’ємів. Елемент реалізується шляхом обчислення базисних функцій, індексів та даних на кожній сутності в домені елемента.

2.2 Існуюча архітектура MoFEM

Після розгляду загальних принципів і концепцій, на яких базується MoFEM, перейдемо до детального аналізу архітектури.

Компоненти екосистеми MoFEM

MoFEM базується на трьох фундаментальних компонентах, кожен з яких відповідає за певний аспект обчислювального процесу:

1. **MoAB (Mesh-Oriented datABase)** — забезпечує управління топологією, геометрією та сіткою. MoAB відповідає за ефективне зберігання інформації про сітку, забезпечуючи оптимальне викори- стання пам'яті. Він надає інтерфейси для створення, модифікації та запиту сіткових даних, підтри- муючи різноманітні типи елементів і топологій.
2. **PETSc (Portable, Extensible Toolkit for Scientific Computation)** — забезпечує функціональ- ність для роботи з розрідженою алгеброю, розв'язання лінійних і нелінійних систем рівнянь.
3. **MoFEM Core** — ядро бібліотеки, що реалізує метод скінченних елементів з інноваційним підхо- дом до організації обчислень. Воно забезпечує абстракції полів, скінченних елементів та операторів даних, що дозволяють гнучко визначати та комбінувати різні апроксимаційні простори.

Ці компоненти тісно інтегровані, утворюючи єдину екосистему. MoFEM використовує PETSc через інтерфейс DMMOFEM, який розширює функціональність PETSc для потреб методу скінченних елементів. Аналогічно, MoAB адаптований для внутрішнього зберігання даних у MoFEM. Така архітектура забезпечує баланс між швидкодією, ефективністю використання пам'яті та гнучкістю розробки.

Внутрішня структура полів і даних

Центральним елементом архітектури MoFEM є організація полів і пов'язаних з ними даних. Система використовує складну ієрархію структур даних для

ефективного зберігання та маніпулювання інформацією про поля, ступені свободи (DOFs) та їх значення.

Основною структурою для зберігання інформації про поля є класи NumeredDofEntity та DofEntity, які інкапсулюють дані про ступені свободи. Ці класи використовують багатоіндексні контейнери (multi-index containers) з бібліотеки Boost для ефективного доступу до даних за різними критеріями.

Структура полів організована наступним чином:

- **Ступені свободи (DOFs)** асоціюються з полями та сутностями (вершини, ребра, грані, об'єми).
- **Поля** мають тип апроксимаційного простору (H1, H-curl, H-div, L2) і базисних функцій.
- **Сутності** мають топологічний тип, локальні та глобальні індекси.

Для ефективного доступу до даних використовуються різні види індексування:

- Індексування за унікальним ідентифікатором (UId), що поєднує інформацію про поле та сутність.
- Індексування за типом сутності та локальним номером.
- Індексування за глобальним номером ступеня свободи. [11]

Структура BlockStructure для матричних обчислень

Одним з найважливіших компонентів архітектури з точки зору обчислювальної ефективності є структура **BlockStructure**, яка відповідає за організацію матричних обчислень.

BlockStructure успадковується від **DiagBlockIndex** і містить інформацію про блочну структуру матриці жорсткості. Вона забезпечує ефективне зберігання, доступ та маніпулювання блоками матриці, що є критичним для методу доповнення Шура та інших блочних алгоритмів.

Основні компоненти BlockStructure включають:

- **blockIndex** — багатоіндексний контейнер, що зберігає інформацію про блоки матриці: їхні розміри, положення, зв’язок з сутностями.
- **dataBlocksPtr** — вказівник на вектор даних, що містить елементи матриці в щільному форматі.
- **ghostX** і **ghostY** — вектори PETSc для обміну даними між процесами в паралельних обчисленнях.
- **preconditionerBlocksPtr** — вказівник на дані передобумовлювача, що використовується для при- скорення збіжності ітераційних методів.

Ця структура є ключовою для ефективної реалізації методу доповнення Шура, який використову- ється для розв’язання систем з блочною структурою, особливо в задачах зі змішаними формулюваннями.

Виклики та обмеження існуючої архітектури

Незважаючи на гнучкість та ефективність існуючої архітектури MoFEM, вона має ряд обмежень, які стають особливо помітними при роботі з великими задачами на сучасних обчислювальних системах:

1. **Обмежена підтримка гетерогенних обчислень** — існуюча архітектура оптимізована для CPU-обчислень і не використовує потенціал сучасних GPU, що обмежує продуктивність на гетерогенних системах.
2. **Послідовна природа критичних алгоритмів** — такі операції, як інверсія блоків матриці в методі доповнення Шура, реалізовані послідовно, що обмежує масштабованість для великих задач.

Передумови для розробки гетерогенної архітектури

Існуюча архітектура MoFEM, незважаючи на обмеження щодо GPU-обчислень, створює міцну основу для розробки гетерогенної архітектури. Кілька аспектів існуючої архітектури особливо сприяють такому розширенню:

1. **Чітка декомпозиція обчислень** — обчислення в MoFEM структуровані через оператори даних, що діють на сутностях, що спрощує ідентифікацію паралельних задач для GPU.
2. **Модульна архітектура** — MoFEM інтегрує зовнішні бібліотеки через чітко визначені інтерфейси, що дозволяє відносно легко додавати підтримку GPU-бібліотек.

2.3 Існуючі алгоритми MoFEM

Алгоритм збірки матриці жорсткості

Збірка матриці жорсткості в MoFEM відбувається на рівні локальних елементів через оператори даних UserDataOperator. Процес включає наступні етапи:

Code 1 Збірка матриці жорсткості в MoFEM

```
1: for all скінченні елементи do
2:     for all сутності елемента do
3:         for all оператори даних на сутності do
4:             Обчислити локальну матрицю елемента
5:         end for
6:     end for
7:     Отримати індекси рядків і стовпців для елементів матриці
8:     Знайти відповідний блок у структурі даних
9:     Виділити під нього пам'ять
10:    if розміри локальної матриці відповідають розмірам блоку then
11:        Використати оптимізовані BLAS-операції (cblas_daxpy, cblas_dcopy)
12:    else
13:        Виконати поелементну збірку
14:    end if
15:    Внести відповідні зміни метаданих блоку
16: end for
```

Цей алгоритм використовує оптимізовані BLAS-операції для ефективного заповнення блоків матриці, коли це можливо.

Алгоритм множення матриці на вектор

Операція множення матриці на вектор є ключовою для ітераційних солверів і має бути максимально ефективною. У MoFEM ця операція реалізована через функцію mult_schur_block_shell, яка спеціально оптимізована для блочних матриць:

Code 2 Множення блочної матриці на вектор

```
1: Копіювати вхідний вектор у вектор-примару для паралельних обчислень
2: Ініціалізувати вектор результату
3: for кожен блок матриці do
4:   if блок відповідає діапазону then
5:       Отримати вказівники на дані блоку та відповідні частини векторів
6:       if розмір блоку > порогового значення then
7:           Використати cblas_dgemv для множення блоку на частину вектора
8:       else
9:           Виконати безпосереднє поелементне множення
10:      end if
11:    end if
12: end for
13: Об'єднати результати з різних процесів
14: Вставити або додати результат до вихідного вектора
```

Цей алгоритм використовує два підходи залежно від розміру блоку: для великих блоків застосовується оптимізована функція cblas_dgemv, а для малих — безпосереднє множення елементів. Такий адаптивний підхід дозволяє ефективно використовувати кеш процесора та мінімізувати накладні витрати.

