Министерство образования Республики Беларусь
Учреждение Образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра Информатики
Дисциплина «Архитектура вычислительных систем»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему:

«РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЕ С ПОМОЩЬЮ ТЕХНОЛОГИИ ‘CUDA’»

Выполнил студент группы 753505
Таланец Артём Витальевич
Руководитель:
ассистент кафедры информатики
Леченко Антон Владимирович

Минск 2019

Содержание

Предисловие

Решение систем линейных алгебраических уравнений - одна из основных задач вычислительной линейной алгебры. Хотя задача решения именно системы линейных уравнений сравнительно редко представляет самостоятельный интерес для прикладных задач, но от умения эффективно решать данные системы часто зависит сама возможность математического моделирования самых разнообразных процессов с применением ЭВМ. Значительная часть численных методов решения различных (в особенности - нелинейных) задач включает в себя решение систем линейных уравнений как элементарный шаг соответствующего алгоритма.

Данный курсовой проект описывает особенности решения СЛАУ с помощью технологии CUDA, реализацию алгоритмов решения и их тестирование.

1. Постановка задачи

Цель работы: реализация алгоритма решения СЛАУ на аппаратно-программной платформе CUDA.
Объект исследования: СЛАУ.
Предмет исследования: решения СЛАУ с помощью технологии CUDA.
Задачи:
Рассмотреть понятие СЛАУ и способы их решения. Проанализировать особенности решения систем линейных алгебраических уравнений на CUDA. Реализовать алгоритмы решения СЛАУ на CUDA.

2. Исторические сведения

Задачи, соответствующие современным задачам на составление и решение систем уравнений с несколькими неизвестными, встречаются еще в вавилонских и египетских рукописях II века до н.э., а также в трудах древнегреческих, индийских и китайских мудрецов. В китайском трактате "Математика в девяти книгах" словесно изложены правила решения систем уравнений, были замечены некоторые закономерности при решении.

Идею общего метода решения систем линейных уравнений высказал Лейбниц в 1693 году. Она была реализована швейцарским математиком Крамером в 1752 году. Он сформулировал и обосновал правило, носящее теперь его имя, которое позволяет решать системы n линейных уравнений с n неизвестными и буквенными коэффициентами. По правилу Крамера каждая неизвестная равна отношению двух определителей. Крамер, фактически, заложил основы теории определителей, хотя и не предложил для них удобного обозначения (это сделал в 1841 году А. Кэли). В 1772 году Вандермонд опубликовал обширное исследование определителей, один из которых носит теперь его имя. Систематическое изложение этой теории принадлежит Бине и Коши. Их труды по теории определителей относятся к периоду 1812-1815 гг.

3. Теоретическая часть

3.1. СЛАУ

Система линейных алгебраических уравнений (линейная система, также употребляются аббревиатуры СЛАУ, СЛУ) — система уравнений, каждое уравнение в которой является линейным — алгебраическим уравнением первой степени.

В классическом варианте коэффициенты при переменных, свободные члены и неизвестные считаются вещественными числами, но все методы и результаты сохраняются (либо естественным образом обобщаются) на случай любых полей, например, комплексных чисел.

Решение систем линейных алгебраических уравнений — одна из классических задач линейной алгебры, во многом определившая её объекты и методы. Кроме того, линейные алгебраические уравнения и методы их решения играют важную роль во многих прикладных направлениях, в том числе в линейном программировании, эконометрике.

