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 проекту на тему:

“Решение задачи Коши с помощью сопроцессора”

Выполнил студент группы 753505
Адасько Эдуард Геннадьевич
Руководитель: ассистент кафедры
информатики Леченко Антон Владимирович

Минск 2019

Содержание

Введение

Дифференциальные уравнения находят применение в различных областях науки: многие физические законы являются дифференциальными уравнениями, относительно некоторых функций; в биологии они используются, например, для описания популяции; широкое применение они находят и в моделях экономической динамики.

С ростом количества входных данных ставится вопрос о ускорении численного решения дифференциальных уравнений, а также их систем. Целью данной работы ставится рассмотрение возможности распараллеливания решения задачи Коши для обыкновенного дифференциального уравнения (ОДУ) первого порядка вида:

$$\begin{cases} y' = f(t, y(t)), t_0 \leq t \leq t_M \\ y(t_0) = y_0 \end{cases} \quad (1)$$

Распараллеливание будет проведено с использованием графического процессора (GPU), так как он имеет значительное преимущество перед CPU при параллельной обработке большого набора информации.

В качестве программного обеспечения будет использован фреймворк Metal, как современный инструмент для параллельных вычислений для платформ Apple.

Использованные аппаратные средства:

CPU - Intel Core i5 2.7 GHz (dual core)

GPU - Intel Iris Graphics 6100 1536 MB

1. Постановка задачи

Работа состоит из двух частей:

1. Реализация метода последовательных приближений Пикара для решения задачи Коши в 2 вариантах: последовательном и параллельном. Сравнение производительности.
2. Реализация параллельного алгоритма Parareal. Анализ производительности.

Для первой части работы в качестве задачи ставится улучшение производительности решения ОДУ с помощью GPU по сравнению с CPU. Для второй - сравнение второго метода распараллеливания задачи с первым и анализ их производительностей.

Для реализации поставленных задач будут использованы языки программирования C и Objective C, для последовательных и параллельных алгоритмов соответсвенно, а также технология Metal для реализации параллельных вычислений с помощью GPU.

Metal - это низкоуровневый интерфейс прикладного программирования для трехмерной графики и шейдеров с аппаратным ускорением, а также для параллельных вычислений с помощью GPU, разработанный Apple. Metal объединяет в одном API функции, подобные OpenGL и OpenCL.

2. Метод последовательных приближений Пикара

Метод последовательных приближений Пикара - один из алгоритмов для решения задачи Коши, который можно распараллелить по времени. Это означает, что вся область поиска решения задачи разбивается на подобласти, количество которых совпадает с количеством задействованных потоков. В каждой подобласти задача Коши решается одновременно и независимо используя заранее найденное приближенное решение задачи на левой границе области. Для ознакомления с этим методом была использована книга "Методы параллельных вычислений" (1, с. 169-172), где можно найти более подробное описание алгоритма. Для начала рассмотрим последовательный вариант алгоритма.

2.1. Последовательный алгоритм

Рассматриваем задачу Коши (1). Интегрируя ДУ, заменим задачу эквивалентным интегральным уравнением:

$$y(t) = y_0 + \int_a^t f(\tau, y(\tau)d\tau$$

Решая уравнение методом последовательных приближений, получим итерационный процесс Пикара:

$$y^{(k)}(t) = y_0 + \int_a^t f(\tau, y^{(k-1)}(\tau)d\tau, y^{(0)}(t) = y_0$$

Численная реализация метода выполняется с помощью квадратурных формул. Пусть

$y_m^{(k)}$ - приближенное решение задачи Коши в узле t_m на k-й итерации.

Используя квадратурную формулу трапеций получим:

$$y_{m+1}^{(k)} = y_m^{(k)} + I_m^{(k-1)}; y_m^{(0)} = y_0;$$

$$I_m^{(k-1)} = \frac{t_{m+1} - t_m}{2} \cdot (f(t_m, y_m^{(k-1)}) + f(t_{m+1}, y_{m+1}^{(k-1)})$$

Итерационный процесс прекращается при достижении заданной точности, например при

$$||y_m^{(k+1)} - y_m^{(k)}|| < \varepsilon$$

2.2. Параллельный алгоритм

Область поиска решенния разобьем на непересекающиеся подобласти количество которых совпадает с количеством потоков p. На каждом интервале приближенное решение расчитывается в M/p узлах.