Алгоритм обчислення доповнення Шура

Метод доповнення Шура є ключовим для розв'язання систем з блочною структурою. У MoFEM він реалізований через клас OpSchurAssembleEndImpl. Алгоритм включає наступні етапи:

Code 3 Обчислення доповнення Шура

```
1:  Створити та заповнити блочну матрицю A00
2:  Обробити граничні умови (занулення рядків/стовпців)
4:  Перетворити блок A00 в густу матрицю
4:  Інвертувати блок A00 за допомогою LAPACK
5:  for кожен блок A10 do
6:      for кожен блок A01 do
7:          Отримати відповідну підматрицю A^-1_00
8:          Обчислити T = A10 · A^-1_00 за допомогою cblas_dgemm
9:          Обчислити R = T · A01 за допомогою cblas_dgemm
10:         Відняти R від відповідного блоку A11
11:         Зібрати результат у глобальну матрицю S
12:     end for
13: end for
```

Цей алгоритм використовує оптимізовані BLAS і LAPACK функції для матричних операцій, що забезпечує високу продуктивність на CPU. Однак, для великих систем інверсія блоку A_{00} є дуже дорогою.

Організація паралельних обчислень

MoFEM підтримує паралельні обчислення, що дозволяє ефективно використовувати обчислювальні кластери та багатоядерні системи. Паралелізм реалізується на кількох рівнях:

1. **Розподіл сітки** — сітка розподіляється між процесами з використанням методів декомпозиції домену, що реалізовані в MoAB. Кожен процес відповідає за певну частину сітки.
2. **Паралельна збірка** — збірка локальних матриць елементів відбувається паралельно на кожному процесі, після чого результати об’єднуються в глобальну матрицю.
3. **Обмін даними через примарні (ghost) сутності** — для забезпечення коректності обчислень на межах підобластей використовуються примарні сутності, які дублюють дані з сусідніх процесівю **[16]**

2.4 Огляд обраних бібліотек для GPGPUs

Для розробки гетерогенної архітектури MoFEM, що ефективно використовує GPU для прискорення математичних обчислень, було обрано три ключові бібліотеки: **MAGMA** для операцій лінійної алгебри (BLAS і LAPACK), **SYCL** для реалізації складних користувацьких обчислювальних ядер, та **OpenMP 5.0 GPU offload** для простих паралельних обчислень.

MAGMA

MAGMA (Matrix Algebra for GPU and Multicore Architectures) — це високопродуктивна бібліотека лінійної алгебри, розроблена в Університеті Теннессі, що забезпечує реалізацію операцій BLAS (Basic Linear Algebra Subprograms) та LAPACK (Linear Algebra Package) для гетерогенних обчислювальних систем, зокрема систем з GPU-прискорювачами. MAGMA є однією з найбільш оптимізованих бібліотек для операцій лінійної алгебри на GPU, що робить її ідеальним вибором для прискорення обчислювально інтенсивних частин методу скінченних елементів. **[7]**

SYCL

SYCL (вимовляється "sickle") — це відкритий стандарт для програмування гетерогенних систем, що базується на C++ і дозволяє писати код, який може виконуватися на різних прискорювачах (GPU, FPGA, DSP тощо) без зміни самого коду. SYCL був розроблений групою Khronos Group і значно розвинутий компанією Codeplay Software з Единбурга, яка створила одну з перших реалізацій цього стандарту — ComputeCpp. Один і той же код C++ використовується як для хост-системи, так і для прискорювачів. SYCL автоматично будує графи залежностей між операціями, що дозволяє ефективно планувати виконання на гетерогенних пристроях**.[14]**

OpenMP 5.0 GPU Offload

OpenMP (Open Multi-Processing) — це API для паралельного програмування на мовах C, C++ та Fortran, що базується на директивах компілятора. Починаючи з

версії 4.0, OpenMP додав підтримку прискорювачів, а версія 5.0 значно розширила можливості для програмування GPU, додавши більш гнучкі та потужні директиви для вивантаження обчислень на прискорювачі. **[13]**

Обґрунтування вибору комбінації бібліотек

Вибір комбінації MAGMA, SYCL та OpenMP 5.0 GPU Offload для паралелізації математичних обчислень у MoFEM базується на їхніх взаємодоповнюючих сильних сторонах та специфічних вимогах різних частин коду:

- MAGMA для BLAS і LAPACK операцій — висока оптимізація для лінійної алгебри робить MAGMA ідеальною для прискорення найбільш обчислювально інтенсивних операцій у методі скінченних елементів, таких як множення матриць, факторизації, та розв'язання систем лінійних рівнянь.
- SYCL для складних користувацьких ядер — потужна абстракція, сучасний C++ та переносимість між різними GPU архітектурами роблять SYCL ідеальним для розробки складних користувацьких обчислювальних ядер, специфічних для методу скінченних елементів.
- OpenMP 5.0 для простих ядер — простота використання та мінімальні зміни коду роблять OpenMP оптимальним для швидкого переносу простих обчислювальних циклів на GPU, особливо для нечасто виконуваних або менш критичних за продуктивністю частин коду.

Всі три технології добре інтегруються між собою. Наприклад, MAGMA може використовуватися для операцій високого рівня, тоді як SYCL і OpenMP можуть застосовуватися для власних спеціалізованих ядер, що взаємодіють з MAGMA. Також усі три технології підтримують основні платформи GPU (NVIDIA, AMD, Intel), що забезпечує переносимість коду між різними архітектурами прискорювачів.

2.5 Розроблена гетерогенна архітектура

На основі аналізу існуючої архітектури MoFEM та вибраних бібліотек для GPGPU обчислень, було розроблено гетерогенну архітектуру, що дозволяє ефективно використовувати як CPU, так і GPU для виконання обчислювально інтенсивних операцій методу скінченних елементів. Основна увага при розробці архітектури приділялася оптимізації операцій методу доповнення Шура та множення матриці на вектор, які є критичними для продуктивності всього процесу розв'язання.

Загальна структура гетерогенної архітектури

Розроблена гетерогенна архітектура базується на розширенні існуючої структури даних BlockStructure та інтеграції для ефективної передачі даних на GPU. Загальний підхід полягає у розділенні обчислень між CPU та GPU так, щоб максимально використати переваги обох обчислювальних пристроїв:

- **CPU** відповідає за керування обчислювальним процесом, підготовку даних, керування пам'яттю та синхронізацію між різними етапами обчислень.
- **GPU** виконує обчислювально інтенсивні операції, такі як множення матриць, операції з великими блоками даних та паралельні обчислення.

