МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра вычислительной математики

ЗЕРНИСТЫЕ ВЕРСИИ АЛГОРИТМОВ ОРГАНИЗАЦИИ ШАБЛОННЫХ ВЫЧИСЛЕНИЙ НА GPU

Курсовая работа

Бобовоза Владислава
Сергеевича
студента 3 курса,
специальность «прикладная математика»

Научный руководитель:
кандидат физ.-мат. наук,
доцент
А.А. Толстиков

Минск, 2024

РЕФЕРАТ

Курсовая работа, 46 с., 5 источников, 19 рис., 3 таблицы, 4 прил.

ЗЕРНИСТЫЕ ВЕРСИИ АЛГОРИТМОВ ОРГАНИЗАЦИИ ШАБЛОННЫХ ВЫЧИСЛЕНИЙ НА GPU, CUDA, TILING, ПЕРЕМНОЖЕНИЕ МАТРИЦ, МЕТОД ЯКОБИ, МЕТОД ЗЕЙДЕЛЯ

Объект исследования – перемножение матриц, метод Якоби, метод Зейделя.

Цель работы – изучить основы вычислений на GPU, реализовать перемножение матриц, метод Якоби и метод Зейделя на CPU, GPU. Сравнить и проанализировать результаты.

Методы исследования – технология CUDA, метод Якоби, метод Зейделя.

Результаты работы: написана программа, реализующая перемножения матриц на GPU с использованием тайлинга. Реализован метод Якоби и метод Зейделя для случайно генерирующихся матриц с диагональным преобладанием. Проведено сравнение полученных результатов.

РЭФЕРАТ

Курсавы праект, 46 с., 5 крыніц, 19 мал., 3 табліцы, 4 дадаткі

КРУПЧАСТЫЯ ВЕРСІІ АЛГАРЫТМАЎ АРГАНІЗАЦЫІ ШАБЛОННЫХ ВЫЛІЧЭННЯЎ НА GPU, CUDA, TILING, ПЕРАМНАЖЭННЕ МАТРЫЦ, МЕТАД ЯКОБІ, МЕТАД ЗАЙДЭЛЯ

Аб'ект даследавання – перамнажэнне матрыц, метад Якобі, метад Зайдэля

Мэта працы – вывучыць асновы вылічэнняў на GPU, рэалізаваць перамнажэнне матрыц, метад Якобі і метад Зайдэля на CPU, GPU. Параўнаць і прааналізаваць вынікі.

Метады даследавання – тэхналогія CUDA, метад Якобі, метад Зайдэля.

Вынікі працы: напісана праграма, што рэалізуе перамнажэнні матрыц на GPU з выкарыстаннем тайлінгу. Рэалізаваны метад Якобі і метад Зайдэля для якія выпадкова генеруюцца матрыц з дыяганальнай перавагай. Праведзена параўнанне атрыманых вынікаў.

SUMMARY

Course work, 46 p., 5 sources, 19 fig., 3 tables, 4 app.

GRANULAR VERSIONS OF ALGORITHMS FOR ORGANIZING PATTERN CALCULATIONS ON GPU, CUDA, TILING, MATRIX MULTIPLICATION, JACOBI METHOD, ZEIDEL METHOD

The object of study – matrix multiplication, Jacobi method, Seidel method.

The purpose of the work – to study the basics of computing on GPU, to realize matrix multiplication, Jacobi method and Seidel method on CPU, GPU. Compare and analyze the results.

Research methods – CUDA technology, Jacobi method, Seidel method.

Results of the work: a program is written that implements matrix multiplications on GPU using tiling. Jacobi method and Seidel method for randomly generated matrices with diagonal dominance were realized. Comparison of the obtained results is carried out.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Зернистые версии алгоритмов организации шаблонных вычислений на графических процессорах (GPU) представляет собой важную и сложную задачу, которая имеет огромнейший потенциал для оптимизации вычислений.

В данной курсовой работе была рассмотрена задача исследования и сравнительного анализа производительности различных алгоритмов, применяемых для выполнения матричных операций и решения систем линейных алгебраических уравнений (СЛАУ) с диагональным преобладанием.

Основной целью работы является разработка и реализация эффективных алгоритмов для выполнения операций перемножения матриц и решения СЛАУ с диагональным преобладанием на различных аппаратных платформах, таких как центральные процессоры (CPU) и графические процессоры (GPU). Также исследуется возможность улучшения производительности путем применения техники тайлинга, которая позволяет более эффективно использовать кэш-память и уменьшить задержки при доступе к ней.

В ходе выполнения курсовой работы была реализована задача перемножения матриц на следующих платформах: CPU, CPU с использованием тайлинга, GPU, GPU с использованием тайлинга. Также были реализованы метод Якоби для решения СЛАУ с диагональным преобладанием на следующих платформах: CPU, CPU с использованием OpenMP, GPU, GPU с использованием тайлинга, а также метод Зейделя для решения СЛАУ с диагональным преобладанием на следующих платформах: CPU, CPU с использованием OpenMP, GPU. Были проведены эксперименты, собраны данные о времени выполнения и построены графики зависимостей производительности от размера входных данных.

Результат курсовой работы позволяет сделать вывод о том, какие алгоритмы и платформы наиболее эффективны для решения поставленных задач. Работа имеет практическую значимость для разработки и оптимизации вычислительных программ.

1 Теоретические сведения о CUDA. Основные понятия

CUDA (Compute Unified Device Architecture) – это программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы NVIDIA. Она предоставляет возможность использовать вычислительную мощность GPU для решения задач, которые традиционно выполнялись на CPU. Главное назначение CUDA – ускорение вычислений за счет параллельной обработки данных тысячами ядер GPU.

Основным понятием в CUDA является поток (thread). Поток является маленьким, независимым блоком кода, который выполняется на GPU. Потоки объединяются в блоки (block), которые выполняются на одном потоковом мультипроцессоре (SM). Каждый SM содержит несколько CUDA ядер – процессорных ядер, которые непосредственно выполняют потоки. Сетка (grid) – это группа блоков, которые выполняются на GPU. Варп (warp) – это группа из 32 потоков, которые выполняются одновременно на одном SM.

Наиболее важным аспектом является управление памятью. В CUDA представлены различные типы памяти:

- Регистровая память (register) является самой быстрой из всех. В CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор.
- Локальная память (local memory) может быть использована компилятором при большом количестве локальных переменных в какой-либо функции. По скорости, она значительно медленнее чем регистровая. Согласно документации NVIDIA, ее рекомендуется использовать только в самых необходимых случаях.
- Глобальная память (global memory) является самым медленным типом памяти, из доступных на GPU. Глобальная память в основном служит для хранения больших объемов данных, поступивших на device с host. В алгоритмах, где необходима высокая производительность, количество операций с глобальной памятью нужно свести к минимуму.
- Разделяемая память (shared memory) относится к быстрому типу памяти. Ее рекомендуется использовать для минимизации обращений к глобальной памяти, а также для хранения локальных переменных функций.
- Константная память (constant memory) является достаточно быстрой. Отличительной особенностью этого типа является возможность

записи данных с host, но при этом, в пределах GPU возможно лишь чтение из этой памяти.

- Текстурная память (texture memory), как и следует из названия, предназначена для работы с текстурами. Текстурная память имеет специфические особенности в адресации, чтении и записи данных.

Существуют также разные вычислительные архитектуры. Приведем их примеры на задаче вычисления значения произвольной функции $f(x)$:

- SISD (Single Instruction Single Data). Берем одно значение x, вычисляем для него $f(x)$. Дальше снова берем значение x, но уже другое и вычисляем для него $f(x)$.
- SIMD (Single Instruction Multiple Data). Берем сразу много значений x_i и вычисляем $f(x_i)$ за один заход. В этом случае все ядра вычисляют не обязательно одно и ту же функцию. К слову, GPU максимально похожа на эту архитектуру.
- MISD (Multiple Instruction Single Data). При такой архитектуре, ядра считают значения разных функций, но для одного и того же значения x.
- MIMD (Multiple Instruction Multiple Data). В этом случае, у каждого ядра своя инструкция и свои входные данные.

Для эффективного использования GPU и достижения высокой производительности, как и в жизненных задачах, важно уметь разбивать большую задачу на мелкие подзадачи, которые могут выполняться независимо друг от друга. Эта техника носит название тайлинга (tiling).

В CUDA представлены механизмы синхронизации, которые позволяют точно контролировать порядок выполнения инструкций и обеспечивает правильное взаимодействие между потоками. Это открывает огромный спектр возможностей для решения сложных задач, требующих параллельной обработки данных.

К основным преимуществом CUDA можно отнести следующее:

- Высокая параллельность. GPU имеет тысячи ядер, которые позволяют запускать огромное количество потоков одновременно;
- Высокая пропускная способность памяти. GPU имеет быструю память, которая позволяет быстро передавать данные между ядрами.
- Низкое энергопотребление. GPU энергоэффективнее CPU, если рассматривать выполнение параллельных задач.

Отсюда и появляется довольно широкий спектр применений данной технологии:

- Научные вычисления;
- Обработка видео и изображений;
- Машинное обучение;

- Криптография;
- Финансовое моделирование.

2 Алгоритм перемножения матриц

В первую очереди рассмотрим задачу перемножения матриц, как вводную, позволяющую ознакомиться с синтаксисом и нюансами технологии CUDA. [1]

2.1 Постановка задачи

Пусть были случайно сгенерированы матрицы A и B размерности $N \times N$. Необходимо перемножить эти матрицы, другими словами, получить $C = AB$, где $c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$.

Требуется реализовать алгоритм на следующих платформах: CPU, CPU используя tiling, GPU, GPU используя tiling.

Необходимо выполнить сравнение времени выполнения программ при разных размерностях матриц, а также при разных размерностях тайла.