Общий вид системы линейных алгебраических уравнений:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

Система линейных алгебраических уравнений может быть представлена в матричной форме как:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

или:

$Ax = b$

Одним из классических итерационных методов решения СЛАУ является метод Гаусса-Зейделя. Чтобы пояснить суть метода перепишем задачу в виде

$$\begin{cases} a_{11}x_1 & = & -a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n + b_1 \\ a_{21}x_1 + a_{22}x_2 & = & -a_{23}x_3 - \ldots - a_{2n}x_n + b_2 \\ \ldots & & \\ a_{(n-1)1}x_1 + a_{(n-1)2}x_2 + \ldots + a_{(n-1)(n-1)}x_{n-1} & = & -a_{(n-1)n}x_n + b_{n-1} \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{n(n-1)}x_{n-1} + a_{nn}x_n & = & b_n \end{cases}$$

Эта запись может быть представлена как

$(\mathrm{L} + \mathrm{D})\vec{x} = -\mathrm{U}\,\vec{x} + \vec{b},$

Итерационный процесс в методе Гаусса — Зейделя строится по формуле

$(\mathrm{L} + \mathrm{D})\vec{x}^{(k+1)} = -\mathrm{U}\vec{x}^{(k)} + \vec{b}, \quad k = 0, 1, 2, \ldots$

после выбора соответствующего начального приближения $\vec{x}^{(0)}$.

Метод Гаусса — Зейделя можно рассматривать как модификацию метода Якоби. Основная идея модификации состоит в том, что новые значения $\vec{x}^{(i)}$ используются здесь сразу же по мере получения, в то время как в методе Якоби они не используются до следующей итерации.

3.2. CUDA

CUDA (Compute Unified Device Architecture) — программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

CUDA SDK позволяет программистам реализовывать на специальных упрощённых диалектах языков программирования Си, C++ и Фортран алгоритмы, выполнимые на графических и тензорных процессорах Nvidia. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического или тензорного ускорителя и управлять его памятью. Функции, ускоренные при помощи CUDA, можно вызывать из различных языков, в т.ч. Python, MATLAB и т.п.

Первоначальная версия CUDA SDK была представлена 15 февраля 2007 года. В основе интерфейса программирования приложений CUDA лежит язык Си с некоторыми расширениями. Для успешной трансляции кода на этом языке в состав CUDA SDK входит собственный Си-компилятор командной строки nvcc компании Nvidia. Компилятор nvcc создан на основе открытого компилятора Open64 и предназначен для трансляции host-кода (главного, управляющего кода) и device-кода (аппаратного кода) (файлов с расширением .cu) в объектные файлы, пригодные в процессе сборки конечной программы или библиотеки в любой среде программирования, например, в NetBeans.

В архитектуре CUDA используется модель памяти грид, кластерное моделирование потоков и SIMD-инструкции. Применима не только для высокопроизводительных графических вычислений, но и для различных научных вычислений с использованием видеокарт nVidia. Учёные и исследователи широко используют CUDA в различных областях, включая астрофизику, вычислительную биологию и химию, моделирование динамики жидкостей, электромагнитных взаимодействий, компьютерную томографию, сейсмический анализ и многое другое. В CUDA имеется возможность подключения к приложениям, использующим OpenGL и Direct3D. CUDA — кроссплатформенное программное обеспечение для таких операционных систем, как Linux, Mac OS X и Windows.

22 марта 2010 года nVidia выпустила CUDA Toolkit 3.0, который содержал поддержку OpenCL.

Платформа CUDA впервые появились на рынке с выходом чипа NVIDIA восьмого поколения G80 и стала присутствовать во всех последующих сериях графических чипов, которые используются в семействах ускорителей GeForce, Quadro и NVidia Tesla.

Первая серия оборудования, поддерживающая CUDA SDK, G8x, имела 32-битный векторный процессор одинарной точности, использующий CUDA SDK как API (CUDA поддерживает тип double языка Си, однако сейчас его точность понижена до 32-битного с плавающей запятой). Более поздние процессоры GT200 имеют поддержку 64-битной точности (только для SFU), но производительность значительно хуже, чем для 32-битной точности (из-за того, что SFU всего два на каждый потоковый мультипроцессор, а скалярных процессоров — восемь). Графический процессор организует аппаратную многопоточность, что позволяет задействовать все ресурсы графического процессора. Таким образом, открывается перспектива переложить функции физического ускорителя на графический ускоритель (пример реализации — PhysX). Также открываются широкие возможности использования графического оборудования компьютера для выполнения сложных неграфических вычислений: например, в вычислительной биологии и в иных отраслях науки.

