Министерство образования и науки

Государственное образовательное учреждение высшего профессионального образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

Факультет вычислительной математики и кибернетики

Кафедра математического обеспечения ЭВМ

Направление: информационные технологии

Отчет по лабораторной работе:

«Метод блочной прогонки для решения задачи построения квадратичного сплайна»

Выполнили: студенты группы 85М21

Замыслов С. А., Мошков С. А.

Нижний Новгород

2011

Содержание

Введение

Постоянное появление и совершенствование на рынке систем, реализующих параллельные вычисления, вызывает необходимость создания новых, параллельных версий алгоритмов.

В данной лабораторной работе рассматривается реализация (на архитектурах NVIDIA CUDA, Intel Threading Building Blocks (TBB) и NVIDIA OpenCL) и исследование параллельного алгоритма блочной прогонки, который, несмотря на линейную сложность последовательной версии алгоритма, позволяет снизить затраты на решение систем линейных уравнений с трехдиагональной матрицей.

Задача построения квадратичного сплайна, сводящаяся, в свою очередь, к такой системе линейных уравнений, также рассматривается в данной работе.

Постановка задачи

В данной лабораторной работе перед нами были поставлены следующие задачи:

1) Реализовать параллельный алгоритм блочной прогонки на архитектурах CUDA, TBB и OpenCL для решения задачи построения квадратичного сплайна и сравнить полученные результаты работы (время) с результатами работы последовательной версии алгоритма.
2) Получить практический опыт работы с технологиями CUDA, TBB и OpenCL.

Описание алгоритмов

Алгоритм параллельной блочной прогонки

Пусть каждый поток обрабатывает m = n / p строк матрицы A (n*n), т.е. k-ый поток обрабатывает строки с номерами 1+(k–1)*m <= i <= k*m. Будем предполагать, что число уравнений в системе кратно числу потоков.

В пределах полосы матрицы можно организовать исключение поддиагональных элементов (*прямой ход* метода): вычитание строки *i*, умноженной на константу b_{i+1}/a_i, из строки *i*+1с тем, чтобы результирующий коэффициент при неизвестной x_i в (*i*+1)-ой строке оказался нулевым.