Архітектурні особливості GPU та необхідність черг

GPU архітектура базується на концепції **потокових мультипроцесорів** (Streaming Multiprocessors, SM), кожен з яких керує групами потоків, організованих у **варпи** (warps по 32 потоки у NVIDIA або wavefronts по 64 потоки у AMD). Ця організація створює унікальні вимоги до керування обчислювальними завданнями:

Черги як механізм керування ресурсами GPU:

- **Обмежена кількість одночасних контекстів**: GPU може одночасно підтримувати лише обмежену кількість активних обчислювальних контекстів.

- **Планування виконання**: GPU драйвер використовує черги для планування виконання kernels, керування залежностями між операціями та оптимізації використання обчислювальних ресурсів
- **Керування пам'яттю**: Кожна черга має свій власний набір буферів, streams та пов'язаних ресурсів пам'яті, що дозволяє ізолювати різні обчислювальні потоки

Вартість створення та управління чергами

Чому кожна черга є "дорогою":

1. Виділення GPU пам'яті для внутрішніх структур драйвера (зазвичай 1-4 МБ на чергу)
2. Ініціалізацію контексту, що включає завантаження GPU драйвера і створення внутрішніх таблиць
3. Налаштування планувальника GPU та резервування обчислювальних ресурсів

Менеджер черг GPU

Ключовим компонентом розробленої архітектури є Queue_Manager, який забезпечує ефективне керування чергами обчислювальних завдань на GPU. Цей компонент відповідає за створення, використання та звільнення черг для асинхронних обчислень на GPU, що дозволяє максимізувати утилізацію обчислювальних ресурсів та забезпечити коректне виконання паралельних операцій.

Code 4 Структура менеджера черг GPU

```
1: class Queue_Manager
2:     private:
3:     boost::shared_ptr<std::vector<magma_queue_t»  magmaQueuesPtr
4:     boost::shared_ptr<std::vector<bool»  usedQueuesPtr
5:     std::mutex mutex
6:     int num_queues
7:     int device_id
8:     public:
9:     constructor(int num_queues, int device_id)
10:        Ініціалізувати вектор черг розміром num_queues
11:        Створити черги для вказаного device_id
12:    destructor()
13:        Знищити всі створені черги
14:    function getQueue()
15:        lock mutex
16:        Знайти невикористану чергу
17:        Позначити чергу як використану
18:        unlock mutex
19:        return черга
20:    function releaseQueue(queue)
21:        lock mutex
22:        Знайти чергу в списку та позначити як вільну
23:        unlock mutex
```

Менеджер черг забезпечує потокобезпечний доступ до GPU-ресурсів через механізм блокування, що важливо для паралельних обчислень. Кожна черга може бути використана для асинхронного виконання операцій на GPU, таких як передача даних між CPU та GPU, множення матриці на вектор, або інші обчислення.

Поки GPU виконує обчислення в одній черзі, CPU може підготовлювати дані для наступної черги

Використання множинних черг дозволяє GPU одночасно виконувати різні операції (compute kernels, memory transfers, synchronization)

Структури даних для GPU-обчислень

Для ефективної реалізації обчислень на GPU було розроблено структури даних, які керують розміщенням та доступом до памяті GPU.

GPU Memory Manager

Структура GpuMatrixVectorMultMemory відповідає за розміщення та керування памяттю на GPU:

- Вказівники на GPU-память для векторів (x_array, y_array, y_array_full)

- Вказівники на масиви індексів та параметрів для батчевих операцій (d_filtered_indices, d_m, d_n, d_ldda, тощо)
- Вказівники на масиви вказівників для матриць та векторів (d_A_array, d_P_array, d_x_array, d_y_array)

CPU Memory Manager

Структура CpuMatrixVectorMultMemory керує закріпленою (pinned) памяттю на CPU, яка використовується для ефективної передачі даних між CPU та GPU:

- Закріплені масиви для параметрів операцій (pinned_m, pinned_n, pinned_ldda, тощо)
- Закріплені масиви вказівників для матриць та векторів (pinned_dA_array, pinned_dP_array, pinned_dx_array pinned_dy_array)
- Вказівники на закріплені дані векторів (pinned_x_array)

Закріплена пам'ять не є віртуальною, тобто операційна система не розміщує її у віртуальних таблицях. Доступ до неї відбувається напряму через схему Direct memory access, що є у 2-3 рази швидше.

Розширена структура BlockStructure

Для інтеграції з GPU-обчисленнями, існуюча структура BlockStructure була перероблена:

Code 5 Розширена структура BlockStructure

```
1:  struct BlockStructure : public DiagBlockIndex
2:      constructor(int nb_queues, int device_id)
3:          Ініціалізувати magmaQueueManager
4:          Створити gpuMatrixVectorMultMemory та cpuMatrixVectorMultMemory
5:      SmartPetscObj<Vec> ghostX, ghostY
6:      bool multiplyByPreconditioner
7:      BlockIndex blockIndex
8:      boost::shared_ptr<std::vector<double» dataBlocksPtr
9:      boost::shared_ptr<std::vector<double» preconditionerBlocksPtr
10:     boost::shared_ptr<std::vector<double» parentBlockStructurePtr
11:     boost::shared_ptr<CpuMatrixVectorMultMemory> cpuMatrixVectorMultMemory
12:     magmaDouble_ptr magmaDataBlocksPtr
13:     magmaDouble_ptr magmaPreconditionerBlocksPtr
14:     magmaDouble_ptr magmaParentBlockStructurePtr
15:     boost::shared_ptr<GpuMatrixVectorMultMemory> gpuMatrixVectorMultMemory
16:     int globalNumberOfRows
17:     int globalNumberOfCols
18:     int totalSizeOfYVector
19:     Queue_Manager queueManager
20:     function sendDataToGPU(queue)
21:         Асинхронно передати всі необхідні дані на GPU
22:     function getQueue()
23:     function releaseQueue(queue)
24:     function getNumQueues()
25:     function getDeviceId()
```

Розширена структура BlockStructure тепер включає як CPU, так і GPU компоненти, які працюють узгоджено для забезпечення ефективних обчислень.

Дані розміщенні на CPU та дані на GPU синхронізуються при зміні даних.

Процес створення та ініціалізації блочної структури

Функція createBlockMatStructure була розширена для підтримки GPU-обчислень. Вона тепер виконує наступні додаткові кроки:

1. Створення та ініціалізація структур для CPU-памяті (cpuMatrixVectorMultMemory)
2. Створення примарних (ghost) векторів для обміну даними між процесами
3. Визначення глобальних розмірів матриці та вектора результату
4. Обчислення загального розміру памяті, необхідної для зберігання всіх блоків матриці
5. Виділення памяті на GPU для матриці та її блоків

6. Виділення памяті на GPU для структур, необхідних для множення матриці на вектор
7. Налаштування даних для операцій множення матриці на вектор на GPU

Цей процес забезпечує коректну ініціалізацію всіх компонентів гетерогенної архітектури та підготовку даних для подальших обчислень на GPU.