По сравнению с традиционным подходом к организации вычислений общего назначения посредством возможностей графических API, у архитектуры CUDA отмечают следующие преимущества в этой области:

- Интерфейс программирования приложений CUDA (CUDA API) основан на стандартном языке программирования Си с некоторыми ограничениями. По мнению разработчиков, это должно упростить и сгладить процесс изучения архитектуры CUDA
- Разделяемая между потоками память (shared memory) размером в 16 Кб может быть использована под организованный пользователем кэш с более широкой полосой пропускания, чем при выборке из обычных текстур
- Более эффективные транзакции между памятью центрального процессора и видеопамятью
- Полная аппаратная поддержка целочисленных и побитовых операций
- Поддержка компиляции кода GPU средствами открытого проекта LLVM

4. Основная часть

4.1. Реализация метода Гаусса-Зейделя с помощью CUDA

Реализуем заполнение матрицы генератором случайных чисел. Главная диагональ всегда будет содержать положительные ненулевые значения:

```
for (int i = 0; i < N; i++) {
    hostA[i + i * N] = rand() % 50 + 1.0f*N;
}
for (int k = 1; k < Num_diag + 1; k++) {
        for (int i = 0; i < N - k; i++) {
            hostA[i + k + i * N] = rand() % 5;
            hostA[i + (i + k)*N] = rand() % 5;
        }
}
```

Заполним столбец свободных членов b:

```
for (int i = 0; i < N; i++) {
    sum = 0.0f;
    for (int j = 0; j < N; j++) sum += hostA[j + i * N] * hostX[j];
    hostF[i] = sum;
}
```

Для численного решения СЛАУ построим точное решение x матрицы A используя генератор случайных чисел:

```
for (int i = 0; i < N; i++) {
    hostX[i] = rand() % 50;
    hostX0[i] = 1.0f;
    hostDelta[i] = 0.0f;
}
```

Идея распараллеливания итерационного процесса для GPU состоит в параллельных вычислениях i компонент вектора $xi(k+1)$ при фиксированном k. Таким образом номер нити на GPU будет соответствовать номеру i

Алгоритм решения СЛАУ на GPU будет работать как на девайсе, так и на хосте. На хост будет выполняться заполнение массива A, нахождение x и b, а так же выполнятся цикл по итерациям k и запускается функция ядро ответственная за итерации. Функция ядро выполняется на девайсе, где параллельно пересчитывает по вектору $xi(k)$ вектор $xi(k+1)$.

```
__global__ void KernelGaussSeidel(float* deviceA, float* deviceF, float* deviceX0, float* deviceX1, int N) {
    float sum = 0.0f;
    for (int i = 0; i < N; i++) deviceX0[i] = deviceX1[i];
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    for (int j = 0; j < t; j++) sum += deviceA[j + t * N] * deviceX1[j];
    for (int j = t + 1; j < N; j++) sum += deviceA[j + t * N] * deviceX0[j];
    deviceX1[t] = (deviceF[t] - sum) / deviceA[t + t * N];
}
```

Функция ядра вызывается на каждой итерациипо начальному приближению $xi(k)$находит следующее приближение решения $xi(k+1)$. . Каждая нить вычисляет свою компонент у вектора $xi(k+1)$.

Так как метод вычисляет компоненты приближения по нитям, то для того чтобы ядро могло однозначно определить номер нити, а значит, и элемент данных, который нужно обработать, необходимо использовать встроенные переменные $threadIdx$, $blockIdx$ типа $dim3$. Каждая из этих переменных является трехмерным целочисленным вектором. Отсюда следует определение типа $dim3$–вектор, принимающий значения сетки и блоков при вызове ядра.

```
int Block = (int)ceil((float)N / Thread);
dim3 Blocks(Block);
dim3 Threads(Thread);
```

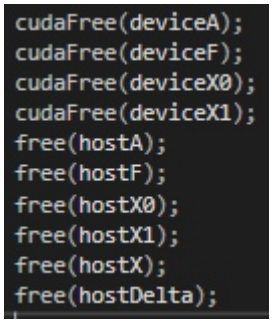
Переменная Thread будет отвечать за количество нитей в блоках. Сами блоки будут вычисляться как округленное ближайшее целое значение деления размерности матрицы на количество нитей.

