МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра вычислительной математики

БОБОВОЗ
Владислав Сергеевич

РЕАЛИЗАЦИЯ НА ГРАФИЧЕСКОМ УСКОРИТЕЛЕ МЕТОДА ГАУССА-ЗЕЙДЕЛЯ ЧИСЛЕННОГО РЕШЕНИЯ ДВУМЕРНЫХ ПАРАБОЛИЧЕСКИХ УРАВНЕНИЙ ГЕМОДИНАМИКИ

Дипломная работа

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

Допущена к защите

«___» _____ 2025 г.
Заведующий кафедрой
вычислительной математики
канд. физ.-мат. наук, доцент В.И. Репников

Минск, 2025

РЕФЕРАТ

Дипломная работа, 67 с., 8 источников, 3 рис., 5 таблицы, 4 приложения

Ключевые слова: ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ, МЕТОД ГАУССА-ЗЕЙДЕЛЯ, ГЕМОДИНАМИКА, GPU, CUDA

Объект исследования – метод Гаусса-Зейделя для двумерных параболических уравнений гемодинамики.

Цель работы – разработать псевдокод и код для последовательного, и код для параллельного (CUDA) алгоритмов метода Гаусса-Зейделя для численного решения, реализовать эти алгоритмы, сравнить и проанализировать результаты.

Методы исследования – технология CUDA, метод Гаусса-Зейделя, анализ эффективности.

Результаты работы: разработан псевдокод последовательного и код параллельного (CUDA) алгоритма Гаусса-Зейделя для численного решения двумерных параболических уравнений гемодинамики, реализованы эти алгоритмы, а также проведено сравнение полученных результатов.

РЭФЕРАТ

Дыпломная праца, 67 с., 8 крыніц, 3 мал., 5 табліцы, 4 дадаткі

Ключавыя словы: ПАРАЛЕЛЬНЫ АЛГАРЫТМ, МЕТАД ГАЎСА-ЗАЙДЭЛЯ, ГЕМАДЫНАМІКА, GPU, CUDA

Аб'ект даследавання – метад Гаўса-Зайдэля для двухвымерных парабаліч-ных раўнанняў гемадынамікі.

Мэта працы – распрацаваць псеўдакод і код для паслядоўнага, і код для паралельнага (CUDA) алгарытмаў метаду Гаўса-Зайдэля для лікавага развязку, рэалізаваць гэтыя алгарытмы, параўнаць і прааналізаваць вынікі.

Метады даследавання – тэхналогія CUDA, метад Гаўса-Зайдэля, аналіз эфектыўнасці.

Вынікі працы: распрацаваны псеўдакоды паслядоўнага і паралельнага (CUDA) алгарытму Гаўса-Зайдэля для лікавага развязку двухвымерных парабалічных раўнанняў гемадынамікі, рэалізаваны гэтыя алгарытмы, а таксама праведзена параўнанне атрыманых вынікаў.

ANNOTATION

Degree paper, 67 p., 8 sources, 3 figures, 5 tables, 4 appendices

Key words: PARALLEL ALGORITHM, GAUSS-SEIDEL METHOD, HEMODYNAMICS, GPU, CUDA

Object of research – Gauss-Seidel method for two-dimensional parabolic equations of hemodynamics.

Purpose of the work – to develop a pseudocode and code of sequential, and code of parallel (CUDA) algorithms of Gauss-Seidel method for numerical solution, to implement these algorithms, to compare and analyze the results.

Research methods – CUDA technology, Gauss-Seidel method, performance analysis.

Results of work: pseudocode of serial and code of parallel (CUDA) Gauss-Seidel algorithm for numerical solution of two-dimensional parabolic equations of hemodynamics was developed, these algorithms were implemented, and the results were compared.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 6
ГЛАВА 1 Теоретические сведения о CUDA. Основные понятия 7
ГЛАВА 2 Представление блочно-трехдиагональных систем уравнений с трехдиагональными блоками 11
2.1 Система двумерных параболических уравнений задачи гемодинамики 11
2.2 Блочно-трехдиагональная СЛАУ специального вида 12
2.3 Блочный последовательный алгоритм Гаусса-Зейделя в неявной форме для систем специального вида 13
2.4 Блочный параллельный алгоритм Гаусса-Зейделя в неявной форме для систем специального вида 18
ГЛАВА 3 Основные положения для распараллеливания алгоритмов при программировании с помощью CUDA 24
3.1 Декомпозиция задач на потоки и блоки 24
3.2 Минимизация узких мест, связанных с памятью 24
3.3 Использование разделяемой памяти и синхронизаций 25
3.4 Оптимизация вычислений (повторное использование данных, развертывание циклов, оптимизация расходования ресурсов на поток) 26
ГЛАВА 4 Применение CUDA технологии для решения СЛАУ специального вида методом Гаусса-Зейделя 28
4.1 Параллелизация блочного алгоритма Гаусса-Зейделя 28
4.2 Наиболее ресурсоемкие участки алгоритма 29
4.3 Синхронизация потоков внутри блока 30
4.4 Выбор параметров блоков и сетки в зависимости от N и M 30
ГЛАВА 5 Вычислительные эксперименты 32
ЗАКЛЮЧЕНИЕ 39
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 40
ПРИЛОЖЕНИЯ 41

ВВЕДЕНИЕ

Гемодинамика – наука о движении крови по сосудам, вызываемом разностью давления в различных участках кровеносной системы. Другими же словами, кровь течет из областей с высоким давлением в области с низким, а характер этого течения зависит от сопротивления сосудистых стенок и вязкости крови.

Моделирование гемодинамических процессов имеет большое практическое значение, т.к. сердечно-сосудистые заболевания остаются одной из главных причин смертей (согласно статистике, свыше 55% летальных исходов), а численное моделирование давления и потока крови позволяет исследовать различные сценарии, что в свою очередь способствует улучшению технологий диагностики и терапии.

Одной из основных проблем таких исследований является высокая трудоемкость вычислений. Это объясняется размерностью систем линейных алгебраических уравнений, которые необходимо решать. Именно высокая трудоемкость и ставит задачу разработки эффективных численных методов при использовании высокопроизводительных вычислений.

В данной работе рассматривается реализация метода Гаусса-Зейделя для численного решения двумерных параболических уравнений гемодинамики. Выбор метода Гаусса-Зейделя обусловлен его относительной простотой и пригодностью для решения больших разреженных СЛАУ, возникающих в данной задаче. Кроме того, стоит отметить, что неявная реализация позволяет значительно ускорить процесс за счет избегания дорогостоящей операции обращения матриц-блоков на каждом шаге. Рассмотренный вариант алгоритма Гаусса-Зейделя для неявных разностных схем обладает естественным параллелизмом и улучшенной устойчивостью по сравнению с явными разностными схемами.

Для ускорения вычислений выбрана технология CUDA и графические процессоры (GPU). Современные GPU способны выполнять тысячи операций параллельно, предоставляя колоссальную скорость для задач, легко поддающихся распараллеливанию. Благодаря массовому параллелизму графического оборудования, подобные симуляции могут работать значительно быстрее чем на CPU. В рассматриваемой задаче применение данных технологий поможет значительно сократить время расчета модели давления в сосудах, что особенно актуально при необходимости приближаться к режиму реального времени, либо для многократных экспериментов в области науки.

ГЛАВА 1
Теоретические сведения о CUDA. Основные понятия

Графические процессоры (GPU – Graphics Processing Unit) являются довольно мощными и широко используемыми многоядерными вычислительными устройствами. Например, NVIDIA L40S, созданная специально для вычисления, оснащенная 18176 CUDA-ядрами, имеет пиковую производительность 1466 терафлопс, что действительно мощно по сравнению с обычными видеокартами из ноутбука или стационарного компьютера. Вычислительные блоки GPU называются потоковыми мультипроцессорами (SM – stream multiprocessor).

Универсальные вычисления на GPU компании NVIDIA обеспечивает технология программирования CUDA (Compute Unified Device Architecture). CUDA предоставляет собой набор расширений для упрощенных языков C, C++, Fortran, позволяющих выражать параллелизм.

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.

Рассмотрим чуть более подробно описанные выше основные понятия.

1. Блоки вычислений.
   Операции исходного алгоритма объединяются в потоки (нити) вычислений, а потоки объединяются в блоки. Блоки же организованы в одномерную, двумерную или трехмерную сетку. Максимальные параметры размера (grid dimension) довольно большие, а более точно их можно указать зная конкретную модель видеокарты. Нумерация блоков вычислений по каждой координате сетки начинается с нуля, синхронизация между потоками разных блоков не предусмотрена. Каждый блок потоков выполняется атомарно на одном из мультипроцессоров, а один мультипроцессор способен одновременно выполнять N блоков.
2. Потоки вычислений.
   Блоки объединяют потоки (нити) вычислений, при этом потоки одного блока помечаются как узлы одномерной, двумерной или трехмерной сетки. Нумерация потоков вычислений по каждой

координате сетки начинается с нуля, а максимальное количество потоков в одном блоке напрямую зависит от конкретной модели видеокарты. При этом, все потоки выполняют одну программу, а каждый поток использует свои индексы для вычислений. Имеется также два механизма взаимодействия потоков внутри блока: разделяемая память и синхронизация (установка барьеров).

3. Варпы.

   Потоки одного блока выполняются на мультипроцессоре частями-пулами, называемыми варп (дословно: warp – канат). Потоки в блоке разбиваются на варпы автоматически согласно своему номеру: 32 подряд идущих потока образуют варп (два полуварпа по 16 потоков). Внутри пула варп потоки выполняются синхронно в режиме SIMT (Single Instruction Multiple Threads – в один момент времени каждый поток выполняет одну и ту же команду над разными данными). С точки зрения аппаратной части, минимальной единицей исполнения является не поток, а варп. Потоки варпа выполняются одновременно (с логической точки зрения, если процессоров в мультипроцессоре меньше чем 32).

   В каждом блоке параллельно (логически) выполняются все варпы. В CUDA каждый мультипроцессор содержит аппаратный блок планирования варпов, поэтому смена одного варпа на другой происходит практически без накладных расходов. Стоит отметить, что для большинства CPU-архитектур верно, что для оптимального быстродействия число ядер должно совпадать с количеством запущенных потоков; связано это с долгим процессом переключения ядра контекста процессора с одного потока на другой.

4. Ядро.

   Множество операций потоков всех независимо выполняемых блоков называют ядром. Для запуска ядра на GPU требуется задать размерность и размер сетки блоков, а также размерность и размер блока. Пока ядро выполняется на GPU, CPU может продолжать выполнять свой код, а может просто ждать, пока GPU закончит. Вычислительное ядро не включает в себя вспомогательные операции, такие, например, как операции по подготовке данных (которые исполняются центральным процессором вне GPU).

5. Обмены между блоками вычислений.

   Мультипроцессоры обмениваются информацией посредством оперативной памяти, называемой глобальной памятью. Глобальная память не кэшируется, работает медленно. Время ожидания варпом доступа к памяти может быть использовано для выполнения других

варпов. Для оптимального использования глобальной памяти желательно, чтобы количество потоков в блоке было кратным 64 и при этом не менее 192.

6. Обмены между потоками одного блока вычислений.
   Потоковые процессоры мультипроцессора взаимодействуют посредством быстрой разделяемой между этими процессорами памяти. Множественное обращение к банку разделяемой памяти порождает конфликты, которые возникают на уровне полуварпа, при этом задачи в разных полуварпах не конфликтуют по разделяемой памяти.

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 7 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.

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.

ГЛАВА 2
Представление блочно-трехдиагональных систем уравнений с трехдиагональными блоками

Пусть диагональные матрицы блочно-трехдиагональной системы линейных алгебраических уравнений (СЛАУ) являются трехдиагональными. Стоит уточнить, что это нередкий для практики частный случай. В частности, такие СЛАУ возникают на каждом временном слое при численном решении двумерных параболических уравнений гемодинамики для получения давления крови в кровеносных сосудах. Явные блочный алгоритм Гаусса-Зейделя включает обращение диагональных матриц-блоков и, на каждом шаге итерационного процесса, включает сначала умножение матриц-блоков на вектор, затем умножение плотных обратных матриц на вектор. Компоненты векторов, на которые умножаются матрицы-блоки, получены на предыдущем шаге, или, если они уже уточнены, на этом шаге.

Рассмотрим для решения такой системы блочный алгоритм Гаусса-Зейделя в неявной форме. В этом случае можно после умножения матриц-блоков на вектор решить (методом прогонки, например) трехдиагональную СЛАУ и обойтись без обращения диагональных матриц-блоков и умножения (на каждом итерационном шаге) плотных обратных матриц на вектор [1]. Алгоритм обладает естественным параллелизмом.

2.1 Система двумерных параболических уравнений задачи гемодинамики

Движение крови по сосуду (т.е. гемодинамику) можно описать системой уравнений в декартовой системе координат x, z:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{1}{\rho}\left(\frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}\right),$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{1}{\rho}\left(\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}\right),$$

$$\left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial z^2}\right) = 2\rho\left(\frac{\partial u}{\partial x}\frac{\partial w}{\partial z} - \frac{\partial u}{\partial z}\frac{\partial w}{\partial x}\right).$$

Здесь u – поперечная компонента скорости, w – продольная компонента скорости, p – давление, ρ – плотность крови, $\tau_{xx}, \tau_{xz}, \tau_{zx}, \tau_{zz}$ – некоторые функции.

Найти для требуемого t компоненты скорости u и w – отдельная задача. Найти давление p сводится к решению рассматриваемой далее системы линейных алгебраических уравнений специального вида (2.1).