2.2 Описание алгоритма

Идея распараллеливания заключается в том, что каждый элемент c_{ij} может быть получен независимо от любого другого элемента матрицы C. Таким образом можно запустить N^2 потоков, а каждый поток посчитает свой c_{ij}.

Программа этого пункта и его подпунктов реализована на C++ с использованием CUDA, текст программы приведен в приложении А.

2.2.1 Реализация на GPU

В матрицах A и B рассмотрим субматрицы A' и B'. Если умножить A' на B', получим набор элементов C'.

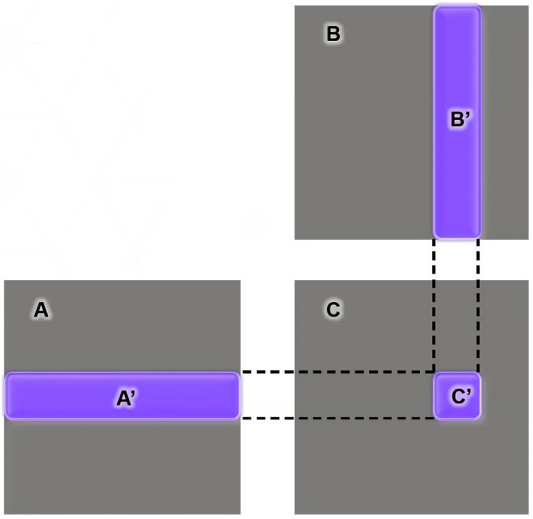

Рисунок 2.2.1.1 – Визуализация алгоритма перемножения матриц на GPU

2.2.2 Реализация на GPU используя tiling

В первую очередь необходимо передать данные в разделенную память. Вся работа производится именно с этим типом памяти, т.к. доступ к нему осуществляется значительно быстрее, чем к любой другой.

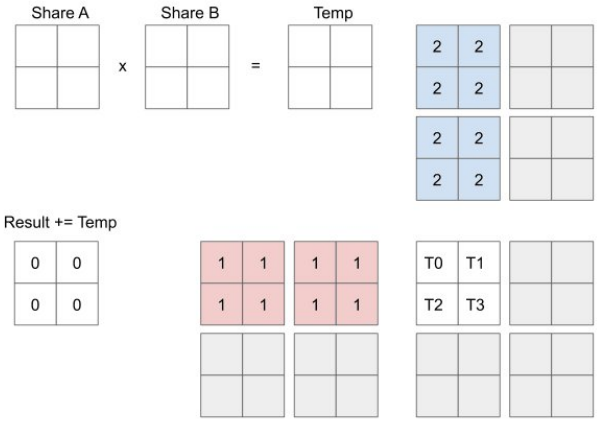

Рисунок 2.2.2.1 – Шаг 1. Вид матрицы

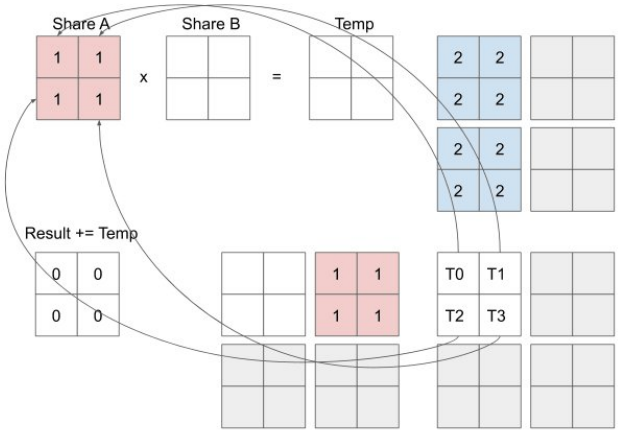

Рисунок 2.2.2.2 – Шаг 2. Перенос первого тайла в разделенную память

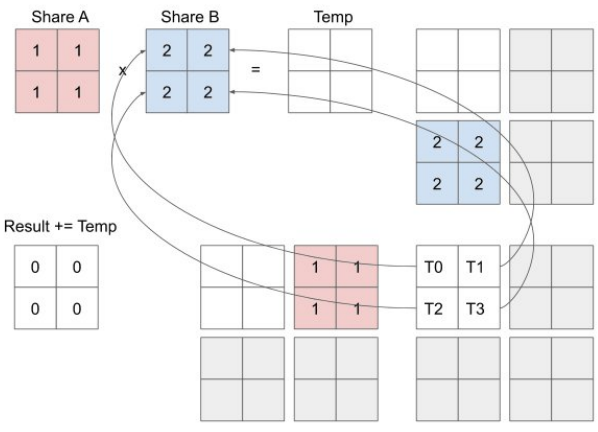

Рисунок 2.2.2.3 – Шаг 3. Перенос второго тайла в разделенную память

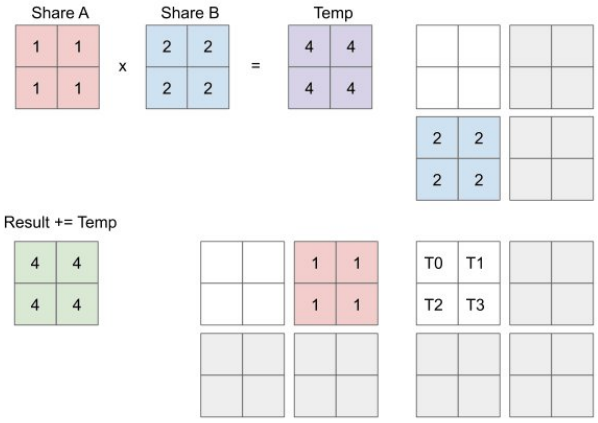

Рисунок 2.2.2.4 – Шаг 4. Вычисление первого частичного значения итоговой матрицы и добавление его в переменную result

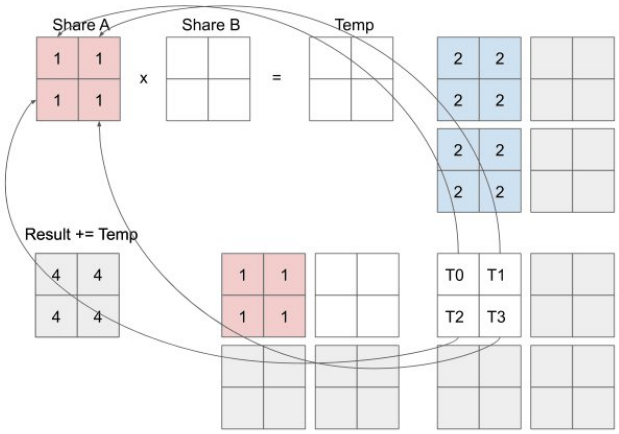

Рисунок 2.2.2.5 – Шаг 5. Перенос первого тайла в разделенную память

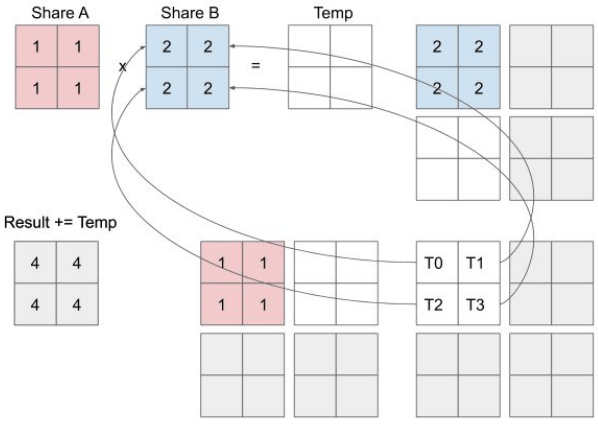

Рисунок 2.2.2.6 – Шаг 6. Перенос второго тайла в разделенную память

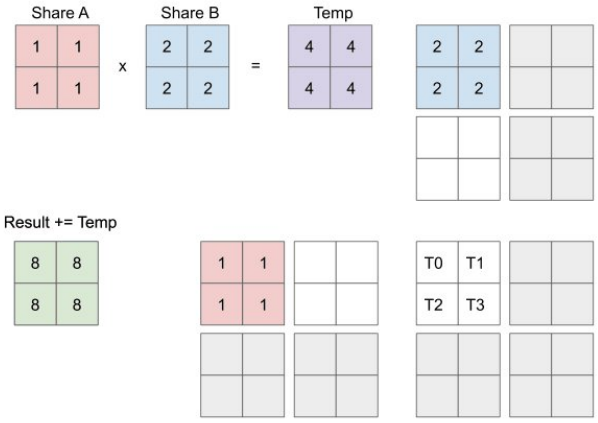

Рисунок 2.2.2.7 – Шаг 7. Вычисление второго частичного значения итоговой матрицы и добавление его в переменную result

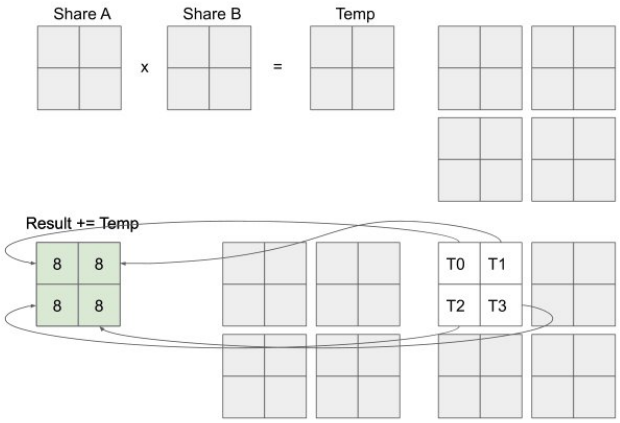

Рисунок 2.2.2.8 – Шаг 8. Присваивание полученного значения тайлу итоговой матрицы

2.3 Получение и анализ результатов

Получение и анализ результатов проведем следующим образом:

1. Запустим программу при разных значениях параметра N. Выберем следующие значения: $N = 16, 64, 256, 1024, 2048$.
2. Заполним таблицу времени выполнения.
3. По собранным данным строим график, позволяющий более четко проанализировать зависимость времени выполнения от размерности.
4. Выполним эти же пункты, но уже меняя параметр $TILE\_DIM$. Выберем следующие значения: $TILE\_DIM = 2, 4, 8, 16$.