Максимальное число нитей в блоке ограничено, так как все нити блока располагаются на одном потоковом процессоре и должны разделять ограниченное число ресурсов этого процессора. Блок нитей может содержать до 1024 нитей. Однако ядро может быть запущено на большем числе блоков. Поэтому общее число нитей будет равно произведению числа нитей в блоке и числа блоков.

Создадим переменные для девайса, в которые будут записываться данные с хоста.

```
cudaMalloc((void**)&deviceA, mem_sizeA);
cudaMalloc((void**)&deviceF, mem_sizeX);
cudaMalloc((void**)&deviceX0, mem_sizeX);
cudaMalloc((void**)&deviceX1, mem_sizeX);
cudaMalloc((void**)&delta, mem_sizeX);
cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);
```

После вычислений на GPU переменные ответственные за начальное приближение будут возвращены на хост.



Алгоритм решения СЛАУ на GPU будет работать как на девайс Переменная Thread будет отвечать за количество нитей в блоках. Сами блоки будут вычисляться как округленное ближайшее целое значение деления размерности матрицы на количество нитей. е, так и на хосте. На хост будет выполняться заполнение массива A, нахождение x и b, а так же выполнятся цикл по итерациям k и запускается функция ядро ответственная за итерации. Функция ядро выполняется на девайсе, где параллельно пересчитывает по вектору $xi(k)$ вектор $xi(k+1)$.

Вывод

В ходе проведенной работы была изучена архитектура видео ускорителей от NVidia, были изучен и реализован метод повышения производительности алгоритмов под графические процессоры Nvidia. Был реализован итерационный алгоритм решения систем линейных уравнений методом Гаусса-Зейделя.

В силы вышесказанного можно заключить, что поставленные задачи выполнены и цели достигнуты.

В заключение можно отметить, что графические процессоры от NVIDIA и программно-аппаратная архитектура параллельных вычислений Nvidia CUDA позволяют программистам использовать тысячи CUDA ядер для параллельных вычислений. Для простых вычислений глубокие знание архитектуры GPU не требуются. Однако только реализация программ под конкретную архитектуру конкретного GPU даст максимальную производительность.

Список использованных источников

1. Свободная энциклопедия Википедия, статья "Cuda" [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/CUDA
2. Свободная энциклопедия Википедия, статья " Система линейных алгебраических уравнений" [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BB%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D1%85_%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D1%85_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9
3. Свободная энциклопедия Википедия, статья " Метод Гаусса — Зейделя решения системы линейных уравнений" [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%93%D0%B0%D1%83%D1%81%D1%81%D0%B0_%E2%80%94_%D0%97%D0%B5%D0%B9%D0%B4%D0%B5%D0%BB%D1%8F_%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B_%D0%BB%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D1%85_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B9
4. Статья " Решение систем линейных алгебраических уравнений с использованием технологии CUDA" [Электронный ресурс]. – Режим доступа: https://dspace.tltsu.ru/bitstream/123456789/1050/1/%D0%9C%D0%B0%D0%BA%D0%B5%D0%B5%D0%B2_%D0%9F.%D0%92._%D0%9F%D0%9C%D0%98%D0%B1_1201.pdf

Приложение 1

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <device_functions.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <cstring>

#define Thread 1024

__global__ void KernelGaussSeidel(float* deviceA, float* deviceF, float*
deviceX0, float* deviceX1, int N) {
	float sum = 0.0f;
	for (int i = 0; i < N; i++) deviceX0[i] = deviceX1[i];
	int t = blockIdx.x * blockDim.x + threadIdx.x;
	for (int j = 0; j < t; j++) sum += deviceA[j + t * N] * deviceX1[j];
	for (int j = t + 1; j < N; j++) sum += deviceA[j + t * N] * deviceX0[j];
	deviceX1[t] = (deviceF[t] - sum) / deviceA[t + t * N];
}