2.2 Блочно-трехдиагональная СЛАУ специального вида

Пусть N, M – некоторые натуральные числа, а $C_1, \dots, C_N$, $A_2, \dots, A_N$, $B_1, \dots, B_{N-1}$ – вещественные матрицы порядка M, а в свою очередь $Y_1, , \dots, Y_N$, $F_1, , \dots, F_N$ – M-мерные векторы. Тогда, рассмотрим блочно-трехдиагональную систему линейных алгебраических уравнений вида [2, 3]:

$$\begin{cases} C_1 Y_1 + B_1 Y_2 = F_1 \\ A_i Y_{i-1} + C_i Y_i + B_i Y_{i+1} = F_i, \quad i = \overline{2, N-1}, \\ A_N Y_{N-1} + C_N Y_N = F_N \end{cases} \tag{2.1}$$

Будем рассматривать важный для практического использования в гемодинамике случай трехдиагональных матриц-блоков C_i:

$$C_i = \begin{pmatrix} c_1^{C_i} & b_1^{C_i} & 0 & 0 & \cdots \\ a_2^{C_i} & c_2^{C_i} & b_2^{C_i} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & a_{M-1}^{C_i} & b_{M-1}^{C_i} & c_{M-1}^{C_i} \\ \cdots & 0 & 0 & a_M^{C_i} & c_M^{C_i} \end{pmatrix}, \qquad i = \overline{1, N}$$

и диагональных матриц-блоков A_i, B_i:

$$A_i = \begin{pmatrix} c_1^{A_i} & 0 & 0 & 0 & \cdots \\ 0 & c_2^{A_i} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & 0 & c_{M-1}^{A_i} & 0 \\ \cdots & 0 & 0 & 0 & c_M^{A_i} \end{pmatrix}, \qquad i = \overline{2, N}$$

$$B_i = \begin{pmatrix} c_1^{B_i} & 0 & 0 & 0 & \cdots \\ 0 & c_2^{B_i} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & 0 & c_{M-1}^{B_i} & 0 \\ \cdots & 0 & 0 & 0 & c_M^{B_i} \end{pmatrix}, \qquad i = \overline{1, N-1}$$

Таким образом, матрица системы имеет порядок NM. Также видно, что матрица имеет блочно-трехдиагональную структуру, где на главной диагонали – N трехдиагональных матриц-блоков порядка M, а на поддиагонали и на наддиагонали – $N-1$ диагональных матриц-блоков порядка M.

Тогда, можем построить схему, которая бы наглядно иллюстрировала схему матрицы. Рассмотрим случай $N = 4,\ M = 4$. В результате получим схему, где значком «×» отмечены ненулевые элементы:

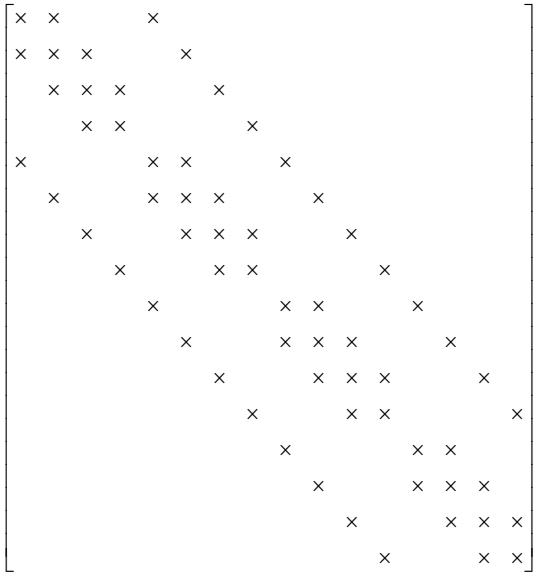

Рисунок 2.1 – Структура рассматриваемой матрицы (общий вид)

Блочные строки и столбцы имеют номера $1, \dots, N$, тогда вектор неизвестных:

$$(Y_1, \dots, Y_N) = \left(y_{1,1}, \dots, y_{1,M}, \dots, y_{N,1}, \dots, y_{N,M}\right),$$

а вектор правой части:

$$(F_1, \dots, F_N) = \left(f_{1,1}, \dots, f_{1,M}, \dots, f_{N,1}, \dots, f_{N,M}\right).$$

2.3 Блочный последовательный алгоритм Гаусса-Зейделя в неявной форме для систем специального вида

Пусть дана система линейных алгебраических уравнений вида $Ax = b$, где A – матрица порядка N, векторы b и x являются N-мерными векторами. Можем выбрать некоторое число r, причем $r < N$ и $n = \frac{N}{r}$ – целое число. Тогда можем разбить матрицу A на квадратные блоки размером $r \times r$, а вектор неизвестных и вектор правой части разобьем на блоки-векторы размерности r.

В свою очередь, блочный метод Гаусса-Зейделя решения системы $Ax = b$ имеет вид:

$$X_i^{l+1} = A_{i,i}^{-1}\left(B_i - \sum_{j=1}^{i-1} A_{i,j}X_j^{l+1} - \sum_{j=i+1}^{n} A_{i,j}X_j^{l}\right), \qquad i = \overline{1,n},\ l = 0,1,2,\dots$$

Получается, что каждая координата $(l+1)$-го приближения сразу после получения используется для вычисления (уточнения) следующих координат.

Тогда можем выписать блочный неявный метод Гаусса-Зейделя решения системы $Ax = b$:

$$A_{i,i}X_i^{l+1} = B_i - \sum_{j=1}^{i-1} A_{i,j}X_j^{l+1} - \sum_{j=i+1}^{n} A_{i,j}X_j^l, \qquad i = \overline{1,n},\ l = 0,1,2,\dots$$

Рассмотрим блочный неявный метод Гаусса-Зейделя решения систем вида (2.1):

$$\begin{cases} C_1Y_1^{l+1} = F_1 - B_1Y_2^l, \\ C_iY_i^{l+1} = F_i - A_iY_{i-1}^{l+1} - B_iY_{i+1}^l, \qquad i = \overline{2,N-1}, \\ C_NY_N^{l+1} = F_N - A_NY_{N-1}^{l+1}, \\ l = 0,1,2,\dots \end{cases}$$

Здесь $Y_1^0, \dots, Y_N^0$ – векторы начального приближения.

Пусть требуется выполнить L итераций метода. Для каждого фиксированного итерационного шага l уточняется каждая из N компонент приближенного вектора решения системы $(Y_1, \dots, Y_N)$, причем компонента Y_i, которая уточнена для очередного i, используется для пересчета следующих компонент. Запишем псевдокод, в котором при реализации блочного метода Гаусса-Зейделя используется только один массив неизвестных:

do $i = \overline{1,N}$
  $Y_i = Y_i^0$ // выбирается начально приближение
enddo
do $l = \overline{0,L-1}$
  do $i = \overline{1,N}$
    if $i = 1$ then $\Phi = F_1 - B_1Y_2^l$ // Φ – временный массив
    if $i \neq 1, i \neq N$ then $\Phi = F_i - A_iY_{i-1}^{l+1} - B_iY_{i+1}^l$
    if $i = N$ then $\Phi = F_N - A_NY_{N-1}^{l+1}$
    Решить СЛАУ $C_iY_i^{l+1} = \Phi$
  enddo
enddo(l)

При программировании существенным является нижний индекс, верхний индекс оставлен для наглядности. Тогда имеем:

do $i = \overline{1,N}$
  $Y_i = Y_i^0$ // выбирается начально приближение
enddo
do $l = \overline{0,L-1}$
  do $i = \overline{1,N}$
    if $i = 1$ then $\Phi = F_1 - B_1Y_2$ // Φ – временный массив
    if $i \neq 1, i \neq N$ then $\Phi = F_i - A_iY_{i-1} - B_iY_{i+1}$

if $i = N$ then $\Phi = F_N - A_N Y_{N-1}$

Решить СЛАУ $C_i Y_i = \Phi$

enddo

enddo(l)

Возникающие на каждом l-м шаге СЛАУ с трехдиагональными матрицами порядка M можно решить методом прогонки [2, 3].

Теория метода (правой) прогонки для решения системы с матрицей вида

$$\begin{pmatrix} c_1 & b_1 & 0 & 0 & \cdots \\ a_2 & c_2 & b_2 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & a_{M-1} & c_{M-1} & b_{M-1} \\ \cdots & 0 & 0 & a_M & c_M \end{pmatrix},$$

вектором неизвестных $(y_1, y_2, \dots, y_M)$ и вектором правой части $(\varphi_1, \varphi_2, \dots, \varphi_M)$ следующее:

Прямая прогонка – вычисление прогоночных коэффициентов по формулам

$$\alpha_2 = -\frac{b_1}{c_1}, \qquad \beta_2 = \frac{\varphi_1}{c_1},$$

$$\alpha_{k+1} = -\frac{b_k}{c_k + \alpha_k a_k}, \qquad \beta_{k+1} = \frac{\varphi_k - a_k \beta_k}{c_k + \alpha_k a_k}, \qquad k = \overline{2, M-1},$$

$$\beta_{M-1} = \frac{\varphi_M - a_M \beta_M}{c_M + \alpha_M a_M}.$$

После вычисления прогоночных коэффициентов, используется обратная прогонка – вычисление решения по формулам

$$y_M = \beta_{M+1}, \qquad y_k = \alpha_{k+1} y_{k+1} + \beta_{k+1}, \qquad k = \overline{M-1, 1}$$

Алгоритм 1. Умножение диагональной матрицы на вектор.

Вход: одномерные массивы

$c(k), 1 \le k \le M$ // элементы диагональной матрицы

$y(k), 1 \le k \le M$ // элементы вектора

Выход: одномерный массив

$d(k), 1 \le k \le M$ // элементы результирующей диагональной матрицы

Псевдокод:

dopar $k = 1, M$ // цикл dopar обозначает возможность (но не обязательность) параллельного выполнения

$d(k) = c(k) * y(k)$

enddopar

Алгоритм 2. Решение СЛАУ с трехдиагональной матрицей методом прогонки.

Вход: одномерные массивы

$a(k), 2 \le k \le M$ // элементы поддиагонали матрицы

$c(k), 1 \le k \le M$ // элементы главной диагонали матрицы

$b(k), 1 \le k \le M-1$ // элементы наддиагонали матрицы

$phi(k), 1 \le k \le M$ // элементы вектора правой части $(\varphi_1, \varphi_2, \dots, \varphi_M)$

Выход: одномерный массив

$y(k), 1 \le k \le M$ // элементы вектора решения $(y_1, y_2, \dots, y_M)$

Псевдокод:

// прямая прогонка:

$gamma = c(1)$ // $\gamma = c_1$

$alpha(2) = -\frac{b(1)}{gamma}$ // $\alpha_2 = -\frac{b_1}{c_1}$

$beta(2) = \frac{phi(1)}{gamma}$ // $\beta_2 = \frac{\varphi_1}{c_1}$

do $k = \overline{2, M-1}$

$gamma = c(k) + a(k)alpha(k)$ // $\gamma = c_k + a_k\alpha_k$

$alpha(k+1) = -\frac{b(k)}{gamma}$ // $\alpha_{k+1} = -\frac{b_k}{c_k + a_k\alpha_k}$

$beta(k+1) = \frac{phi(k) - a(k)beta(k)}{gamma}$ // $\beta_{k+1} = \frac{\varphi_k - a_k\beta_k}{c_k + a_k\alpha_k}$

enddo

$gamma = c(M) + a(M)alpha(M)$ // $\gamma = c_M + a_M\alpha_M$

$beta(M+1) = \frac{phi(M) - a(M)beta(M)}{gamma}$ // $\beta_{M-1} = \frac{\varphi_M - a_M\beta_M}{c_M + a_M\alpha_M}$

// обратная прогонка:

$y(M) = beta(M+1)$ // $y_M = \beta_{M+1}$

do $k = \overline{M-1, 1}$ // цикл с шагом -1

$y(k) = alpha(k+1)y(k+1) + beta(k+1)$ // $y_k = \alpha_{k+1}y_{k+1} + \beta_{k+1}$

enddo

Трудоемкость: примерно $5M$ операций деления и умножения

Алгоритм 3. Заданное число итераций блочного неявного метода Гаусса-Зейделя решения системы (2.1).

Вход: двумерные массивы

$cC(i,k), 1 \le i \le N, 1 \le k \le M$ // $cC(i,k) = c_k^{C_i}$

$aC(i,k), 1 \le i \le N, 2 \le k \le M$ // $aC(i,k) = a_k^{C_i}$

$bC(i,k), 1 \le i \le N, 1 \le k \le M-1$ // $bC(i,k) = b_k^{C_i}$

$cA(i,k), 1 \le i \le N, 1 \le k \le M$ // $cA(i,k) = c_k^{A_i}$

$cB(i,k), 1 \le i \le N, 1 \le k \le M$ // $cB(i,k) = c_k^{B_i}$

$fslae(i,k), 1 \le i \le N, 1 \le k \le M$ // элементы векторов F_i из правой части (2.1)

$Y_1^0, \dots, Y_N^0$ – векторы начального приближения

Выход: двумерный массив

$y(i,k), 0 \le i \le N, 1 \le k \le M$ // приближенное решение $(Y_1, \dots, Y_N)$ системы (2.1), при этом $Y_i = (y_{i,1}, \dots, y_{i,M})$

Псевдокод:

do $i = \overline{1,N}$

$Y_i = Y_i^0$ // выбирается начальное приближение

enddo

do $l = \overline{0,L-1}$

do $i = \overline{1,N}$

if $i = 1$ do // вычисление вектора $\Phi = F_1 - B_1 Y_2$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы B_1 с коэффициентами $cB(1,k), 1 \le k \le M$, на текущий вектор $Y_2 = (y_{2,1}, \dots, y_{2,M})$

$\Phi = F_1 - t$ // F_1 – вектор с элементами $fslae(1,k), 1 \le k \le M$

endif($i = 1$)

if $i \ne 1, i \ne N$ do // вычисление вектора $\Phi = F_i - A_i Y_{i-1} - B_i Y_{i+1}$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы A_i с коэффициентами $cA(i,k), 1 \le k \le M$, на текущий вектор $Y_{i-1} = (y_{i-1,1}, \dots, y_{i-1,M})$

$\Phi = F_i - t$ // вектор с элементами $fslae(1,k), 1 \le k \le M$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы B_i с коэффициентами $cB(i,k), 1 \le k \le M$, на текущий вектор $Y_{i+1} = (y_{i+1,1}, \dots, y_{i+1,M})$

$\Phi = \Phi - t$

endif($i \ne 1, i \ne N$)

if $i = N$ do // вычисление вектора $\Phi = F_N - A_N Y_{N-1}$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы A_N с коэффициентами $cA(N,k)$, $1 \le k \le M$ на текущий вектор $Y_{N-1} = (y_{N-1,1}, \dots, y_{N-1,M})$

$\Phi = F_N - t$ // F_N – вектор с элементами $fslae(N,k), 1 \le k \le M$,

endif($i = N$)

$Y_i = \cdots$ // вектор $(y_{i,1}, \dots, y_{i,M})$, полученный по алгоритму 2 решения методом прогонки СЛАУ с трехдиагональной матрицей C_i и правой частью Φ

enddo(i)

enddo(k)

Трудоемкость: сложность выполнения заданного числа L итераций блочного неявного метода Гаусса-Зейделя решения системы (2.1) – примерно $LN(2M + 5M) = 7LNM$ операций умножения и деления.