Таблица 2.3.1 – Время выполнения программы при разных параметрах

N	TILE_DIM	CPU, мсек	CPU(tiling), мсек	GPU, мсек	GPU(tiling), мсек
16	2	0.009408	0.019456	0.032224	0.744768

16	4	0.009984	0.014784	0.032576	0.055936
16	8	0.010048	0.012672	0.032416	0.055168
16	16	0.010560	0.011488	0.032160	0.068192
64	2	0.459488	1.085344	0.050496	0.080640
64	4	0.479264	0.731712	0.047200	0.068320
64	8	0.473312	0.640800	0.048928	0.068768
64	16	0.465024	0.540320	0.046400	0.062080
256	2	31.517984	70.204193	0.693152	1.339104
256	4	31.645599	47.845791	0.431936	0.488800
256	8	32.172222	40.329922	0.389376	0.362848
256	16	31.497503	34.204704	0.475744	0.570080
1024	2	3244.554199	5616.709473	32.681599	123.267197
1024	4	3280.276123	3188.432373	13.742944	19.508608
1024	8	3174.305420	2899.360107	7.812416	7.539232
1024	16	3233.545654	2910.169922	6.821568	6.280448
2048	2	32378.373047	55114.375000	260.737732	843.160156
2048	4	32589.306641	25749.925781	96.835999	147.075623
2048	8	33206.726562	23647.255859	49.484001	43.869343
2048	16	33479.957031	23455.181641	43.198849	35.913246

Анализировать таблицу в таком виде не совсем удобно, поэтому зафиксируем размерность и посмотрим графики зависимости времени выполнения от размера тайла, для каждой размерности. Будем также учитывать, что матрицы одинаковой размерности каждый раз генерируются разные. Программа этого пункта для $N = 2048$ реализована на C++ с использованием MatplotlibCPP, текст программы приведен в приложении Б. Для остальных параметров N данные заполняются аналогично.

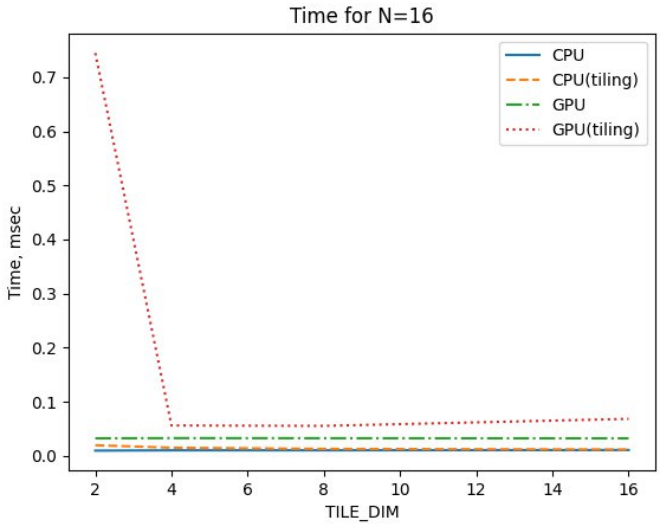

Рисунок 2.3.1 – График зависимости времени выполнения от размерности (N=16)

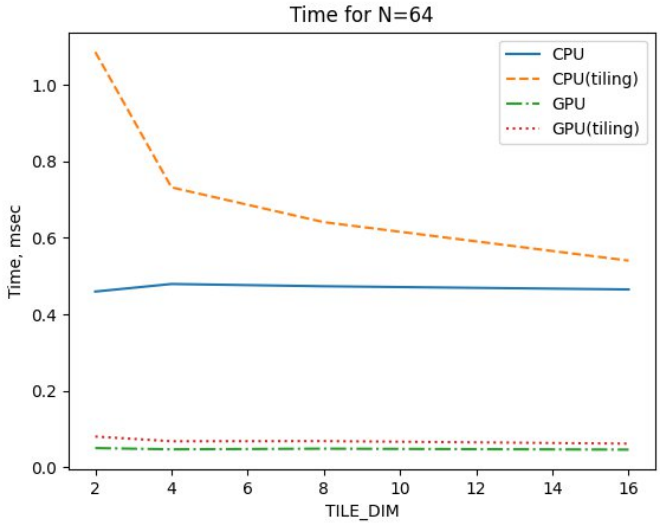

Рисунок 2.3.2 – График зависимости времени выполнения от размерности (N=64)

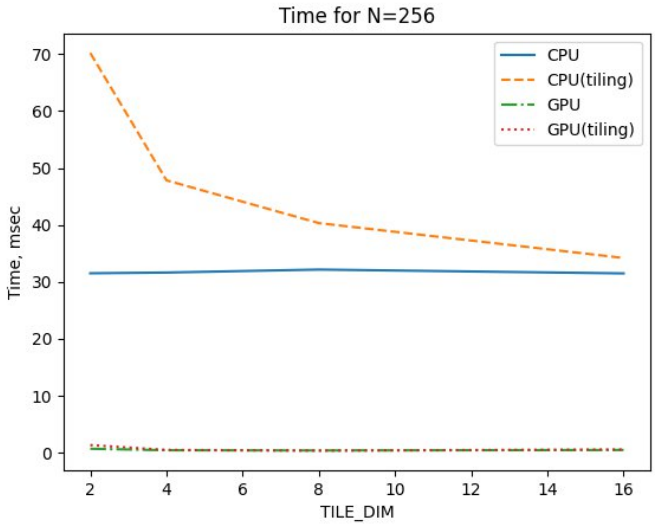

Рисунок 2.3.3 – График зависимости времени выполнения от размерности (N=256)

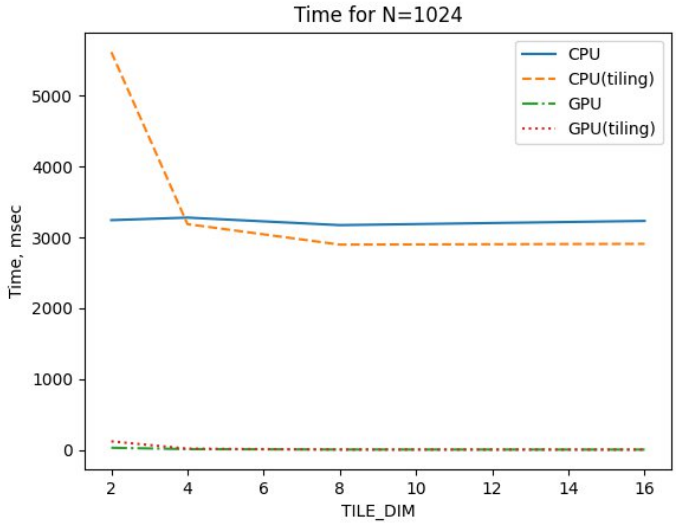

Рисунок 2.3.4 – График зависимости времени выполнения от размерности (N=1024)

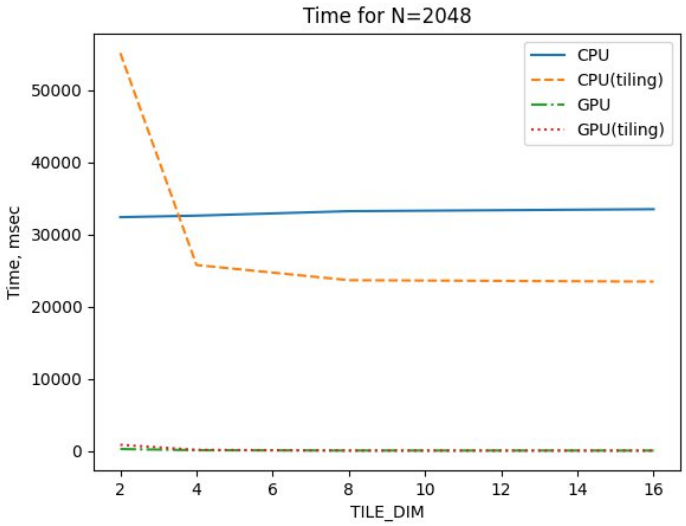

Рисунок 2.3.5 – График зависимости времени выполнения от размерности (N=2048)

Из графиков и таблицы можно сделать вывод, что для больших матриц наиболее эффективным и производительным является реализация на GPU с использованием тайлинга. При этом, размерность тайла также влияет на результат. Вариант реализации на GPU также довольно хорош, потому что дает серьезное ускорение относительно CPU и CPU с использованием тайлинга.

3 Метод Якоби

Метод Якоби – это итерационный метод решения системы линейных алгебраических уравнений (СЛАУ) вида:

$$Ax = b,$$

где A – матрица размера N, x – вектор-столбец размерности N, содержащий неизвестные, b – вектор-столбец размерности N, содержащий правые части уравнений.

3.1 Постановка задачи

Пусть генерируется матрица с диагональным преобладанием размерности $N \times N$, и вектор-столбец b размерности N. Необходимо решить задачу методом Якоби.

3.2 Описание алгоритма

Суть метода заключается в следующем [2]:

- Начальное приближение $x^{(0)}$ задается произвольно.
- На каждой итерации k, для каждого x_i вычисляется новое значение по формуле:

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{N} a_{ij} x_j^{(k-1)}\right).$$

- Итерации продолжаются до тех пор, пока не будет достигнута заданная точность.

Основное преимущество:

- Простота реализации.

Недостаток метода:

- Медленная скорость сходимости для некоторых систем.

В качестве критерия остановки будем использовать следующее:

$$\left\| Ax^{(k)} - b \right\| \leq eps.$$

Программа этого пункта и его подпунктов реализована на C++ с использованием CUDA и OpenMP, текст программы приведен в приложении В.