__global__ void EpsGaussSeidel(float *deviceX0, float *deviceX1, float *delta,
int N) {
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	delta[i] += fabs(deviceX0[i] - deviceX1[i]);
	deviceX0[i] = deviceX1[i];
}

int main() {
	srand(time(NULL));
	float *hostA, *hostX, *hostX0, *hostX1, *hostF, *hostDelta;
	float sum, eps;
	float EPS = 1.e-5;
	int N = 10000;
	float size = N * N;
	int count;

	int Block = (int)ceil((float)N / Thread);
	dim3 Blocks(Block);
	dim3 Threads(Thread);

	int Num_diag = 0.5f*(int)N*0.3f;
	float mem_sizeA = sizeof(float)*size;
	unsigned int mem_sizeX = sizeof(float)*(N);
	hostA = (float*)malloc(mem_sizeA);
	hostF = (float*)malloc(mem_sizeX);
	hostX = (float*)malloc(mem_sizeX);
	hostX0 = (float*)malloc(mem_sizeX);
	hostX1 = (float*)malloc(mem_sizeX);
	hostDelta = (float*)malloc(mem_sizeX);
	for (int i = 0; i < size; i++) {
		hostA[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		hostA[i + i * N] = rand() % 50 + 1.0f*N;
	}
	for (int k = 1; k < Num_diag + 1; k++) {
```

```
		for (int i = 0; i < N - k; i++) {
			hostA[i + k + i * N] = rand() % 5;
			hostA[i + (i + k)*N] = rand() % 5;
		}
	}

	for (int i = 0; i < N; i++) {
		hostX[i] = rand() % 50;
		hostX0[i] = 1.0f;
		hostDelta[i] = 0.0f;
	}

	for (int i = 0; i < N; i++) {
		sum = 0.0f;
		for (int j = 0; j < N; j++) sum += hostA[j + i * N] * hostX[j];
		hostF[i] = sum;
	}

	float *deviceA, *deviceX0, *deviceX1, *deviceF, *delta;
	for (int i = 0; i < N; i++) hostX1[i] = 1.0f;

	cudaMalloc((void**)&deviceA, mem_sizeA);
	cudaMalloc((void**)&deviceF, mem_sizeX);
	cudaMalloc((void**)&deviceX0, mem_sizeX);
	cudaMalloc((void**)&deviceX1, mem_sizeX);
	cudaMalloc((void**)&delta, mem_sizeX);
	cudaMemcpy(deviceA, hostA, mem_sizeA, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceF, hostF, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX0, hostX0, mem_sizeX, cudaMemcpyHostToDevice);
	cudaMemcpy(deviceX1, hostX1, mem_sizeX, cudaMemcpyHostToDevice);

	count = 0; eps = 1.0f;
	while (eps > EPS)
	{
		count++;
		cudaMemcpy(delta, hostDelta, mem_sizeX, cudaMemcpyHostToDevice);
		KernelGaussSeidel << < Blocks, Threads >> > (deviceA, deviceF,
deviceX0,
			deviceX1, N);
		EpsGaussSeidel << < Blocks, Threads >> > (deviceX0, deviceX1, delta,
N);
		cudaMemcpy(hostDelta, delta, mem_sizeX, cudaMemcpyDeviceToHost);
		eps = 0.0f;
		for (int j = 0; j < N; j++) {
			eps += hostDelta[j]; hostDelta[j] = 0;
		}
		eps = eps / N;
	}
	cudaMemcpy(hostX1, deviceX1, mem_sizeX, cudaMemcpyDeviceToHost);

	cudaFree(deviceA);
	cudaFree(deviceF);
	cudaFree(deviceX0);
	cudaFree(deviceX1);
	free(hostA);
	free(hostF);
	free(hostX0);
	free(hostX1);
	free(hostX);
	free(hostDelta);

	return 0;
}
```