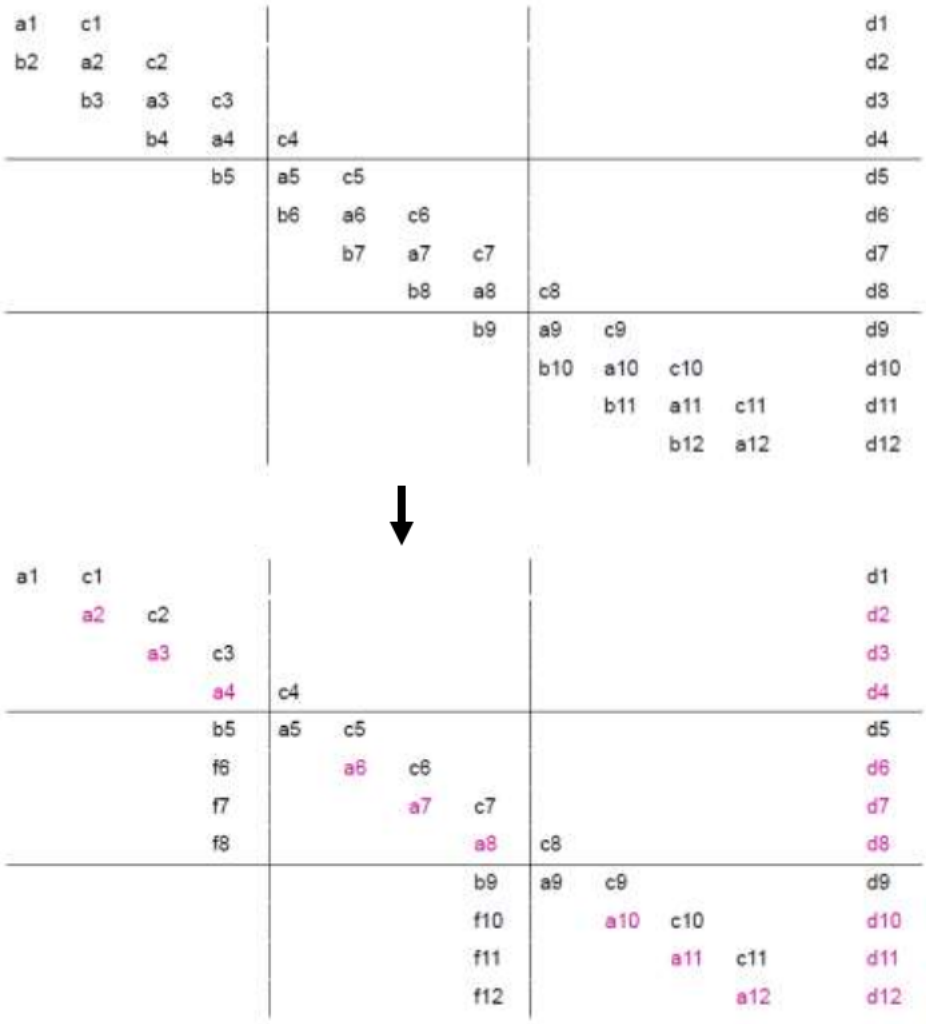

Рисунок 1. Прямой ход метода

Если исключение первым потоком поддиагональных переменных не добавит в матрицу новых коэффициентов, то исключение поддиагональных элементов в остальных потоках приведет к возникновению столбца отличных от нуля коэффициентов: во всех блоках (кроме первого) число ненулевых элементов в строке не изменится, но изменится структура уравнений. Модификации также подвергнутся элементы вектора правой части.

Затем выполняется обратный ход алгоритма – каждый поток исключает наддиагональные элементы, начиная со своего предпоследнего, и заканчивая последним для предыдущего блока матрицы.