3.2.1 Реализация на GPU

Суть реализации метода Якоби на GPU заключается в параллельном выполнении итераций метода Якоби. Это позволяет значительно ускорить вычисления за счет параллельной обработки большого объема данных. [3]

Основные шаги:

- Подготовка данных. Матрица A, векторы x и b, а также другие необходимые параметры загружаются на GPU
- Инициализация начального приближения.
- Итерационный процесс. Сам метод Якоби применяется к текущему приближению, чтобы получить новое приближение. Этот шаг выполняется параллельно для всех элементов вектора x.
- Проверка условий сходимости.

3.2.2 Реализация на GPU используя tiling

Суть реализации метода Якоби на GPU используя tiling заключается в разбиении матрицы коэффициентов на небольшие блоки (тайлы), над которыми выполняются параллельные вычисления. [3]

Основные преимущества:

- Улучшенная локальность данных. Благодаря тайлингу, каждый блок потоков загружает в разделяемую память только необходимые данные для вычислений. Это уменьшает количество обращений к глобальной памяти и улучшает локальность данных.
- Эффективное использование памяти. Так как данные хранятся в разделяемой памяти, к которой имеется довольно быстрый доступ,

можно получить сниженное время ожидания при доступе, что в свою очередь повысит производительность.

3.3 Получение и анализ результатов

Получение и анализ результатов проведем следующим образом:

1. Запустим программу при разных значениях параметра N. Выберем следующие значения: $N = 32, 64, 128, 256, 512$.
2. Заполним таблицу времени выполнения.
3. По собранным данным строим график, позволяющий более четко проанализировать зависимость времени выполнения от размерности
4. Выполним эти же пункты, но уже меняя параметр *TILE_DIM*. Выберем следующие значения: $TILE\_DIM = 2, 4, 8, 16$.

Таблица 3.3.1 – Время выполнения программы при разных параметрах. Величина погрешности установлена равной 1e-5.

N	TILE_DIM	CPU, msec	CPU(OpenMP), msec	GPU, msec	GPU(tiling), msec
32	2	77.298691	110.444542	456.664917	591.845459
32	4				571.566223
32	8				564.571777
32	16				564.391541
64	2	632.857605	351.202301	1351.281616	1209.848511
64	4				1187.305054
64	8				1144.456299
64	16				1142.435913
128	2	5321.412598	990.581787	4865.458008	2878.362793
128	4				2978.305664
128	8				2847.568115
128	16				2846.475342
256	2	43858.902344	8772.993164	27729.259766	8823.482422
256	4				9034.385742
256	8				8815.123047
256	16				8739.131836
512	2	366274.582	50919.527344	200963.75	32603.59375
512	4				29796.52734
512	8				32785.92968
512	16				32774.47656

Как и в предыдущем пункте, анализировать таблицу в таком виде не совсем удобно, поэтому зафиксируем размерность и посмотрим графики зависимости времени выполнения от размера тайла, для каждой размерности. Программа этого пункта аналогична уже реализованной на C++ с использованием MatplotlibCPP. Текст программы приведен в приложении Б.

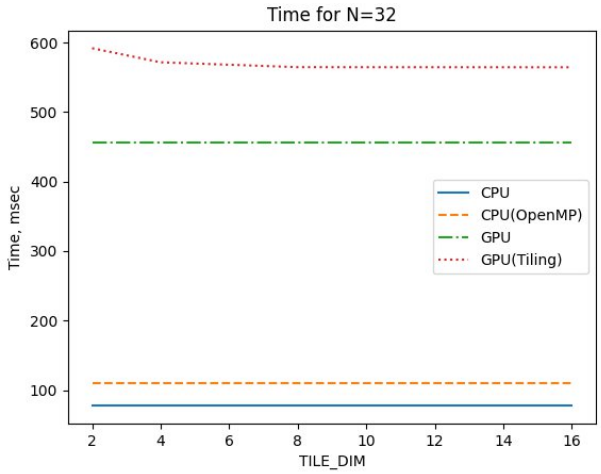

Рисунок 3.3.1 – График зависимости времени выполнения от размерности. (N=32)

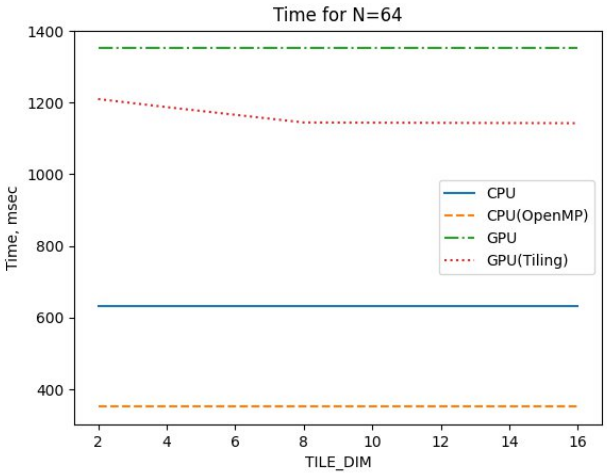

Рисунок 3.3.2 – График зависимости времени выполнения от размерности. (N=64)

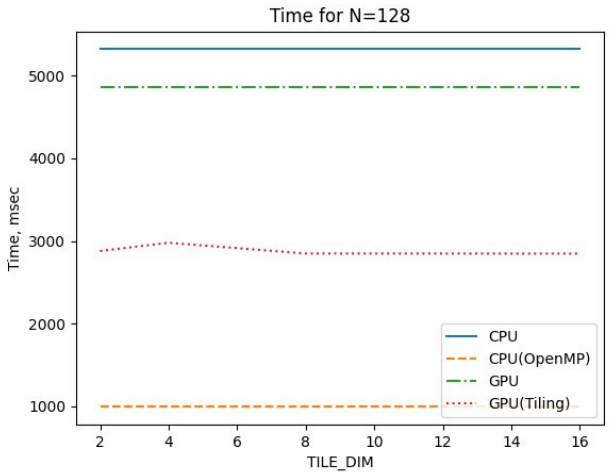

Рисунок 3.3.3 – График зависимости времени выполнения от размерности. (N=128)

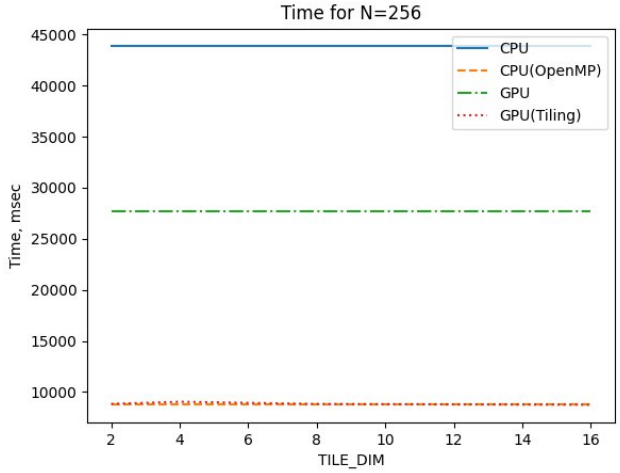

Рисунок 3.3.4 – График зависимости времени выполнения от размерности. (N=256)

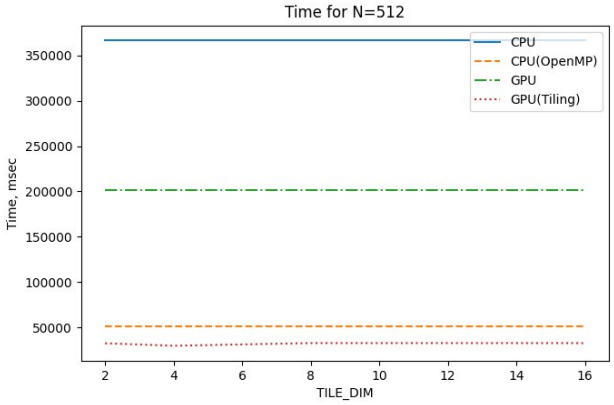

Рисунок 3.3.5 – График зависимости времени выполнения от размерности. (N=512)

Из графиков и таблицы видно, что при больших размерностях матрицы, лучше всего показывает себя реализация на GPU с использованием тайлинга.

Реализация на CPU с OpenMP также хороша в силу простоты реализации. Но если нужно добиться максимальной производительности, лучше всего использовать реализацию на GPU с тайлингом.

4 Метод Зейделя

Метод Зейделя – это итерационный метод решения системы линейных алгебраических уравнений (СЛАУ) вида:

$$Ax = b,$$

где A – матрица размера N, x – вектор-столбец размерности N, содержащий неизвестные, b – вектор-столбец размерности N, содержащий правые части уравнений.

4.1 Постановка задачи

Пусть генерируется матрица с диагональным преобладанием размерности $N \times N$, и вектор-столбец b размерности N. Необходимо решить задачу методом Зейделя.

4.2 Описание алгоритма

Суть метода заключается в следующем [2]:

- Начальное приближение $x^{(0)}$ задается произвольно.
- На каждой итерации k, для каждого x_i вычисляется новое значение по формуле:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(k-1)} \right).$$

- Итерации продолжаются до тех пор, пока не будет достигнута заданная точность.

Основное преимущество:

- Более быстрая сходимость, в сравнении с методом Якоби.

Недостаток метода:

- Реализация немного сложнее, в сравнении с методом Якоби.

В качестве критерия остановки будем использовать следующее:

$$\left\| Ax^{(k)} - b \right\| \leq eps.$$

Программа этого пункта и его подпунктов реализована на C++ с использованием CUDA и OpenMP, текст программы приведен в приложении Г.

