Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Кваліфікаційна робота

на здобуття ступеня вищої освіти «бакалавр»

«Паралельний алгоритм дробового інтегрування»

«A parallel fractional integration algorithm»

Виконав: здобувач денної форми навчання
спеціальності 113 Прикладна математика
Освітня програма «Прикладна математика»
Полуектов Іван Олександрович

Керівник: канд. фіз.-мат. наук, доц. Вербіцький В.В. _____
Рецензент: канд. фіз.-мат. наук, доц. Таірова М.С.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ 2025 р.

Завідувач кафедри

_____ _____

Захищено на засіданні ЕК № _____

Протокол № ___ від _____ 2025 р.

Оцінка _____ / _____ / _____

Голова ЕК

_____ _____

Одеса — 2025 р.

ЗМІСТ

ВСТУП

Паралельне програмування стає все більш важливим для розробки програмного забезпечення. Паралельне програмування та розробка ефективних паралельних програм добре зарекомендували себе у високопродуктивних наукових обчисленнях протягом багатьох років.

Моделювання наукових проблем є важливою галуззю природничих та інженерних наук, значення якої зростає. Більш точне моделювання або моделювання більших проблем призводить до зростання попиту на обчислювальну потужність та обсяг пам'яті. В останні десятиліття високопродуктивні дослідження також включали розробку нових паралельних апаратних та програмних технологій, і можна спостерігати стабільний прогрес у паралельних високопродуктивних обчисленнях. Популярними прикладами є моделювання прогнозу погоди на основі складних математичних моделей, що включають диференціальні рівняння з частинними похідними, або моделювання аварій в автомобільній промисловості на основі методів скінченних елементів.

Комп'ютерне моделювання часто вимагає великих обчислювальних зусиль. Таким чином, низька продуктивність використовуваної комп'ютерної системи може значно обмежити моделювання та точність отриманих результатів. Використання високопродуктивної системи дозволяє проводити більші симуляції, що призводить до кращих результатів, і тому для виконання комп'ютерного моделювання зазвичай використовуються паралельні комп'ютери. Сьогодні кластерні системи, побудовані з серверних вузлів, широко доступні та також часто використовуються для паралельного моделювання. Крім того, багатоядерні процесори у вузлах забезпечують додатковий паралелізм, який можна використовувати для швидких обчислень. Для використання паралельних комп'ютерів або кластерних систем обчислення, що виконуються, повинні бути розділені на кілька частин, які призначені паралельним ресурсам для виконання.

Зазвичай це стосується наукового моделювання, яке часто використовує одно- або багатовимірні масиви як структури даних та організовує їхні обчислення у вкладених циклах. Щоб отримати паралельну програму для

паралельного виконання, алгоритм має бути сформульований відповідною мовою програмування. Паралельне виконання часто контролюється спеціальними бібліотеками середовища виконання або директивами компілятора, які додаються до стандартної мови програмування, такої як C, Fortran або Java.

Паралельне програмування є важливим аспектом високопродуктивних наукових обчислень, але раніше воно було нішею в усій галузі апаратних та програмних продуктів. Однак, останнім часом паралельне програмування покинуло цю нішу та стане основним напрямком методів розробки програмного забезпечення через радикальну зміну в апаратних технологіях.

Великі виробники мікросхем почали випускати процесори з кількома енергоефективними обчислювальними блоками на одному кристалі, які мають незалежне керування та можуть одночасно отримувати доступ до однієї й тієї ж пам'яті. Зазвичай термін "ядро"використовується для позначення окремих обчислювальних блоків, а термін "багатоядерний"– для всього процесора, що має кілька ядер. Таким чином, використання багатоядерних процесорів робить кожен настільний комп'ютер невеликою паралельною системою. Технологічний розвиток у напрямку багатоядерних процесорів був зумовлений фізичними причинами, оскільки тактову частоту мікросхем з більшою кількістю транзисторів неможливо збільшити з попередньою швидкістю без перегріву.

Багатоядерні архітектури у вигляді окремих багатоядерних процесорів, систем спільної пам'яті з кількох багатоядерних процесорів або кластерів багатоядерних процесорів з ієрархічною мережею взаємозв'язків матимуть великий вплив на розробку програмного забезпечення. Ще однією тенденцією в паралельних обчисленнях є використання графічних процесорів (GPU) для ресурсоємних програм. Архітектури GPU забезпечують сотні спеціалізованих обчислювальних ядер, які можуть виконувати обчислення паралельно.

Користувачі комп'ютерної системи зацікавлені у використанні переваг підвищення продуктивності, яке забезпечують багатоядерні процесори. Якщо цього вдасться досягти, вони можуть очікувати, що їхні прикладні програми будуть працювати швидше та отримуватимуть все більше додатко-

вих функцій, які не можна було інтегрувати в попередні версії програмного забезпечення, оскільки вони потребували занадто великої обчислювальної потужності. Щоб забезпечити це, операційна система обов'язково повинна підтримувати їх, наприклад, використовуючи виділені ядра за призначенням або запускаючи кілька користувацьких програм паралельно, якщо вони доступні. Але коли передбачено велику кількість ядер, що станеться найближчим часом, також виникає потреба виконувати одну прикладну програму на кількох ядрах. Найкращою ситуацією для розробника програмного забезпечення було б наявність автоматичного перетворювача, який приймає послідовну програму як вхідні дані та генерує паралельну програму, що ефективно працює на нових архітектурах. Якби такий перетворювач був доступний, розробка програмного забезпечення могла б продовжуватися як і раніше. Але, на жаль, досвід досліджень у паралелізації компіляторів протягом останніх 20 років показав, що для багатьох послідовних програм неможливо автоматично витягти достатню кількість паралелізму. Тому потрібна певна допомога від програміста, і прикладні програми необхідно відповідно реструктуризувати.

Для розробника програмного забезпечення розробка нового обладнання в напрямку багатоядерних архітектур є викликом, оскільки існуюче програмне забезпечення має бути реструктуризоване в бік паралельного виконання, щоб скористатися перевагами додаткових обчислювальних ресурсів.

Метою даної роботи є розробка паралельних алгоритмів обчислення визначеного інтеграла з використанням сучасних технологій паралельного програмування для гетерогенних паралельних комп'ютерних систем.

Об'єктом роботи є методи обчислення визначених інтегралів.

Предметом дослідження є ефективні паралельні алгоритми обчислення визначених інтегралів для гетерогенних комп'ютерних систем.

РОЗДІЛ 1

ПОСТАНОВКА ЗАДАЧІ

Задача обчислення визначених інтегралів широко використовується при розв'язуванні складних задач математичної фізики та техніки. Наприклад, при побудові скінченно-елементних апроксимацій крайових задач.

Важливою особливістю числового інтегрування є можливість оцінити точність обчислення. З появою паралельних обчислювальних систем постає питання щодо ефективної реалізації відомих алгоритмів числового інтегрування для різних паралельних архітектур.

В кваліфікаційній роботі треба розробити паралельні алгоритми обчислення визначеного інтегралу з використанням складеної квадратурної формули Ньютона-Котеса. Точність обчислень треба контролювати за правилом Рунге.

Треба розробити паралельні алгоритми для комп'ютера з багатоядерним процесором та для комп'ютера з використанням обчислень на графічних процесорах.

Треба провести обчислювальні експерименти щодо встановлення ефективності паралельних реалізацій.

РОЗДІЛ 2

КВАДРАТУРНІ ФОРМУЛИ НЬЮТОНА-КОТЕСА

Інтеграл

$$I(f) = \int_a^b f(x)dx, \tag{2.1}$$

в якому функція $f(x)$ відома, можна наближено обчислити наступним чином: Представимо функцію $f(x)$ у вигляді

$$f(x) = L_{n-1}(x) + R_{n-1}(x),$$

$$L_{n-1}(x) = \sum_{i=1}^{n} f(x_i) \prod_{j=1(j\neq i)}^{n} \frac{x - x_j}{x_i - x_j},$$

— інтерполяційний поліном Лагранжа, побудований по вузлах $x_1, \ldots, x_n$ на проміжку $[a, b]$,

$$R_{n-1}(x) = f(x_1; \ldots; x_n; x)(x - x_1) \ldots (x - x_n).$$

— залишковий член інтерполяційного полінома Лагранжа у формі з поділеними різницями. Тоді

$$I(f) = \int_a^b f(x)dx = \int_a^b L_{n-1}(x)dx + \int_a^b R_{n-1}(x)dx = S_n(f) + r_n(f),$$

де

$$S_n(f) = \sum_{i=1}^{n} \int_a^b \prod_{j=1(j\neq i)}^{n} \frac{x - x_j}{x_i - x_j} dx f(x_i), \tag{2.2}$$

$$r_n(f) = \int_a^b f(x_1; \ldots; x_n; x)(x - x_1) \ldots (x - x_n)dx. \tag{2.3}$$

$$I(f) \approx S_n(f).$$

$S_n(f)$ - квадратурна формула Ньютона-Котеса (або Котса).
$r_n(f)$ - залишок, який визначає точність квадратурної формули.