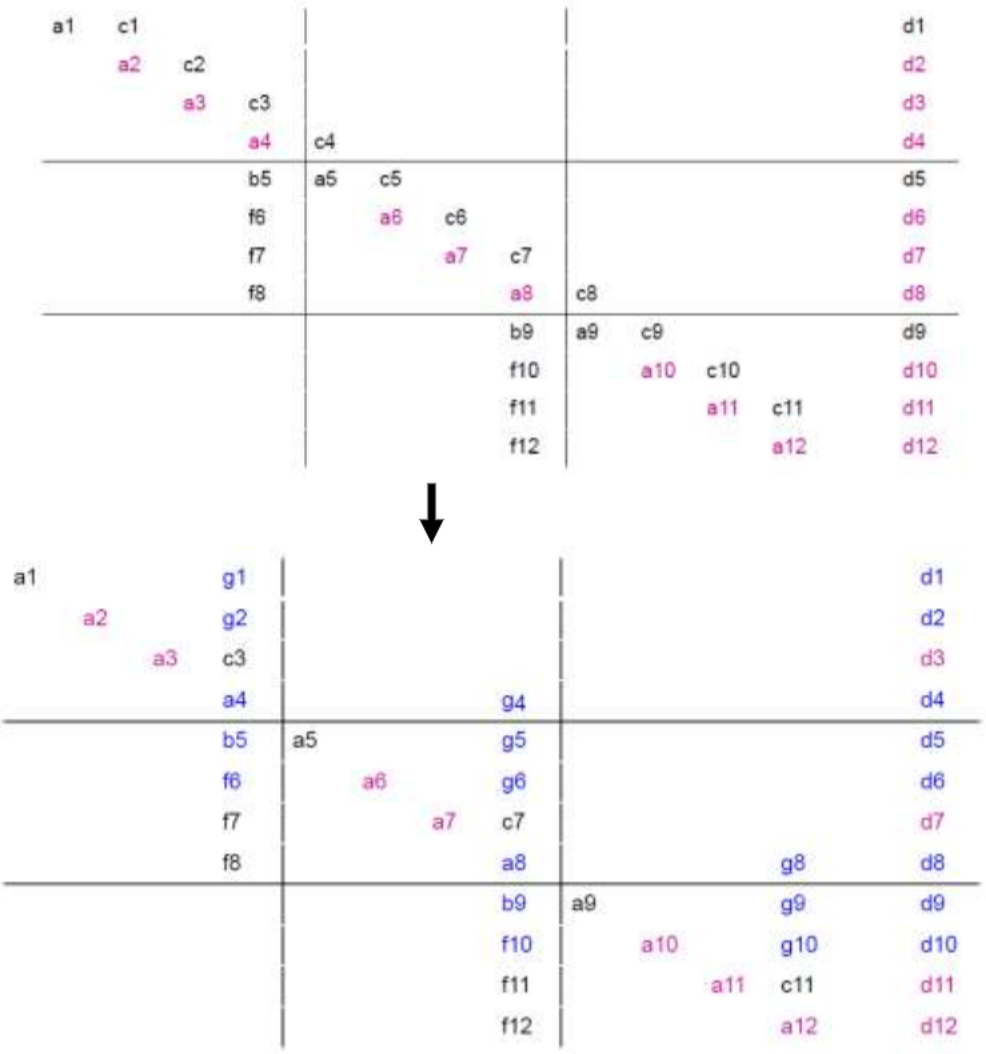

Рисунок 2. Обратный ход метода

Исключим из матрицы все строки каждой полосы, кроме последней, в результате получим систему уравнения относительно части исходных неизвестных.

Рисунок 3. Исключение внутренних строк блоков

Данная система будет содержать p уравнений и будет трехдиагональной. Ее можно решить последовательным методом прогонки. После того, как эта система будет решена, станут известны значения неизвестных на нижних границах полос разделения данных. Далее можно за один проход найти значения внутренних переменных в каждом потоке.

Построение квадратичного сплайна

Для решения задачи построения квадратичного сплайна используется метод Субботина.

Пусть аппроксимирующая кривая ($Q(x)$) может менять свою форму в точках $a = t_0 < t_1 < \ldots < t_n = b$. Определим узлы интерполяции (точки, в которых значения аппроксимируемой и аппроксимирующей кривых совпадают, $Q(p_i) = y_i$) следующим образом:

$$\begin{cases} p_0 = t_0, \\ p_{n+1} = t_n, \\ p_{i+1} = (t_i + t_{i+1}) / 2,\ 0 <= i <= n - 1. \end{cases}$$

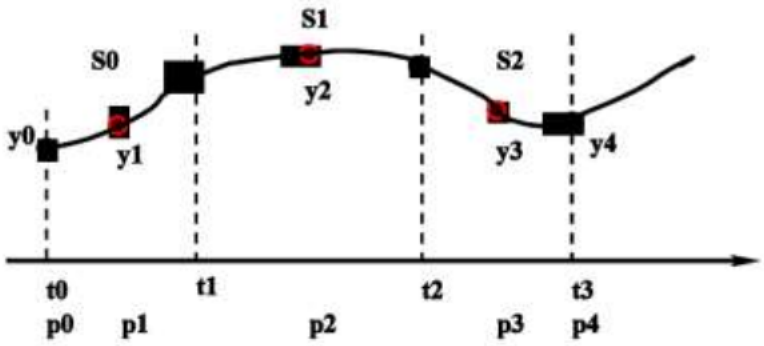

Рисунок 4. Выбор узлов интерполяции.

Функция $Q_i(x)$, определенная на интервале $[t_i, t_{i+1}]$ может быть записана следующим образом:

$$Q_i(x) = y_{i+1} + (z_{i+1} + z_i) * (x - p_{i+1}) / 2 + (z_{i+1} - z_i) * (x - p_{i+1})^2 / (2*h),$$ где $h = t_{i+1} - t_i$, $Q_i'(t_i) = z_i$.

Для каждой $Q_i(x)$ должны выполняться следующие условия:

$$\begin{cases} Q_i(p_{i+1}) = y_{i+1}, \\ Q_i'(t_i) = z_i, \\ Q_i'(t_{i+1}) = z_{i+1}, \\ \lim\limits_{x \to t_i^-} Q_{i-1}(x) = \lim\limits_{x \to t_i^+} Q_i(x) \quad (1 \le i \le n-1) \end{cases}$$

После нетрудных преобразований получаем рекурсивную зависимость:

$$h_{i-1} z_{i-1} + 3(h_{i-1} + h_i) z_i + h_i z_{i+1} = 8(y_{i+1} - y_i)$$, $1 <= i <= n - 1$.

Учитывая $Q(p_0) = y_0$, $Q(p_{n+1}) = y_{n+1}$ получаем еще два уравнения:

$$3h_0 z_0 + h_0 z_1 = 8(y_1 - y_0)$$
$$h_{n-1} z_{n-1} + 3h_{n-1} z_n = 8(y_{n+1} - y_n)$$

Матрица A полученной системы линейных уравнений $Ax=b$ является трехдиагональной. Найдя неизвестные $z_0, z_1, \ldots, z_n$ методом прогонки (при наличии достаточных условий – диагонального преобладания матрицы A) и подставив их в $Q_i(x)$, получим решение.