2.4 Блочный параллельный алгоритм Гаусса-Зейделя в неявной форме для систем специального вида

Алгоритм (2), как и алгоритм (3) можно использовать для получения реализации на графическом процессор. Для реализации алгоритма на GPU множество операций алгоритма должно быть разбито на блоки, а блоки в свою очередь – на потоки (нити) вычислений.

Пусть один поток выполняет одну итерацию цикла с параметром i, I^{block} – число потоков в одном блоке, причем это число одинаковое для всех блоков (I^{block} – параметр CUDA-реализации алгоритма). Тогда $Q^{block}I^{block} = N$, где Q^{block} – число блоков вычислений. Потоки одного блока выполняются мультипроцессорами GPU частями-пулами, называемыми варпами, по 32 потока. Поэтому, будем предполагать, что число $Q^{warp} = \frac{I^{block}}{32}$ является целым, а Q^{warp} – количество варпов в одном блоке.

При фиксированном параметре l внешнего цикла, независимо друг от друга можно выполнять итерации цикла с параметром i, определяющего потоки и блоки вычислений:

dopar $i = \overline{1,N}$
 $Y_i = Y_i^0 = \cdots$ // выбирается начальное приближение
enddopar
do $l = \overline{0,L-1}$
 запуск ядра // всего L запусков
 dopar $q = \overline{0,Q^{block}-1}$ // независимое выполнение Q^{block} блоков
  $Bl(q)$
 enddopar
enddo(l)

$Bl(q)$:
// Параллельное выполнение I^{block} потоков
do $i^{block} = \overline{0,I^{block}-1}$
 $Thr^q(i^{block})$
enddo

При этом, все потоки одного блока должны выполняться параллельно, но не обязательно, как в данном случае, независимо. Достаточно независимого выполнения только фрагментов потоков, затем после обмена данными (посредством разделяемой памяти и синхронизации), - последующих фрагментов. Поток вычислений $Thr^q(i^{block})$ q-го блока имеет следующий вид:

$i = qI^{block} + i^{block} + 1$ // наименьшее $i = 1$, а наибольшее $i = (Q^{block} - 1)I^{block} + (I^{block} - 1) + 1 = I^{block}Q^{block} = N$

if $i = 1$ then $\Phi = F_1 - B_1Y_2$ // Φ – временный массив

if $i \neq 1, i \neq N$ then $\Phi = F_i - A_iY_{i-1} - B_iY_{i+1}$

if $i = N$ then $\Phi = F_N - A_NY_{N-1}$

Решить СЛАУ $C_iY_i = \Phi$

На практике же, необходимо провести так называемое выделение используемых блоками массивов. В данном случае, необходимо выделить векторы Y_i, F_i, матрицы $C_i, A_i\ (i \neq 1), B_i\ (i \neq N), i \leq i \leq N$, и в каждом блоке начать нумерацию по i с нуля.

В q-м блоке вычисляются I^{block} векторов Y_i, $qI^{block} + 1 \leq i \leq qI^{block} + I^{block}$, при вычислениях используются Y_{i-1} (если $i \neq 1$) и Y_{i+1} (если $i \neq N$). Кроме того, в каждом q-м блоке, кроме 0-го и $(Q^{block} - 1)$-го, будем использовать векторы YQ_{iq} такие, что $YQ_{iq} = Y_i,\ iq = i - qI^{block},\ qI^{block} \leq i \leq qI^{block} + I^{block} + 1$. Это $I^{block} + 2$ векторов YQ_iq, т.е. нулевой вектор YQ_0 – для Y_{i-1} (при $i = qI^{block}$) и $(I^{block} + 1)$-й вектор $YQ_{I^{block}+1}$ – для Y_{i+1} (при $i = qI^{block} + I^{block} + 1$).

В 0-м блоке, фактически, используются $I^{block} + 1$ векторов YQ_{iq}: $YQ_{iq} = Y_i$, $iq = i,\ 1 \leq i \leq I^{block} + 1$, а нулевой вектор YQ_0 не используется. Тем не менее, будем для нулевого вектора YQ_0 резервировать память, т.е. считать что $0 \leq i \leq I^{block} + 1$. Тогда нумерация YQ_{iq} во всех блоках будет единообразной и, как следствие, код будет более читаемым.

В $(Q^{block} - 1)$-м блоке будем использовать $I^{block} + 1$ векторов YQ_{iq}: $YQ_{iq} = Y_i,\ iq = i - (Q^{block} - 1)I^{block},\ N - I^{block} \leq i \leq N$. Последнее двойное неравенство, с учетом $Q^{block}I^{block} = N$ означает, что $(Q^{block} - 1)I^{block} \leq i \leq (Q^{block} - 1)I^{block} + I^{block}$. Нулевой вектор YQ_0 – для Y_{i-1} (при $i = (Q^{block} - 1)I^{block} = N - I^{block}$), вектор Y_{i+1} отсутствует.

При выделении остальных массивов учтем, что в каждом блоке векторов F_i, матриц C_i, A_i (в 0-м блоке A_i отсутствует), B_i (в $(Q^{block} - 1)$-м блоке B_N отсутствует) столько же, сколько и вычисляется векторов Y_i. В q-м блоке, кроме 0-го для случая матриц A_i и $(Q^{block} - 1)$-го для случая матриц B_i, при вычислениях используются, до выделения массивов, по I^{block} векторов F_i, матриц $C_i, A_i, B_i, qI^{block} + 1 \leq i \leq qI^{block} + I^{block}$. В каждом q-м блоке, кроме 0-го и $(Q^{block} - 1)$-го, будем теперь использовать векторы FQ_{iq1}, матрицы $CQ_{iq1}, AQ_{iq1}, BQ_{iq1}$ такие, что $FQ_{iq1} = F_i,\ CQ_{iq1} = C_i,\ AQ_{iq1} = A_i,\ BQ_{iq1} = B_i,\ iq1 = i - qI^{block} - 1$, где $qI^{block} + 1 \leq i \leq qI^{block} + I^{block}$. Это по I^{block} векторов FQ_{iq} и матриц $CQ_{iq1},\ AQ_{iq1},\ BQ_{iq1}$ (тут $iq1$ на 1 меньше, чем iq для Y_i).

В 0-м блоке фактически используются $I^{block}-1$ матриц AQ_{iq1}: $AQ_{iq1}=A_i$, $iq1=i-1, 2\le i\le I^{block}$, при этом нулевая матрица AQ_0 не используется. Тем не менее, будем для нулевой матрицы AQ_0 резервировать память, т.е. считать $1\le i\le I^{block}$. Тогда нумерация AQ_{iq} во всех блоках будет единообразной.

В $(Q^{block}-1)$-м блоке будем использовать $I^{block}-1$ матриц BQ_{iq1}: $BQ_{iq1}=B_i,\ iq1=i-N+I^{block}-1$ ($iq1=i-(Q^{block}-1)I^{block}-1$, $Q^{block}I^{block}=N$), где $N-I^{block}+1\le i\le N-1$ (последнее двойное неравенство означает $(Q^{block}-1)I^{block}+1\le i\le (Q^{block}-1)I^{block}+I^{block}-1$).

Поток вычислений $Thr^q(i^{block})$ q-го блока после выделения массивов:

$iq=i^{block}+1$ // iq, при которых вычисляются YQ_{iq}; наименьшее в блоке такое $iq=1$, а наибольшее $iq=I^{block}$

if $q=0, iq=1$ then $\Phi=FQ_0-BQ_0YQ_2$

if $q\ne 0\ or\ \{q=0, iq>1\}$ then $\Phi=FQ_{i^{block}}-AQ_{i^{block}}YQ_{i^{block}}-BQ_{i^{block}}YQ_{iq+1}$

if $q\ne Q^{block}-1\ or\ \{q=Q^{block}-1,\ iq<I^{block}\}$ then $\Phi=FQ_{i^{block}}-AQ_{i^{block}}YQ_{i^{block}}-BQ_{i^{block}}YQ_{iq+1}$

if $q=Q^{block}-1,\ iq=I^{block}$ then $\Phi=FQ_{i^{block}}-AQ_{i^{block}}YQ_{i^{block}}$

Решить СЛАУ $C_{i^{block}}YQ_{iq}=\Phi$

Запишем вариант алгоритма для реализации на графическом ускорителе.

Алгоритм 3GPU. Заданное число итераций блочного неявного метода Гаусса-Зейделя решения системы.

Вход:

N, M – параметры СЛАУ, двумерные массивы

$cC(i,k), 1\le i\le N, 1\le k\le M$ // $cC(i,k)=c_k^{C_i}$

$aC(i,k), 1\le i\le N, 2\le k\le M$ // $aC(i,k)=a_k^{C_i}$

$bC(i,k), 1\le i\le N, 1\le k\le M-1$ // $bC(i,k)=b_k^{C_i}$

$cA(i,k), 1\le i\le N, 1\le k\le M$ // $cA(i,k)=c_k^{A_i}$

$cB(i,k), 1\le i\le N, 1\le k\le M$ // $cB(i,k)=c_k^{B_i}$

$fslae(i,k), 1\le i\le N, 1\le k\le M$ // элементы векторов F_i из правой части (2.1)

$Y_1^0, \dots, Y_N^0$ – векторы начального приближения, I^{block} – количество потоков в одном блоке вычислений. Предполагается, что число I^{block} одинаковое для всех блоков и число $\frac{I^{block}}{32}$ является целым.

Выход: двумерный массив

$y(i,k), 0\le i\le N, 1\le k\le M$ // приближенное решение $(Y_1, \dots, Y_N)$ системы (2.1), при этом $Y_i=(y_{i,1}, \dots, y_{i,M})$

Псевдокод:

dopar $i = \overline{1, N}$

$Y_i = Y_i^0 = \cdots$ // выбирается начальное приближение

enddopar

do $l = \overline{0, L-1}$

запуск ядра // всего L запусков

при этом, для q выделены массивы:

if $q = 0$ do

$Y_i, 0 \le i \le I^{block} + 1 : YQ_{iq} = Y_i, iq = i$

$F_i, C_i, A_i, B_i, 1 \le i \le I^{block}$:

$FQ_{iq1} = F_i, CQ_{iq1} = C_i, AQ_{iq1} = A_i, BQ_{iq1} = B_i, iq1 = i - 1$

endif

if $q \neq 0$ or $q \neq Q^{block} - 1$ do

$Y_i, qI^{block} \le i \le qI^{block} + I^{block} + 1 : YQ_{iq} = Y_i, iq = i - qI^{block}$

$F_i, C_i, A_i, B_i, qI^{block} + 1 \le i \le qI^{block} + I^{block}$:

$FQ_{iq1} = F_i, CQ_{iq1} = C_i, AQ_{iq1} = A_i, BQ_{iq1} = B_i, iq1 = i - qI^{block} - 1$

endif

if $q = Q^{block} - 1$ do

$Y_i, N - I^{block} \le i \le N : YQ_{iq} = Y_i, iq = i - N + I^{block}$

$F_i, C_i, A_i, N - I^{block} + 1 \le i \le N$:

$FQ_{iq1} = F_i, CQ_{iq1} = C_i, AQ_{iq1} = A_i, iq1 = i - N + I^{block} - 1$

$B_i, N - I^{block} + 1 \le i \le N - 1 : BQ_{iq1} = B_i, iq1 = i - N + I^{block} - 1$

endif

dopar $q = 0, Q^{block} - 1$ // независимое выполнение Q^{block} блоков

$Bl(q)$

enddopar

enddo(l)

Bl(q):

// параллельное выполнение I^{block} потоков

do $i^{block} = 0, I^{block} - 1$

$Thr^q(i^{block})$

enddo

$Thr^q(i^{block})$:

$iq = i^{block} + 1$ // iq, при которых вычисляются YQ_{iq} ($iq1 = i^{block}$) наименьшее такое iq в блоке равно 1, а наибольшее равно I^{block}

if $q = 0, iq = 1$ do // вычисление вектора $\Phi = FQ_0 - BQ_0 YQ_2 = F_1 - B_1 Y_2$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $BQ_0 = B_1$ с коэффициентами $cB(1,k), 1 \le k \le M$ на текущий вектор $YQ_2 = Y_2 = (y_{2,1}, \dots, y_{2,M})$

$\Phi = FQ_0 - t$ // $FQ_0 = F_1$ – вектор с элементами $fslae(1,k), 1 \le k \le M$

endif

if $q \ne 0\ or\ \{q = 0, iq > 1\}$ do // вычисление вектора $\Phi = FQ_{i^{block}} - AQ_{i^{block}}YQ_{i^{block}} - BQ_{i^{block}}YQ_{iq+1} = F_i - A_iY_{i-1} - B_iY_{i+1}$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $AQ_{i^{block}} = A_i$ с коэффициентами $cA(i,k), 1 \le k \le M$ на текущий вектор $YQ_{i^{block}} = Y_{i-1} = (y_{i-1,1}, \dots, y_{i-1,M})$

$\Phi = FQ_{i^{block}} - t$ // $FQ_{i^{block}} = F_i$ – вектор с элементами $fslae(i,k), 1 \le k \le M$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $BQ_{i^{block}} = B_i$ с коэффициентами $cB(i,k), 1 \le k \le M$ на текущий вектор $YQ_{iq+1} = Y_{i+1} = (y_{i+1,1}, \dots, y_{i+1,M})$

$\Phi = \Phi - t$

endif

if $q \ne Q^{block} - 1\ or\ \{q = Q^{block} - 1, iq < I^{block}\}$ do // вычисление вектора $\Phi = FQ_{i^{block}} - AQ_{i^{block}}YQ_{i^{block}} - BQ_{i^{block}}YQ_{iq+1} = F_i - A_iY_{i-1} - B_iY_{i+1}$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $AQ_{i^{block}} = A_i$ с коэффициентами $cA(i,k), 1 \le k \le M$ на текущий вектор $YQ_{i^{block}} = Y_{i-1} = (y_{i-1,1}, \dots, y_{i-1,M})$

$\Phi = FQ_{i^{block}} - t$ // $FQ_{i^{block}} = F_i$ – вектор с элементами $fslae(i,k), 1 \le k \le M$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $BQ_{i^{block}} = B_i$ с коэффициентами $cB(i,k), 1 \le k \le M$ на текущий вектор $YQ_{iq+1} = Y_{i+1} = (y_{i+1,1}, \dots, y_{i+1,M})$

$\Phi = \Phi - t$

endif

if $q = Q^{block} - 1, iq = I^{block}$ do // вычисление вектора $\Phi = FQ_{i^{block}} - AQ_{i^{block}}YQ_{i^{block}} = F_N - A_NY_{N-1}$

$t = \cdots$ // вектор, полученный по алгоритму 1 умножения диагональной матрицы $AQ_{i^{block}} = A_N$ с коэффициентами $cA(N,k), 1 \le k \le M$ на текущий вектор $YQ_{i^{block}} = Y_{N-1} = (y_{N-1,1}, \dots, y_{N-1,M})$

$\Phi = FQ_{i^{block}} - t$ // $FQ_{i^{block}} = F_N$ – вектор с элементами $fslae(N,k), 1 \le k \le M$

endif

$YQ_{iq} = Y_i = \cdots$ // вектор $(y_{i,1}, \dots, y_{i,M})$, полученный по алгоритму 2 решения методом прогонки СЛАУ с трехдиагональной матрицей $C_{i^{block}} = C_i$ и правой частью Ф.