Покладемо

$$f(x) = f(a) + f(a;b)(x-a) + f(a;b;x)(x-a)(x-b).$$

Тоді ми отримуємо квадратурну формулу трапецій

$$S_2(f) = \int_a^b f(a) + f(a;b)(x-a)dx = \frac{f(a)+f(b)}{2}(b-a). \tag{2.4}$$

Якщо скористатись другою теоремою про середнє та зв'язком поділених різниць з похідними, можна отримати її точність.

$$r_2(f) = \int_a^b f(a;b;x)(x-a)(x-b)dx =$$

$$= f(a;b;\xi)\int_a^b (x-a)(x-b)dx = -\frac{f''(\xi_1)}{12}(b-a)^3, \tag{2.5}$$

де ξ, ξ_1 деякі точки проміжку (a,b).

2.1 Складені квадратурні формули

Для більш точних результатів можна розбити відрізок $[a,b]$ на n рівних відрізків довжиною $h = (b-a)/n$. Позначимо $x_i = a + ih$, $i = \overline{0,n}$. Для того, щоб обчислити

$$\int_{x_{i-1}}^{x_i} f(x)dx, \quad i = \overline{1,n},$$

застосуємо квадратурну формулу трапецій. Будемо застосовувати її до кожного з проміжків та складемо усі результати - це складена квадратурна формула. Аналогічно можна отримати її похибку. Наприклад, нехай є

інтеграл

$$I(f) = \int_a^b f(x)dx.$$

Для його наближенного обчислення маємо складену квадратурну формулу трапецій

$$S_h(f) = \frac{h}{2}(f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)), \tag{2.6}$$

з похибкою

$$r_h(f) = -\frac{h^3}{12}\sum_{i=1}^{n} f''(\xi_i)$$

Її оцінка:

$$|r_h(f)| \leq \frac{h^2(b-a)}{12}\max_{x\in(a,b)}|f''(x)|, \tag{2.7}$$

при умові, що підінтегральна функція $f(x)$ є гладкою.

Можна побачити, що складені квадратурні формули трапецій мають другий порядок точності.

2.2 Принцип Рунге

За допомогою принципа Рунге можна отримати апостеріорну оцінку похибки для складених квадратурних формул.

Запишемо похибку формули трапецій наступним чином

$$r_h(f) = -\frac{h^3}{12}\sum_{i=1}^{n} f''(\xi_i) = -\frac{h^2}{12}(h\sum_{i=1}^{n} f''(\xi_i)) =$$

$$= -\frac{h^2}{12}\left(\int_a^b f''(x)dx + O(h^\alpha)\right) = -\frac{h^2}{12}\int_a^b f''(x)dx + o(h^2),$$

де $0 < \alpha \leq 1$.

Покладемо

$$I(f) = S_h(f) + r_h(f) = S_h(f) + C_m h^m + o(h^m).$$

Розділимо проміжок $[a, b]$ на n_1 відрізків, кожний довжиною $h_1 = (b-a)/n_1$, а також на $n_2 = \lambda n_1$ відрізків, кожний довжиною $h_2 = (b-a)/n_2$ $(\lambda > 1)$. Тоді, якщо враховувати тільки головний член похибки, можна отримати наближені рівності:

$$I(f) \approx S_{h_1}(f) + C_m h_1^m,$$
$$I(f) \approx S_{h_2}(f) + C_m h_2^m. \tag{2.8}$$

Прирівнюючи праві частини цих наближених рівностей, після кількох перетворень можна отримати

$$C_m \approx \frac{S_{h_2}(f) - S_{h_1}(f)}{h_1^m - h_2^m}.$$

Підставляючи це в (2.8), отримуємо

$$I(f) \approx S_{h_2}(f) + \frac{S_{h_2}(f) - S_{h_1}(f)}{\lambda^m - 1}. \tag{2.9}$$

Отриману наближену формулу тепер можна перетворити на апостеріорний оцінювач похибки чисельного інтегрування (апостеріорний оцінювач Рунге):

$$e(f) = \frac{S_{h_2}(f) - S_{h_1}(f)}{\lambda^m - 1} \tag{2.10}$$

Тепер, коли треба обчислити інтеграл

$$I(f) = \int_a^b f(x)dx$$

з заданою точністю ε, користуючись (2.10), слід обчислити значення квадратурних формул $S_{h_1}(f)$ та $S_{h_2}(f)$. Якщо виконується умова

$$|e(f)| = \left| \frac{S_{h_2}(f) - S_{h_1}(f)}{\lambda^m - 1} \right| < \varepsilon, \tag{2.11}$$

то для остаточного наближенного значення інтеграла користуємось формулою (2.9). В іншому випадку переходимо до наступної ітерації, де кладемо $h_1 = h_2$, $S_{h_1}(f) = S_{h_2}(f)$, $h_2 = h_1/\lambda$ та знову знаходимо $S_{h_2}(f)$, після чого перевіряємо умову (2.11).

2.3 Чисельні методи дробового інтегрування

Розглянемо ефективні алгоритмічні підходи для обчислення дробових інтегралів типу Рімана-Ліувілля.

Оператор дробового інтегрування посідає значне місце в теорії дробового числення, зокрема, він використовується для перетворення дробових диференціальних рівнянь в інтегральні рівняння зі слабко сингулярним ядром. У цьому розділі представлено числові методики, що застосовуються для наближеного обчислення дробових інтегралів і ґрунтуються на поліноміальній інтерполяції.

Припустимо, що функція $f(t)$ є неперервною на відрізку $I = [0, T]$, тобто $f(t) \in C(I)$. Нехай Δt позначає крок дискретизації, де $\Delta t = T/n_T$, $n_T \in N$, а $t_k = k\Delta t$ — вузли сітки. Далі ми зосередимося на методах чисельного обчислення наступного інтеграла:

$$J^{\alpha}_{0,t} f(t) = \frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} f(s) ds, \quad \alpha > 0. \tag{2.12}$$

Один із підходів до чисельного розрахунку (2.12) полягає в заміні функції $f(t)$ деякою апроксимуючою функцією $\widetilde{f}(t)$ такою, щоб інтеграл $J^{\alpha}_{0,t}\widetilde{f}(t)$ можна було легко обчислити аналітично. Логічним вибором є поліноміальна апроксимація функції $f(t)$ на інтервалі $[0,T]$. Інтеграл $J^{\alpha}_{0,t}\widetilde{f}(t)$ можна обчислити точно, якщо $\widetilde{f}(t)$ є багаточленом. Для точки $t = t_n$, $n \in N$, перепишемо вираз $\left[J^{\alpha}_{0,t} f(t)\right]_{t=t_n}$ у такій формі:

$$\begin{aligned} \left[J^{\alpha}_{0,t} f(t)\right]_{t=t_n} &= \tfrac{1}{\Gamma(\alpha)} \int_0^{t_n} (t_n - s)^{\alpha-1} f(s) ds \\ &= \tfrac{1}{\Gamma(\alpha)} \sum_{k=0}^{n-1} \int_{t_k}^{t_{k+1}} (t_n - s)^{\alpha-1} f(s) ds. \end{aligned} \tag{2.13}$$

Далі ми визначимо чисельні методи, що базуються на поліноміальній інтерполяції, щоб обчислити вираз (2.13).

У цьому розділі ми поширимо ідеї чисельних методів для класичних інтегралів на випадок дробових інтегралів.

Дробова формула прямокутників

На кожному з підінтервалів $[t_k, t_{k+1}]$, де $k = 0,1,\ldots,n-1$, функція $f(t)$ апроксимується сталою величиною, а саме:

$$f(t)|_{[t_k,t_{k+1})} \approx \widetilde{f}(t)|_{[t_k,t_{k+1})} = f(t_k), \tag{2.14}$$

Тоді матимемо:

$$\begin{aligned}\left[J^{\alpha}_{0,t} f(t)\right]_{t=t_n} &= \tfrac{1}{\Gamma(\alpha)} \sum_{k=0}^{n-1} \int_{t_k}^{t_{k+1}} (t_n - s)^{\alpha-1} f(s) ds \\ &\approx \tfrac{1}{\Gamma(\alpha)} \sum_{k=0}^{n-1} \int_{t_k}^{t_{k+1}} (t_n - s)^{\alpha-1} f(t_k) ds \qquad (2.15) \\ &= \sum_{k=0}^{n-1} b_{n-k-1} f(t_k) ds,\end{aligned}$$

де вагові коефіцієнти b_k визначаються як:

$$b_k = \frac{\Delta t^{\alpha}}{\Gamma(\alpha+1)} \left[(k+1)^{\alpha} - k^{\alpha}\right]. \tag{2.16}$$

Отже, отримуємо наближену формулу:

$$\left[J^{\alpha}_{0,t} f(t)\right]_{t=t_n} \approx \sum_{k=0}^{n-1} b_{n-k-1} f(t_k). \tag{2.17}$$

Аналогічно з класичною формулою лівих прямокутників, вираз (2.17) називається дробовою формулою лівих прямокутників. Аналогічно, якщо апроксимувати функцію як:

$$f(t)|_{(t_k,t_{k+1}]} \approx \widetilde{f}(t)|_{(t_k,t_{k+1}]} = f(t_{k+1}), \tag{2.18}$$

то маємо дробову формулу правих прямокутників:

$$\left[J^{\alpha}_{0,t}f(t)\right]_{t=t_n} \approx \sum_{k=0}^{n-1} b_{n-k-1} f(t_{k+1}). \tag{2.19}$$

Формули (2.18) та (2.19) — це частинні випадки дробової зваженої формули прямокутників:

$$\left[J^{\alpha}_{0,t}f(t)\right]_{t=t_n} \approx \sum_{k=0}^{n-1} b_{n-k-1}(\theta f(t_k) + (1-\theta) f(t_{k+1})), \quad 0 \le \theta \le 1. \tag{2.20}$$

Також отримаємо формулу:

$$\left[J^{\alpha}_{0,t}f(t)\right]_{t=t_n} \approx \sum_{k=0}^{n-1} b_{n-k-1} f(t_k + (1-\theta)\Delta t), \quad 0 \le \theta \le 1. \tag{2.21}$$

Якщо у формулі (2.20) (або (2.21)) покласти $\theta = 1$ (або $\theta = 0$), то відновиться дробова формула лівих прямокутників (2.17) (або дробова формула правих прямокутників (2.19)). Якщо ж у формулі (2.20) (або (2.21)) взяти $\alpha = 1$ та $\theta = 1/2$, то ця формула (2.20) (або (2.21)) перетвориться на складену формулу трапецій для класичного інтеграла.