Реализация на CUDA

В данном разделе приведены основные моменты реализации описанного алгоритма параллельной блочной прогонки на CUDA C.

```
/* Данная функция выполняет решение системы уравнений с трехдиагональной матрицей
 * методом параллельной блочной прогонки.
 * Параметры:
 *     equationCount – число уравнений,
 *     subDiagonal   – массив поддиагональных элементов матрицы,
 *     mainDiagonal  – массив диагональных элементов матрицы,
 *     supDiagonal   – массив наддиагональных элементов матрицы,
 *     b             – массив свободных членов,
 *     x             – массив для значений неизвестных.
 *
 * Предполагаем, что число уравнений системы кратно числу строк матрицы в блоке,
 * а число блоков матрицы кратно числу потоков.
 */
void solveTDM_Block(int equationCount,
                    float* subDiagonal, float* mainDiagonal, float* supDiagonal,
                    float* b,
                    float* x)
{
    // Вычисляем число блоков матрицы.
    int matrixBlockCount = equationCount / ROWS_PER_MATRIX_BLOCK;

    // Инициализируем данные в памяти GPU.
    size_t dataElementSize = sizeof(float);

    float* deviceSubDiagonal = 0;
    float* deviceMainDiagonal = 0;
    float* deviceSupDiagonal = 0;
    float* deviceB = 0;
    float* deviceX = 0;
    float* deviceAuxiliaryMatrix = 0;

    cudaMalloc(&deviceSubDiagonal, dataElementSize * (equationCount - 1));
    cudaMalloc(&deviceMainDiagonal, dataElementSize * equationCount);
    cudaMalloc(&deviceSupDiagonal, dataElementSize * (equationCount - 1));
    cudaMalloc(&deviceB, dataElementSize * equationCount);
    cudaMalloc(&deviceX, dataElementSize * equationCount);
    cudaMalloc(&deviceAuxiliaryMatrix, dataElementSize * matrixBlockCount * 4);

    cudaMemcpy(deviceSubDiagonal, subDiagonal,
            dataElementSize * (equationCount - 1), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceMainDiagonal, mainDiagonal,
```

```
        dataElementSize * equationCount, cudaMemcpyHostToDevice);
cudaMemcpy(deviceSupDiagonal, supDiagonal,
        dataElementSize * (equationCount - 1), cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, b,
        dataElementSize * equationCount, cudaMemcpyHostToDevice);

// Вычисляем количество блоков CUDA.
int cudaBlockCount = matrixBlockCount / THREADS_PER_BLOCK;

/* Запускаем прямой ход метода (ядро).
 * В каждом потоке обрабатывается свой блок матрицы. */
solveTDM_BlockForward<<<cudaBlockCount, THREADS_PER_BLOCK>>>(
        equationCount,
        deviceSubDiagonal, deviceMainDiagonal, deviceSupDiagonal, deviceB);
cudaThreadSynchronize();

/* Запускаем обратный ход метода (ядро).
 * В каждом потоке обрабатывается свой блок матрицы. */
solveTDM_BlockBackward<<<cudaBlockCount, THREADS_PER_BLOCK>>>(
        equationCount,
        deviceSubDiagonal, deviceMainDiagonal, deviceSupDiagonal, deviceB);
cudaThreadSynchronize();

/* Для удобства собираем все коэффициенты
 * (deviceSubDiagonal, deviceMainDiagonal, deviceSupDiagonal, deviceB) в один
 * массив (ядро).
 * Каждый поток записывает коэффициенты своего блока матрицы. */
getAuxiliaryMatrixCoeffs<<<cudaBlockCount, THREADS_PER_BLOCK>>>(
        matrixBlockCount,