4.2.1 Реализация на GPU

Суть реализации метода Зейделя на GPU заключается в параллельном выполнеии итераций. Основное отличие от метода Якоби заключается в том, что метод Зейделя использует уже вычисленные значения текущей итерации для обновления остальных значений, что ускоряет сходимость. Но тогда появляется главный минус: обновления не являются независимыми, что усложняет параллельную реализацию на GPU. [3]

Основные шаги:

- Подготовка данных. Матрица A, векторы x и b, а также другие необходимые параметры загружаются на GPU
- Инициализация начального приближения.
- Итерационный процесс. Обновление значений выполняется последовательно, а каждое $x_i^{(k+1)}$ зависит от уже обновленных $x_j^{(k+1)}$ при $j < i$ и старых $x_j^{(k)}$ при $j > i$
- Проверка условий сходимости.

4.3 Получение и анализ результатов

Получение и анализ результатов проведем следующим образом:

1. Запустим программу при разных значениях параметра N. Выберем следующие значения: $N = 32, 64, 128, 256, 512$.
2. Заполним таблицу времени выполнения.

Таблица 4.3.1 – Время выполнения программы при разных размерностях. Величина погрешности установлена равной 1e-5.

N	CPU, msec	CPU(OpenMP), msec	GPU, msec
32	0.098304	0.542720	298.403015
64	0.292864	0.653312	1060.930176
128	1.073152	0.711680	4157.780273
256	4.559872	2.378752	20226.386719
512	18.890753	3.175424	184573.687500

Стоит отметить, что в реализации на CPU и CPU(OpenMP) явно есть некие проблемы: для матрицы любой размерности число итераций равно 12, что очевидно не является верным, хотя полученное решение удовлетворяет критерию остановки и является истинным. Безусловно, программная реализация нуждается в доработке.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе были исследованы и сравнены различные алгоритмы для выполнения матричных операций и решения систем линейных алгебраических уравнений (СЛАУ) с диагональным преобладанием на разных аппаратных платформах: CPU и GPU.

Основной целью была разработка и реализаций эффективных алгоритмов для решения поставленных задач.

В ходе выполнения работы были получены следующие результаты:

- Реализованы алгоритмы перемножения матриц на CPU, CPU с использованием тайлинга, GPU, GPU с использованием тайлинга.
- Реализован метод Якоби для решения СЛАУ с диагональным преобладанием на CPU, CPU с использованием OpenMP, GPU, GPU с использованием тайлинга.
- Реализован метод Зейделя для решения СЛАУ с диагональным преобладанием на CPU, CPU с использованием OpenMP, GPU.
- Проведены эксперименты, собраны данные о времени выполнения программ, а также построены графики зависимостей производительности от размера входных данных.

Результаты экспериментов показали следующее:

- Использование GPU позволяет существенно повысить производительность по сравнению с CPU.
- Применение техники тайлинга может дополнительно улучшить производительность как на CPU, так и на GPU.
- Метод Зейделя, как правило, работает быстрее метода Якоби, особенно для больших систем уравнений.

Тогда можно сделать следующие выводы:

- Для задач, связанных с перемножением матриц, стоит реализовывать код на GPU с использованием тайлинга, если имеется необходимое оборудование. Если же оборудования нет, лучше всего реализовывать используя CPU tiling.
- Для решения СЛАУ больших размерностей с диагональным преобладанием наиболее эффективным является метод Зейделя, реализованный на CPU используя OpenMP. Но если сравнивать время выполнения на CPU и GPU, GPU реализация явно выигрывает для больших матриц.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Вводный курс ШАД «Основы CUDA вычислений», [электронный ресурс]. – Режим доступа: https://shad.yandex.ru/

2. Репников, В.И. Методы численного анализа / В.И. Репников. – Минск: БГУ, 2009. – 378 с.

3. Форум разработчиков, [электронный ресурс]. – Режим доступа: https://forums.developer.nvidia.com/

4. Дж. Сандерс, Э. Кэндрот. Технология CUDA в примерах: введение в программирование графических процессоров. Пер. с англ. Слинкина А.А., научный редактор Боресков А.В. – М.: ДМК Пресс, 2015. - 232 с.

5. CUDA C++ Programming Guide, [электронный ресурс]. – Режим доступа: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Код программы на языке C++ с использованием CUDA

```
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"


#include <stdio.h>
#include <stdlib.h> // Для функции rand()
#include <time.h>  // Для инициализации генератора случайных чисел


#define TILE_DIM 16


// Функция-ядро без тайлинга
__global__ void mulKernelGlobal(int *A, int *B, int *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;


    if(row < N && col < N) {
        int sum = 0;
        for(int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}


// Функция-ядро с tiling
__global__ void mulKernelTiling(int *A, int *B, int *C, int N) {
    // Создание 2 тайлов для матрицы A и B в shared memory
    __shared__ int ATile[TILE_DIM][TILE_DIM];
    __shared__ int BTile[TILE_DIM][TILE_DIM];


    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;


    int thrX = threadIdx.x;
    int thrY = threadIdx.y;


    int elementC = 0;


    for(int t = 0; t < (N - 1) / TILE_DIM + 1; ++t) {
        // Потоки для загрузки матрицы A в shared memory
        if(row < N && t * TILE_DIM + thrX < N)
            ATile[thrY][thrX] = A[row * N + t * TILE_DIM + thrX];
        else
            ATile[thrY][thrX] = 0.0f;


        // Потоки для загрузки матрицы B в shared memory
        if(t * TILE_DIM + thrY < N && col < N)
```

```
            BTile[thrY][thrX] = B[(t * TILE_DIM + thrY) * N + col];
        else
            BTile[thrY][thrX] = 0.0f;


        __syncthreads();


        // Вычисление частичного значения для матрицы C
        for(int i = 0; i < TILE_DIM; ++i)
            elementC += ATile[thrY][i] * BTile[i][thrX];


        __syncthreads();


    }
    // Копирование конечного значения в матрицу C
    if(row < N && col < N)
        C[row * N + col] = elementC;


}


// Получение результатов на CPU + tiling
void matrix_mul_cpu_tiling(int *a, int *b, int *c, int N) {
    // Обнуление результирующей матрицы
    for(int i = 0; i < N * N; i++) {
        c[i] = 0;
    }


    // Перемножение матриц с использованием разбиения на плитки
    for(int i = 0; i < N; i += TILE_DIM) { // строки
        for(int j = 0; j < N; j += TILE_DIM) { // столбцы
            for(int k = 0; k < N; k += TILE_DIM) { // внутренний размер плитки


                // Подплиточные границы
                const int minI = std::min(i + TILE_DIM, N);
                const int minJ = std::min(j + TILE_DIM, N);
                const int minK = std::min(k + TILE_DIM, N);


                // Перемножение в пределах плитки
                for(int ii = i; ii < minI; ii++) { // строки плитки
                    for(int jj = j; jj < minJ; jj++) { // столбцы плитки
                        int sum = 0; // промежуточное значение
                        for(int kk = k; kk < minK; kk++) { // перемножение в
пределах плитки
                            sum += a[ii * N + kk] * b[kk * N + jj];
                        }
                        c[ii * N + jj] += sum; // накопление результата
                    }
                }
            }
        }
    }
}


// Получение результатов на CPU
```

```
void matrix_mul_cpu(int *a, int *b, int *c, int N) {
    int tmp;
    for(int i = 0; i < N; ++i) { // Для каждой строки
        for(int j = 0; j < N; ++j) { // Для каждого столбца
            tmp = 0;
            for(int k = 0; k < N; ++k) { // Для каждого элемента в этой строке и
столбце
                tmp += a[i * N + k] * b[k * N + j];
            }
            c[i * N + j] = tmp;
        }
    }
}


int main() {
    int *hostA, *hostB, *hostCGPUTiling, *hostCGPU, *hostCCPUTiling, *hostCCPU;
    int *deviceA, *deviceB, *deviceC;
    int N;


    // Создание переменных-событий
    float timerValueGPU, timerValueGPUTiling, timerValueCPU,
timerValueCPUTiling;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);


    N = 2048; // Размерность матриц


    // Инициализация генератора случайных чисел
    srand(time(NULL));

    // Выделение памяти на хосте
    hostA = (int *)malloc(N * N * sizeof(int));
    hostB = (int *)malloc(N * N * sizeof(int));
    hostCGPU = (int *)malloc(N * N * sizeof(int));
    hostCCPU = (int *)malloc(N * N * sizeof(int));
    hostCGPUTiling = (int *)malloc(N * N * sizeof(int));
    hostCCPUTiling = (int *)malloc(N * N * sizeof(int));


    // Выделение памяти на устройстве (GPU)
    cudaMalloc((void **) &deviceA, N * N * sizeof(int));
    cudaMalloc((void **) &deviceB, N * N * sizeof(int));
    cudaMalloc((void **) &deviceC, N * N * sizeof(int));

    // Конфигурация сетки и блока
    dim3 DimGrid((N - 1) / TILE_DIM + 1, (N - 1) / TILE_DIM + 1, 1);
    dim3 DimBlock(TILE_DIM, TILE_DIM, 1);


    // Генерация случайных матриц
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            hostA[i * N + j] = ((int)rand() % 101) - 50; // значения от -50 до
50
        }
    }
```

```
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            hostB[i * N + j] = ((int)rand() % 101) - 50; // значения от -50 до
50
        }
    }


    // Вывод времени
    printf("\nTIME:");


    // ------------------------ GPU-tiling-вариант ------------------------
    // Запуск таймера
    cudaEventRecord(start, 0);


    // Копирование данных на GPU
    cudaMemcpy(deviceA, hostA, N * N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceB, hostB, N * N * sizeof(int), cudaMemcpyHostToDevice);


    // Запуск ядра
    mulKernelTiling<<<DimGrid,DimBlock>>>(deviceA, deviceB, deviceC, N);
    cudaDeviceSynchronize();


    // Копирование результата обратно на хост
    cudaMemcpy(hostCGPUTiling, deviceC, N * N * sizeof(int),
cudaMemcpyDeviceToHost);


    // Оценка времени вычисления GPU-варианта
    cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueGPUTiling, start, stop);
    printf("\n GPU-tiling    %f msec    ", timerValueGPUTiling);
    // ------------------------------------------------------------------


    // ------------------------ GPU-no-tiling-вариант ------------------------
    // Запуск таймера
    cudaEventRecord(start, 0);


    // Копирование данных на GPU
    cudaMemcpy(deviceA, hostA, N * N * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceB, hostB, N * N * sizeof(int), cudaMemcpyHostToDevice);


    // Запуск ядра
    mulKernelGlobal<<<DimGrid, DimBlock>>>(deviceA, deviceB, deviceC, N);
    cudaDeviceSynchronize();


    // Копирование результата обратно на хост
    cudaMemcpy(hostCGPU, deviceC, N * N * sizeof(int), cudaMemcpyDeviceToHost);


    // Оценка времени вычисления GPU-варианта без tiling
    cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
```

```
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueGPU, start, stop);
    printf("\n GPU-No-Tiling %f msec ", timerValueGPU);
    // ------------------------------------------------------------------------


    // Освобождение памяти на устройстве
    cudaFree(deviceA);
    cudaFree(deviceB);
    cudaFree(deviceC);


    // ------------------------ CPU_tiling-вариант ------------------------
    // Запуск таймера
    cudaEventRecord(start, 0);


    // Запуск функции
    matrix_mul_cpu_tiling(hostA, hostB, hostCCPUTiling, N);


    // Оценка времени вычисления GPU-варианта
    cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPUTiling, start, stop);
    printf("\n CPU-tiling    %f msec    ", timerValueCPUTiling);
    // ------------------------------------------------------------------------


    // ------------------------ CPU_no_tiling-вариант ------------------------
    // Запуск таймера
    cudaEventRecord(start, 0);


    // Запуск функции
    matrix_mul_cpu(hostA, hostB, hostCCPU, N);


    // Оценка времени вычисления GPU-варианта
    cudaThreadSynchronize();
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPU, start, stop);
    printf("\n CPU-No-Tiling %f msec \n", timerValueCPU);
    // ------------------------------------------------------------------------

    // Вывод ускорения
    printf("\nACCELERATION:");
    printf("\n NoTilingCPU / TilingGPU   %fx", timerValueCPU /
timerValueGPUTiling);
    printf("\n NoTilingCPU / NoTilingGPU %fx", timerValueCPU / timerValueGPU);
    printf("\n NoTilingCPU / TilingCPU   %fx", timerValueCPU /
timerValueCPUTiling);
    printf("\n TilingCPU   / TilingGPU   %fx", timerValueCPUTiling /
timerValueGPUTiling);
    printf("\n TilingCPU   / NoTilingGPU %fx", timerValueCPUTiling /
timerValueGPU);
    printf("\n NoTilingGPU / TilingGPU   %fx", timerValueGPU /
timerValueGPUTiling);


    // Освобождение памяти на хосте
```

```
    free(hostA);
    free(hostB);
    free(hostCCPU);
    free(hostCGPU);
    free(hostCCPUTiling);
    free(hostCGPUTiling);


    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Код программы на языке C++ с использованием MatplotlibCPP