Дробова формула трапеції

Апроксимуючи функцію $f(t)$ на кожному підінтервалі $[t_k, t_{k+1}]$ лінійним многочленом:

$$f(t)|_{[t_k,t_{k+1}]} \approx \widetilde{f}(t)|_{[t_k,t_{k+1}]} = \frac{t_{k+1}-t}{t_{k+1}-t_k} f(t_k) + \frac{t-t_k}{t_{k+1}-t_k} f(t_{k+1}), \tag{2.22}$$

отримуємо дробову формулу трапецій:

$$\left[J^{\alpha}_{0,t}f(t)\right]_{t=t_n} \approx \left[J^{\alpha}_{0,t}\widetilde{f}(t)\right]_{t=t_n}$$

$$= \frac{1}{\Gamma(\alpha)} \sum_{k=0}^{n-1} \int_{t_k}^{t_{k+1}} (t_n - t)^{\alpha-1} \left(\frac{t_{k+1}-t}{t_{k+1}-t_k} f(t_k) + \frac{t-t_k}{t_{k+1}-t_k} f(t_{k+1}) \right) dt \tag{2.23}$$

$$= \sum_{k=0}^{n} a_{k,n} f(t_k),$$

тут

$$a_{k,n} = \frac{1}{\Gamma(\alpha)} \begin{cases} \int\limits_0^{t_1} (t_n - t)^{\alpha-1} \frac{t_1 - t}{t_1 - t_0} dt, & k = 0, \\ \int\limits_{t_k}^{t_{k+1}} (t_n - t)^{\alpha-1} \frac{t_{k+1} - t}{t_{k+1} - t_k} dt + \int\limits_{t_{k-1}}^{t_k} (t_n - t)^{\alpha-1} \frac{t - t_{k-1}}{t_k - t_{k-1}} dt, & 1 \le k \le n-1, \\ \int\limits_{t_{n-1}}^{t_n} (t_n - t)^{\alpha-1} \frac{t - t_{n-1}}{t_n - t_{n-1}} dt, & k = n. \end{cases} \tag{2.24}$$

Звідси отримуємо

$$a_{k,n} = \frac{\Delta t^\alpha}{\Gamma(\alpha+2)} \begin{cases} (n-1)^{\alpha+1} - (n-1-\alpha)n^\alpha, & k = 0, \\ (n-k+1)^{\alpha+1} + (n-1-k)^{\alpha+1} - 2(n-k)^{\alpha+1}, & 1 \le k \le n-1, \\ 1, & k = n. \end{cases} \tag{2.25}$$

РОЗДІЛ 3

OPENMP: ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ ДЛЯ БАГАТОЯДЕРНИХ СИСТЕМ

3.1 Призначення технології OpenMP

OpenMP являє собою інтерфейс прикладного програмування (API), призначений для створення багатопотокових застосунків, які здебільшого орієнтовані на паралельні обчислювальні системи, що характеризуються спільною пам'яттю. Цей інтерфейс включає набір компіляторних директив та бібліотеки спеціальних функцій. Розробка стандартів OpenMP для архітектур із загальною пам'яттю тривала впродовж останніх п'ятнадцяти років. У новітній період активно ведеться робота над розширенням цих стандартів для забезпечення підтримки паралельних обчислювальних систем з розподіленою пам'яттю.

3.2 Розпаралелювання потоків в OpenMP

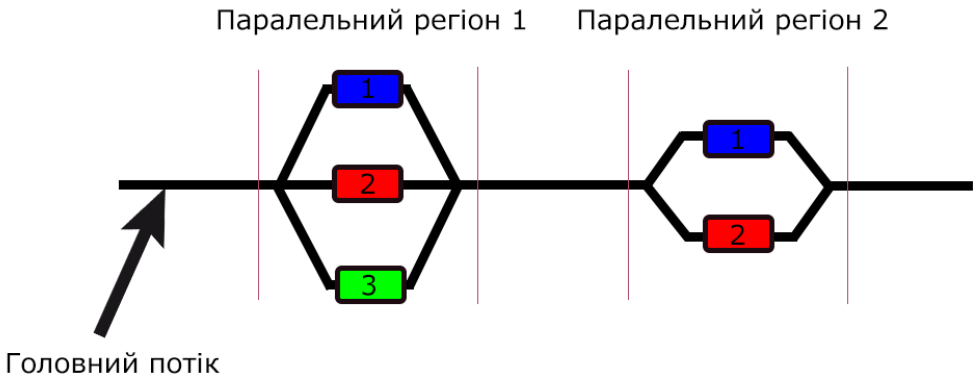

Рис. 3.1. Схема розгалуження паралельної програми

Найбільш важливою частиною OpenMP є паралельні регіони. Розробник може сам обирати та налаштовувати їх під свою задачу. Що таке паралельний регіон? Це регіон коду, який буде виконуватися декількома потоками, залежно від налаштування, та через це підвищувати швидкість

його виконання. Мови програмування, які підтримує OpenMP - це C, C++ та Fortran. Налаштування паралельного регіону відбувається за допомогою параметрів (clauses). Приклади параметрів: `private()`, `shared()`, `for`, `reduction()`, `num_threads()`, тощо.

`omp_get_thread_num()` повертає номер потоку, який виконує цю команду. Номери потоків - цілі числа від 0 до n-1, якщо n - це кількість потоків.

`omp_get_num_threads()` повертає кількість потоків, які існують в момент виклику цієї команди.

Паралельні області

OpenMP використовує "fork-join"модель. Ця модель працює наступним чином: спочатку існує один master thread, який виконує код поступово. Коли цей потік зустрічає паралельний регіон, він створює кілька додаткових потоків для його виконання. Після завершення паралельного регіону всі потоки, окрім головного, завершують свою роботу.

Приклад

```
#pragma omp parallel
{   // Code to execute in parallel
    printf("Hello from thread %d\n", omp_get_thread_num());
}
```

Спільні та приватні змінні

В OpenMP змінні можуть бути спільними або приватними. Всі потоки мають доступ до спільних змінних. Кожен потік має власний екземпляр приватних змінних.

Приклад

```
int shared_var = 0;
int private_var = 0;
#pragma omp parallel private(private_var)
{   // Code with shared and private variables
    shared_var++;
    private_var++;
}
printf("shared: %d, private: %d\n", shared_var, private_var);
// Out: shared: 6, private: 0
```

Конструкції розподілу роботи

За допомогою цих конструкцій можна розподіляти завдання між потоками.

Паралелізація циклів `for`

Для того, щоб поділити цикл `for` на паралельні потоки, необхідно створити паралельний регіон та прописати параметр `for`. Параметр `num_threads()` дозволяє обрати певну кількість потоків для виконання циклу. Якщо не передати параметр `num_threads()`, цикл розподілиться на всі ядра процесору. Можна ділити тільки цикли в яких ітерації не залежать одна від одної і для яких загальна кількість ітерацій відома заздалегідь.

Приклад

```
int x = 0;
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
    // Loop iterations are divided among threads
    // Using all cores available
    x += i*i;
}
printf("x = %d", x);
```

```
// Out: x = 285
```

Розділи (Sections)

Кожний розділ - це окрема задача, яку виконує певний потік, причому розділи виконуються паралельно. Це можна порівняти з тим, як цикли `for` поділяються на паралельні потоки.