ГЛАВА 3
Основные положения для распараллеливания алгоритмов при программировании с помощью CUDA

Для грамотного использования технологии CUDA стоит понимать основные положения и рекомендации. Рассмотрим некоторые из них.

3.1 Декомпозиция задач на потоки и блоки

Одним из основных и ключевых шагов при распараллеливании на CUDA является правильная декомпозиция задачи. Задачу следует разбивать на большое число независимых (или условно независимых) подзадач, чтобы их можно было выполнять параллельно.

Каждый поток (нить) обрабатывает свою часть данных или вычислений, а потоки группируются в блоки. Потоки внутри одного блока могут кооперироваться друг с другом (например, обмениваться данными через общую память и синхронизироваться). Это позволяет совместно решать подзадачи. В это же время, блоки выполняются независимо и параллельно друг от друга на разных мультипроцессорах GPU.

В вопросах выбора конфигурации потоков следует учитывать баланс нагрузки. Каждый поток должен выполнять сопоставимый объем работы, чтобы избежать ситуации, когда один поток стоит, а другие перегружены. Кроме этого, размер блоков (число потоков в блоке) обычно выбирают кратным размеру варпа. Это связано с архитектурой, т.к. варп – группа из 32 потоков, которая выполняет одну инструкцию одновременно. В то же время, неполный варп все равно занимает ресурсы, эквивалентные полному. Поэтому часто используют блоки размером 32, 64, 128 или же 256 потоков. Число блоков (размер сетки), подбирается так, чтобы покрыть весь объем данных. В идеале, общее число потоков (число блоков умножить на число потоков в блоке) должно значительно превышать количество ядер GPU, чтобы все ресурсы были задействованы параллельно.

3.2 Минимизация узких мест, связанных с памятью

Производительность алгоритмов на GPU очень часто зависит от пропускной способности и задержек доступа к памяти. Глобальная память (Global Memory) расположена в видеопамяти и довольно медленная, т.е. ее задержка доступа значительно выше, чем у регистров и разделяемой памяти. Для минимизации этого эффекта следует придерживаться нескольких рекомендаций:

- Коалесцированный доступ к памяти.

Необходимо располагать обращения потоков к глобальной памяти так, чтобы они были последовательными по адресам. Если потоки одного варпа обращаются к идущим подряд адресам в памяти, контроллер объединяет эти запросы в один широкий, вместо множества мелких. Это значительно повышает эффективность передачи данных. Так, например, если 32 потока варпа читают элементы с адресами 0,1,2,..,31, то данные будут переданы одной групповой операцией. Иначе, если адреса разбросаны (2, 5, 11, 17, …), то необходимо будет выполнять несколько операций в памяти. То есть можно сказать, что важно организовывать структуры данных и индексы потоков таким образом, чтобы обеспечивалась максимальная последовательность доступа к глобальной памяти.

- Максимальное использование пропускной способности через параллелизм.

  Даже если оптимизировать схему доступа, полностью избежать задержек памяти нельзя, поэтому важно чтобы в каждый момент было достаточно других активных вычислений. Это достигается путем, запуска множества потоков на каждом мультипроцессоре, чтобы во время когда одни ждут данные из памяти, другие выполняются. Для GPU это означает запуск как можно большего числа потоков, чтобы конвейер команд и запросов к памяти всегда был загружен. С практической стороны, это реализуется подбором оптимального числа потоков на блок, и блоков на мультипроцессор.
- Избежание конфликтов памяти.

  При использовании разделяемой памяти важно учитывать организацию банков памяти. Одновременный доступ нескольких потоков к разным данным, находящимся в одном банке, приводит к задержкам, или так называемым «конфликтам банков». По возможности, алгоритм должен быть спроектирован так, чтобы потоки обращались к разным банкам. Это довольно тонкая оптимизация, но в некоторых случаях она может очень сильно влиять на производительность.

3.3 Использование разделяемой памяти и синхронизаций

Разделяемая память – память, расположенная на чипе GPU, к которой имеют совместный доступ все потоки внутри одного блока. Она работает на порядок быстрее глобальной (примерно в 100 раз меньшая задержка по сравнению с некешированной глобальной памятью). По сути, разделяемая память представляет собой программируемый кеш, т.е. данные явно загружаются в нее

из глобальной памяти и затем многократно используются. Если алгоритм предполагает многократное обращение к одним и тем же данным, или обмен между потоками, имеет смысл воспользоваться разделяемой памятью. Если же какой-то элемент читается единожды и только своим потоком, копировать его в разделяемую память нет смысла.

Типичные применения разделяемой памяти:

- Буферизация.
  Часто алгоритмы обрабатывают данные блоками (тайлами). Например, при умножении матриц, каждый блок потоков загружает фрагмент матрицы в разделяемую память, после чего все потоки блока многократно используют эти значения, вычисляя результаты для своего подмножества выходной матрицы. Такой подход позволяет многократно переиспользовать данные из быстрой памяти вместо повторного чтения из глобальной.
- Кооперативные алгоритмы.
  Некоторые параллельные алгоритмы, такие как редукция, сортировка и т.п., требуют обмена данными между потоками. Разделяемая память предоставляет возможность для этого.
- Временное хранение результатов для последующих итераций.
  Если алгоритм итеративный, потоки могут сохранять промежуточные результаты в разделяемую память, чтобы использовать их на следующем шаге вместо того, чтобы каждый раз выгружать в глобальную память.

Для корректности использования общей памяти нужна синхронизация потоков. Несмотря на то, что потоки в блоке логически работают параллельно, на практике они используются варпами. Без явной синхронизации один поток может попытаться прочитать данные из разделяемой памяти, которые другой поток еще не успел туда записать.

Чтобы предотвратить такие проблемы, в CUDA имеется механизм __syncthreads(). Когда поток достигает ее вызова, он останавливается и выполнение продолжится только после того, как все потоки блока достигнут этой точки.

3.4 Оптимизация вычислений (повторное использование данных, развертывание циклов, оптимизация расходования ресурсов на поток)

Помимо оптимизации доступа к памяти, значительную роль играет оптимизация самих вычислений внутри ядра. Есть несколько подходов:

- Повторное использование данных.

Этот принцип уже отчасти описан выше: если одни и те же данные нужны для нескольких операций, их стоит загрузить один раз и хранить локально (в регистре или разделяемой памяти), а не запрашивать каждый раз из памяти.

- Развертывание циклов.
  Развертывание – это замена цикла набором повторяющихся команд. Эта операция убирает затраты на управление циклом и может улучшить работоспособность. Особенно эффективно развертывание для коротких циклов с известным числом итераций. Но нужно очень осторожно подходить к этому вопросу, так как избыточное развертывание увеличивает размер кода и потребление ресурсов.
- Оптимизация расходования ресурсов на поток.
  Если каждый поток выполняет слишком мало работы, накладные расходы (создание потока, переключение контекста, простаивание) могут доминировать. Поэтому иногда нужно увеличивать работу на поток путем увеличения последовательной работы на поток.

ГЛАВА 4
Применение CUDA технологии для решения СЛАУ специального вида методом Гаусса-Зейделя

Рассмотрим здесь применение выше описанных принципов при решении нашей конкретной задачи. Алгоритм решения использует итерационный метод Гаусса-Зейделя блочного типа, т.е. на каждой итерации последовательно уточняются решения для каждого блочного ряда. Прямая реализация такого алгоритма на CPU выполняет обновление блоков последовательно (что обеспечивает использование новых значений немедленно), но на GPU столь последовательный подход неэффективен. Ниже описано, как алгоритм модифицируется и оптимизируется для параллельного выполнения на графическом процессоре.

4.1 Параллелизация блочного алгоритма Гаусса-Зейделя

В блочном алгоритме Гаусса-Зейделя для блочно-трехдиагональной СЛАУ присутствует зависимость между вычислениями в соседних блоках. Например, при обновлении i-го блочного неизвестного используются данные из i-1 (предыдущего) и i+1 (следующего) блоков. Однако, можно разделить обновление блоков на группы, которые не имеют непосредственной зависимости друг от друга, а обрабатывать их параллельно.

Предположим, что выбрано Q блоков для покрытия блочных строк. Тогда, в каждом запуске CUDA ядра распараллеливается обработка Q блочных строк, т.е. запускается сетка из Q блоков, и блок с индексом i^{gl} отвечает за вычисления для блочных строк с номерами примерно от i^{gl} до $D * i^{gl+1}$, где $D = \frac{N}{Q}$. Внутри каждого такого блока GPU работает несколько потоков (нитей), а их число равно L или близко к тому. Обычно выбирают количество потоков равным размеру группы блочных строк. В упоминавшемся алгоритме, например, каждая группа содержит 32 строки ($L = 32$), и внутри GPU-блока запускаются 32 потока, каждый из которых обрабатывает одну блочную строку системы. Поток с индексом T внутри блока обрабатывает блочную строку с глобальным номером $i = 32 * i^{gl} + T + 1$.

Однако полностью независимого выполнения, по сути нет, так как метод Гаусса-Зейделя предполагает использование обновленных значений соседних блоков на той же итерации. Чтобы реализовать это в параллельном режиме, применяется разбиение вычислений на фазы. Идея заключается в том, что внутри одного блока потоки выполняют часть вычислений для своих строк независимо, не требуя еще данных соседей, затем останавливаются и обмениваются полученной информацией, а после этого продолжают вычисление следующей

части с учетом обмененных данных. В нашем случае каждый поток сначала может выполнить ту часть обновления, которая опирается только на внутренние данные блочного уравнения или данные с предыдущей итерации. Затем результаты, необходимые соседним строкам, выкладываются в разделяемую память, и выполняется синхронизация потоков блока (__syncthreads()). После, каждый поток считывает из разделяемой памяти значения, вычисленные его соседями, и завершает вычисления для своей строки.

4.2 Наиболее ресурсоемкие участки алгоритма

К наиболее ресурсоемким операциям данного алгоритма относятся:

- Доступ к памяти для вектора неизвестных и коэффициентов.
  Суммарный объем данных всей задачи – матрица, имеющая при оценке сверху около $5NM$ ненулевых элементов. На каждой итерации происходит чтение коэффициентов и чтение/запись текущего приближения для решения. Эти обращения к глобальной памяти тоже могут портить общую картину, если это не оптимизировано. В рассмотренной реализации, доступ устроен достаточно эффективно: каждый поток обращается к своему участку массивов коэффициентов, причем последовательно, что как правило коалесцирует память. При этом, обмен значениями Y между соседними блоками происходит через разделяемую или глобальную память с относительно небольшим объемом данных.
- Решение локальных систем для каждого блочного ряда.
  Вспомним, что исходная задача сводится к решению блочно-трехдиагональной СЛАУ размер $N \times N$ с блоками $M \times M$. Для каждого блочного уравнения на итерации Гаусса-Зейделя требуется решить трехдиагональную систему размерности M (диагональный блок – трехдиагональная матрица $M \times$M). В явном блочном алгоритме для этого приходилось обращать матрицы-блоки и умножать их на вектор, что очень затратно. В оптимизированной версии (неявный алгоритм), вместо обращения выполняется прямое решение трехдиагональной СЛАУ, например, методом прогонки.
- Умножение матриц-блоков на векторы.
  Для формирования правых частей требуются результаты с соседних блоков: выражение вида $A_i Y_{i-1}$ и $B_i Y_{i+1}$, где A_i и B_i – под- и наддиагональные блоки, Y_{i-1}, Y_{i+1} – решения соседних блочных строк. Вычисление этих произведений тоже достаточно ресурсоемкий процесс. Каждое такое умножение порядка $O(M)$ операций, а суммарно по всем блокам $O(N * M)$ операций.

4.3 Синхронизация потоков внутри блока

Критически важным элементом корректности алгоритма Гаусса-Зейделя является внутриблочная синхронизация. Как упоминалось выше, в каждом блоке потоки должны выполнять части работы последовательно с точки зрения алгоритма, несмотря на общее параллельное выполнение. Для этого в CUDA используется синхронизация __syncthreads().

В нашем случае, каждый блок решает группу из D блочных уравнений. Эти уравнения являются последовательно зависимыми, т.е. результат i-ой строки влияет на следующую, $(i+1)$-ую строку в рамках одной группы. Это решается двухфазным выполнением, что требует синхронизацию между ними. В начале блока потоки параллельно вычисляют предварительные значения, а после того как i-й поток вычислил временное значение, он записывает это значение в разделяемую память, которая доступна всем потокам. После этого вызывается __syncthreads(), который останавливает каждый поток до тех пор, пока все остальные не сделают то же самое. В результате, когда выполнение возобновляется, мы имеем гарантию, что массив заполнен: для каждый i-ой строки там лежит обновленная информация. Тогда потоки могут начать вторую фазу. Суть второй фазы заключается в том, что, например, i-й поток считывает из разделяемой памяти значения от предыдущего $(i-1)$-го потока и завершает вычисление своей строчки.

Очень важно правильно выбрать расположение вызовов __syncthreads(). Они должны быть вне каких-то ветвлений, зависящих от индекса потока, чтобы все потоки дошли до барьера синхронно. Для рассматриваемого алгоритма это уже выполнено.

Также, важно отметить, что синхронизация действует только внутри одного блока. Нельзя напрямую синхронизировать все блоки где-то в середине ядра, поэтому параллельные блоки в рассматриваемой задаче синхронизируются не явным барьером, а неявным. Когда ядро отработало, мы уверены что все блоки закончили свои группы, и можно начать следующую итерацию, т.е. новый запуск ядра. На границе этих запуском и происходит глобальная синхронизация.