```
#include <matplotlibcpp.h>
#include <vector>


namespace plt = matplotlibcpp;


int main() {
    std::vector<double> x = {2, 4, 8, 16};
    std::vector<double> y1 = {
        32378.373047,
        32589.306641,
        33206.726562,
        33479.957031
    };
    std::vector<double> y2 = {
        55114.375000,
        25749.925781,
        23647.255859,
        23455.181641
    };
    std::vector<double> y3 = {
        260.737732,
        96.835999,
        49.484001,
        43.198849
    };
    std::vector<double> y4 = {
        843.160156,
        147.075623,
        43.869343,
        35.913246
    };


    plt::plot(x, y1, {{"label", "CPU"}, {"ls", "-"}});
    plt::plot(x, y2, {{"label", "CPU(tiling)"}, {"ls", "--"}});
    plt::plot(x, y3, {{"label", "GPU"}, {"ls", "-."}});
    plt::plot(x, y4, {{"label", "GPU(tiling)"}, {"ls", ":"}});


    plt::xlabel("TILE_DIM");
    plt::ylabel("Time, msec");
    plt::title("Time for N=2048");
    plt::legend();


    plt::show();


    return 0;
}
```

ПРИЛОЖЕНИЕ В

Код программы на языке C++ с использованием CUDA и OpenMP

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
#include <omp.h>


using namespace std;


#define TILE_DIM 16


// ---------------------------------- NORM-CALCULATION -------------------------
-------
void multiplyMatrixVector(double* A, double* x, double* result, int N) {
    for(int i = 0; i < N; ++i) {
        result[i] = 0;
        for(int j = 0; j < N; ++j) {
            result[i] += A[i * N + j] * x[j];
        }
    }
}


double computeNorm(double* Ax, double* b, int N) {
    double norm = 0;
    for(int i = 0; i < N; ++i) {
        norm += (Ax[i] - b[i]) * (Ax[i] - b[i]);
    }
    return sqrt(norm);
}


void multiplyMatrixVectorOMP(double* A, double* x, double* result, int N) {
    #pragma omp parallel for
    for(int i = 0; i < N; ++i) {
        result[i] = 0;
        for(int j = 0; j < N; ++j) {
            result[i] += A[i * N + j] * x[j];
        }
    }
}


double computeNormOMP(double* Ax, double* b, int N) {
    double norm = 0;


    #pragma omp parallel for reduction(+:norm)
    for(int i = 0; i < N; ++i) {
        norm += (Ax[i] - b[i]) * (Ax[i] - b[i]);
    }
    return sqrt(norm);
}
// ---------------------------------------------------------------------------
-------
```

```
// ---------------------------------- CPU-NO-TILING ---------------------------
--------
// Метод Якоби для решения СЛАУ Ax = f
void CPU_NO_TILING_Jacobi_Method(double* A, double* f, double* x, int N, double
eps) {
    double* x_prev = new double[N]();
    double* Ax = new double[N]();
    int iterations = 0;


    while(true) { // цикл продолжается до достижения критерия остановки
        // Копируем текущее значение x в x_prev перед каждой итерацией
        for(int i = 0; i < N; ++i) {
            x_prev[i] = x[i];
        }


        for(int i = 0; i < N; ++i) {
            double sum = 0;


            for(int j = 0; j < N; ++j) {
                if(j != i) {
                    sum += A[i * N + j] * x_prev[j];
                }
            }

            // Получаем новое приближение x
            x[i] = (f[i] - sum) / A[i * N + i];
        }


        iterations++;


        // Вычисляем Ax^(k)
        multiplyMatrixVector(A, x, Ax, N);


        // Проверка условия остановки
        if(computeNorm(Ax, f, N) <= eps) {
            break;
        }
    }


    delete[] x_prev; // освобождаем память
    delete[] Ax; // освобождаем память
    cout << "Iteration count: " << iterations;
}
// ---------------------------------------------------------------------------
--------


// ------------------------------------ CPU-OMP -------------------------------
------
// Метод Якоби + OpenMP
void CPU_Parallel_Jacobi_Method(double* A, double* f, double* x, int N, double
eps) {
    double* x_prev = new double[N]();
    double* Ax = new double[N]();
    int iterations = 0;
```

```
    while(true) {
        // Копируем текущее значение x в x_prev параллельно
        #pragma omp parallel for
        for(int i = 0; i < N; ++i) {
            x_prev[i] = x[i];
        }


        // Новое приближение x параллельно
        #pragma omp parallel for
        for(int i = 0; i < N; ++i) {
            double sum = 0;


            // Суммируем элементы в строке, исключая диагональ
            #pragma omp simd
            for(int j = 0; j < N; ++j) {
                if(j != i) {
                    sum += A[i * N + j] * x_prev[j];
                }
            }


            x[i] = (f[i] - sum) / A[i * N + i];
        }


        iterations++;


        // Вычисляем Ax^(k)
        multiplyMatrixVectorOMP(A, x, Ax, N);


        // Проверка условия остановки
        if(computeNormOMP(Ax, f, N) <= eps) {
            break;
        }
    }


    delete[] x_prev; // освобождаем память
    delete[] Ax; // освобождаем память
    cout << "\nIteration count: " << iterations;
}
// ------------------------------------------------------------------------
------


// ---------------------------------- GPU-No-Tiling ----------------------------
------
__global__ void jacobi_no_tiling_Kernel(double* x_next, const double* A, const
double* x_now, const double* b_h, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;


    if(i < N) {
        double sigma = 0.0;
        for(int j = 0; j < N; j++) {
            if(i != j) {
                sigma += A[i * N + j] * x_now[j];
```

```
            }
        }
        x_next[i] = (b_h[i] - sigma) / A[i * N + i];
    }
}


void GPU_NO_TILING_Jacobi_Method(double* A, double* f, double* x, int N, double
eps) {
    // Размеры блока и сетки
    int threads_per_block = 16;
    int blocks_per_grid = (N + threads_per_block - 1) / threads_per_block;


    // Выделяем память на устройстве
    double* d_A;
    double* d_x_now;
    double* d_x_next;
    double* d_f;
    cudaMalloc(&d_A, N * N * sizeof(double));
    cudaMalloc(&d_x_now, N * sizeof(double));
    cudaMalloc(&d_x_next, N * sizeof(double));
    cudaMalloc(&d_f, N * sizeof(double));


    // Копируем данные Host->Device
    cudaMemcpy(d_A, A, N * N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_x_now, x, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_f, f, N * sizeof(double), cudaMemcpyHostToDevice);


    double* x_prev = new double[N];
    double* Ax = new double[N]();
    int iteration = 0;


    while(true) {
        // Копируем текущее приближение в x_prev
        cudaMemcpy(x_prev, d_x_now, N * sizeof(double), cudaMemcpyDeviceToHost);


        // Запускаем ядро Якоби
        jacobi_no_tiling_Kernel<<<threads_per_block,
blocks_per_grid>>>(d_x_next, d_A, d_x_now, d_f, N);


        // Копируем результат Device->Host для проверки условия остановки
        cudaMemcpy(x, d_x_next, N * sizeof(double), cudaMemcpyDeviceToHost);

        iteration++;


        // Вычисляем Ax^(k)
        multiplyMatrixVector(A, x, Ax, N);


        // Проверка условия остановки
        if(computeNorm(Ax, f, N) <= eps) {
            break;
        }


        // Обновляем x_now для следующей итерации
```

```
        cudaMemcpy(d_x_now, d_x_next, N * sizeof(double),
cudaMemcpyHostToDevice);
    }


    // Вывод числа итераций
    cout << "\nIteration count: " << iteration;


    // Освобождаем память
    cudaFree(d_A);
    cudaFree(d_x_now);
    cudaFree(d_x_next);
    cudaFree(d_f);
    delete[] x_prev;
    delete[] Ax;
}
// ------------------------------------------------------------------------
------


// ------------------------------------ GPU-Tiling ---------------------------
---------
__global__ void jacobiTilingKernel(double* x_next, const double* A, const
double* x_now, const double* b, int N) {
    __shared__ double x_tile[TILE_DIM];


    int tx = threadIdx.x;
    int bx = blockIdx.x;
    int i = bx * blockDim.x + tx;


    if(i < N) {
        double sigma = 0.0;
        for(int j = 0; j < N; j += TILE_DIM) {
            if(j + tx < N) {
                x_tile[tx] = x_now[j + tx];
            }
            __syncthreads();


            for(int k = 0; k < TILE_DIM && (j + k) < N; ++k) {
                if(i != (j + k)) {
                    sigma += A[i * N + (j + k)] * x_tile[k];
                }
            }
            __syncthreads();
        }
        x_next[i] = (b[i] - sigma) / A[i * N + i];