Приклад

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // Task 1: Compute sum of the array
        for (int i = 0; i < n; i++) {
            sum += a[i];
        }
        printf("Sum: %d\n", sum);
    }

    #pragma omp section
    {
        // Task 2: Find max value in the array
        for (int i = 0; i < n; i++) {
            if (a[i] > max_val) {
                max_val = a[i];
            }
        }
        printf("Max value: %d\n", max_val);
    }
}
```

Операції Злиття/Редукції (Reduction Operations)

Злиття комбінує результати з декількох потоків в один результат. Параметр `reduction` працює для таких конструкцій як `parallel`, `sections` та `for`. Для того, щоб користуватись злиттям, потрібно обрати одну з операцій (`+`, `-`, `*`, `/`, `&`, `^`, `|`, `<<`, `>>`) та одну або декілька спільних змінних, до яких треба застосувати цю операцію й передати це параметром як `reduction(op: var)`. Для того, щоб продемонструвати, як цим можна користуватись, ось приклад:

Приклад

```
int s = 5;
#pragma omp parallel for reduction(+:s)
for (int i = 1; i < 4; i++)
{
    s += i*i;
    printf("s = %d\n", s);
}
printf("reducted s = %d\n", s);

// Out: s = 1
//      s = 9
//      s = 4
//      reducted s = 19
```

РОЗДІЛ 4

CUDA: ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ ДЛЯ БАГАТОЯДЕРНИХ СИСТЕМ

4.1 Графічні процесори, CUDA та їхня структура

Графічні процесори (GPU – Graphics Processing Units) розроблялися та застосовувалися для того, щоб обробляти графічні дані. Але сьогодні навіть більш актуальним варіантом використання графічних процесорів є розпаралелювання завдань. Часто застосування GPU для цих задач позначається як "GPGPU General Purpose GPU, тобто графічний процесор загального призначення.

Спочатку було дуже важко проводити симуляції та звичайні розрахунки на графічних процесорах - на той момент не існувало середовищ програмування як CUDA та OpenCL, але були тільки такі, як DirectX та OpenGL - вони створені майже виключно для роботи з графічними задачами. Більш нові середовища: CUDA (Compute Unified Device Architecture) та OpenCL (Open Computing Language) надали можливість розширити функціонал графічних процесорів. З їхньою допомогою можна розпаралелювати звичайні задачі, але про це трошки пізніше.

CUDA являє собою більш універсальну платформу для паралельних обчислень, що підтримується новими поколіннями графічних процесорів NVIDIA, починаючи з 2007 року. Також її можна емулювати на центральних процесорах (CPU).

В графічних застосунках завжди був великий потенціал для паралельних обчислень. Це призвело до того, що графічні процесори значно раніше почали використовувати багато процесорних ядер на відміну від центральних процесорів. Через це сьогодні саме GPU використовуються для паралельних обчислень. Один графічний процесор може мати тисячі обчислювальних ядер, що набагато більше, ніж в CPU.

Графічні процесори існують двох базових типів - дискретні та інтегровані. Інтегровані графічні процесори (iGPU) розміщуються безпосередньо на кристалі центрального процесора у вигляді окремого модуля. Вони не мають власної виділеної пам'яті, а використовують спільну пам'ять із центральним процесором. Такий підхід спрощує обмін даними між CPU та iGPU. Дискретні графічні процесори реалізовані на окремій мікросхемі. Відповідно, вони оснащені власною пам'яттю, незалежною від пам'яті CPU. За таких умов, дані для обробки на GPU необхідно попередньо копіювати з оперативної пам'яті CPU до пам'яті GPU, що може призводити до значних затримок, особливо під час роботи з великими обсягами даних.

Середовища CUDA та OpenCL структурують програму, розділяючи її на **програму для центрального процесора (хост-програму)**, що виконує операції введення/виведення та взаємодію з користувачем, та **програму для графічного процесора (програму пристрою)**, яка реалізує обчислення, призначені для виконання на GPU. У найпростішому сценарії взаємодія між хост-програмою та програмою пристрою відбувається наступним чином: хост-програма спершу передає дані до глобальної пам'яті GPU, після чого викликає функції пристрою для запуску обчислень на графічному процесорі. Функції пристрою розроблені з урахуванням паралельної архітектури GPU для ефективного розподілу паралельних операцій з даними між обчислювальними ядрами.

До складу графічного процесора входить декілька багатопотокових SIMD-процесорів (Single Instruction, Multiple Data), кожен з яких виконує незалежні потоки обчислювальних команд. Компанія NVIDIA позначає такі SIMD-процесори терміном "потокові мультипроцесори"(SM – Streaming Multiprocessors). Реальна кількість SM, інтегрованих в GPU, варіюється залежно від конкретної моделі графічного процесора. Кожен SIMD-процесор містить декілька функціональних блоків SIMD (FU – Function Units), здатних виконувати одну й ту ж SIMD-інструкцію над різними наборами даних. Оброблювані дані повинні знаходитися в локальних регістрах, асоційованих з відповідним функціональним блоком SIMD. Для запуску процесу переміщення даних з пам'яті GPU до регістрів існують спеціалізовані операції передачі.

Сама передача може тривати декілька машинних тактів, адже дані можуть розташовуватися в глобальній пам'яті GPU, звідки їх необхідно завантажити. Останні архітектури GPU містять ієрархію кеш-пам'яті, завдяки чому окремі операції передачі даних можуть обробляти кешовану інформацію, що значно прискорює доступ порівняно з некешованими даними. Слід зазначити, що для одного SIMD-процесора недостатньо одного потоку управління (SIMD-потоку, або "варпа"в термінології NVIDIA). Паралелізм досягається за рахунок виконання кожним функціональним блоком SIMD тієї самої інструкції над різними даними, **однак операції передачі даних спричинять час очікування невизначеної тривалості.** Щоб приховати час очікування, процесори SIMD можуть виконувати кілька незалежних потоків SIMD. Планувальник потоків SIMD (Warp Scheduler) вибирає потік SIMD, готовий до виконання, і ініціює виконання наступної інструкції SIMD цього потоку. Кожен з потоків SIMD використовує незалежний набір регістрів. На кожному етапі виконання планувальник SIMD-потоків може обрати інший SIMD-потік, оскільки ці потоки є взаємно незалежними. Для забезпечення цього вибору планувальник SIMD-потоків застосовує механізм табло (Scoreboard). Для кожного SIMD-потоку табло зберігає поточну інструкцію для виконання та відомості про наявність її операндів у регістрах. Максимально можлива кількість підтримуваних SIMD-потоків визначається розміром табло. Розмір табло є характеристикою архітектури, і його типове значення – 32. Реальна кількість незалежних SIMD-потоків задається прикладною програмою.

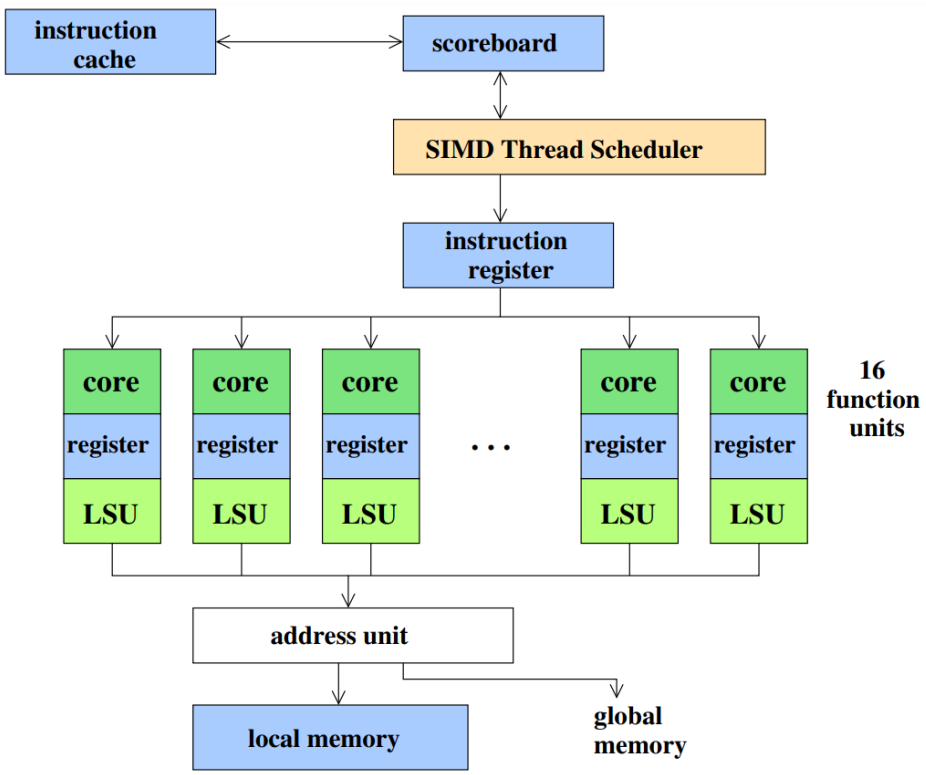

Рис. 4.1. Блок-схема SIMD-процесора. SIMD-процесор має 16 функціональних блоків (FU), кожен FU має свій набір регістрів і Load-Store-Unit (LSU)

4.2 Створення паралельних додатків за допомогою технології CUDA

4.2.1 Загальна архітектура програми

Загальна архітектура програми CUDA передбачає наявність етапів, що виконуються або на хост-системі, або на одному з обчислювальних пристроїв. Хост-система (host) представлена традиційним центральним процесором (CPU), а програмні компоненти, призначені для хоста, є програмами мовою C, які компілюються за допомогою стандартних C-компіляторів. Обчислювальні пристрої (devices) – це масивно-паралельні процесори, такі як графічні процесори, що володіють значною кількістю виконавчих блоків і призначені для обробки великих обсягів даних на паралельних програмних етапах. Код для пристрою розробляється мовою C з використанням спеціальних розширень CUDA, які дають змогу описувати паралельну обробку даних.

Функція, що реалізує паралельну обробку даних і виконується на пристрої, має назву функція ядра або просто ядро (Kernel). Паралелізм у програмі CUDA реалізується саме через функції ядра, які, як правило, породжують багато потоків CUDA. Ці потоки є "легковагими"і потребують лише кількох тактових циклів для свого створення та планування, на відміну від потоків CPU, для яких аналогічні операції вимагають тисяч циклів.

Програма CUDA, зазвичай збережена у файлі з розширенням *.cu, містить як головну програму, так і функції ядра. Вона обробляється компілятором NVIDIA C (nvcc), який здійснює розділення цих двох компонентів. Функції ядра транслюються у проміжний асемблерний код PTX (Parallel Thread Execution). PTX є низькорівневою мовою, розробленою NVIDIA, яка, подібно до асемблера x86, визначає набір інструкцій, що забезпечують сумісність між різними поколіннями графічних процесорів NVIDIA. Виклики функцій ядра з головної програми перетворюються на системні виклики середовища виконання CUDA для запуску відповідної функції на GPU. Після цього головна програма компілюється стандартним C-компілятором.

Виконання програми CUDA розпочинається із запуску головної програми, яка, у свою чергу, викликає функції ядра для здійснення паралельних обчислень на GPU. Кожен виклик функції ядра ініціює створення сукупності потоків CUDA, відомої як сітка потоків (grid). Робота сітки завершується, коли всі потоки, що входять до її складу, виконали свою частину функції ядра. Після ініціації виклику функції ядра, центральний процесор продовжує виконання головної програми, що може включати виклики інших функцій ядра.

4.2.2 Програмування за допомогою технології CUDA

CUDA доповнює стандартний синтаксис оголошення функцій мови C для розрізнення між функціями хоста та функціями ядра. Спеціальне ключове слово CUDA `__global__` позначає, що функція є ядром, яке може бути викликане з хост-функції для виконання на GPU. Ключове слово `__device__` вказує, що функція є ядром (або функцією пристрою), призначеним для виклику з іншого ядра або функції пристрою. У таких функціях пристрою рекурсивні виклики та непрямі виклики через вказівники на

функції не допускаються. Функція хоста оголошується за допомогою ключового слова `__host__` і є стандартною функцією C, що виконується на хості та може бути викликана лише іншою хост-функцією. За замовчуванням, усі функції, оголошені без спеціальних ключових слів, вважаються хост-функціями.

Для виконання функції ядра на пристрої, необхідні дані мають бути розміщені в пам'яті цього пристрою. Відповідно, програма CUDA зазвичай включає операції передачі даних з пам'яті хоста до пам'яті GPU, а також з пам'яті GPU до пам'яті CPU для повернення результатів обчислень на хост. Ці операції передачі даних явно програмуються в CUDA-застосунках за допомогою спеціалізованих функцій. Перш ніж здійснювати передачу даних з хоста до пам'яті графічного процесора, необхідно зарезервувати відповідний обсяг пам'яті на пристрої. Це виконується за допомогою функції:

```
cudaMalloc(void **, size_bytes)
```

яка викликається з головної програми для виділення простору в глобальній пам'яті GPU. Функція `cudaMalloc()` приймає два аргументи: перший – це вказівник на вказівник, що вказуватиме на виділену ділянку пам'яті, а другий – розмір необхідної пам'яті в байтах. Функція:

```
cudaFree(void *)
```

використовується для звільнення пам'яті, що була виділена для об'єктів даних (на які вказує її параметр), після завершення обчислень. Запит на передачу даних, наприклад, від хоста до GPU, реалізується через виклик функції копіювання:

```
cudaMemcpy(void *, const void *, size_bytes, enum cudaMemcpyKind)
```

з параметрами: `void *` - вказівник на місце призначення операції копіювання, `const void *` - вказівник на джерело даних для копіювання, `size_bytes` - кількість байтів для передачі, `enum cudaMemcpyKind` - тип передачі: `cudaMemcpyHostToDevice` для копіювання з хоста на пристрій та `cudaMemcpyDeviceToHost` для копіювання з пристрою на хост

Після того, як дані передано до глобальної пам'яті графічного процесора, хост-програма може ініціювати виклик функції ядра GPU, яка буде оперувати цими даними. Виклик функції ядра супроводжується конфі-

гурацією виконання, що визначає структуру сітки потоків, які необхідно створити для виконання цієї функції.

Потоки в межах сітки організовані за дворівневою ієрархією. На першому рівні сітка складається з певної кількості блоків потоків, причому кожен блок містить однакову кількість потоків. Другий рівень представляє організацію потоків всередині кожного окремого блоку, яка є однаковою для всіх блоків у межах однієї сітки.

Блоки, що формують сітку, можуть мати двовимірну (або тривимірну, починаючи з CUDA версії 3) структуру. Кожен блок потоків ідентифікується унікальними двовимірними (або тривимірними) координатами, які надаються через вбудовані змінні CUDA: `blockIdx.x`, `blockIdx.y` (та `blockIdx.z`). Потоки всередині блоку, своєю чергою, також організовані у тривимірну структуру, і унікальні тривимірні координати кожного потоку доступні через змінні CUDA: `threadIdx.x`, `threadIdx.y` та `threadIdx.z`. Максимально допустима кількість потоків у блоці становить 512 (для версій до 2.x) та 1024 (починаючи з версії 3.x). Відповідно до цієї дворівневої ієрархічної моделі, кожен потік у сітці однозначно ідентифікується координатами блоку, до якого він належить, та своїми власними координатами всередині цього блоку. Ці координатні значення можуть використовуватися у функціях ядра для диференціації паралельних потоків, що забезпечує паралельну обробку даних.

Розміри сітки потоків та блоків потоків, які генеруються для виконання конкретного виклику ядра, задаються у конфігурації виконання. Для визначення конфігурації виконання оголошуються дві змінні структурного типу `dim3`. Тип `dim3` є цілочисельним векторним типом, похідним від векторного типу `uint3`, і за замовчуванням ініціалізується значенням (1,1,1), якщо не вказано інше. Ці параметри описують дво- або тривимірну організацію сітки потоків та блоків і включаються в синтаксис виклику ядра, обрамлені символами <<< та >>>, як у наступному прикладі:

```
dim3 grid_size(gx, gy);
dim3 block_size(bx, by, bz);
Kernel<<<grid_size, block_size>>>(...);
```

У наведеному прикладі двовимірна структура сітки для виконання ядра

`Kernel` визначається змінною `grid_size` і має розмірність `gx` ×`gy`. Тривимірна структура блоку визначається змінною `block_size` і має розмірність `bx` ×`by` ×`bz`.

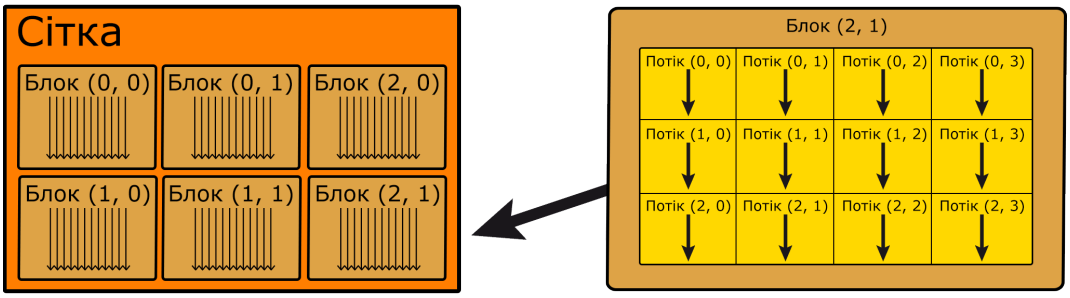

Рис. 4.2. Ілюстрація конфігурації виконання CUDA для `grid_size(3, 2)` та `block_size(4, 3)`.

Розмір сітки та її блоків будуть знаходитися в змінних `gridDim` та `blockDim`. Наприклад, `blockDim.y` буде зберігати `by` в цьому випадку. Якщо для задачі достатньо одновимірної сітки або блоків, можна замість змінної типу `dim3` передати число: `Kernel<<<5, 10>>>` - в цьому прикладі `5` буде означати змінну типу `dim3` зі значенням `(5, 1, 1)`

Необхідно ще зазначити функцію CUDA `__syncthreads()`. Це називається операцією бар'єрної синхронізації. Ця операція буквально виступає в якості бар'єру для потоків - потоки будуть чекати, доки останній потік дійде до цього моменту. **Бар'єр дозволяє потокам йти далі по алгоритму тільки якщо всі потоки його досягли.** Синхронизація потоків цим або іншим чином є необхідною частиною паралелизації алгоритмів, це можна побачити в реалізації алгоритму в наступній секції.

РОЗДІЛ 5

ПАРАЛЕЛЬНИЙ ДОДАТОК ІНТЕГРУВАННЯ ДЛЯ CPU З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ OPENMP

5.1 Функція calc_intg_cpu

Функція `calc_intg_cpu` створена для паралельного (або не паралельного) рахування інтегралу заданої функції f на відрізку $[a, b]$ за допомогою складеної квадратурної формули трапецій S_h (2.6) із заданою кількістю проміжків n з використанням технології OpenMP - на CPU

```
double calc_intg_cpu(
    bool use_parallel = true,
    long long unsigned int n = 1000000000,
    double (*f)(double) = host_bessel_j1,
    long double a_ld = 0.0,
    long double b_ld = 50.0) {
```

Алгоритм складеної квадратурної формули трапецій:

```
for (i = 1; i <= n - 1; i++) {
    double x_val = a + (double)i * h;
    sum += f(x_val);
}

double intg = (h / 2.0) * (f(a) + f(b) + 2.0 * sum);
return intg;
```

Це обертається в `if-else`, щоб вмикати паралелізацію:

```
if (use_parallel) {
    #pragma omp parallel for private(x) reduction(+:sum)
    // <Алгоритм складеної квадратурної формули трапецій> //
}
else // <Той самий алгоритм, але без паралелизації> //
double intg = (h / 2.0) * (f(a) + f(b) + 2.0 * sum);
```