4.4 Выбор параметров блоков и сетки в зависимости от N и M

Параметры конфигурации запуска CUDA-ядер оказывают значительное влияние на производительность и эффективность. Очень важно выбирать их с учетом размеров задачи, в рассматриваемом случае, учитывая N блочных строк и размер M каждого блока. Рассмотрим более подробно, что лежит в основе выбора конфигурации:

- Размер блока (число потоков в блоке).
  В рассматриваемой реализации, размер блока был выбран равным 32.

Это обусловлено двумя причинами:

1. Уже известно, что 32 потока составляют 1 варп. Таким образом, блок исполняется без избыточной разбивки на несколько варпов. Все потоки идут синхронно, что упрощает логику обмена данными. Кроме этого, размер блока кратен 32, а это тоже соответствует общим рекомендациям эффективного использования.
2. Стоит также отметить, что 32 выбрано еще и из-за соответствия «1 поток = 1 блочная строка», что значительно упрощает отображения алгоритма на потоки.

Однако, выбор не является жестко фиксированным, т.е. при других условиях N, M можно было бы рассмотреть другие размеры. Например, если M небольшое число, а N очень большое, то можно увеличить число потоков на блок до 64 или 128, опять же, чтобы было кратно 32. Это уже было подробно описано выше. Но в общем случае, при типичных N, M, блок в 32 или 64 потока будет являться хорошим компромиссом, т.к. он дает достаточную параллелизацию, не требуя слишком большого объема разделяемой памяти.

- Размер сетки (число блоков).

  Размер сетки обычно выбирается исходя из размера N и выбранного размера блока: $gridDim = ceil\left(\frac{N}{threadsPerBlock}\right)$. В нашем случае, при 32 потоках на блок, и N блочных строк, число блоков примерно равно $\frac{N}{32}$ (округление вверх).

  Когда N велико, число блоков будет большим, что очень хорошо для GPU, потому что он сможет запустить их на все доступные SM параллельно. Например, если $N = 1024$, а блок размером в 32, мы получим 32 блока, которые распределяться по 2 блока на SM, если у GPU условные 16 мультипроцессоров. Это обеспечит хорошую загрузку.

  Если же N невелико, например $N = 30$ (что меньше 32), у нас получится всего 1 блок на весь запуск. Это значит что параллелизм будет ограничен, а некоторые ресурсы останутся неиспользованными.

ГЛАВА 5
Вычислительные эксперименты

Перед началом экспериментов, приведем характеристики графического ускорителя:

Параметр	Значение
Версия драйвера CUDA	12.8
Архитектура CUDA (Compute Capability)	8.6
Объем глобальной памяти	6144 МБ (6 ГБ)
Количество мультипроцессоров (SM)	30
Количество CUDA-ядер	3840 (по 128 ядер на 1 SM)
Размер L2 кеша	3 МБ
Максимум потоков на SM	1536
Максимум потоков в одном блоке	1024
Максимальные размеры блока (x, y, z)	(1024, 1024, 64)
Максимальные размеры сетки (x, y, z)	(2147483647, 65535, 65535)

Таблица 5.1 – Характеристики используемого графического ускорителя RTX 3060.

При проведении вычислительных экспериментов будем выполнять заданное число итераций, т.е. не будем достигать критерия остановки. В качестве вектора Y^0 можно брать, например, правую часть системы уравнений, или же нулевой вектор.

Рассмотрим первый вычислительный эксперимент. В качестве вектора Y^0 будем брать нулевой вектор.

Элементы остальных матриц-блоков системы можно задать так, чтобы получилось строгое диагональное преобладание.

Заполним матрицу следующим образом:

все диагональные элементы диагональных матриц-блоков равны 4:

$$cC(i,k) = 4, \qquad 1 \le i \le N, \qquad 1 \le k \le M;$$

другие элементы:

$$aC(i,k) = \frac{2i+k}{2N+M}, \qquad 1 \le i \le N, \qquad 2 \le k \le M;$$

$$bC(i,k) = \frac{2i+k}{2N+M}, \qquad 1 \le i \le N, \qquad 1 \le k \le M-1;$$

$$cA(i,k) = \frac{2i+k}{2N+M}, \qquad 1 \le i \le N, \qquad 1 \le j \le M;$$

$$cB(i,k) = \frac{2i+k}{2N+M}, \qquad 1 \le i \le N, \qquad 1 \le j \le M;$$

Будем считать, что решение системы такое:

$$(y_{1,1}, \dots, y_{1,M}, \dots, y_{N,1}, \dots, y_{N,M}) = (1,1, \dots, 1).$$

Тогда правая часть системы $fslae(i,k), 1 \le i \le N,\ 1 \le k \le M$ – произведение матрицы системы и вектора $(1,1, \dots, 1)$. Можно воспользоваться следующим утверждением:

Пусть A – матрица системы вида (2.1), а X – вектор вида:

$$X = (X_1, \dots, X_N) = (x_{1,1}, \dots, x_{1,M}, \dots, x_{N,1}, \dots, x_{N,M}).$$

Тогда элементы вектора $Z = AX$:

$$Z = (Z_1, \dots, Z_N) = (z_{1,1}, \dots, z_{1,M}, \dots, z_{N,1}, \dots, z_{N,M})$$

Могут быть вычислены следующим образом:

Вход: двумерные массивы

$cC(i,k),\ 1 \le i \le N,\ 1 \le k \le M$ // $cC(i,k) = c_k^{C_i}$

$aC(i,k),\ 1 \le i \le N,\ 2 \le k \le M$ // $aC(i,k) = a_k^{C_i}$

$bC(i,k),\ 1 \le i \le N,\ 1 \le k \le M-1$ // $bC(i,k) = b_k^{C_i}$

$cA(i,k),\ 1 \le i \le N,\ 1 \le k \le M$ // $cA(i,k) = c_k^{A_i}$

$cB(i,k),\ 1 \le i \le N,\ 1 \le k \le M$ // $cB(i,k) = c_k^{B_i}$

$x(i,k),\ 1 \le i \le N,\ 1 \le k \le M$

Выход: двумерный массив

$z(i,k),\ 1 \le i \le N,\ 1 \le k \le M$

Псевдокод:

do $i = \overline{1,N}$:

if $i = 1$ do:

$z_{i,1} = c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

do $k = \overline{2,M-1}$:

$z_{i,k} = a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i = 1$)

if $i \ne 1, i \ne N$ do:

$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

do $k = \overline{2,M-1}$:

$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i \ne 1, i \ne N$)

if $i = N$ do:

$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2}$
do $k = \overline{2, M-1}$:
$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1}$
enddo
$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M}$
endif($i = N$)
enddo

Тогда, можем реализовать последовательный и параллельный алгоритмы метода Гаусса-Зейделя для системы описанного выше вида. Код программы приведен в приложении А, Б.

Рассмотрим $N, M = 128, 128$ соответственно. Это значит, что матрица будет содержать $N = 128$ блоков в строке и столбце, и $M = 128$ строк и столбцов в блоке.

Число итераций	Число потоков	Послед. метод, сек	Паралл. метод, сек	Наилучшее ускорение (последовательный к параллельному)
8	16	0.0071064	0.020672	0.3438
	32		0.021952	
	64		0.021974	
16	16	0.0184338	0.038556	0.4781
	32		0.039401	
	64		0.039792	
32	16	0.0386533	0.065165	0.5931
	32		0.069154	
	64		0.073250	

Таблица 5.2 – Время выполнения программы при фиксированном $N, M = 128, 128$ соответственно.

Рассмотрим $N, M = 512, 512$ соответственно. Это значит, что матрица будет содержать $N = 512$ блоков в строке и столбце, и $M = 512$ строк и столбцов в блоке.

Число итераций	Число потоков	Послед. метод, сек	Паралл. метод, сек	Наилучшее ускорение (последовательный к параллельному)
16	16	0.205872	0.129723	1.5870

	32		0.141701	
	64		0.143524	
32	16	0.416178	0.214211	1.9620
	32		0.227436	
	64		0.245922	
64	16	1.39038	0.390137	3.5638
	32		0.412896	
	64		0.442887	

Таблица 5.3 – Время выполнения программы при фиксированном $N, M = 512, 512$ соответственно.

Рассмотрим $N, M = 1024, 1024$ соответственно. Это значит, что матрица будет содержать $N = 1024$ блока в строке и столбце, и $M = 1024$ строк и столбцов в блоке.

Число итераций	Число потоков	Послед. метод, сек	Паралл. метод, сек	Наилучшее ускорение (последовательный к параллельному)
16	16	0.79605	0.235103	3.3859
	32		0.239242	
	64		0.252268	
32	16	2.65437	0.428167	6.1993
	32		0.440775	
	64		0.466778	
64	16	5.41033	0.771089	7.0164
	32		0.811737	
	64		0.842981	

Таблица 5.4 – Время выполнения программы при фиксированном $N, M = 1024, 1024$ соответственно.

Сделаем выводы о полученных результатах первого вычислительного эксперимента.

Если проанализировать полученные в таблице 5.2, можно явно заметить, что для систем таких размерностей не имеет смысла тратить время и ресурсы на реализацию данного алгоритма на GPU. Последовательная реализация на порядок быстрее, что можно объяснить тем, что на маленьких матрицах почти вся работа относится к «служебным» затратам GPU. В них входит запуск ядер и копирование данных.

После анализа результатов таблицы 5.3 и 5.4 становится понятно, что смысл реализовывать данный алгоритм на GPU появляется только для матриц больших размерностей. На таких размерах описанные выше «служебные» затраты значительно уменьшаются из-за более равномерной нагрузки.

Также, важно оптимальным образом выбирать число потоков. Это стоит делать таким образом, чтобы GPU был полностью загружен.

Проведем второй вычислительный эксперимент. Заполним элементы матриц-блоков следующим образом:

Внутренние строки внутренних блочных строк, $2 \le i \le N-1, 2 \le k \le M-1$:

$$cC(i,k) = -4, aC(i,k) = 1, bC(i,k) = 1, cA(i,k) = 1, cB(i,k) = 1.$$

Первая строка внутренних блочных строк, $2 \le i \le N-1, k = 1$:

$$cC(i,k) = -5, bC(i,k) = 1, cA(i,k) = 1, cB(i,k) = 1.$$

Последняя строка внутренних блочных строк, $2 \le i \le N-1, k = M$:

$$cC(i,k) = -5, aC(i,k) = 1, cA(i,k) = 1, cB(i,k) = 1.$$

Внутренние строки первой блочной строки, $i = 1, 2 \le k \le M-1$:

$$cC(i,k) = -3, aC(i,k) = 1, bC(i,k) = 1, cB(i,k) = 1.$$

Первая строка первой блочной строки, $i = 1,\ k = 1$:

$$cC(i,k) = -4, bC(i,k) = 1, cB(i,k) = 1.$$

Последняя строка первой блочной строки, $i = 1,\ k = M$:

$$cC(i,k) = -4, aC(i,k) = 1, cB(i,k) = 1.$$

Внутренние строки последней блочной строки, $i = N,\ 2 \le k \le M-1$:

$$cC(i,k) = -3, aC(i,k) = 1, bC(i,k) = 1, cA(i,k) = 1.$$

Первая строка последней блочной строки, $i = N,\ k = 1$:

$$cC(i,k) = -4, bC(i,k) = 1, cA(i,k) = 1.$$

Последняя строка последней блочной строки, $i = N,\ k = M$:

$$cC(i,k) = -4, aC(i,k) = 1, cA(i,k) = 1.$$

Правая часть системы: $fslae(i,k) = -2, k = 0\ or\ k = M-1$ и $fslae(i,k) = 0, else.$

Решение системы:

$$(y_{1,1}, \dots, y_{1,M}, \dots, y_{N,1}, \dots, y_{N,M}) = (1,1, \dots, 1).$$

Тогда, можем реализовать последовательный и параллельный алгоритмы метода Гаусса-Зейделя для системы описанного выше вида. Код программы приведен в приложении В, Г.

Приведем расширенную матрицу системы для данного примера, для случая $N = 4,\ M = 4$:

$$\left(\begin{array}{cccccccccccccccc|c}
-4 & 1 & & & 1 & & & & & & & & & & & & -2 \\
1 & -3 & 1 & & & 1 & & & & & & & & & & & 0 \\
 & 1 & -3 & 1 & & & 1 & & & & & & & & & & 0 \\
 & & 1 & -4 & & & & 1 & & & & & & & & & -2 \\
1 & & & & -5 & 1 & & & 1 & & & & & & & & -2 \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & & 0 \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & 0 \\
 & & & 1 & & & 1 & -5 & & & & 1 & & & & & -2 \\
 & & & & 1 & & & & -5 & 1 & & & 1 & & & & -2 \\
 & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & & 0 \\
 & & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & 0 \\
 & & & & & & & 1 & & & 1 & -5 & & & & 1 & -2 \\
 & & & & & & & & 1 & & & & -4 & 1 & & & -2 \\
 & & & & & & & & & 1 & & & 1 & -3 & 1 & & 0 \\
 & & & & & & & & & & 1 & & & 1 & -3 & 1 & 0 \\
 & & & & & & & & & & & 1 & & & 1 & -4 & -2
\end{array}\right)$$

Рисунок 5.1 – Структура рассматриваемой матрицы при $N = 4,\ M = 4$.

Применим предобусловливание Якоби: разделим каждый коэффициент СЛАУ на диагональный элемент матрицы. Таким образом, расширенная матрица полученной системы для данного примера изображена на следующем рисунке:

$$\left(\begin{array}{cccccccccccccccc|c}
1 & -\frac{1}{4} & & & -\frac{1}{4} & & & & & & & & & & & & \frac{1}{2} \\
-\frac{1}{3} & 1 & -\frac{1}{3} & & & -\frac{1}{3} & & & & & & & & & & & \frac{0}{1} \\
 & -\frac{1}{3} & 1 & -\frac{1}{3} & & & -\frac{1}{3} & & & & & & & & & & \frac{0}{1} \\
 & & -\frac{1}{4} & 1 & & & & -\frac{1}{4} & & & & & & & & & \frac{1}{2} \\
-\frac{1}{5} & & & & 1 & -\frac{1}{5} & & & -\frac{1}{5} & & & & & & & & \frac{2}{5} \\
 & -\frac{1}{4} & & & -\frac{1}{4} & 1 & -\frac{1}{4} & & & -\frac{1}{4} & & & & & & & \frac{0}{1} \\
 & & -\frac{1}{4} & & & -\frac{1}{4} & 1 & -\frac{1}{4} & & & -\frac{1}{4} & & & & & & \frac{0}{1} \\
 & & & -\frac{1}{5} & & & -\frac{1}{5} & 1 & & & & -\frac{1}{5} & & & & & \frac{2}{5} \\
 & & & & -\frac{1}{5} & & & & 1 & -\frac{1}{5} & & & -\frac{1}{5} & & & & \frac{2}{5} \\
 & & & & & -\frac{1}{4} & & & -\frac{1}{4} & 1 & -\frac{1}{4} & & & -\frac{1}{4} & & & \frac{0}{1} \\
 & & & & & & -\frac{1}{4} & & & -\frac{1}{4} & 1 & -\frac{1}{4} & & & -\frac{1}{4} & & \frac{0}{1} \\
 & & & & & & & -\frac{1}{5} & & & -\frac{1}{5} & 1 & & & & -\frac{1}{5} & \frac{2}{5} \\
 & & & & & & & & -\frac{1}{4} & & & & 1 & -\frac{1}{4} & & & \frac{1}{2} \\
 & & & & & & & & & -\frac{1}{3} & & & -\frac{1}{3} & 1 & -\frac{1}{3} & & \frac{0}{1} \\
 & & & & & & & & & & -\frac{1}{3} & & & -\frac{1}{3} & 1 & -\frac{1}{3} & \frac{0}{1} \\
 & & & & & & & & & & & -\frac{1}{4} & & & -\frac{1}{4} & 1 & \frac{1}{2}
\end{array}\right)$$

Рисунок 5.2 – Структура рассматриваемой предобусловленной матрицы при $N = 4,\ M = 4$.