    }
}


__global__ void multiplyMatrixVectorKernel(const double* A, const double* x,
double* result, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;


    if(i < N) {
        double sum = 0.0;
        for(int j = 0; j < N; ++j) {
            sum += A[i * N + j] * x[j];
```

```
        }
        result[i] = sum;
    }
}


__global__ void computeNormKernel(const double* Ax, const double* b, double*
norm, int N) {
    __shared__ double partialSum[TILE_DIM];


    int tx = threadIdx.x;
    int bx = blockIdx.x;
    int i = bx * blockDim.x + tx;


    double diff = 0.0;
    if(i < N) {
        diff = Ax[i] - b[i];
        partialSum[tx] = diff * diff;
    } else {
        partialSum[tx] = 0.0;
    }
    __syncthreads();


    for(int i = blockDim.x / 2; i > 0; i /= 2) {
        if(tx < i) {
            partialSum[tx] += partialSum[tx + i];
        }
        __syncthreads();
    }


    if(tx == 0) {
        norm[blockIdx.x] = partialSum[0];
    }
}


void GPU_TILING_Jacobi_Method(double* A, double* b, double* x, int N, double
eps) {
    int threads_per_block = TILE_DIM;
    int blocks_per_grid = (N + threads_per_block - 1) / threads_per_block;


    double* d_A;
    double* d_x_now;
    double* d_x_next;
    double* d_b;
    double* d_Ax;
    double* d_partial_norm;
    cudaMalloc(&d_A, N * N * sizeof(double));
    cudaMalloc(&d_x_now, N * sizeof(double));
    cudaMalloc(&d_x_next, N * sizeof(double));
    cudaMalloc(&d_b, N * sizeof(double));
    cudaMalloc(&d_Ax, N * sizeof(double));
    cudaMalloc(&d_partial_norm, blocks_per_grid * sizeof(double));


    cudaMemcpy(d_A, A, N * N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_x_now, x, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N * sizeof(double), cudaMemcpyHostToDevice);
```

```
    double* x_prev = new double[N];
    double* partial_norm = new double[blocks_per_grid];
    int iteration = 0;


    while(true) {
        cudaMemcpy(x_prev, d_x_now, N * sizeof(double), cudaMemcpyDeviceToHost);
        jacobiTilingKernel<<<blocks_per_grid, threads_per_block>>>(d_x_next,
d_A, d_x_now, d_b, N);
        cudaMemcpy(x, d_x_next, N * sizeof(double), cudaMemcpyDeviceToHost);


        multiplyMatrixVectorKernel<<<blocks_per_grid, threads_per_block>>>(d_A,
d_x_next, d_Ax, N);
        computeNormKernel<<<blocks_per_grid, threads_per_block>>>(d_Ax, d_b,
d_partial_norm, N);


        cudaMemcpy(partial_norm, d_partial_norm, blocks_per_grid *
sizeof(double), cudaMemcpyDeviceToHost);
        double norm = 0.0;
        for(int i = 0; i < blocks_per_grid; ++i) {
            norm += partial_norm[i];
        }
        norm = sqrt(norm);


        iteration++;


        if(norm <= eps) {
            break;
        }


        cudaMemcpy(d_x_now, d_x_next, N * sizeof(double),
cudaMemcpyHostToDevice);
    }


    cout << "Iteration count: " << iteration;


    cudaFree(d_A);
    cudaFree(d_x_now);
    cudaFree(d_x_next);
    cudaFree(d_b);
    cudaFree(d_Ax);
    cudaFree(d_partial_norm);
    delete[] x_prev;
    delete[] partial_norm;
}
// ------------------------------------------------------------------------------
---------


// ------------------------------------ GENERATION --------------------------
---------
// Генерация матрицы с диагональным доминированием
void generateMatrix(double* A, int N) {
    double min = 0;
```

```
    double max = 50;
    for(int i = 0; i < N * N; ++i) {
        double tmp = (double)rand() / RAND_MAX; // значение в пределах [0; 1]
        A[i] = min + tmp * (max - min); // присваивание значения в пределах [0;
50]
    }


    for(int i = 0; i < N; ++i) {
        double sum = 0;
        for(int j = 0; j < N; ++j) {
            if(i != j) {
                sum += A[i * N + j];
            }
        }
        // A[i * N + i] += sum + 1; // диагональное доминирование
        A[i * N + i] = sum + 1; // диагональное доминирование
    }
}


// Генерация вектора
void generateVector(double* f, int N) {
    double min = 0;
    double max = 50;
    for(int i = 0; i < N; ++i) {
        double tmp = (double)rand() / RAND_MAX; // значение в пределах [0; 1]
        f[i] = min + tmp * (max - min); // присваивание значения в пределах [0;
50]
    }
}
// ------------------------------------------------------------------------------
---------


// ---------------------------------------- PRINT -------------------------------
--------
void printMatrix(double* A, int N, int precision) {
    cout << fixed << setprecision(precision); // устанавливаем точность вывода
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            cout << A[i * N + j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}


void printVector(double* v, int N, int precision) {
    cout << fixed << setprecision(precision); // устанавливаем точность вывода
    for(int i = 0; i < N; ++i) {
        cout << v[i] << " ";
    }
    cout << endl << endl;
}
// ------------------------------------------------------------------------------
--------


int main() {
    srand(time(NULL)); // инициализация генератора случайных чисел
```

```
    int N = 128; // размер матрицы и вектора
    double eps = 1e-5; // погрешность
    int precision = 10; // количество знаков после запятой в выводе


    double* A = new double[N * N]; // матрица A
    double* f = new double[N]; // вектор f
    double* x_cpu = new double[N](); // начальное приближение CPU_NO_TILING
(нулевой вектор)
    double* x_cpu_omp = new double[N](); // начальное приближение CPU_OMP
(нулевой вектор)
    double* x_gpu = new double[N](); // начальное приближение GPU_NO_TILING
(нулевой вектор)
    double* x_gpu_tiling_1 = new double[N](); // начальное приближение
GPU_NO_TILING (нулевой вектор)
    double* x_gpu_tiling_2 = new double[N](); // начальное приближение
GPU_NO_TILING (нулевой вектор)
    double* x_gpu_tiling_3 = new double[N](); // начальное приближение
GPU_NO_TILING (нулевой вектор)
    double* x_gpu_tiling_4 = new double[N](); // начальное приближение
GPU_NO_TILING (нулевой вектор)


    // События для измерения времени
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);


    // Генерация матрицы и вектора
    generateMatrix(A, N);
    generateVector(f, N);


    // Вывод времени
    printf("\nTIME:");


    // -------------------------------- CPU --------------------------------
    cudaEventRecord(start, 0); // старт таймера
    CPU_NO_TILING_Jacobi_Method(A, f, x_cpu, N, eps); // вызов метода Якоби


    float timerValueCPU;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPU, start, stop);
    printf("\n CPU         %f msec", timerValueCPU);
    // -------------------------------------------------------------------


    // -------------------------------- CPU-OMP --------------------------------
    cudaEventRecord(start, 0); // старт таймера
    CPU_Parallel_Jacobi_Method(A, f, x_cpu_omp, N, eps); // вызов метода Якоби


    float timerValueCPUOpenMP;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPUOpenMP, start, stop);
    printf("\n CPU(OpenMP) %f msec", timerValueCPUOpenMP);
    // -------------------------------------------------------------------------
```

```
    // -------------------------------- GPU --------------------------------
    cudaEventRecord(start, 0); // старт таймера
    GPU_NO_TILING_Jacobi_Method(A, f, x_gpu, N, eps); // вызов метода Якоби


    float timerValueGPU;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueGPU, start, stop);
    printf("\n GPU         %f msec", timerValueGPU);
    // ---------------------------------------------------------------------


    // -------------------------------- GPU-Tiling -----------------------------
---
    cudaEventRecord(start, 0); // старт таймера
    GPU_TILING_Jacobi_Method(A, f, x_gpu_tiling_1, N, eps); // вызов метода
Якоби


    float timerValueGPUTiling;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueGPUTiling, start, stop);
    printf("\n GPU(Tiling) %f msec", timerValueGPUTiling);
    // ---------------------------------------------------------------------------
---


    // Освобождение памяти
    delete[] A;
    delete[] f;
    delete[] x_cpu;
    delete[] x_cpu_omp;
    delete[] x_gpu;
    delete[] x_gpu_tiling_1;
    delete[] x_gpu_tiling_2;
    delete[] x_gpu_tiling_3;
    delete[] x_gpu_tiling_4;


    return 0;
}
```