```
return intg;
```

5.2 Функція calc_intg_final

Створюється функція, яка буде рахувати цей інтеграл доки не буде виконана умова (2.11)

```
enum IntegrationMode { CPU_SINGLE_THREAD, CPU_PARALLEL }
double calc_intg_final(
    IntegrationMode mode,
    double epsilon = 10e-15,
    long long unsigned int n_initial = 1,
    int lambda = 2,
    int m = 2) {
```

Алгоритм, який подвоює кількість відрізків, доки $e(f)$ не буде менше ніж ϵ

```
Sh1 = calc_intg_cpu(mode == CPU_PARALLEL, current_n);
current_n *= lambda;
Sh2 = calc_intg_cpu(mode == CPU_PARALLEL, current_n);
double ef = (Sh2 - Sh1) / (pow(lambda, m) - 1.0);
while (fabs(ef) > epsilon) {
    current_n *= lambda;
    Sh1 = Sh2;
    Sh2 = calc_intg(mode == CPU_PARALLEL, current_n);
    ef = (Sh2 - Sh1) / (pow(lambda, m) - 1.0);
}
double ans = Sh2 + ef;
return(ans);
```

5.3 Результати обчислювального експерименту

Обчислювальний експеримент проводився з використанням хмарних обчислень за допомогою платформи Kaggle. Платформа Kaggle дозволяє проводити обчислення за допомогою віртуального CPU з двома фізичними та чотирма логічними ядрами.

Записник Kaggle дозволяє створювати програми на мовах C/C++ та компілювати їх за допомогою компілятора nvcc фірми NVIDIA.

Результати обчислювального експерименту наведено в таблицях 5.1, 5.2.

Табл. 5.1. Результати обчислювального експерименту з CPU (1 потік)

Iter	N	Integral Value	Iter Time	Error Estimate (ef)
1	100	0.934946887111277	0.000043	N/A
2	200	0.941882639391906	0.000040	2.31e-03
3	400	0.943611735371371	0.000045	5.76e-04
4	800	0.944043708148307	0.000085	1.44e-04
5	1600	0.944151682538374	0.000140	3.60e-05
6	3200	0.944178674960980	0.000248	9.00e-06
7	6400	0.944185422993206	0.000449	2.25e-06
8	12800	0.944187109996670	0.000872	5.62e-07
9	25600	0.944187531747246	0.001723	1.41e-07
10	51200	0.944187637184871	0.003524	3.51e-08
11	102400	0.944187663544288	0.006901	8.79e-09
12	204800	0.944187670134115	0.014071	2.20e-09
13	409600	0.944187671781594	0.030961	5.49e-10
14	819200	0.944187672193468	0.056196	1.37e-10
15	1638400	0.944187672296485	0.114238	3.43e-11
16	3276800	0.944187672322015	0.229290	8.51e-12
17	6553600	0.944187672328720	0.452668	2.23e-12
18	13107200	0.944187672330023	0.880351	4.34e-13
19	26214400	0.944187672330761	1.778822	2.46e-13
20	52428800	0.944187672330773	3.524031	3.74e-15

Total Time: 7.095037 seconds

Табл. 5.2. Результати обчислювального експерименту з CPU (4 потоки)

Iter	N	Integral Value	Iter Time	Error Estimate (ef)
1	100	0.934946887111277	0.000297	N/A
2	200	0.941882639391906	0.000010	2.31e-03
3	400	0.943611735371372	0.000014	5.76e-04
4	800	0.944043708148305	0.000024	1.44e-04
5	1600	0.944151682538376	0.000052	3.60e-05
6	3200	0.944178674960978	0.000089	9.00e-06
7	6400	0.944185422993203	0.000176	2.25e-06
8	12800	0.944187109996671	0.000365	5.62e-07
9	25600	0.944187531747238	0.000705	1.41e-07
10	51200	0.944187637184871	0.001463	3.51e-08
11	102400	0.944187663544274	0.002900	8.79e-09
12	204800	0.944187670134124	0.005797	2.20e-09
13	409600	0.944187671781591	0.011437	5.49e-10
14	819200	0.944187672193481	0.024698	1.37e-10
15	1638400	0.944187672296464	0.053177	3.43e-11
16	3276800	0.944187672322130	0.098398	8.56e-12
17	6553600	0.944187672328574	0.192950	2.15e-12
18	13107200	0.944187672330247	0.379116	5.58e-13
19	26214400	0.944187672330712	0.762761	1.55e-13
20	52428800	0.944187672330580	1.503404	-4.40e-14

Total Time: 3.038255 seconds

В таблиці 5.1 наведені результати обчислень інтеграла

$$I = \int_0^{50} J_1(x)dx, \tag{5.1}$$

де $J_1(x)$ — функція Бесселя першого роду, послідовним алгоритмом. Точність обчислення інтеграла визначалась за принципом Рунге. Інтеграл обчислено з точністю 5×10^{-14} за 20 кроків. В таблиці 5.2 наведені результати обчислень інтеграла (5.1) паралельним алгоритмом з використанням чотирьох потоків. Загальний час обчислення паралельним алгоритмом в 2.3 рази менший.

РОЗДІЛ 6

ПАРАЛЕЛЬНИЙ ДОДАТОК ІНТЕГРУВАННЯ ДЛЯ GPU З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ CUDA

6.1 Функція ядра integrate_kernel

Аналогічно реалізовано паралельний алгоритм обчислення інтегралу для функції f на відрізку $[a, b]$ за допомогою складеної квадратурної формули трапецій (2.6) для GPU з використанням технології CUDA.

```
__global__ void integrate_kernel(double a, double h,
long long int num_sum_terms, double* d_block_sums) {

    __shared__ double s_cache[THREADS_PER_BLOCK_INTEGRATE];

    int tid_in_block = threadIdx.x;
    int block_id = blockIdx.x;
    int threads_per_block_dim = blockDim.x;
    int total_threads_in_grid = gridDim.x * threads_per_block_dim;

    double my_sum = 0.0;

    for (long long int i =
         (long long int)block_id * threads_per_block_dim
         + tid_in_block + 1;
         i <= num_sum_terms;
         i += total_threads_in_grid) {
        double x = a + (double)i * h;
        my_sum += device_bessel_j1(x);
    }

    s_cache[tid_in_block] = my_sum;
    __syncthreads();
```

```
    for (int s = threads_per_block_dim / 2; s > 0; s >>= 1) {
        if (tid_in_block < s) {
            s_cache[tid_in_block] += s_cache[tid_in_block + s];
        }
        __syncthreads(); // Synchronize after each step of reduction
    }

    if (tid_in_block == 0) {
        d_block_sums[block_id] = s_cache[0];
    }
}
```

Це є функція ядра. Ця функція буде виконуватись кожним потоком на GPU. Кожен потік обчислює значення функції $J_1(x)$ у точці $x = a + ih$ у циклі, де a та h є константами, а i збільшується на загальну кількість потоків на кожній ітерації (у випадку, якщо потрібно додати більше членів, ніж загальна кількість потоків, інакше кожен потік виконає лише одне додавання), доки i не досягне загальної кількості членів, які потрібно додати. Потім результат додавання зберігається у списку `s_cache`, який є спільним для всіх потоків у блоці, кожен блок має свій власний `s_cache`. Усі потоки чекають, поки кожен потік завершить додавання. Далі відбувається редукція, результатом якої є `s_cache`, де нульовий елемент містить суму всіх елементів, що були в ньому раніше, всі інші елементи дорівнюють 0. Далі ця сума знову зберігається в іншому списку `d_block_sums`. Цей список передається параметром в ядро, тобто його можна використати за межами функції.

6.2 Функція calc_intg_cuda

Ця функція буде виконуватись на CPU та запускти ядро, яке було створено раніше.

```
double calc_intg_cuda(
    long long unsigned int n_intervals = 1000000000,
```

```
    double (*f_host)(double) = host_bessel_j1,
    long double a_ld = 0.0,
    long double b_ld = 50.0) {
```

Готується функція ядра

```
double h = (b - a) / n_intervals;
long long int num_sum_terms = n_intervals - 1;

if (num_sum_terms > 0) {
    int threads_per_block = THREADS_PER_BLOCK_INTEGRATE;
    int num_blocks = (num_sum_terms + threads_per_block - 1)
                     / threads_per_block;
    if (num_blocks > 2048) num_blocks = 2048;
    if (num_blocks == 0 && num_sum_terms > 0) num_blocks = 1;

    double *d_block_sums;
    CUDA_CHECK(cudaMalloc((void **)&d_block_sums,
                          num_blocks * sizeof(double)));
```

Запускається ядро

```
integrate_kernel<<<num_blocks, threads_per_block>>>
(a, h, num_sum_terms, d_block_sums);
CUDA_CHECK(cudaDeviceSynchronize());
```

Результати роботи GPU копіюються на CPU

```
double *h_block_sums = (double *)malloc(num_blocks * sizeof(double));
CUDA_CHECK(cudaMemcpy(h_block_sums, d_block_sums,
num_blocks * sizeof(double), cudaMemcpyDeviceToHost));
```

Це список сум результатів потоків для кожного блоку. Тепер для знаходження значення інтегралу необхідно знайти суму всіх елементів цього списку та використати функцію трапецій:

```
for (int i = 0; i < num_blocks; i++) {
    gpu_sum += h_block_sums[i];
}
(...)
```