Рассмотрим $N, M = 128, 128$ соответственно. Это значит, что матрица будет содержать $N = 128$ блоков в строке и столбце, и $M = 128$ строк и столбцов в блоке.

Число итераций	Число потоков	Послед. метод, сек	Паралл. метод, сек	Наилучшее ускорение (последовательный к параллельному)
8192	16	7.2754	9.63634	0.75499
	32		10.05440	
	64		10.31396	
12288	16	10.6632	14.55884	0.73242
	32		15.10426	
	64		15.32312	
16384	16	14.5167	19.23600	0.75466
	32		20.00645	
	64		20.25727	

Таблица 5.5 – Время выполнения программы при фиксированном $N, M = 128, 128$ соответственно.

При рассмотрении матриц больших размерностей требуется слишком большое число итераций для получения удовлетворительной погрешности, из чего вытекает слишком большое время выполнения.

Если анализировать полученный результат из таблицы 5.5, можно увидеть, что аналогично прошлому примеру, на матрице такой размерности не стоит использовать параллельную реализацию.

ЗАКЛЮЧЕНИЕ

В данной работе рассмотрен вопрос численного моделирования двумерных параболических уравнений гемодинамики с помощью с помощью метода Гаусса-Зейделя и его эффективная реализация на GPU с использованием CUDA.

Был выбран неявный блочный алгоритм Гаусса-Зейделя, обладающий преимуществами перед явными схемами: повышенной устойчивостью и наличием естественного параллелизма.

Для ускорения вычислений была применена технология CUDA, позволяющая раскрыть потенциал графических ускорителей, а за счет высокой степени параллелизма, GPU реализация алгоритма показывает меньшее время выполнения расчета, особенно для задач больших размерностей. Были реализованы последовательная (CPU) и параллельная (GPU) версии алгоритма, проведено сравнение этих реализаций. В ходе анализа подтвердилась эффективность графических ускорителей для задач больших размерностей.

Кроме того, были тщательно изучены основные теоретические аспекты задачи: разработаны и описаны псевдокоды используемых методов и исследована структура матриц, возникающих в поставленной задаче.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Yang W, Li K, Li K. A parallel solving method for block-tridiagonal equations on CPU-GPU heterogeneous computing systems // Supercomputing. 2017. Vol. 73, No 5. P. 1760-1781.

2. Самарский А.А. Николаев Е.С. Методы решения сеточных уравнений. М: Наука. 1978. 592 с.

3. Лекционный курс по численным методам (5 семестр, 1 поток), «Метод прогонки», [электронный ресурс]. – Режим доступа: https://edufpmi.bsu.by/.

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

5. Информационные статьи, [электронный ресурс]. Режим доступа: https://developer.nvidia.com/

6. 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.

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

8. Открытая энциклопедия свойств алгоритмов, [электронный ресурс]. – Режим доступа: https://algowiki-project.org/

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Код последовательного метода Гаусса-Зейделя для систем специального вида на языке C++. Первый вычислительный эксперимент

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <cmath>
#include <chrono>

namespace cfg
{
   constexpr int N = 1024;    // число блоков
   constexpr int M = 1024;    // размер блока
   constexpr int MAX_IT = 32; // число итераций
}

using Vec = std::vector<double>;
using Mat1 = std::vector<Vec>;

// Индексация
inline int gIndex(int i, int k) { return i * cfg::M + k; }

// Прогонка
void solveTridiagonal(const Vec &a, const Vec &c, const Vec &b, Vec rhs, Vec &x)
{
   int m = cfg::M;
   Vec alpha(m, 0.0), beta(m + 1, 0.0);

   double gamma = c[0];
   alpha[1] = -b[0] / gamma;
   beta[1] = rhs[0] / gamma;

   for (int k = 1; k < m - 1; ++k)
   {
      gamma = c[k] + a[k] * alpha[k];
      alpha[k + 1] = -b[k] / gamma;
      beta[k + 1] = (rhs[k] - a[k] * beta[k]) / gamma;
```

```
    }
    gamma = c[m - 1] + a[m - 1] * alpha[m - 1];
    beta[m] = (rhs[m - 1] - a[m - 1] * beta[m - 1]) / gamma;

    x[m - 1] = beta[m];
    for (int k = m - 2; k >= 0; --k)
        x[k] = alpha[k + 1] * x[k + 1] + beta[k + 1];
}

// Заполнение матрицы по формулам + вектор правой части
void fillSystem(Mat1 &a, Mat1 &c, Mat1 &b, Mat1 &Adiag, Mat1 &Bdiag, Mat1
&F)
{
    const int N = cfg::N, M = cfg::M;
    a.assign(N, Vec(M, 0.0));
    b.assign(N, Vec(M, 0.0));
    c.assign(N, Vec(M, 4.0)); // главная диагональ C_i = 4
    Adiag.assign(N, Vec(M, 0.0));
    Bdiag.assign(N, Vec(M, 0.0));
    F.assign(N, Vec(M, 0.0));

    const double denom = 2.0 * N + M; // общий знаменатель

    // Коэффициенты блоков
    for (int i = 0; i < N; ++i)
        for (int k = 0; k < M; ++k)
        {
            double val = (2.0 * (i + 1) + (k + 1)) / denom;
            if (k > 0)
                a[i][k] = val; // поддиагональ C_i
            if (k < M - 1)
                b[i][k] = val; // наддиагональ C_i
            if (i > 0)
                Adiag[i][k] = val; // A_i
            if (i < N - 1)
                Bdiag[i][k] = val; // B_i
        }

    // Правая часть F для получения точного единичного решения
    for (int i = 0; i < N; ++i)
```

```
        for (int k = 0; k < M; ++k)
        {
            double rhs = c[i][k];
            if (k > 0)
                rhs += a[i][k];
            if (k < M - 1)
                rhs += b[i][k];
            if (i > 0)
                rhs += Adiag[i][k];
            if (i < N - 1)
                rhs += Bdiag[i][k];
            F[i][k] = rhs;
        }
}

// Красивый вывод матрицы
void printGlobalMatrix(const Mat1 &a, const Mat1 &c, const Mat1 &b, const Mat1
&Ad, const Mat1 &Bd)
{
    const int dim = cfg::N * cfg::M;
    const int W = 11; // ширина столбца

    auto Acoeff = [&](int gi, int gk)
    {
        int bi = gi / cfg::M, ki = gi % cfg::M;
        int bj = gk / cfg::M, kj = gk % cfg::M;

        if (bi == bj)
        {
            if (ki == kj)
                return c[bi][ki];
            if (kj == ki - 1)
                return a[bi][ki];
            if (kj == ki + 1)
                return b[bi][ki];
        }
        if (bj == bi - 1 && ki == kj)
            return Ad[bi][ki];
        if (bj == bi + 1 && ki == kj)
            return Bd[bi][ki];
```

```
        return 0.0;
    };

    std::cout << "\nПолная матрица (" << dim << " x " << dim << "):\n";
    std::cout.setf(std::ios::fixed);
    std::cout << std::setprecision(6);

    for (int r = 0; r < dim; ++r)
    {
        for (int ccol = 0; ccol < dim; ++ccol)
        {
            double v = Acoeff(r, ccol);
            if (std::fabs(v) < 1e-12)
                std::cout << std::setw(W) << '.';
            else
                std::cout << std::setw(W) << v;
        }
        std::cout << '\n';
    }
}

// Последовательный блочный неявный метод Гаусса-Зейделя
void solveGaussSeidel(const Mat1 &a, const Mat1 &c, const Mat1 &b, const Mat1
&Ad, const Mat1 &Bd, const Mat1 &F, Mat1 &Y)
{
    const int N = cfg::N, M = cfg::M;
    Vec Phi(M), tmp(M);

    for (int it = 0; it < cfg::MAX_IT; ++it)
    {
        for (int i = 0; i < N; ++i)
        {
            for (int k = 0; k < M; ++k)
            {
                Phi[k] = F[i][k];
                if (i > 0)
                    Phi[k] -= Ad[i][k] * Y[i - 1][k];
                if (i < N - 1)
                    Phi[k] -= Bd[i][k] * Y[i + 1][k];
            }
```

```
            solveTridiagonal(a[i], c[i], b[i], Phi, tmp);
            Y[i] = tmp;
        }
    }
}

int main()
{
    Mat1 a, c, b, Ad, Bd, F;
    fillSystem(a, c, b, Ad, Bd, F); // заполняем систему

    // Вывод полной матрицы
    // printGlobalMatrix(a, c, b, Ad, Bd);

    // Вывод вектора правой части
    // std::cout << "\nВектор правой части:\n";
    // int idx = 0;
    // for (int i = 0; i < cfg::N; ++i)
    //     for (int k = 0; k < cfg::M; ++k, ++idx)
    //         std::cout << "F[" << std::setw(2) << idx + 1 << "] = "
    //                   << std::setw(8) << F[i][k] << '\n';

    Mat1 Y(cfg::N, Vec(cfg::M, 0.0)); // начальное приближение

    // Таймер начала
    auto start = std::chrono::high_resolution_clock::now();

    // Вызов метода
    solveGaussSeidel(a, c, b, Ad, Bd, F, Y);

    // Таймер окончания
    auto stop = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> duration_ms = stop - start;
    std::cout << "\nВремя работы: " << duration_ms.count() << " мс\n";

    // Вывод вектора решений
    // std::cout << "\nВектор решений:\n";
    // int counter = 0;
    // for (int i = 0; i < cfg::N; ++i)
    // for (int k = 0; k < cfg::M; ++k, ++counter)
```

```
    // std::cout << "Y[" << std::setw(2) << counter + 1 << "] = " << std::setw(8) <<
Y[i][k] << '\n';

    // Наибольшая погрешность с точным решением (1 значение)
    std::cout << "\nНаибольшая погрешность с точным решением: ";
    double max_err = 0.0;
    for (int i = 0; i < cfg::N; ++i)
        for (int k = 0; k < cfg::M; ++k)
            max_err = std::max(max_err, std::fabs(Y[i][k] - 1.0));
    std::cout << std::setw(8) << max_err << '\n';

    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Код параллельного метода Гаусса-Зейделя с использованием технологии CUDA для систем специального вида на языке C++. Первый вычислительный эксперимент

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <clocale>
#include <cuda_runtime.h>
namespace cfg
{
   constexpr int N = 4;       // число блоков
   constexpr int M = 4;       // размер блока
   constexpr int MAX_IT = 16; // число итераций
}
// Вывод вектора
void printVector(const double *v)
{
   const size_t tot = static_cast<size_t>(cfg::N) * cfg::M;
   std::cout << std::fixed << std::setprecision(6);
   for (size_t i = 0; i < tot; ++i)
      std::cout << v[i] << '\n';
}
// Вывод матрицы
void printGlobalMatrix(const double *a, const double *c, const double *b, const
double *Ad, const double *Bd)
{
   const int N = cfg::N, M = cfg::M, dim = N * M, W = 11;
   auto coef = [&](int gi, int gk) -> double
   {
      int bi = gi / M, ki = gi % M, bj = gk / M, kj = gk % M;
      if (bi == bj)
      {
         if (ki == kj)
            return c[bi * M + ki];
         if (kj == ki - 1)
            return a[bi * M + ki];
         if (kj == ki + 1)
            return b[bi * M + ki];
      }
      if (bj == bi - 1 && ki == kj)
         return Ad[bi * M + ki];
      if (bj == bi + 1 && ki == kj)
```

```
            return Bd[bi * M + ki];
        return 0.0;
    };
    std::cout << "\nПолная матрица (" << dim << " x " << dim << "):\n"
              << std::fixed << std::setprecision(6);
    for (int r = 0; r < dim; ++r)
    {
        for (int ccol = 0; ccol < dim; ++ccol)
        {
            double v = coef(r, ccol);
            if (std::fabs(v) < 1e-12)
                std::cout << std::setw(W) << '.';
            else
                std::cout << std::setw(W) << v;
        }
        std::cout << '\n';
    }
}
// CUDA-ядро для обновления нечетных строк
__global__ void updateOdd(const double *a, const double *c, const double *b, const
double *Ad, const double *Bd, double *Y, const double *F)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalOdd = (cfg::N + 1) / 2;
    if (tid >= totalOdd)
        return;
    int i = 2 * tid;
    int iOff = i * cfg::M;
    double alpha[cfg::M + 2], beta[cfg::M + 2];
    double rhs = F[iOff];
    if (i)
        rhs -= Ad[iOff] * Y[(i - 1) * cfg::M];
    if (i < cfg::N - 1)
        rhs -= Bd[iOff] * Y[(i + 1) * cfg::M];
    double inv = 1.0 / c[iOff];
    alpha[1] = (cfg::M > 1 ? -b[iOff] * inv : 0.0);
    beta[1] = rhs * inv;
    for (int k = 1; k < cfg::M - 1; ++k)
    {
        int idx = iOff + k;
        rhs = F[idx] - (i ? Ad[idx] * Y[(i - 1) * cfg::M + k] : 0.0) - (i < cfg::N - 1 ?
Bd[idx] * Y[(i + 1) * cfg::M + k] : 0.0);
        double denom = c[idx] + a[idx] * alpha[k];
        inv = 1.0 / denom;
        alpha[k + 1] = -b[idx] * inv;
```