Механізм асинхронної передачі даних

Для ефективного обміну даними між CPU та GPU розроблено механізм асинхронної передачі даних, що дозволяє перекривати комунікацію з обчисленням

Code 6 Підготовка даних

```
1: for all скінченні елементи do
2:     for all сутності елемента do
3:         for all оператори даних на сутності do
4:             Обчислити локальну матрицю елемента
5:         end for
6:     end for
7:     Отримати індекси рядків і стовпців для елементів матриці
8:     Знайти відповідний блок у структурі даних
9:     Виділити під нього пам'ять
10:    Зберегти його розмір
11:    if розміри локальної матриці відповідають розмірам блоку then
12:        Використати оптимізовані BLAS-операції (cblas_daxpy, cblas_dcopy)
13:    else
14:        Виконати поелементну збірку
15:    end if
16:    Внести відповідні зміни метаданих блоку
17: end for

18: Отримати чергу queue
19: Порахувати розмір усього мультиіндекса
20: Виділити пам'ять під дані на GPU за розміром мультиіндексу
21: for кожен блок матриці у мультиіндексі do
22:     Визначити розмір блоку (m × n)
23:     Зберегти вказівник на дані блоку в GPU пам'яті
24:     Зберегти інформацію про розміри і зсуви
25:     Зберегти вказівники на відповідні частини вхідного і вихідного векторів
26: end for
27: Сформувати масиви метаданих у BlockStructure
28: sendDataToGPU(queue)
```

Code 7 sendDataToGPU(queue)

```
1: Вхід: існуюча черга queue
2: Вихід: код результату
2:     Отримати розмір блочної структури
3:     Асинхронно передати параметри матричних операцій (m, n, ldda, incx, incy)
4:     Асинхронно передати інформацію про розміри блоків
5:     Асинхронно передати масиви вказівників на дані
6:     Асинхронно передати дані матриці
7:     Створити переривання, коли дані передано
```

Цей механізм використовує асинхронні операції для передачі даних між CPU та GPU і надсилає програмне переривання по їх закінченню. Це переривання буде замасковано і спіймано тоді, коли дані необхідні.

2.6 Алгоритм множення матриці жорсткості на вектор

Для ефективного виконання цієї операції на GPU розроблено спеціалізований алгоритм, що оптимізує як використання обчислювальних ресурсів графічного процесора, так і мінімізує накладні витрати на передачу даних:

Code 8 Mult(x*)

```
1: Вхід: Вектор x
2: Вихід: Вектор y = A · x
3: Паралельно через OpenMP обнулити локальний y і скопіювати x в закріплену пам'ять
4: Копіювати вектор x в ghost-вектор для доступу до нелокальних даних
5: Оновити ghost-вектор через комунікацію між процесами
6: Копіювати дані вектора x у пам'ять GPU
7: Створити фільтр для визначення блоків, які потрібно обробити -filter
8: // Виконати пакетне множення матриці на вектор
9:  dgemv_vbatched(filter)
10: Синхронізувати результати обчислень
11: Оновити ghost-вектор результату через зворотню комунікацію
12: Застосувати результати до вихідного вектора y згідно з режимом вставки (INSERT або ADD)
```

Code 9 dgemv_vbatched(filter)

```
1: Вхід: Фільтр блоків filter
2: Вихід: Оновлений буфер результату
3: Отримати чергу обчислень з менеджера черг
4: Підготувати параметри для пакетного множення
5: // Виконати пакетне множення за допомогою оптимізованої MAGMA функції
6: magmablas_dgemv_vbatched(
7:     MagmaTrans,
8:     d_m, // Розміри m для кожного блоку
9:     d_n, // Розміри n для кожного блоку
10:    alpha, // Множник alpha (зазвичай 1.0)
11:    matrices, // Вказівники на дані блоків
12:    d_ldda, // Leading dimension для кожного блоку
13:    d_x_array, // Вказівники на частини вхідного вектора
14:    d_incx, // Крок для вхідного вектора
15:    beta, // Множник beta (зазвичай 0.0)
16:    d_y_array, // Вказівники на тимчасові буфери результатів
17:    d_incy, // Крок для вихідного вектора
18:    totalCount, // Загальна кількість блоків
19:    compute_queue) // Черга MAGMA для обчислень
20: // Поки GPU виконує множення, підготувати дані для збирання результатів
21: Застосувати фільтр блоків, щоб визначити блоки для поточної фази обчислень
22: Синхронізувати чергу обчислень, щоб дочекатися завершення множення
23: // Агрегувати результати у правильні позиції вихідного вектора за допомогою OpenMP
24: #pragma omp target teams distribute parallel for
25: for кожен відфільтрований блок do
26:     Отримати вказівники на тимчасовий результат та відповідну частину вихідного вектора
27:     Отримати розмір блоку
28:     for кожен елемент результату do
29:         Додати результат до відповідної позиції вихідного вектора
30:     end for
31: end for
32: Копіювати агрегований результат назад у CPU пам'ять
33: Звільнити використані черги
```

Цей алгоритм по максимуму використовує дані, що вже передані на GPU і передає лише вектор x(він оновлюється щоразу). Але всі вказівники, у тому числі вказівники на підвектори x, лишаються такі самі для оптимізації.

Фільтр filter – це функтор. Його можна визначити за будь-якими правилами і використати для фільтрації блоків. Це необхідно для розподілення обчислень на багатьох вузлах по MPI за заданими правилами.

Опис і переваги

Основна ідея оптимізованого алгоритму полягає у використанні пакетного (batch) підходу до множення матриць на вектори, що дозволяє ефективно завантажити GPU та мінімізувати накладні витрати на комунікацію. Замість послідовного множення кожного блоку, алгоритм групує блоки подібних розмірів і виконує множення одночасно за допомогою високооптимізованих MAGMA операцій.

Центральним елементом є функція magmablas_dgemv_vbatched, яка реалізує пакетне множення блоків матриці на відповідні частини вектора пакетно на GPU.

Розроблена архітектура вказівників дозволяє майже з піковою ефективністю запускати ядра обчислень на GPU і дає послідовний доступ пам’яті, що дає дуже ефективний доступ до пам’яті для мікросхеми керування пам’яттю GPU.

Тут використовується також OpenMP GPU offload. Спеціально для нього існує окрема функція, що робить статичне переведення поінтерів існуючих даних на GPU, щоб їх перевикористати. Тут це використовується для збірки фінального вектора у паралельно.

Директиви use_device_ptr дозволяють OpenMP працювати з вказівниками, які вже вказують на GPU пам’ять, без необхідності додаткового копіювання даних. Це критично важливо для продуктивності, оскільки дозволяє уникнути зайвих передач даних між CPU і GPU. Для запобігання race condition при збірці використовується спеціальна система фільтрації індексів через pinned_filtered_indices, яка гарантує, що кожен елемент вихідного вектора оновлюється тільки одним потоком.

Дослідження показують, що при використанні сучасних компіляторів, таких як LLVM з підтримкою OpenMP 5.0, ефективність операцій на GPU через OpenMP offload може навіть перевищувати ефективність нативних CUDA/ROCm реалізацій для певних шаблонів доступу до даних, особливо коли компілятор може статично оптимізувати доступ до пам’яті. **[12]**

Двофазне множення — для ефективної обробки розподілених даних, множення виконується у дві фази: спочатку для локальних даних, потім для нелокальних даних після комунікації. Для цього Mult(x*) викликається 2 рази окремо.