```
double integral_val = (h / 2.0) *
                      (f_host(a) + f_host(b) + 2.0 * gpu_sum);
return integral_val;
```

6.3 Результати обчислювального експерименту

Табл. 6.1. Результати обчислювального експерименту з GPU

Iter	N	Integral Value	Iter Time	Error Estimate (ef)
1	100	0.934946887111166	0.137720	N/A
2	200	0.941882639391746	0.000300	2.31e-03
3	400	0.943611735371189	0.000288	5.76e-04
4	800	0.944043708148111	0.000274	1.44e-04
5	1600	0.944151682538177	0.000270	3.60e-05
6	3200	0.944178674960775	0.000266	9.00e-06
7	6400	0.944185422992997	0.000258	2.25e-06
8	12800	0.944187109996462	0.000284	5.62e-07
9	25600	0.944187531747042	0.000311	1.41e-07
10	51200	0.944187637184669	0.000349	3.51e-08
11	102400	0.944187663544076	0.000458	8.79e-09
12	204800	0.944187670133927	0.000622	2.20e-09
13	409600	0.944187671781387	0.001020	5.49e-10
14	819200	0.944187672193255	0.001648	1.37e-10
15	1638400	0.944187672296223	0.002893	3.43e-11
16	3276800	0.944187672321964	0.005431	8.58e-12
17	6553600	0.944187672328397	0.010532	2.14e-12
18	13107200	0.944187672330007	0.020731	5.37e-13
19	26214400	0.944187672330410	0.041039	1.34e-13
20	52428800	0.944187672330510	0.073243	3.33e-14

Total Time: 0.298297 seconds

В таблиці 6.1 наведені результати обчислень інтеграла (5.1) паралельним алгоритмом на GPU з використанням технології CUDA. Загальний час обчислення цим алгоритмом в 10 разів менший, порівняно з паралельним алгоритмом на CPU. Якщо не розглядати час, витрачений на створення контексту CUDA на першій ітерації, загальний час обчислення цим алгоритмом в 20 разів менший.

На рисунку 6.1 приведено порівняння часу виконання послідовного на паралельних алгоритмів на різних пристроях (CPU, CPU - OpenMP, GPU - CUDA)

На рисунку 6.2 приведено порівняння похибки ef послідовного на паралельних алгоритмів на різних пристроях (CPU, CPU - OpenMP, GPU - CUDA)

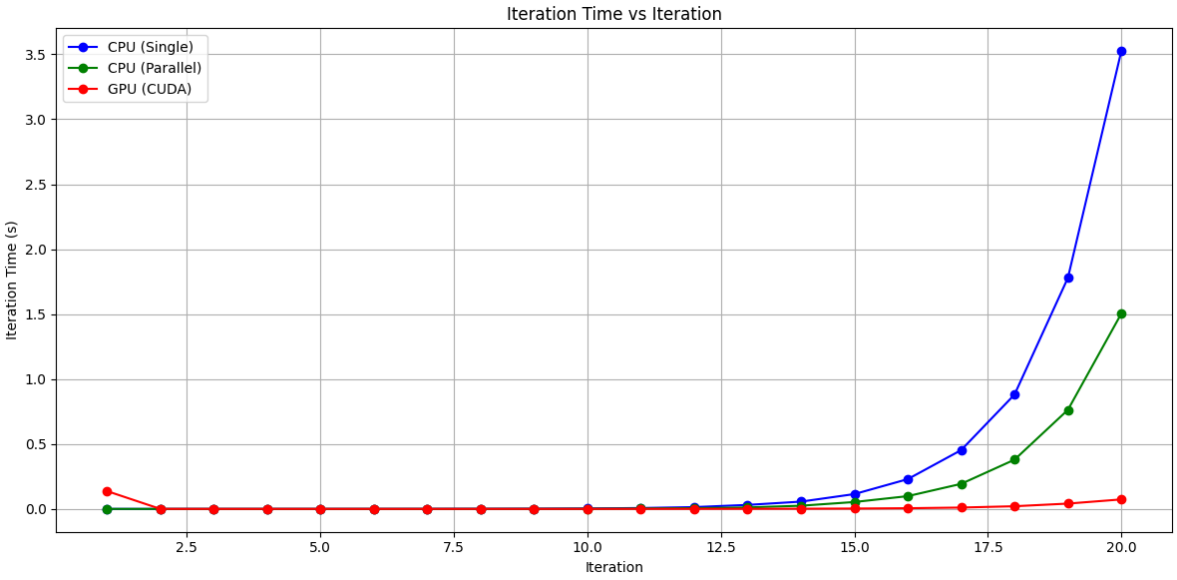

Рис. 6.1. Порівняння часу виконання послідовного на паралельних алгоритмів на різних пристроях (CPU, CPU - OpenMP, GPU - CUDA)

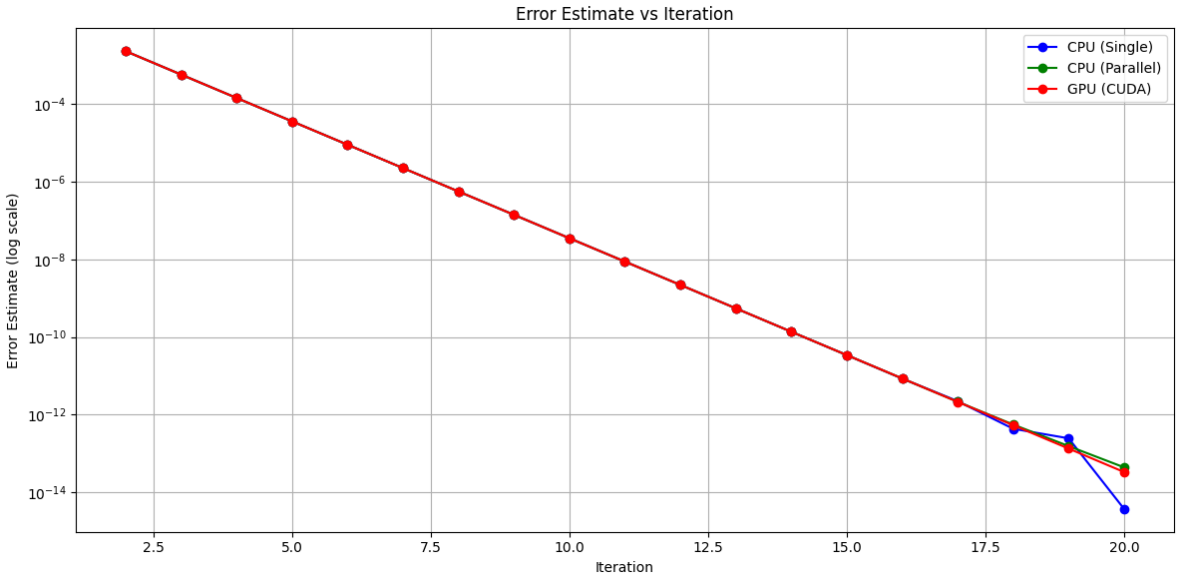

Рис. 6.2. Порівняння похибки ef послідовного на паралельних алгоритмів на різних пристроях (CPU, CPU - OpenMP, GPU - CUDA)

ВИСНОВКИ

В кваліфікаційній роботі були розроблені паралельні алгоритми обчислення визначеного інтегралу з використанням складеної квадратурної формули Ньютона-Котеса. Точність обчислень була контрольована за правилом Рунге.

Були розроблені паралельні алгоритми для комп'ютера з багатоядерним процесором та для комп'ютера з використанням обчислень на графічних процесорах.

Були проведені обчислювальні експерименти щодо встановлення ефективності паралельних реалізацій.

СПИСОК ЛІТЕРАТУРИ

1. Вербіцький В.В., Реут В.В. Введення в чисельні методи аналізу і диференціальних рівнянь: навчальний посібник. Одеса: Одеський національний університет імені І.І. Мечникова, 2018. 116 с.
2. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с.
3. Andersch M., Palmer G.,Krashinsky R., Stam N., Mehta V., Brito G., Ramaswamy S.. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopperarchitecture-in-depth/, 2022. Nvidia Developer.
4. Rauber T., Rünger G., Zenger C.. Parallel Programming: for Multicore and Cluster Systems. 3rd ed., Springer, 2023. 563 p.
5. Chapman B., Jost G., Ruud van der Pas. Using OpenMP: portable shared memory parallel programming. The MIT Press, 2007. 378 p.
6. Förster M.. Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP. Springer Vieweg © Springer Fachmedien Wiesbaden, 2014. 411 p.
7. OpenMP: Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-APISpecification-5.0.pdf
8. Ruud van der Pas, Stotzer E., and Terboven C.. Using OpenMP-The Next Step. Affinity, Accelerators, Tasking, and SIMD. The MIT Press, 2017. 381 p.
9. Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges. The OpenMP common core : making OpenMP simple again / Cambridge, Massachusetts : The MIT Press, 2019. 277 p.
10. T.M. Aamodt, W.W.L. Fung, and T.G. Rogers. General-Purpose Graphics Processor Architectures. Morgan & Claypool Publishers, 2018
11. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 4.0. www.mpi-forum.org, 2021.

12. T.G. Mattson, Y. He, and A.E. Koniges. The OpenMP Common Core. MIT Press, 2019.

ДОДАТОК А

Код програми

```
#include <stdio.h>
#include <math.h> // For host j1, pow, fabs
#include <stdlib.h> // For exit, EXIT_FAILURE
#include <time.h>   // For timing

// For OpenMP (CPU version)
#include "omp.h"

// For CUDA
#include <cuda_runtime.h>
#include <device_launch_parameters.h> // For threadIdx, blockIdx etc.