```
        beta[k + 1] = (rhs - a[idx] * beta[k]) * inv;
    }
    int kL = cfg::M - 1, idxL = iOff + kL;
    rhs = F[idxL] - (i ? Ad[idxL] * Y[(i - 1) * cfg::M + kL] : 0.0) - (i < cfg::N - 1 ?
Bd[idxL] * Y[(i + 1) * cfg::M + kL] : 0.0);
    double yL = (cfg::M == 1) ? rhs / c[idxL] : (rhs - a[idxL] * beta[kL]) / (c[idxL] +
a[idxL] * alpha[kL]);
    Y[idxL] = yL;
    double yNext = yL;
    for (int k = cfg::M - 2; k >= 0; --k)
    {
        double yk = alpha[k + 1] * yNext + beta[k + 1];
        Y[iOff + k] = yk;
        yNext = yk;
    }
}
// CUDA-ядро для обновления четных строк
__global__ void updateEven(const double *a, const double *c, const double *b,
                           const double *Ad, const double *Bd,
                           double *Y, const double *F)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalEven = cfg::N / 2;
    if (tid >= totalEven)
        return;
    int i = 2 * tid + 1;
    int iOff = i * cfg::M;
    double alpha[cfg::M + 2], beta[cfg::M + 2];
    double rhs = F[iOff] - Ad[iOff] * Y[(i - 1) * cfg::M] - (i < cfg::N - 1 ? Bd[iOff] *
Y[(i + 1) * cfg::M] : 0.0);
    double inv = 1.0 / c[iOff];
    alpha[1] = (cfg::M > 1 ? -b[iOff] * inv : 0.0);
    beta[1] = rhs * inv;
    for (int k = 1; k < cfg::M - 1; ++k)
    {
        int idx = iOff + k;
        rhs = F[idx] - Ad[idx] * Y[(i - 1) * cfg::M + k] - (i < cfg::N - 1 ? Bd[idx] * Y[(i
+ 1) * cfg::M + k] : 0.0);
        double denom = c[idx] + a[idx] * alpha[k];
        inv = 1.0 / denom;
        alpha[k + 1] = -b[idx] * inv;
        beta[k + 1] = (rhs - a[idx] * beta[k]) * inv;
    }
    int kL = cfg::M - 1, idxL = iOff + kL;
```

```
    rhs = F[idxL] - Ad[idxL] * Y[(i - 1) * cfg::M + kL] - (i < cfg::N - 1 ? Bd[idxL] *
Y[(i + 1) * cfg::M + kL] : 0.0);
    double yL = (cfg::M == 1) ? rhs / c[idxL] : (rhs - a[idxL] * beta[kL]) / (c[idxL] +
a[idxL] * alpha[kL]);
    Y[idxL] = yL;
    double yNext = yL;
    for (int k = cfg::M - 2; k >= 0; --k)
    {
        double yk = alpha[k + 1] * yNext + beta[k + 1];
        Y[iOff + k] = yk;
        yNext = yk;
    }
}
int main()
{
    std::setlocale(LC_ALL, "Russian");
    const int N = cfg::N, M = cfg::M;
    size_t size = static_cast<size_t>(N) * M;
    // Хостовые массивы
    auto alloc = [&](size_t n)
    { return new double[n](); };
    double *h_a = alloc(size), *h_b = alloc(size), *h_c = alloc(size), *h_Ad =
alloc(size), *h_Bd = alloc(size), *h_F = alloc(size), *h_Y = alloc(size);
    // Заполнение коэффициентов
    double denom = 2.0 * N + M;
    for (int i = 0; i < N; ++i)
    {
        for (int k = 0; k < M; ++k)
        {
            h_c[i * M + k] = 4.0;
            double v = (2.0 * (i + 1) + k + 1) / denom;
            h_a[i * M + k] = (k ? v : 0.0);
            h_b[i * M + k] = (k < M - 1 ? v : 0.0);
            h_Ad[i * M + k] = (i ? v : 0.0);
            h_Bd[i * M + k] = (i < N - 1 ? v : 0.0);
        }
    }
    for (int i = 0; i < N; ++i)
    {
        for (int k = 0; k < M; ++k)
        {
            double rhs = h_c[i * M + k];
            if (k)
                rhs += h_a[i * M + k];
            if (k < M - 1)
```

```
            rhs += h_b[i * M + k];
        if (i)
            rhs += h_Ad[i * M + k];
        if (i < N - 1)
            rhs += h_Bd[i * M + k];
        h_F[i * M + k] = rhs;
    }
}
// Device-память
double *d_a, *d_b, *d_c, *d_Ad, *d_Bd, *d_F, *d_Y;
cudaMalloc(&d_a, size * sizeof(double));
cudaMalloc(&d_b, size * sizeof(double));
cudaMalloc(&d_c, size * sizeof(double));
cudaMalloc(&d_Ad, size * sizeof(double));
cudaMalloc(&d_Bd, size * sizeof(double));
cudaMalloc(&d_F, size * sizeof(double));
cudaMalloc(&d_Y, size * sizeof(double));
cudaMemcpy(d_a, h_a, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_c, h_c, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Ad, h_Ad, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Bd, h_Bd, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_F, h_F, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Y, h_Y, size * sizeof(double), cudaMemcpyHostToDevice);
int threads = 32;
int blocksOdd = ((N + 1) / 2 + threads - 1) / threads;
int blocksEven = (N / 2 + threads - 1) / threads;
// Замер времени
cudaEvent_t evStart, evStop;
cudaEventCreate(&evStart);
cudaEventCreate(&evStop);
cudaEventRecord(evStart, 0);
for (int it = 0; it < cfg::MAX_IT; ++it)
{
   updateOdd<<<blocksOdd, threads>>>(d_a, d_c, d_b, d_Ad, d_Bd, d_Y, d_F);
   updateEven<<<blocksEven, threads>>>(d_a, d_c, d_b, d_Ad, d_Bd, d_Y, d_F);
}
cudaEventRecord(evStop, 0);
cudaEventSynchronize(evStop);
float gpu_ms = 0.0f;
cudaEventElapsedTime(&gpu_ms, evStart, evStop);
cudaMemcpy(h_Y, d_Y, size * sizeof(double), cudaMemcpyDeviceToHost);
// Результаты
double maxErr = 0.0;
for (size_t i = 0; i < size; ++i)
```

```
        maxErr = std::max(maxErr, std::fabs(h_Y[i] - 1.0));
    std::cout << std::fixed << std::setprecision(3) << "Время работы: " << gpu_ms <<
"мс\n";
    std::cout << std::scientific << std::setprecision(6) << "Наибольшая погрешность
с точным решением : " << maxErr << '\n';
    // printGlobalMatrix(h_a, h_c, h_b, h_Ad, h_Bd);
    // std::cout << "\nF:\n"; printVector(h_F);
    // std::cout << "\nY:\n"; printVector(h_Y);
    // Чистка памяти
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(d_Ad);
    cudaFree(d_Bd);
    cudaFree(d_F);
    cudaFree(d_Y);
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    delete[] h_Ad;
    delete[] h_Bd;
    delete[] h_F;
    delete[] h_Y;
    return 0;
}
```

ПРИЛОЖЕНИЕ В

Код последовательного метода Гаусса-Зейделя для систем специального вида на языке C++. Второй вычислительный эксперимент

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <cmath>
#include <chrono>

namespace cfg
{
   constexpr int N = 4;       // число блоков
   constexpr int M = 4;       // размер блока
   constexpr int MAX_IT = 16; // число итераций
}

using Vec = std::vector<double>;
using Mat1 = std::vector<Vec>;

// Индексация
inline int gIndex(int i, int k) { return i * cfg::M + k; }

// Прогонка
void solveTridiagonal(const Vec &a, const Vec &c, const Vec &b, Vec rhs, Vec &x)
{
   int m = cfg::M;
   Vec alpha(m, 0.0), beta(m + 1, 0.0);

   double gamma = c[0];
   alpha[1] = -b[0] / gamma;
   beta[1] = rhs[0] / gamma;

   for (int k = 1; k < m - 1; ++k)
   {
      gamma = c[k] + a[k] * alpha[k];
      alpha[k + 1] = -b[k] / gamma;
      beta[k + 1] = (rhs[k] - a[k] * beta[k]) / gamma;
   }
   gamma = c[m - 1] + a[m - 1] * alpha[m - 1];
   beta[m] = (rhs[m - 1] - a[m - 1] * beta[m - 1]) / gamma;

   x[m - 1] = beta[m];
   for (int k = m - 2; k >= 0; --k)
```

```
        x[k] = alpha[k + 1] * x[k + 1] + beta[k + 1];
}

// Заполнение матрицы по формулам + вектор правой части
void fillSystem(Mat1 &a, Mat1 &c, Mat1 &b, Mat1 &Adiag, Mat1 &Bdiag, Mat1
&F)
{
   const int N = cfg::N, M = cfg::M;
   a.assign(N, Vec(M, 0.0));
   b.assign(N, Vec(M, 0.0));
   c.assign(N, Vec(M, 0.0));
   Adiag.assign(N, Vec(M, 0.0));
   Bdiag.assign(N, Vec(M, 0.0));
   F.assign(N, Vec(M, 0.0));

   for (int i = 0; i < N; ++i)
   {
      for (int k = 0; k < M; ++k)
      {
         bool firstBlockRow = (i == 0);
         bool lastBlockRow = (i == N - 1);
         bool firstCol = (k == 0);
         bool lastCol = (k == M - 1);

         // Центральные внутренние точки (2<=i<=N-1, 2<=k<=M-1)
         if (!firstBlockRow && !lastBlockRow && !firstCol && !lastCol)
         {
            c[i][k] = 4;
            a[i][k] = -1;
            b[i][k] = -1;
            Adiag[i][k] = -1;
            Bdiag[i][k] = -1;
         }
         // Внутренние блочные ряды, первый столбец (2<=i<=N-1, k=1)
         else if (!firstBlockRow && !lastBlockRow && firstCol)
         {
            c[i][k] = 5;
            b[i][k] = -1;
            Adiag[i][k] = -1;
            Bdiag[i][k] = -1;
         }
         // Внутренние блочные ряды, последний столбец (2<=i<=N-1, k=M)
         else if (!firstBlockRow && !lastBlockRow && lastCol)
         {
            c[i][k] = 5;
```

```
    a[i][k] = -1;
    Adiag[i][k] = -1;
    Bdiag[i][k] = -1;
}
// Первая блочная строка, внутренние столбцы (i=1,2<=k<=M-1)
else if (firstBlockRow && !firstCol && !lastCol)
{
    c[i][k] = 3;
    a[i][k] = -1;
    b[i][k] = -1;
    Bdiag[i][k] = -1;
}
// Первая блочная строка, первый столбец (i=1, k=1)
else if (firstBlockRow && firstCol)
{
    c[i][k] = 4;
    b[i][k] = -1;
    Bdiag[i][k] = -1;
}
// Первая блочная строка, последний столбец (i=1, k=M)
else if (firstBlockRow && lastCol)
{
    c[i][k] = 4;
    a[i][k] = -1;
    Bdiag[i][k] = -1;
}
// Последняя блочная строка, внутренние столбцы (i=N,2<=k<=M-1)
else if (lastBlockRow && !firstCol && !lastCol)
{
    c[i][k] = 3;
    a[i][k] = -1;
    b[i][k] = -1;
    Adiag[i][k] = -1;
}
// Последняя блочная строка, первый столбец (i=N, k=1)
else if (lastBlockRow && firstCol)
{
    c[i][k] = 4;
    b[i][k] = -1;
    Adiag[i][k] = -1;
}
// Последняя блочная строка, последний столбец (i=N, k=M)
else
{
    c[i][k] = 4;
```

```
            a[i][k] = -1;
            Adiag[i][k] = -1;
        }

        F[i][k] = (k == 0 || k == M - 1) ? 2.0 : 0.0;
    }
  }

  for (int i = 0; i < N; ++i)
  {
    for (int k = 0; k < M; ++k)
    {
      double diag = c[i][k];
      if (std::fabs(diag) < 1e-16)
        continue;

      a[i][k] /= diag;
      b[i][k] /= diag;
      Adiag[i][k] /= diag;
      Bdiag[i][k] /= diag;
      F[i][k] /= diag;
      c[i][k] = 1.0;
    }
  }
}

// Красивый вывод матрицы
void printGlobalMatrix(const Mat1 &a, const Mat1 &c, const Mat1 &b, const Mat1
&Ad, const Mat1 &Bd)
{
  const int dim = cfg::N * cfg::M;
  const int W = 11; // ширина столбца

  auto Acoeff = [&](int gi, int gk)
  {
    int bi = gi / cfg::M, ki = gi % cfg::M;
    int bj = gk / cfg::M, kj = gk % cfg::M;

    if (bi == bj)
    {
      if (ki == kj)
        return c[bi][ki];
      if (kj == ki - 1)
        return a[bi][ki];
      if (kj == ki + 1)
```

```
                return b[bi][ki];
            }
            if (bj == bi - 1 && ki == kj)
                return Ad[bi][ki];
            if (bj == bi + 1 && ki == kj)
                return Bd[bi][ki];
            return 0.0;
        };

        std::cout << "\nПолная матрица (" << dim << " x " << dim << "):\n";
        std::cout.setf(std::ios::fixed);
        std::cout << std::setprecision(6);

        for (int r = 0; r < dim; ++r)
        {
            for (int ccol = 0; ccol < dim; ++ccol)
            {
                double v = Acoeff(r, ccol);
                if (std::fabs(v) < 1e-12)
                    std::cout << std::setw(W) << '.';
                else
                    std::cout << std::setw(W) << v;
            }
            std::cout << '\n';
        }
    }

    // Последовательный блочный неявный метод Гаусса-Зейделя
    void solveGaussSeidel(const Mat1 &a, const Mat1 &c, const Mat1 &b, const Mat1
    &Ad, const Mat1 &Bd, const Mat1 &F, Mat1 &Y)
    {
        const int N = cfg::N, M = cfg::M;
        Vec Phi(M), tmp(M);

        for (int it = 0; it < cfg::MAX_IT; ++it)
        {
            for (int i = 0; i < N; ++i)
            {
                for (int k = 0; k < M; ++k)
                {
                    Phi[k] = F[i][k];
                    if (i > 0)
                        Phi[k] -= Ad[i][k] * Y[i - 1][k];
                    if (i < N - 1)
                        Phi[k] -= Bd[i][k] * Y[i + 1][k];
```

```
            }
            solveTridiagonal(a[i], c[i], b[i], Phi, tmp);
            Y[i] = tmp;
        }
    }
}

int main()
{
    Mat1 a, c, b, Ad, Bd, F;
    fillSystem(a, c, b, Ad, Bd, F); // заполняем систему

    // Вывод полной матрицы
    // printGlobalMatrix(a, c, b, Ad, Bd);

    // Вывод вектора правой части
    // std::cout << "\nВектор правой части:\n";
    // int idx = 0;
    // for (int i = 0; i < cfg::N; ++i)
    //     for (int k = 0; k < cfg::M; ++k, ++idx)
    //         std::cout << "F[" << std::setw(2) << idx + 1 << "] = "
    //                   << std::setw(8) << F[i][k] << '\n';

    Mat1 Y(cfg::N, Vec(cfg::M, 0.0)); // начальное приближение

    // Таймер начала
    auto start = std::chrono::high_resolution_clock::now();

    // Вызов метода
    solveGaussSeidel(a, c, b, Ad, Bd, F, Y);

    // Таймер окончания
    auto stop = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> duration_ms = stop - start;
    std::cout << "\nВремя работы: " << duration_ms.count() / 1000 << " c\n";

    // Вывод вектора решений
    // std::cout << "\nВектор решений:\n";
    // int counter = 0;
    // for (int i = 0; i < cfg::N; ++i)
    //     for (int k = 0; k < cfg::M; ++k, ++counter)
    //         std::cout << "Y[" << std::setw(2) << counter + 1 << "] = " << std::setw(8)
<< Y[i][k] << '\n';

    // Наибольшая погрешность с точным решением (1 значение)
```

```
    std::cout << "\nНаибольшая погрешность с точным решением: ";
    double max_err = 0.0;
    for (int i = 0; i < cfg::N; ++i)
        for (int k = 0; k < cfg::M; ++k)
            max_err = std::max(max_err, std::fabs(Y[i][k] - 1.0));
    std::cout << std::setw(8) << max_err << '\n';

    return 0;
}
```