        deviceSubDiagonal, deviceMainDiagonal, deviceSupDiagonal, deviceB,
        deviceAuxiliaryMatrix);

// Выделяем память под измененную матрицу на хосте и копируем туда результаты
// из памяти устройства.
float* auxiliaryMatrix[5];
auxiliaryMatrix[0] = new float[matrixBlockCount - 1];
auxiliaryMatrix[1] = new float[matrixBlockCount];
auxiliaryMatrix[2] = new float[matrixBlockCount - 1];
auxiliaryMatrix[3] = new float[matrixBlockCount];
auxiliaryMatrix[4] = new float[matrixBlockCount];
cudaThreadSynchronize();

cudaMemcpy(auxiliaryMatrix[0], &deviceAuxiliaryMatrix[0 * matrixBlockCount],
        dataElementSize * (matrixBlockCount - 1), cudaMemcpyDeviceToHost);
cudaMemcpy(auxiliaryMatrix[1], &deviceAuxiliaryMatrix[1 * matrixBlockCount],
        dataElementSize * matrixBlockCount, cudaMemcpyDeviceToHost);
cudaMemcpy(auxiliaryMatrix[2], &deviceAuxiliaryMatrix[2 * matrixBlockCount],
```

```
        dataElementSize * (matrixBlockCount - 1), cudaMemcpyDeviceToHost);
cudaMemcpy(auxiliaryMatrix[3], &deviceAuxiliaryMatrix[3 * matrixBlockCount],
        dataElementSize * matrixBlockCount, cudaMemcpyDeviceToHost);

// Решаем полученную систему уравнений  обычным
// методом прогонки (число уравнений равно числу блоков матрицы).
solveTDM(matrixBlockCount,
        auxiliaryMatrix[0], auxiliaryMatrix[1], auxiliaryMatrix[2],
        auxiliaryMatrix[3],
        auxiliaryMatrix[4]);

// Копируем результаты обратно на устройство.
cudaMemcpy(deviceAuxiliaryMatrix, auxiliaryMatrix[4],
        dataElementSize * matrixBlockCount, cudaMemcpyHostToDevice);

/* Подставляем найденные неизвестные в общий вектор решений (ядро).
 * Каждый поток записывает коэффициенты своего блока матрицы. */
setX<<<cudaBlockCount, THREADS_PER_BLOCK>>>(
        deviceAuxiliaryMatrix,
        deviceX);
cudaThreadSynchronize();

/* Вычисляем оставшиеся значения неизвестных (ядро).
 * В каждом потоке вычисляются оставшиеся неизвестные в своем блоке матрицы. */
calculateX<<<cudaBlockCount, THREADS_PER_BLOCK>>>(
        equationCount,
        deviceSubDiagonal, deviceMainDiagonal, deviceSupDiagonal, deviceB,
        deviceX);
cudaThreadSynchronize();

// Копируем результаты на устройство.
cudaMemcpy(x, deviceX,
        dataElementSize * equationCount, cudaMemcpyDeviceToHost);

// ---

// Освобождаем память на хосте.
delete[] auxiliaryMatrix[0];
delete[] auxiliaryMatrix[1];
delete[] auxiliaryMatrix[2];
delete[] auxiliaryMatrix[3];
delete[] auxiliaryMatrix[4];

// Освобождаем память на устройстве.
if(deviceSubDiagonal != 0) {
         cudaFree(deviceSubDiagonal);
         deviceSubDiagonal = 0;
```

```
    }
    if(deviceMainDiagonal != 0) {
            cudaFree(deviceMainDiagonal);
            deviceMainDiagonal = 0;
    }
    if(deviceSupDiagonal != 0) {
            cudaFree(deviceSupDiagonal);
            deviceSupDiagonal = 0;
    }
    if(deviceB != 0) {
            cudaFree(deviceB);
            deviceB = 0;
    }
    if(deviceX != 0) {
            cudaFree(deviceX);
            deviceX = 0;
    }
    if(deviceAuxiliaryMatrix != 0) {
            cudaFree(deviceAuxiliaryMatrix);
            deviceAuxiliaryMatrix = 0;
    }
}
```