// CUDA error checking macro
#define CUDA_CHECK(err) { \
    if (err != cudaSuccess) { \
        fprintf(stderr, "CUDA Error: %s at %s:%d\n", \
        cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(EXIT_FAILURE); \
    } \
}

// --- Function to be integrated (Bessel function j1) ---
double host_bessel_j1(double x) {
    return j1(x);
}

__device__ double device_bessel_j1(double x) {
```

```
    return j1(x);
}


// --- CUDA Kernel for Integration ---
#define THREADS_PER_BLOCK_INTEGRATE 256

__global__ void integrate_kernel(double a, double h,
long long int num_sum_terms, double* d_block_sums) {
    __shared__ double s_cache[THREADS_PER_BLOCK_INTEGRATE];

    int tid_in_block = threadIdx.x;
    int block_id = blockIdx.x;
    int threads_per_block_dim = blockDim.x;
    int total_threads_in_grid = gridDim.x * threads_per_block_dim;

    double my_sum = 0.0;

    for (long long int i = (long long int)block_id *
                           threads_per_block_dim + tid_in_block + 1;
         i <= num_sum_terms;
         i += total_threads_in_grid) {
        double x = a + (double)i * h;
        my_sum += device_bessel_j1(x);
    }

    s_cache[tid_in_block] = my_sum;
    __syncthreads();

    for (int s = threads_per_block_dim / 2; s > 0; s >>= 1) {
        if (tid_in_block < s) {
            s_cache[tid_in_block] += s_cache[tid_in_block + s];
        }
        __syncthreads(); // Synchronize after each step of reduction
```

```
    }

    if (tid_in_block == 0) {
        d_block_sums[block_id] = s_cache[0];
    }
}

// --- Host function to manage CUDA integration ---
double calc_intg_cuda(
    long long unsigned int n_intervals = 1000000000,
    double (*f_host)(double) = host_bessel_j1,
    long double a_ld = 0.0,
    long double b_ld = 50.0) {

    double a = static_cast<double>(a_ld);
    double b = static_cast<double>(b_ld);

    if (n_intervals == 0) return 0.0;

    double h = (b - a) / n_intervals;
    long long int num_sum_terms = n_intervals - 1;

    double gpu_sum = 0.0;

    if (num_sum_terms > 0) {
        int threads_per_block = THREADS_PER_BLOCK_INTEGRATE;
        int num_blocks = (num_sum_terms + threads_per_block -1)
                         / threads_per_block;
        if (num_blocks > 2048) num_blocks = 2048;
        if (num_blocks == 0 && num_sum_terms > 0) num_blocks = 1;

        double *d_block_sums;
        CUDA_CHECK(cudaMalloc((void **)&d_block_sums,
                              num_blocks * sizeof(double)));
```

```
        integrate_kernel<<<num_blocks, threads_per_block>>>(a, h,
        num_sum_terms, d_block_sums);
        CUDA_CHECK(cudaGetLastError());
        CUDA_CHECK(cudaDeviceSynchronize());

        double *h_block_sums = (double *)malloc(num_blocks
                                                * sizeof(double));
        CUDA_CHECK(cudaMemcpy(h_block_sums, d_block_sums,
        num_blocks * sizeof(double), cudaMemcpyDeviceToHost));

        for (int i = 0; i < num_blocks; i++) {
            gpu_sum += h_block_sums[i];
        }

        free(h_block_sums);
        CUDA_CHECK(cudaFree(d_block_sums));
    }

    double integral_val = (h / 2.0) *
                          (f_host(a) + f_host(b) + 2.0 * gpu_sum);
    return integral_val;
}


// --- Original CPU calculation (renamed for clarity) ---
double calc_intg_cpu(
    bool use_parallel = true,
    long long unsigned int n = 1000000000,
    double (*f)(double) = host_bessel_j1,
    long double a_ld = 0.0,
    long double b_ld = 50.0) {

    double a = static_cast<double>(a_ld);
```

```
    double b = static_cast<double>(b_ld);

    if (n == 0) return 0.0;

    double h;
    long long int i;
    h = (b - a) / n;
    double sum = 0.0;

    if (use_parallel) {
        #pragma omp parallel for reduction(+:sum)
        for (i = 1; i <= n - 1; i++) {
            double x_val = a + (double)i * h;
            sum += f(x_val);
        }
    } else {
        for (i = 1; i <= n - 1; i++) {
            double x_val = a + (double)i * h;
            sum += f(x_val);
        }
    }

    double intg = (h / 2.0) * (f(a) + f(b) + 2.0 * sum);
    return intg;
}


// --- Final calculation logic (Richardson Extrapolation style) ---
enum IntegrationMode { CPU_SINGLE_THREAD, CPU_PARALLEL, GPU_CUDA };

double calc_intg_final(
    IntegrationMode mode,
    double epsilon = 5 * 10e-15,
    long long unsigned int n_initial = 1,
```

```
int lambda = 2,
int m = 2) {

double S_time = omp_get_wtime();

long double a_val = 0.0;
long double b_val = 50.0;
double (*func_ptr)(double) = host_bessel_j1;

long long unsigned int current_n = n_initial;
if (current_n == 0) current_n = 1;

double Sh1, Sh2;
double s_iter_time, f_iter_time;

printf("Mode: ");
switch(mode) {
    case CPU_SINGLE_THREAD: printf("CPU (Single Thread)\n"); break
    case CPU_PARALLEL:      printf("CPU (OpenMP Parallel)\n"); bre
    case GPU_CUDA:          printf("GPU (CUDA)\n"); break;
}
printf("Epsilon: %.2e, Initial N: %llu, Lambda: %d,
        Order m: %d\n", epsilon, n_initial, lambda, m);
printf("Function: j1(x), Range: [%.1Lf, %.1Lf]\n\n", a_val, b_val)
printf("Iter |         N | Integral Value    | Iter Time (s)
        | Error Estimate (ef)\n");
printf("-----------------------------------------------------------

s_iter_time = omp_get_wtime();
if (mode == GPU_CUDA) Sh1 = calc_intg_cuda(current_n, func_ptr,
a_val, b_val);
else Sh1 = calc_intg_cpu(mode == CPU_PARALLEL, current_n,
func_ptr, a_val, b_val);
f_iter_time = omp_get_wtime();
```

```
printf("%4d | %9llu | %.15f | %13.6f | N/A\n", 1, current_n,
Sh1, f_iter_time - s_iter_time);

current_n *= lambda;
s_iter_time = omp_get_wtime();
if (mode == GPU_CUDA) Sh2 = calc_intg_cuda(current_n, func_ptr,
a_val, b_val);
else Sh2 = calc_intg_cpu(mode == CPU_PARALLEL, current_n,
func_ptr, a_val, b_val);
f_iter_time = omp_get_wtime();

double ef = (Sh2 - Sh1) / (pow(lambda, m) - 1.0);
printf("%4d | %9llu | %.15f | %13.6f | %.2e\n", 2, current_n,
Sh2, f_iter_time - s_iter_time, ef);

int iter_count = 2;
while (fabs(ef) > epsilon) {
    iter_count++;
    if (iter_count > 25) { // Safety break for iterations
        printf("Max iterations reached, stopping.\n");
        break;
    }

    current_n *= lambda;
    Sh1 = Sh2;

    s_iter_time = omp_get_wtime();
    if (mode == GPU_CUDA) Sh2 = calc_intg_cuda(current_n,
    func_ptr, a_val, b_val);
    else Sh2 = calc_intg_cpu(mode == CPU_PARALLEL, current_n,
    func_ptr, a_val, b_val);
    f_iter_time = omp_get_wtime();

    ef = (Sh2 - Sh1) / (pow(lambda, m) - 1.0);
```

```
            printf("%4d | %9llu | %.15f | %13.6f | %.2e\n", iter_count,
            current_n, Sh2, f_iter_time - s_iter_time, ef);
        }
        printf("-------------------------------------------------------

        double ans = Sh2 + ef;
        // Richardson Extrapolation: Sh2 + (Sh2 - Sh1) / (lambda^m - 1)
        // which is Sh2 + ef
        double F_time = omp_get_wtime();

        printf("Final Result: %.15f\n", ans);
        printf("Total Time: %.6f seconds\n", F_time - S_time);
        printf("Converged in %d iterations with N_final = %llu.\n",
        iter_count, current_n);
        return ans;
}

int main() {
        printf("===== Running Numerical Integration =====\n");

        double epsilon = 5 * 10e-15;
        long long unsigned int n_initial = 100;

        // CPU Single Thread
        printf("\n--- CPU Single-Threaded ---\n");
        calc_intg_final(CPU_SINGLE_THREAD, epsilon, n_initial);

        // CPU Parallel (OpenMP)
        printf("\n--- CPU Parallel (OpenMP) ---\n");
        calc_intg_final(CPU_PARALLEL, epsilon, n_initial);

        // GPU CUDA
        printf("\n--- GPU CUDA ---\n");
        long long unsigned int n_initial_gpu = n_initial;
```

```
    calc_intg_final(GPU_CUDA, epsilon, n_initial_gpu);

    printf("\n===== Program Finished =====\n");
    return 0;
}
```