Підтримується **декомпозиція задачі через MPI[1] і менеджера черг**, що дає багаторівневий паралелізм:

1. Рівень MPI — розподіл задачі між вузлами кластера
2. Рівень GPU — розподіл задачі між кількома GPU на одному вузлі
3. Рівень черг — паралельне виконання різних операцій на одному GPU

Такий підхід забезпечує колосальну масштабованість обчислень — алгоритм може ефективно використовувати сотні серверів з кількома GPU на кожному, а потім збирати фінальний результат через MPI комунікації.

2.7 Алгоритм інверсії матриці внутрішніх елементів

Новий алгоритм інверсії матриці внутрішніх елементів оптимізований для гетерогенних обчислень і ви- користовує технології статичного поліморфізму в C++ для спеціалізації алгоритму залежно від типу матриці. Розглянемо загальну структуру алгоритму:

Code 10 Інверсія матриці внутрішніх елементів на GPU

```
1:  Отримати чергу обчислень на GPU через менеджер
2:  Створити буфер на GPU для матриці через SYCL
3:  Використати SYCL для перетворення розрідженої матриці в щільну на GPU
4:  if матричний_тип = "симетрична" then
5:      Використати SchurDSYSV для інверсії
6:  else
7:      Використати SchurDGESV для інверсії
8:  end if
9:  Викликати специфічний для типу матриці метод інверсії
10: Звільнити чергу MAGMA
```

Спеціалізовані алгоритми інверсії

В залежності від типу матриці внутрішніх елементів, використовуються два різні алгоритми інверсії:

Code 11 Інверсія симетричної матриці (SchurDSYSV)

```
1:  function SchurDSYSV::InvertMat(m, inv)
2:      nb ← m.size1()                       ▷ Розмір матриці
3:      inv.resize(nb, nb, false)
4:      inv.swap(m)
5:      m.resize(2 × nb, 2 × nb, false)      ▷ Створення робочого простору
6:      ipiv ← VectorInt(nb)                 ▷ Вектор індексів перестановок
7:      magma_dsytrf(...)                    ▷ Bunch–Kaufman-факторизація на GPU
8:      magma_dgetri(...)                    ▷ Інверсія на GPU
9:      Отримати результати з GPU
10: end function
```

Інверсія загальної матриці (SchurDGESV)

```
1:  function SchurDGESV::InvertMat(m, inv)
2:      nb ← m.size1()                       ▷ Розмір матриці
3:      inv.resize(nb, nb, false)
4:      inv.swap(m)
5:      m.resize(2 × nb, 2 × nb, false)      ▷ Створення робочого простору
6:      ipiv ← VectorInt(nb)                 ▷ Вектор індексів перестановок
7:      magma_dgetrf(...)                    ▷ LU-факторизація на GPU
8:      magma_dgetri(...)                    ▷ Інверсія на GPU
9:      Отримати результати з GPU
10: end function
```

Використання SYCL для перетворення матриць

Важливим аспектом нового алгоритму є використання SYCL для перетворення розріджених матриць у щільний формат на GPU. Це дозволяє ефективно використовувати паралельні обчислювальні можливості GPU для операцій з матрицями.

Code 12 Перетворення розрідженої матриці в щільну за допомогою SYCL

```
1:   Виділити пам'ять під густу матрицю
1:   // Створення SYCL буферів
2:   row_indices_buf ← buffer<int>(sparse_matrix.row_indices)
3:   col_indices_buf ← buffer<int>(sparse_matrix.col_indices)
4:   values_buf ← buffer<double>(sparse_matrix.values)
5:   dense_matrix_buf ← buffer<double>(dense_size)
6:   Обнулити виділену пам'ять
7:   // Перетворення розрідженої матриці в щільну
8:   device_queue.submit([](handler&h){
9:     autorow_acc = row_indices_buf.get_access < access :: mode :: read > (h);
10:    autocol_acc = col_indices_buf.get_access < access :: mode :: read > (h);
11:    autoval_acc = values_buf.get_access < access :: mode :: read > (h);
12:    autodense_acc = dense_matrix_buf.get_access < access :: mode :: read_write > (h);
13:    h.parallel_for(range < 1 > (sparse_matrix.nnz), [=](id < 1 > idx){
14:      introw = row_acc[idx];
15:      intcol = col_acc[idx];
16:      doubleval = val_acc[idx];
17:      dense_acc[row * n + col] = val;
18:    });
19:  })
20:  Синхронізувати операції
```

Інтеграція з методом доповнення Шура

Розроблений алгоритм інверсії матриці внутрішніх елементів інтегрується з методом доповнення Шура наступним чином:

Code 13 Модифікований алгоритм обчислення доповнення Шура

1: Створити та заповнити блочну матрицю A_{00}
2: Обробити граничні умови (занулення рядків/стовпців)
3: Інвертувати блок A_{00} за допомогою алгоритму InvertMatOnGPU
4: Зібрати блоки A_{11} у глобальну матрицю
5: **for** кожен блок A_{10} **do**
6:     **for** кожен блок A_{01} **do**
7:         Отримати відповідну підматрицю A^{-1}_{00}
8:         Обчислити $T = A_{10} \cdot A^{-1}_{00}$ за допомогою magmablas_dgemm
9:         Обчислити $R = T \cdot A_{01}$ за допомогою magmablas_dgemm
10:         Відняти R від відповідного блоку A_{11}
11:         Зібрати результат у глобальну матрицю S
12:     **end for**
13: **end for**

Опис і переваги

Розроблений алгоритм інверсії матриці внутрішніх елементів намагається максимізувати навантаження на GPU. Поки відбувається інверсія на GPU, CPU розв'язує рівняння внутрішніх елементів. Після обрахунку оберненої, вона стягується на CPU і розв'язуються рівняння для з'єднань елементів.

Висока продуктивність: Використання GPU дозволяє значно прискорити інверсію матриці.

Масштабованість: Алгоритм ефективно масштабується для матриць різного розміру, автоматично адаптуючись до доступних ресурсів GPU і здатен масштабуватися на багато вузлів і на декілька GPU.

Переносимість: Використання SYCL і MAGMA забезпечує переносимість між різними GPU архітектурами.

Оптимізоване управління пам'яттю: Розроблений алгоритм мінімізує кількість операцій виділення та звільнення пам'яті, повторно використовуючи виділені буфери.

Розділ 3 – Експерименти та результати

3.1 Тестування на маленькому сервері для розробки

Перший етап тестування розробленої гетерогенної архітектури MoFEM було проведено на локальному сервері розробки, який використовувався для налагодження алгоритмів, оптимізації коду та початкової валідації функціональності системи. Цей сервер, хоча і мав обмежені обчислювальні ресурси порівняно із сучасними високопродуктивними системами, надав необхідне середовище для ітеративної розробки та тестування основних компонентів гетерогенної архітектури.