Реализация на TBB

В данном разделе приведены основные моменты реализации описанного алгоритма параллельной блочной прогонки на TBB.

```
/* Данная функция выполняет решение системы уравнений с трехдиагональной матрицей
 * методом параллельной блочной прогонки.
 * Параметры:
 *     equationCount – число уравнений,
 *     subDiagonal   – массив поддиагональных элементов матрицы,
 *     mainDiagonal  – массив диагональных элементов матрицы,
 *     supDiagonal   – массив наддиагональных элементов матрицы,
 *     b             – массив свободных членов,
 *    blockCount     – число блоков матрицы,
 *     x             – массив для значений неизвестных.
 *
 * Предполагаем, что число уравнений системы кратно числу строк матрицы в блоке,
 * а число блоков матрицы равно числу потоков.
 */

void solveTDM_Block(int equationCount,
                    float* _subDiagonal, float* _mainDiagonal, float* _supDiagonal,
                    float* _b,
                    int blockCount, float* x)
{
    size_t dataElementSize = sizeof(float);

    // Вычисляем количество строк в блоке
    int blockRowCount = equationCount / blockCount;
    float* auxiliaryMatrix[5];

    // Копируем исходные данные во вспомогательные массивы.
    float *subDiagonal  = clone(_subDiagonal, equationCount - 1);
    float *mainDiagonal = clone(_mainDiagonal, equationCount);
    float *supDiagonal  = clone(_supDiagonal, equationCount - 1);
    float *b            = clone(_b, equationCount);

    // Выполняем параллельно прямой ход метода (BlockForward – функтор)
    parallel_for(blocked_range<int>(0, blockCount, GRAINSIZE),
        BlockForward(subDiagonal, mainDiagonal, supDiagonal, b, blockRowCount));
      /*** end of step one ***/

    // Выполняем параллельно обратный ход метода (BlockBackward – функтор)
    parallel_for(blocked_range<int>(0, blockCount, GRAINSIZE),
        BlockBackward(subDiagonal, mainDiagonal, supDiagonal, b, blockRowCount));
      /*** end of step two ***/

    // Формируем вспомогательную матрицу из нижних уравнений блоков
    // (BlockAuxilaryMatrix – функтор)
    auxiliaryMatrix[0] = (float*)malloc(dataElementSize * (blockCount - 1));
    auxiliaryMatrix[1] = (float*)malloc(dataElementSize * blockCount);
    auxiliaryMatrix[2] = (float*)malloc(dataElementSize * (blockCount - 1));
    auxiliaryMatrix[3] = (float*)malloc(dataElementSize * blockCount);
    auxiliaryMatrix[4] = (float*)malloc(dataElementSize * blockCount);

    parallel_for(blocked_range<int>(0, blockCount, GRAINSIZE),
        BlockAuxilaryMatrix(subDiagonal, mainDiagonal, supDiagonal, b, blockRowCount,
                            auxiliaryMatrix, blockCount));
```

```
    // Решаем вспомогательную систему из нижних уравнений блоков методом прогонки
    solveTDM(blockCount,
             auxiliaryMatrix[0], auxiliaryMatrix[1], auxiliaryMatrix[2],
             auxiliaryMatrix[3],
             auxiliaryMatrix[4]);

    // Формируем результирующий вектор X
    // (BlockX - функтор)
    parallel_for(blocked_range<int>(0, blockCount, GRAINSIZE),
        BlockX(auxiliaryMatrix[4], x, blockRowCount));

    // Вычисляем оставшиеся элементы вектора X
    // (BlockCalculate - функтор)
    parallel_for(blocked_range<int>(0, blockCount, GRAINSIZE),
        BlockCalculate(subDiagonal, mainDiagonal, supDiagonal, b, x, blockRowCount));

    // Освобождаем ресурсы
    free(auxiliaryMatrix[0]);   free(auxiliaryMatrix[1]);   free(auxiliaryMatrix[2]);
    free(auxiliaryMatrix[3]);   free(auxiliaryMatrix[4]);
    free(subDiagonal);          free(mainDiagonal);         free(supDiagonal);
    free(b);
}
```

Результаты экспериментов

Для вычисления производительности работы последовательной и параллельной версий алгоритма были проведены эксперименты[1] на различных по объему входных данных. Результаты проведенных экспериментов представлены на рисунке 5.