ПРИЛОЖЕНИЕ Г

Код параллельного метода Гаусса-Зейделя с использованием технологии CUDA для систем специального вида на языке C++. Второй вычислительный эксперимент

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <clocale>
#include <cuda_runtime.h>
namespace cfg
{
   constexpr int N = 128;       // число блоков
   constexpr int M = 128;       // размер блока
   constexpr int MAX_IT = 8192; // число итераций
}
// Вывод вектора
void printVector(const double* v)
{
   const size_t tot = static_cast<size_t>(cfg::N) * cfg::M;
   std::cout << std::fixed << std::setprecision(6);
   for (size_t i = 0; i < tot; ++i)
      std::cout << v[i] << '\n';
}
// Вывод матрицы
void printGlobalMatrix(const double* a, const double* c, const double* b, const
double* Ad, const double* Bd)
{
   const int N = cfg::N, M = cfg::M, dim = N * M, W = 11;
   auto coef = [&](int gi, int gk) -> double
      {
         int bi = gi / M, ki = gi % M, bj = gk / M, kj = gk % M;
         if (bi == bj)
         {
            if (ki == kj)
               return c[bi * M + ki];
            if (kj == ki - 1)
               return a[bi * M + ki];
            if (kj == ki + 1)
               return b[bi * M + ki];
```

```
            }
            if (bj == bi - 1 && ki == kj)
                return Ad[bi * M + ki];
            if (bj == bi + 1 && ki == kj)
                return Bd[bi * M + ki];
            return 0.0;
        };
    std::cout << "\nПолная матрица (" << dim << " x " << dim << "):\n"
        << std::fixed << std::setprecision(6);
    for (int r = 0; r < dim; ++r)
    {
        for (int ccol = 0; ccol < dim; ++ccol)
        {
            double v = coef(r, ccol);
            if (std::fabs(v) < 1e-12)
                std::cout << std::setw(W) << '.';
            else
                std::cout << std::setw(W) << v;
        }
        std::cout << '\n';
    }
}
// CUDA-ядро для обновления нечетных строк
__global__ void updateOdd(const double* a, const double* c, const double* b, const
double* Ad, const double* Bd, double* Y, const double* F)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalOdd = (cfg::N + 1) / 2;
    if (tid >= totalOdd)
        return;
    int i = 2 * tid;
    int iOff = i * cfg::M;
    double alpha[cfg::M + 2], beta[cfg::M + 2];
    double rhs = F[iOff];
    if (i)
        rhs -= Ad[iOff] * Y[(i - 1) * cfg::M];
    if (i < cfg::N - 1)
        rhs -= Bd[iOff] * Y[(i + 1) * cfg::M];
    double inv = 1.0 / c[iOff];
    alpha[1] = (cfg::M > 1 ? -b[iOff] * inv : 0.0);
```

```
    beta[1] = rhs * inv;
    for (int k = 1; k < cfg::M - 1; ++k)
    {
        int idx = iOff + k;
        rhs = F[idx] - (i ? Ad[idx] * Y[(i - 1) * cfg::M + k] : 0.0) - (i < cfg::N - 1 ? Bd[idx]
* Y[(i + 1) * cfg::M + k] : 0.0);
        double denom = c[idx] + a[idx] * alpha[k];
        inv = 1.0 / denom;
        alpha[k + 1] = -b[idx] * inv;
        beta[k + 1] = (rhs - a[idx] * beta[k]) * inv;
    }
    int kL = cfg::M - 1, idxL = iOff + kL;
    rhs = F[idxL] - (i ? Ad[idxL] * Y[(i - 1) * cfg::M + kL] : 0.0) - (i < cfg::N - 1 ?
Bd[idxL] * Y[(i + 1) * cfg::M + kL] : 0.0);
    double yL = (cfg::M == 1) ? rhs / c[idxL] : (rhs - a[idxL] * beta[kL]) / (c[idxL] +
a[idxL] * alpha[kL]);
    Y[idxL] = yL;
    double yNext = yL;
    for (int k = cfg::M - 2; k >= 0; --k)
    {
        double yk = alpha[k + 1] * yNext + beta[k + 1];
        Y[iOff + k] = yk;
        yNext = yk;
    }
}
// CUDA-ядро для обновления четных строк
__global__ void updateEven(const double* a, const double* c, const double* b,
    const double* Ad, const double* Bd,
    double* Y, const double* F)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int totalEven = cfg::N / 2;
    if (tid >= totalEven)
        return;
    int i = 2 * tid + 1;
    int iOff = i * cfg::M;
    double alpha[cfg::M + 2], beta[cfg::M + 2];
    double rhs = F[iOff] - Ad[iOff] * Y[(i - 1) * cfg::M] - (i < cfg::N - 1 ? Bd[iOff] *
Y[(i + 1) * cfg::M] : 0.0);
    double inv = 1.0 / c[iOff];
```

```
    alpha[1] = (cfg::M > 1 ? -b[iOff] * inv : 0.0);
    beta[1] = rhs * inv;
    for (int k = 1; k < cfg::M - 1; ++k)
    {
        int idx = iOff + k;
        rhs = F[idx] - Ad[idx] * Y[(i - 1) * cfg::M + k] - (i < cfg::N - 1 ? Bd[idx] * Y[(i +
1) * cfg::M + k] : 0.0);
        double denom = c[idx] + a[idx] * alpha[k];
        inv = 1.0 / denom;
        alpha[k + 1] = -b[idx] * inv;
        beta[k + 1] = (rhs - a[idx] * beta[k]) * inv;
    }
    int kL = cfg::M - 1, idxL = iOff + kL;
    rhs = F[idxL] - Ad[idxL] * Y[(i - 1) * cfg::M + kL] - (i < cfg::N - 1 ? Bd[idxL] *
Y[(i + 1) * cfg::M + kL] : 0.0);
    double yL = (cfg::M == 1) ? rhs / c[idxL] : (rhs - a[idxL] * beta[kL]) / (c[idxL] +
a[idxL] * alpha[kL]);
    Y[idxL] = yL;
    double yNext = yL;
    for (int k = cfg::M - 2; k >= 0; --k)
    {
        double yk = alpha[k + 1] * yNext + beta[k + 1];
        Y[iOff + k] = yk;
        yNext = yk;
    }
}
int main()
{
    std::setlocale(LC_ALL, "Russian");
    const int N = cfg::N, M = cfg::M;
    size_t size = static_cast<size_t>(N) * M;
    // Хостовые массивы
    auto alloc = [&](size_t n)
        { return new double[n](); };
    double* h_a = alloc(size), * h_b = alloc(size), * h_c = alloc(size), * h_Ad =
alloc(size), * h_Bd = alloc(size), * h_F = alloc(size), * h_Y = alloc(size);
    // Заполнение коэффициентов
    for (int i = 0; i < N; ++i)
    {
        bool firstBlockRow = (i == 0);
```

```
bool lastBlockRow = (i == N - 1);
for (int k = 0; k < M; ++k)
{
   bool firstCol = (k == 0);
   bool lastCol = (k == M - 1);
   size_t idx = static_cast<size_t>(i) * M + k;

   if (!firstBlockRow && !lastBlockRow && !firstCol && !lastCol)
   {
      // центральные внутренние
      h_c[idx] = 4.0;
      h_a[idx] = -1.0;
      h_b[idx] = -1.0;
      h_Ad[idx] = -1.0;
      h_Bd[idx] = -1.0;
   }
   else if (!firstBlockRow && !lastBlockRow && firstCol)
   {
      // внутренние блоки, первый столбец
      h_c[idx] = 5.0;
      h_b[idx] = -1.0;
      h_Ad[idx] = -1.0;
      h_Bd[idx] = -1.0;
   }
   else if (!firstBlockRow && !lastBlockRow && lastCol)
   {
      // внутренние блоки, последний столбец
      h_c[idx] = 5.0;
      h_a[idx] = -1.0;
      h_Ad[idx] = -1.0;
      h_Bd[idx] = -1.0;
   }
   else if (firstBlockRow && !firstCol && !lastCol)
   {
      // первая блочная строка, внутренние столбцы
      h_c[idx] = 3.0;
      h_a[idx] = -1.0;
      h_b[idx] = -1.0;
      h_Bd[idx] = -1.0;
   }
```

```
    else if (firstBlockRow && firstCol)
    {
        // первая блочная строка, первый столбец
        h_c[idx] = 4.0;
        h_b[idx] = -1.0;
        h_Bd[idx] = -1.0;
    }
    else if (firstBlockRow && lastCol)
    {
        // первая блочная строка, последний столбец
        h_c[idx] = 4.0;
        h_a[idx] = -1.0;
        h_Bd[idx] = -1.0;
    }
    else if (lastBlockRow && !firstCol && !lastCol)
    {
        // последняя блочная строка, внутренние столбцы
        h_c[idx] = 3.0;
        h_a[idx] = -1.0;
        h_b[idx] = -1.0;
        h_Ad[idx] = -1.0;
    }
    else if (lastBlockRow && firstCol)
    {
        // последняя блочная строка, первый столбец
        h_c[idx] = 4.0;
        h_b[idx] = -1.0;
        h_Ad[idx] = -1.0;
    }
    else
    {
        // последняя блочная строка, последний столбец
        h_c[idx] = 4.0;
        h_a[idx] = -1.0;
        h_Ad[idx] = -1.0;
    }

    // Правая часть F
    h_F[idx] = (firstCol || lastCol) ? 2.0 : 0.0;
}
```

```
}
for (size_t idx = 0; idx < size; ++idx)
{
   double diag = h_c[idx];
   if (std::fabs(diag) < 1e-16)
      continue;

   h_a[idx] /= diag;
   h_b[idx] /= diag;
   h_Ad[idx] /= diag;
   h_Bd[idx] /= diag;
   h_F[idx] /= diag;
   h_c[idx] = 1.0;
}
// Device-память
double* d_a, * d_b, * d_c, * d_Ad, * d_Bd, * d_F, * d_Y;
cudaMalloc(&d_a, size * sizeof(double));
cudaMalloc(&d_b, size * sizeof(double));
cudaMalloc(&d_c, size * sizeof(double));
cudaMalloc(&d_Ad, size * sizeof(double));
cudaMalloc(&d_Bd, size * sizeof(double));
cudaMalloc(&d_F, size * sizeof(double));
cudaMalloc(&d_Y, size * sizeof(double));
cudaMemcpy(d_a, h_a, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_c, h_c, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Ad, h_Ad, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Bd, h_Bd, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_F, h_F, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_Y, h_Y, size * sizeof(double), cudaMemcpyHostToDevice);
int threads = 16;
int blocksOdd = ((N + 1) / 2 + threads - 1) / threads;
int blocksEven = (N / 2 + threads - 1) / threads;
// Замер времени
cudaEvent_t evStart, evStop;
cudaEventCreate(&evStart);
cudaEventCreate(&evStop);
cudaEventRecord(evStart, 0);
for (int it = 0; it < cfg::MAX_IT; ++it)
{
```

```
        updateOdd <<<blocksOdd, threads>>> (d_a, d_c, d_b, d_Ad, d_Bd, d_Y, d_F);
        updateEven <<<blocksEven, threads>>> (d_a, d_c, d_b, d_Ad, d_Bd, d_Y, d_F);
    }
    cudaEventRecord(evStop, 0);
    cudaEventSynchronize(evStop);
    float gpu_ms = 0.0f;
    cudaEventElapsedTime(&gpu_ms, evStart, evStop);
    double gpu_sec = gpu_ms * 1e-3;
    cudaMemcpy(h_Y, d_Y, size * sizeof(double), cudaMemcpyDeviceToHost);
    // Результаты
    double maxErr = 0.0;
    for (size_t i = 0; i < size; ++i)
        maxErr = std::max(maxErr, std::fabs(h_Y[i] - 1.0));
    std::cout << std::fixed << std::setprecision(6) << "Время работы: " << gpu_sec <<
"c\n";
    std::cout << std::scientific << std::setprecision(6) << "Наибольшая погрешность с
точным решением : " << maxErr << '\n';
    //printGlobalMatrix(h_a, h_c, h_b, h_Ad, h_Bd);
    //std::cout << "\nF:\n";
    //printVector(h_F);
    //std::cout << "\nY:\n";
    //printVector(h_Y);
    // Чистка памяти
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cudaFree(d_Ad);
    cudaFree(d_Bd);
    cudaFree(d_F);
    cudaFree(d_Y);
    delete[] h_a;
    delete[] h_b;
    delete[] h_c;
    delete[] h_Ad;
    delete[] h_Bd;
    delete[] h_F;
    delete[] h_Y;
    return 0;
}
```