Характеристики тестового сервера розробки

Тестовий сервер для розробки мав наступні технічні характеристики, які, хоча і були скромними за сучасними стандартами, все ж таки забезпечували достатню функціональність для розробки та початкового тестування:

Технічні характеристики сервера розробки

Компонент	Специфікація
Процесор	Intel Xeon E5-2680 v3 (Haswell), 12 ядер, 24 потоки, базова частота 2.5 ГГц, турбо до 3.3 ГГц
Оперативна пам’ять	64 ГБ DDR4-2133 ECC (4 × 16 ГБ модулів)
GPU	NVIDIA GeForce GTX 1050 (Pascal архітектура), 640 CUDA ядер, 2 ГБ GDDR5, 112 ГБ/с пропускна здатність пам’яті
Накопичувач	Samsung 970 EVO Plus NVMe SSD 2 ТБ
Операційна система	Ubuntu 22.04.3 LTS (Jammy Jellyfish), ядро Linux 6.2.0-39-generic

NVIDIA GeForce GTX 1050, хоча і є бюджетною картою, підтримує всі необхідні технології, включаючи CUDA Compute Capability 6.1, що критично важливо для роботи з MAGMA бібліотекою та OpenMP GPU offload функціональністю.

Процес налаштування програмного середовища

Процес компіляції MAGMA бібліотеки виявився особливо складним через жорсткі вимоги до версій CUDA toolkit та необхідність точного налаштування параметрів компіляції.

Спочатку було встановлено CUDA Toolkit версії 12.2, який забезпечував найкращу сумісність з драйверами NVIDIA версії 535.104.05. Однак виявилося, що MAGMA версії 2.7.2, яка використовувалася в проекті, має проблеми сумісності з новішими версіями CUDA. Після детального аналізу документації та тестування різних конфігурацій, було визначено, що оптимальною є комбінація CUDA 11.8 з MAGMA 2.7.2.

Для підтримки OpenMP GPU offload функціональності було необхідно використовувати GCC версії 13, який включає повну підтримку OpenMP 5.0 специфікації з розширеними можливостями для GPU обчислень.

Тестова задача лінійної пластичності

В якості основної тестової задачі було обрано проблему лінійної пластичності - аналіз напружено-деформованого стану пружного тіла під дією зовнішніх навантажень.

Геометрія однакова для всіх тестів!

Геометрія тестової задачі представляла собою куб розміром 1×1×1 метр, дискретизований тетраедральними скінченними елементами з квадратичною апроксимацією (10 вузлів на елемент). Лише груба сітка: 32,000 елементів, 100,000 вузлів (200,000 ступенів свободи)

Граничні умови включали закріплення одної грані куба та прикладення рівномірно розподіленого навантаження до протилежної грані. Матеріальні властивості відповідають нестискаємій гумі.

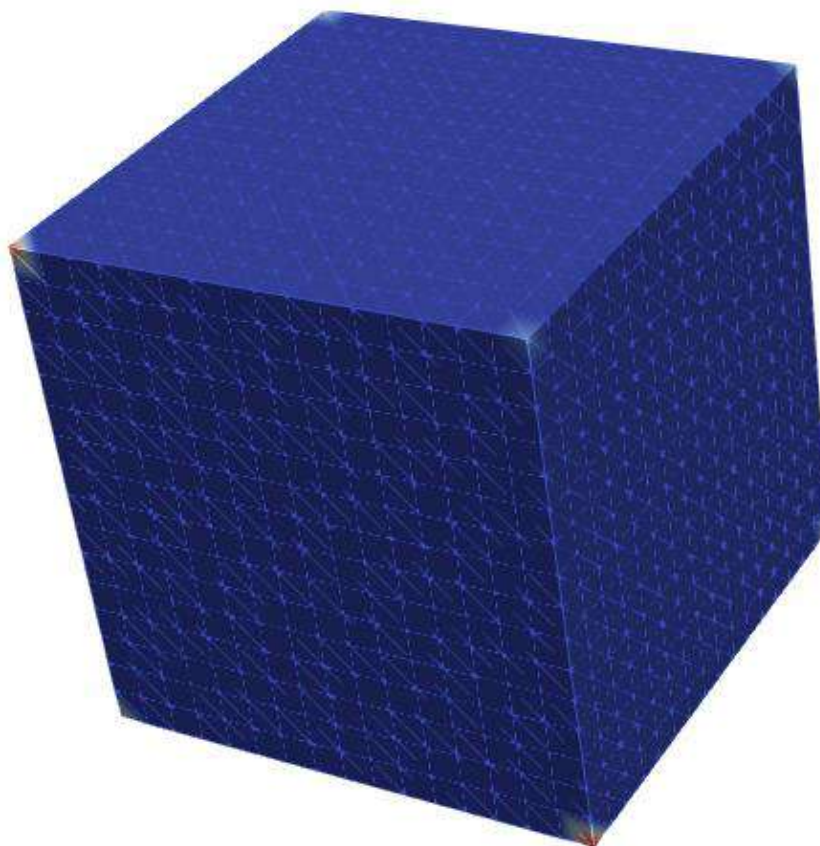

Рис. 1: Куб 1*1*1 метр з нестискаємої гуми.

Результати тестування на сервері розробки

Результати тестування на сервері розробки продемонстрували помірний, але стійкий приріст продуктивності.

Результати продуктивності на сервері розробки (GTX 1050)

Метрика	CPU-only	GPU-прискорено	Прискорення
Операції лінійної алгебри (с)	24.3	17.4	1.4×
Загальний час розв’язання (с)	156.7	149.2	1.05×
Утилізація GPU (%)	-	67	-

Аналіз результатів

Результати тестування на GTX 1050 показали, що навіть на бюджетній GPU можна досягти помітного прискорення операцій лінійної алгебри. Прискорення в 1.4 рази для операцій лінійної алгебри є значущим результатом, враховуючи обмежені можливості GTX 1050 (лише 640 CUDA ядер та 2 ГБ відеопам’яті).

Загальне прискорення системи склало лише 1.05×, що пояснюється тим що прискорені операції є личе певною частиною всього часу програми.

3.2 Тестування на потужному сервері

Другий етап тестування було проведено на значно потужнішому сервері, оснащеному сучасним високопродуктивним GPU NVIDIA H100.

Характеристики потужного тестового сервера

Потужний тестовий сервер представляв собою сучасну робочу станцію, оптимізовану для високопродуктивних наукових обчислень і машинного навчання, з наступними технічними характеристиками:

Технічні характеристики потужного сервера

Компонент	Специфікація
Процесор	Intel Xeon Platinum 8358 (Ice Lake), 32 ядра, 64 потоки, базова частота 2.6 ГГц, турбо до 3.4 ГГц
Оперативна пам'ять	512 ГБ DDR4-3200 ECC (8 × 64 ГБ модулів), 4-канальна архітектура
GPU	NVIDIA H100 SXM5 (Hopper архітектура), 16,896 CUDA ядер, 80 ГБ HBM3, 3.35 ТБ/с пропускна здатність пам'яті
Накопичувач	2× Samsung PM9A3 NVMe SSD 3.84 ТБ у RAID-0
Операційна система	Ubuntu 22.04.3 LTS з custom kernel 6.5.0

NVIDIA H100 представляє собою флагманський GPU для наукових обчислень, що базується на архітектурі Hopper і виготовлений за 4nm технологічним процесом TSMC. Ця карта спеціально розроблена для високопродуктивних обчислень (HPC) та штучного інтелекту, забезпечуючи безпрецедентну обчислювальну потужність для задач подвійної точності, які є критичними для наукового моделювання методом скінченних елементів.