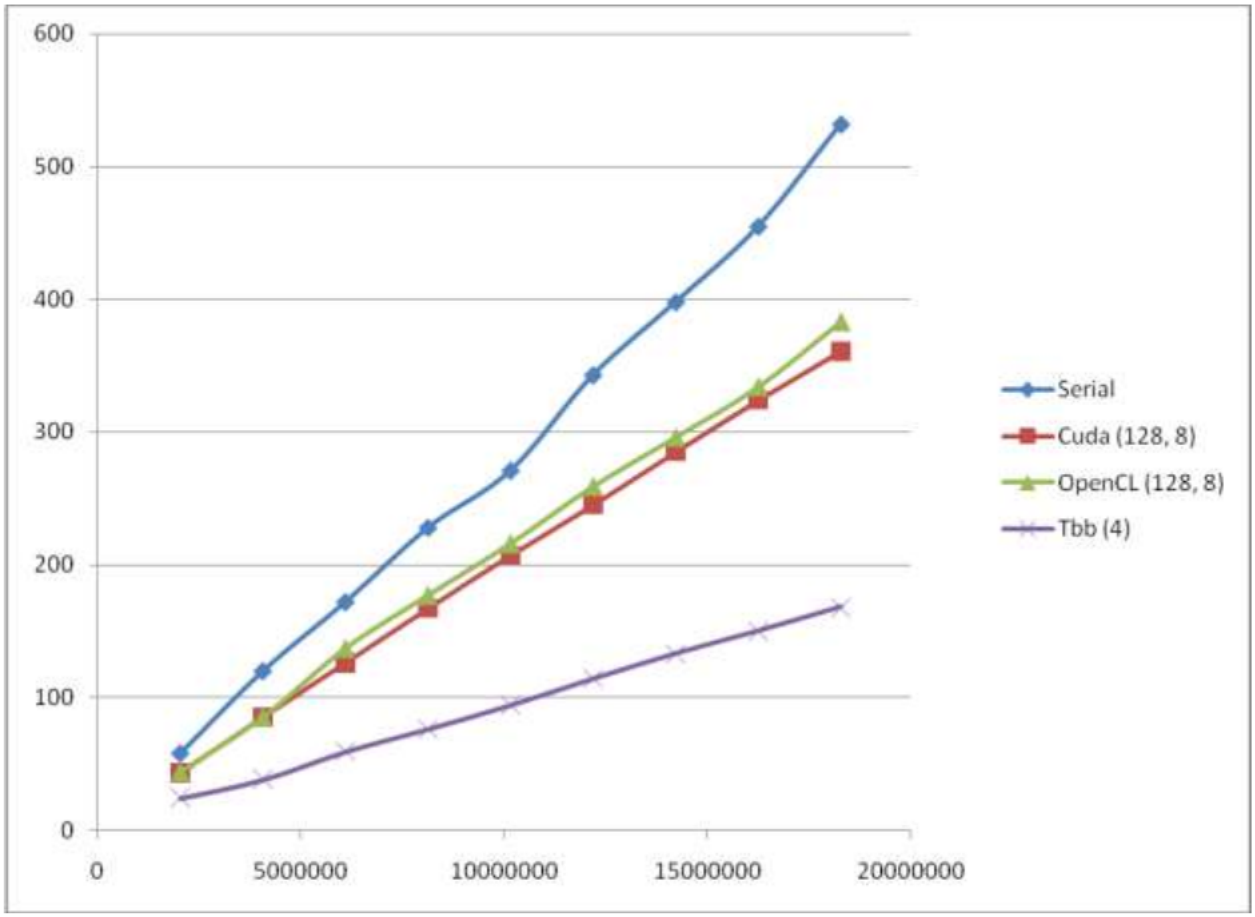

Рисунок №5. Сравнение времени работы последовательной (CPU) и параллельной (GPU) версий алгоритма; ось абсцисс – количество точек, в которых значение функции известно, ось ординат – время работы алгоритма (в миллисекундах); исходная функция $f(x) = \cos(x)/x$. Параметры экспериментов для Cuda и OpenCL (128, 8) означают число блоков потоков, которые будут запущены, и число потоков на блок, соответственно; параметр экспериментов для Tbb (4) означает число используемых потоков.

[1] Эксперименты проводились в системе с CPU Intel Core i5-2400 и GPU NVIDIA GeForce GTX460.

Для проверки корректности работы алгоритма был построен сплайн для функции *cos(x) / x* (рисунок 6).