ПРИЛОЖЕНИЕ Г

Код программы на языке C++ с использованием CUDA и OpenMP

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <omp.h>


using namespace std;


#define TILE_DIM 16



// ------------------------------------- PRINT ------------------------------
--------
void printMatrix(double* A, int N, int precision) {
    cout << fixed << setprecision(precision); // устанавливаем точность вывода
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < N; ++j) {
            cout << A[i * N + j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}


void printVector(double* v, int N, int precision) {
    cout << fixed << setprecision(precision); // устанавливаем точность вывода
    for(int i = 0; i < N; ++i) {
        cout << v[i] << " ";
    }
    cout << endl << endl;
}
// ------------------------------------------------------------------------
--------


// ---------------------------------- NORM-CALCULATION -----------------------
-------
void multiplyMatrixVector(double* A, double* x, double* result, int N) {
    for(int i = 0; i < N; ++i) {
        result[i] = 0;
        for(int j = 0; j < N; ++j) {
            result[i] += A[i * N + j] * x[j];
        }
    }
}


double computeNorm(double* Ax, double* b, int N) {
    double norm = 0;
    for(int i = 0; i < N; ++i) {
        norm += (Ax[i] - b[i]) * (Ax[i] - b[i]);
    }
    return sqrt(norm);
}
```

```
void multiplyMatrixVectorOMP(double* A, double* x, double* result, int N) {
    #pragma omp parallel for
    for(int i = 0; i < N; ++i) {
        result[i] = 0;
        for(int j = 0; j < N; ++j) {
            result[i] += A[i * N + j] * x[j];
        }
    }
}


double computeNormOMP(double* Ax, double* b, int N) {
    double norm = 0;
    #pragma omp parallel for reduction(+:norm)
    for(int i = 0; i < N; ++i) {
        norm += (Ax[i] - b[i]) * (Ax[i] - b[i]);
    }
    return sqrt(norm);
}
// ------------------------------------------------------------------------
-------


// ---------------------------------- CPU-NO-TILING ---------------------------
--------
void CPU_NO_TILING_Seidel_Method(double* A, double* f, double* x, int N, double
eps) {
    int iterations = 0;
    double* x_old = new double[N];
    double* Ax = new double[N]();


    while(true) {
        for(int i = 0; i < N; ++i) {
            x_old[i] = x[i];
        }


        for(int i = 0; i < N; ++i) {
            double sigma = 0.0;
            for(int j = 0; j < N; ++j) {
                if(j != i) {
                    sigma += A[i * N + j] * x[j];
                }
            }
            x[i] = (f[i] - sigma) / A[i * N + i];
        }

        iterations++;


        // Вычисляем Ax^(k)
        multiplyMatrixVector(A, x, Ax, N);


        // Проверка условия остановки
        if(computeNorm(Ax, f, N) <= eps) {
            break;
        }
    }


    cout << "\nNumber of iterations: " << iterations;
```

```
    delete[] x_old;
}
// ------------------------------------------------------------------------
--------


// ------------------------------------ CPU-OMP ------------------------------
------
void CPU_Parallel_Seidel_Method(double* A, double* f, double* x, int N, double
eps) {
    int iterations = 0;
    double* x_old = new double[N];
    double* Ax = new double[N]();


    while(true) {
        // Копируем текущее решение в x_old
        #pragma omp parallel for
        for(int i = 0; i < N; ++i) {
            x_old[i] = x[i];
        }


        // Основной цикл метода Зейделя
        #pragma omp parallel for
        for(int i = 0; i < N; ++i) {
            double sigma = 0.0;
            for(int j = 0; j < N; ++j) {
                if(j != i) {
                    sigma += A[i * N + j] * x[j];
                }
            }
            x[i] = (f[i] - sigma) / A[i * N + i];
        }


        iterations++;


        // Вычисляем Ax^(k)
        multiplyMatrixVectorOMP(A, x, Ax, N);


        // Проверка условия остановки
        if(computeNormOMP(Ax, f, N) <= eps) {
            break;
        }
    }


    std::cout << "\nNumber of iterations: " << iterations;
    delete[] x_old;
    delete[] Ax;
}
// ------------------------------------------------------------------------
------


// --------------------------------- GPU-No-Tiling ---------------------------
------
__global__ void seidel_no_tiling_Kernel(double* A, double* f, double* x, double*
x_old, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if(i < N) {
        double sigma = 0.0;
        for(int j = 0; j < N; ++j) {
            if(j != i) {
                sigma += A[i * N + j] * x[j];
            }
        }
        x[i] = (f[i] - sigma) / A[i * N + i];
    }
}


void GPU_NO_TILING_Seidel_Method(double* A, double* f, double* x, int N, double
eps) {
    int iterations = 0;
    double* x_old = new double[N];
    double* Ax = new double[N]();


    int threads_per_block = 16;
    int blocks_per_grid = (N + threads_per_block - 1) / threads_per_block;


    double *d_A, *d_f, *d_x, *d_x_old;
    cudaMalloc(&d_A, N * N * sizeof(double));
    cudaMalloc(&d_f, N * sizeof(double));
    cudaMalloc(&d_x, N * sizeof(double));
    cudaMalloc(&d_x_old, N * sizeof(double));
    cudaMemcpy(d_A, A, N * N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_f, f, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_x, x, N * sizeof(double), cudaMemcpyHostToDevice);


    while(true) {
        cudaMemcpy(d_x_old, d_x, N * sizeof(double), cudaMemcpyDeviceToDevice);
        seidel_no_tiling_Kernel<<<threads_per_block, blocks_per_grid>>>(d_A,
d_f, d_x, d_x_old, N);
        cudaDeviceSynchronize();


        iterations++;


        cudaMemcpy(x, d_x, N * sizeof(double), cudaMemcpyDeviceToHost);
        multiplyMatrixVector(A, x, Ax, N);
        if(computeNorm(Ax, f, N) <= eps) {
            break;
        }
    }


    std::cout << "\nNumber of iterations: " << iterations;


    cudaFree(d_A);
    cudaFree(d_f);
    cudaFree(d_x);
    cudaFree(d_x_old);
    delete[] x_old;
}
```

```
// ------------------------------------------------------------------------
------


// ------------------------------------ GENERATION --------------------------
---------
// Генерация матрицы с диагональным доминированием
void generateMatrix(double* A, int N) {
    double min = 0;
    double max = 50;
    for(int i = 0; i < N * N; ++i) {
        double tmp = (double)rand() / RAND_MAX; // значение в пределах [0; 1]
        A[i] = min + tmp * (max - min); // присваивание значения в пределах [0;
50]
    }


    for(int i = 0; i < N; ++i) {
        double sum = 0;
        for(int j = 0; j < N; ++j) {
            if(i != j) {
                sum += A[i * N + j];
            }
        }
        // A[i * N + i] += sum + 1; // диагональное доминирование
        A[i * N + i] = sum + 1; // диагональное доминирование
    }
}


// Генерация вектора
void generateVector(double* f, int N) {
    double min = 0;
    double max = 50;
    for(int i = 0; i < N; ++i) {
        double tmp = (double)rand() / RAND_MAX; // значение в пределах [0; 1]
        f[i] = min + tmp * (max - min); // присваивание значения в пределах [0;
50]
    }
}
// ------------------------------------------------------------------------
---------


int main() {
    srand(time(NULL)); // инициализация генератора случайных чисел

    int N = 512; // размер матрицы и вектора
    double eps = 1e-5; // погрешность
    int precision = 10; // количество знаков после запятой в выводе


    double* A = new double[N * N]; // матрица A
    double* f = new double[N]; // вектор f
    double* x_cpu = new double[N](); // начальное приближение CPU_NO_TILING
(нулевой вектор)
    double* x_cpu_omp = new double[N](); // начальное приближение CPU_OMP
(нулевой вектор)
    double* x_gpu = new double[N](); // начальное приближение GPU_NO_TILING
(нулевой вектор)


    // События для измерения времени
```

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);


    // Генерация матрицы и вектора
    generateMatrix(A, N);
    generateVector(f, N);


    // Вывод времени
    printf("\nTIME:");


    // --------------------------------- CPU ---------------------------------
    cudaEventRecord(start, 0); // старт таймера
    CPU_NO_TILING_Seidel_Method(A, f, x_cpu, N, eps); // вызов метода Якоби


    float timerValueCPU;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPU, start, stop);
    printf("\nCPU         %f msec", timerValueCPU);
    // -----------------------------------------------------------------------


    // --------------------------------- CPU-OMP ---------------------------------
    cudaEventRecord(start, 0); // старт таймера
    CPU_Parallel_Seidel_Method(A, f, x_cpu_omp, N, eps); // вызов метода Якоби


    float timerValueCPUOpenMP;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueCPUOpenMP, start, stop);
    printf("\nCPU(OpenMP) %f msec\n", timerValueCPUOpenMP);
    // -----------------------------------------------------------------------


    // --------------------------------- GPU ---------------------------------
    cudaEventRecord(start, 0); // старт таймера
    GPU_NO_TILING_Seidel_Method(A, f, x_gpu, N, eps); // вызов метода Якоби


    float timerValueGPU;
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&timerValueGPU, start, stop);
    printf("\nGPU         %f msec\n", timerValueGPU);
    // -----------------------------------------------------------------------


    // Освобождение памяти
    delete[] A;
    delete[] f;
    delete[] x_cpu;
    delete[] x_cpu_omp;
    delete[] x_gpu;


    return 0;
}
```