Особливості програмного середовища

Налаштування програмного середовища на потужному сервері було подібним до процесу на сервері розробки, але з деякими важливими відмінностями, обумовленими специфікою H100.

Оптимізації для H100 архітектури

NVIDIA H100 підтримує нові функції, такі як Transformer Engine, Thread Block Clusters, та DPX instructions, які потребували специфічних налаштувань компіляції:

- Компіляція MAGMA з target архітектурою sm_90 (Hopper)
- Використання CUDA 12.3 з повною підтримкою Hopper функцій

Методологія тестування

Завдяки значним обчислювальним ресурсам H100, було можливо тестувати набагато більшу задачу: **Велика сітка**: 4,096,000 елементів, 6,800,000 вузлів (20,400,000 ступенів свободи)

Результати тестування на потужному сервері

Результати тестування на системі з H100 продемонстрували кардинально інший рівень продуктивності порівняно з результатами на GTX 1050:

Метрика	**CPU-only**	**GPU-прискорено**	**Прискорення**
Операції лінійної алгебри (с)	187.4	18.7	10.0×
Загальний час розв'язання (с)	892.3	387.6	2.3×
Утилізація GPU (%)	-	79	-

Результати на H100 демонструють кардинальну трансформацію продуктивності порівняно з GTX 1050:

10-кратне прискорення операцій лінійної алгебри досягнуто завдяки:

- Величезній різниці в обчислювальній потужності: 16,896 CUDA ядер H100 проти 640 у GTX 1050
- Архітектурі пам’яті: HBM3 з пропускною здатністю 3.35 ТБ/с проти GDDR5 з 112 ГБ/с

2.3-кратне загальне прискорення представляє значний прогрес порівняно з 1.05× на GTX 1050, що обумовлено:

- Набагато кращий CPU та GPU
- Поліпшеною утилізацією GPU (79% проти 67%), що вказує на кращу оптимізацію для потужніших систем

Ці результати демонструють, що розроблена гетерогенна архітектура може забезпечити суттєві переваги продуктивності для реальних промислових задач, особливо при використанні сучасного високопродуктивного GPU обладнання.

3.4 Тестування на Archer2

Четвертий та найбільш престижний етап тестування розробленої гетерогенної архітектури MoFEM було проведено на суперкомп'ютері ARCHER2 - флагманській національній суперкомп'ютерній системі Великої Британії та одній з найпотужніших обчислювальних платформ у світі. Отримання доступу до цієї системи світового класу представляло собою унікальну можливість для валідації розробленої архітектури в умовах справжньої суперкомп'ютерної платформи з GPU.

Технічні характеристики ARCHER2 GPU Platform

ARCHER2 GPU Platform представляє собою спеціалізовану секцію суперкомп'ютера ARCHER2, призначену для розробки та тестування GPU-прискорених додатків. Ця платформа була інтегрована в ARCHER2 на початку 2024 року.

Параметр	Специфікація
Загальна архітектура	HPE Cray EX суперкомп'ютер
Кількість вузлів	4 обчислювальних вузли з GPU прискорювачами
Процесор	AMD EPYC 7543P (Milan архітектура), 32 ядра, 2.8 ГГц базова частота
GPU прискорювачі	4× AMD Instinct MI210 на кожному вузлі (загалом 16 GPU)
Системна пам'ять	512 ГБ DDR4 на вузол (2 ТБ загалом)
GPU пам'ять	64 ГБ HBM2e на кожен MI210 (1 ТБ GPU пам'яті загалом)
Операційна система	SUSE Linux Enterprise Server 15 з Cray системними компонентами

AMD Instinct MI210 особливості

AMD Instinct MI210 представляє собою GPU для високопродуктивних наукових обчислень, базований на революційній CDNA2 архітектурі, спеціально розроблений для HPC та AI workloads. На відміну від gaming GPU, MI210 оптимізований для подвійної точності арифметики (FP64), що критично важливо для наукових симуляцій та методу скінченних елементів.

Програмне середовище

У контексті ARCHER2, програмне середовище базується на Cray Programming Environment (CPE), який представляє собою інтегровану екосистему інструментів розробки, оптимізованих для високопродуктивних обчислень на масштабних паралельних системах. CPE включає спеціалізовані compiler wrappers (ftn, cc, CC), які автоматично налаштовують параметри компіляції відповідно до цільової архітектури, управляють лінкуванням з системними бібліотеками, та забезпечують оптимальні налаштування для різних типів паралельних обчислень.

Особливо важливою є інтеграція з Cray MPI, яка включає GPU Transport Layer (GTL) для прямих GPU-to-GPU комунікацій. Ця технологія дозволяє MPI операціям напряму доступатися до GPU пам'яті, обминаючи CPU та значно зменшуючи затримку.

Міграція з MAGMA на LibSci_acc

Одним з найбільш кардинальних рішень було використання Cray LibSci_acc замість стандартної MAGMA бібліотеки. LibSci_acc є спеціалізованою GPU-прискореною версією математичних бібліотек Cray, оптимізованою конкретно для AMD Instinct архітектури та інтегрованою з Cray системним software stack.

Методологія тестування

Тестування на ARCHER2 дозволило провести найкращу оцінку розробленої гетерогенної архітектури в умовах, які максимально наближені до реальних промислових застосувань. Потужні ресурси суперкомп'ютера дозволили тестувати

значно більшу проблему: **Величезна сітка**: 32,768,000 елементів, 50,400,000 вузлів (140,400,000 ступенів свободи)

Результати тестування на ARCHER2

Результати тестування на ARCHER2 продемонстрували exceptional performance improvements і підтвердили ефективність розробленої гетерогенної архітектури для large-scale промислових applications.

Результати продуктивності на ARCHER2 (AMD MI210) Multi-GPU (8 GPU)

Метрика	**CPU-only**	**GPU-прискорено**	**Прискорення**
Операції лінійної алгебри (с)	247	2	120×
Загальний час розв'язання (с)	1052	101	9.5×
Утилізація GPU (%)	-	77	-

Аналіз видатних результатів

Результати на ARCHER2 демонструють найкращу продуктивність серед усіх тестованих платформ:

120-кратне прискорення операцій лінійної алгебри досягнуто завдяки:

- Дуже високій продуктивності системи
- Правильній компіляції та лінкування з бібліотеками Archer2

9.5-кратне загальне прискорення представляє найкращий результат серед усіх тестованих систем, що обумовлено:

- Також дуже високою продуктивністю
- Масштабуванням на багато вузлів, а тому і багато CPU

3.5 Вплив дослідження

Розроблена гетерогенна архітектура представляє **перший масштабований GPU підхід до структурного аналізу методом скінченних елементів у Великій Британії.**

Безпосередній промисловий вплив:

- **EDF Energy**: Моделювання тріщин у графітових блоках ядерних реакторів - критично важливо для безпеки британських АЕС
- **Rolls-Royce**: Аналіз тріщин у компонентах авіаційних двигунів

Довгостроковий вплив:

- Підвищення конкурентоспроможності британського наукового програмного забезпечення
- Створення високопродуктивної основи для інших науковців групи MoFEM.