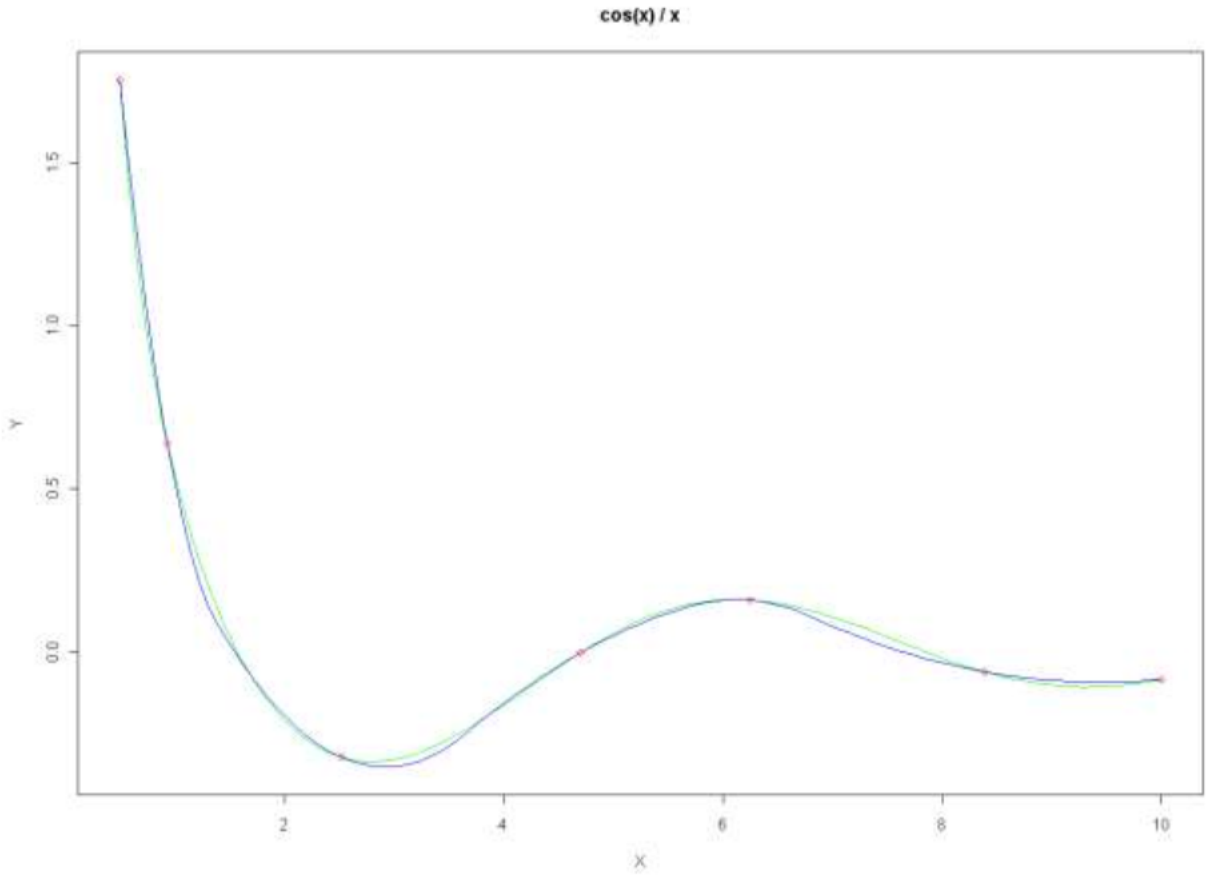

Рисунок №6. Функция *cos(x) / x* и сплайн, построенный по 7 узлам.

Выводы

На основании полученных результатов можно сделать следующие выводы:

- На малых объемах данных последовательная версия на CPU работает быстрее параллельной версии на GPU. Это связано с тем, что затраты на пересылку данных существенно больше затрат на вычисление.
- При увеличении объема выборки (объем выборки равен числу уравнений системы) параллельная версия работает быстрее.
- Трудоемкость алгоритма не зависит от вида аппроксимируемой функции.

Заключение

В результате выполнения данной работы были достигнуты следующие цели:

1. Реализован алгоритм параллельной блочной прогонки на CUDA, TBB и OpenCL.
2. Получен практический опыт работы с соответствующими технологиями.
3. Достигнутые показатели реализованного алгоритма параллельной блочной прогонки при больших объемах исходных данных превосходят аналогичные показатели последовательной версии алгоритма, что говорит о целесообразности использования параллельных алгоритмов и архитектур, поддерживающих параллельные вычисления в вычислительных задачах большой размерности.

Список литературы

1. Баркалов К. А., Мееров И. Б., Сысоев А. В. Презентация «Метод прогонки» по курсу «Параллельные численные методы», Н. Новгород, 2010.
2. Бастраков С. И. Материалы лекций по CUDA. Н. Новгород, 2011.
3. Сиднев А.А. Инструменты параллельного программирования в системах с общей памятью. Раздел 3. Разработка параллельных программ в системах с общей памятью с использованием библиотеки Intel Threading Building Blocks (TBB). Н. Новгород, 2010.
4. Numerical analysis. Approximation by Spline Functions. (http://www.cs.uky.edu/~jzhang/CS537/lecture6.pdf).