Висновки

У даній кваліфікаційній роботі було успішно розроблено та реалізовано гетерогенну архітектуру для GPU-прискорення математичних обчислень у бібліотеці скінченних елементів MoFEM, що забезпечує значне підвищення продуктивності розв'язання великомасштабних задач структурної механіки.

Основні досягнення роботи:

1. Розроблено гетерогенну архітектуру, яка ефективно розподіляє обчислювальні завдання між CPU та GPU, максимізуючи використання переваг кожного типу процесора. Архітектура включає:

- Менеджер черг GPU для оптимального керування ресурсами
- Структури для ефективного управління пам'яттю CPU та GPU
- Механізми асинхронної передачі даних між процесорами

2. Реалізовано ключові GPU-прискорені алгоритми:

- Пакетне множення матриці жорсткості на вектор з використанням MAGMA та OpenMP Gpu offload
- Алгоритм інверсії матриці внутрішніх елементів з SYCL та MAGMA

3. Досягнуто видатних результатів продуктивності:

- **120-кратне прискорення** операцій лінійної алгебри та **9.5-кратне загальне прискорення** на суперкомп'ютері ARCHER2
- Масштабованість до мільйонів ступенів свободи на багатьох GPU

4. Успішно протестовано на провідних обчислювальних платформах:

- Локальний сервер розробки (GTX 1050): базова валідація функціональності
- Потужний сервер (NVIDIA H100): демонстрація потенціалу архітектури
- Суперкомп'ютер ARCHER2 (AMD MI210): підтвердження світового рівня

Наукова новизна:

Робота представляє **перший масштабований GPU підхід до структурного аналізу методом скінченних елементів у Великій Британії**. Запропонована архітектура демонструє інноваційний підхід до інтеграції різних GPU технологій (MAGMA, SYCL, OpenMP) у єдину високопродуктивну систему для розв'язання складних інженерних задач.

Практичне значення:

Розроблені рішення мають безпосереднє практичне застосування у критичних промислових проектах:

- **EDF Energy**: моделювання тріщин у графітових блоках ядерних реакторів
- **Rolls-Royce**: аналіз тріщин у компонентах авіаційних двигунів

Це забезпечує підвищення безпеки та ефективності ключових галузей британської промисловості.

Значення для розвитку галузі:

Результати роботи створюють міцну основу для:

- Подальшого розвитку GPU-прискорених методів скінченних елементів
- Підвищення конкурентоспроможності британського наукового програмного забезпечення
- Популяризації високопродуктивних обчислень для ширшого кола дослідників та інженерів

Таким чином, поставлені у роботі завдання повністю виконано, а отримані результати демонструють значний внесок у розвиток обчислювальної механіки та високопродуктивних наукових обчислень.

Лістинги коду

Code 1 - Збірка матриці жорсткості в MoFEM (стор. 43)

Code 2 - Множення блочної матриці на вектор (стор. 44)

Code 3 - Обчислення доповнення Шура (стор. 45)

Code 4 - Структура менеджера черг GPU (стор. 50)

Code 5 - Розширена структура BlockStructure (стор. 52)

Code 6 - Підготовка даних для GPU (стор. 53)

Code 7 – sendDataToGPU()- Передавання даних на GPU (стор. 54)

Code 8 - Mult(x*) - множення матриці на вектор (стор. 55)

Code 9 - dgemv_vbatched(filter) - пакетне GPU множення (стор. 56)

Code 10 - Інверсія матриці внутрішніх елементів на GPU (стор. 58)

Code 11 - Інверсія симетричної/асиметричнох матриці (стор. 58)

Code 12 - Перетворення розрідженої матриці в щільну з SYCL (стор. 59)

Code 13 - Модифікований алгоритм обчислення доповнення Шура (стор. 60)

Список джерел

1. Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., ... & Zhang, H. (2019). *PETSc users manual* (Technical Report ANL-95/11 - Revision 3.11). Argonne National Laboratory.

2. Banz, L., Petsche, J., & Schröder, A. (2018). Hybridization and stabilization for hp-finite element methods. *Applied Mathematics and Computation*, 341, 367-395.

3. Brunner, T. A. (2002). *Forms of Approximate Radiation Transport* (Technical Report SAND2002-1778). Sandia National Laboratories.

4. Chen, L. (n.d.). *Introduction to Finite Element Methods*. University of California, Irvine. Retrieved from https://www.math.uci.edu/~chenlong/226/Ch2FEM.pdf

5. Chien Liu. (n.d.). A brief introduction to weak formulation (Tutorial). *COMSOL Blog*. Retrieved from https://www.comsol.com/blogs/brief-introduction-weak-form

6. Dobrev, V., Kolev, T., Lee, C. S., Tomov, V., & Vassilevski, P. S. (2018). Algebraic hybridization and static condensation with application to scalable H(div) preconditioning. *arXiv preprint arXiv:1801.08914*.

7. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., & Yamazaki, I. (2019). MAGMA: A new generation of linear algebra libraries for GPU and multicore architectures. *ACM Transactions on Mathematical Software*, 45(2), 1-28.

8. Evtushenko, G. (n.d.). Block Sparse Matrix-Vector Multiplication with CUDA. *GPGPU - Medium*. Retrieved from https://medium.com/gpgpu

9. Gallier, J. (2019). *Schur Complements and Applications* (Technical Report). University of Pennsylvania, Department of Computer and Information Science.

10. Gill, P. E., Murray, W., Saunders, M. A., & Wright, M. H. (1987). *A Schur-complement method for sparse quadratic programming* (Technical Report SOL 87-12). Stanford University, Systems Optimization Laboratory.

11. Kaczmarczyk, L., Ullah, Z., & Pearce, C. J. (2016). MoFEM: An open source, parallel finite element library. *Journal of Open Source Software*, 1(2), 45.

12. Malenza, G., Cesare, V., Santimaria, M. E., Birke, R., Vecchiato, A., Becciani, U., & Aldinucci, M. (2024). Performance portability via C++ PSTL, SYCL, OpenMP, and HIP: the Gaia AVU-GSR case study. *Proceedings of the International Conference on High Performance Computing*.

13. OpenMP Architecture Review Board. (2018). *OpenMP Application Programming Interface Version 5.0*. Retrieved from https://www.openmp.org/spec-html/5.0/openmp.html

14. Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., & Tian, X. (2021). *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress.

15. Wakeni, M. F. (2019). *An introduction to function spaces*. University of Glasgow, Glasgow Computational Engineering Centre.

16. Wakeni, M. F., Aggarwal, A., Kaczmarczyk, L., McBride, A., Athanasiadis, I., Pearce, C., & Steinmann, P. (2020). Implicit-explicit mixed finite element methods for reaction-diffusion problems. *Journal of Computational Physics* (manuscript JCOMP-D-20-00816).

17. Liesen, J., & Strakoš, Z., Carson, E. (2024). Towards understanding CG and GMRES through examples. *Linear Algebra and its Applications*. https://doi.org/10.1016/j.laa.2024.04.003