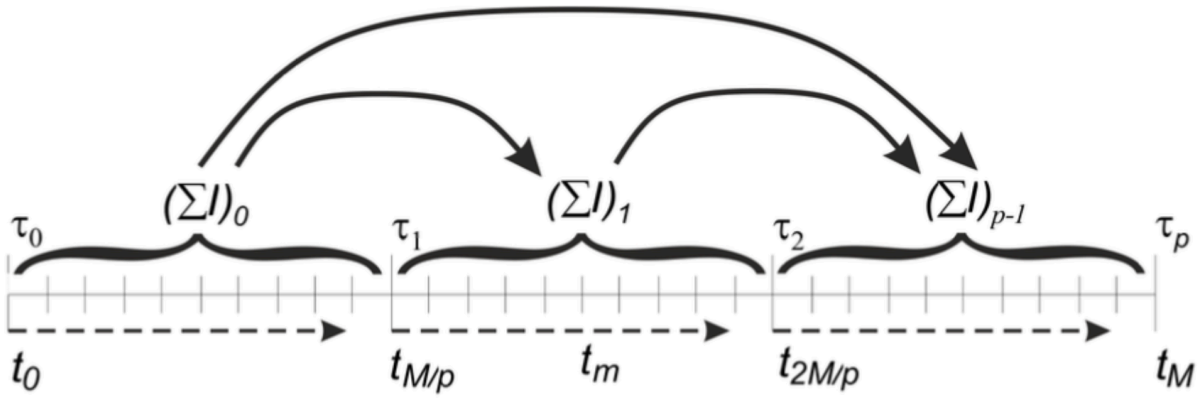
На каждой k-й итерации μ-й поток выполняет следующие действия:

1. Одновременно с другими потоками вычисляет значения определенных интегралов своей подобласти по квадратурной формуле.
2. Выполняет суммирование полученных значений и осуществляет пересылку рассчитанных сумм потокам с номерами больше μ.
3. Получив значения, производит вычисление y на левой границе интервала, а также все значения приближенного решения в своей подобласти:

$$y_{l+1+\mu M/p}^{(k)} = y_{l+\mu M/p}^{(k)} + I_{l+\mu M/p}^{(k-1)}, l = 0, 1, 2, M/p - 1$$

Точность вычислений в методе Пикара существенно зависит от величины интервала интегрирования и может быть повышена за счет применения адаптивных квадратурных формул.

Наглядно алгоритм можно увидеть на изображении:



2.3. Сравнение производительностей программных реализаций

Последовательный алгоритм был реализован с использованием языка C, а параллельный с использованием Objective C + Metal. После нахождения оптимального количества потоков удалось добиться ускорения в среднем в 3 раза по сравнению с последовательным алгоритмом. С исходным кодом kernel - функции, а также самого алгоритма можно ознакомиться в приложении.

Для демонстрации возьмем следующую задачу Коши:

$$\begin{cases} y' = \frac{2xy}{1+x^2}, 0 \leq x \leq 10 \\ y(0) = 1 \end{cases}$$

Аналитическим решением которого является функция $y(x) = 1 + x^2$.

Область решения разбивалась на 2^{20} точек. Количество потоков в параллельном алгоритме - 512. Итерационный процесс заканчивался, когда максимум разностей предыдущего и текущего решения в каждой точке становился меньше 0.0001. Было измерено время нахождения решения 50 уравнений каждого алгоритма. Результат:

```
Parallel time: 5.344612
y[xN] = 101.000290
Iterative time: 17.047558
y[xN] = 100.948753
```

где y[xN] - значения y в последней точке интервала. Необходимы для того, чтобы сверять, корректны ли найденные ответы.

Можно заметить, что при реализации параллельного метода после того как потоки параллельно высчитали значения интегралов и их сумму происходит пересылка суммы следующим потокам. Эта операция выполняется последовательно в промежутке между итерациями за $O(p^2)$, где p - число потоков. Поэтому с некоторого достаточно большого количества потоков алгоритм может начать терять свою производительность и вовсе начать выполняться медленнее последовательной версии.

3. Алгоритм Parareal

Parareal - параллельный алгоритм численного анализа, используемый для решения задач Коши. Изобретен в 2001 году. Также как и метод последовательных приближений Пикара является алгоритмом с временным параллелизмом. Рассмотрим описание алгоритма, а также его программную реализацию.

3.1. Описание алгоритма

Алгоритм основан на итеративном применении двух методов интегрирования. Один из них, обозначим как F, должен иметь высокую точность и соответственно высокие вычислительные затраты. Второй, обозначим как G, наоборот должен быть вычислительно дешевым, но может иметь малую точность.

Как и в предыдущем алгоритме разбиваем область поиска на подобласти с количеством равным количеству потоков. Далее сначала на каждой итерации с помощью дешевого алгоритма G вычисляем значения функции в граничных узлах. Этот алгоритм используем большой шаг равный длине подобласти, который обозначим за h. После чего, зная значения функции в граничных узлах, каждую подобласть разбиваем с помощью некоторого малого шага t на еще более мелкие подобласти и параллельно вычисляем значения для каждой подобласти с шагом t. В итоге в граничных узлах мы получили 2 значения функции - решения с использованием дешевого алгоритма с большим шагом и более точного алгоритма с меньшим шагом. Находим дефект:

$$\delta = F(y_\mu, \tau_\mu, \tau_{\mu+1}) - G(y_\mu, \tau_\mu, \tau_{\mu+1})$$

который будет использоваться для вычисления значения функции в граничных узлах в следующей итерации.

Следующая итерация начинается с нахождения граничных значений по формуле:

$$y_{\mu+1}^{k+1} = G(y_\mu^{k+1}, \tau_\mu, \tau_{\mu+1}) + F(y_\mu^k, \tau_\mu, \tau_{\mu+1}) - G(y_\mu^k, \tau_\mu, \tau_{\mu+1})$$

или

$$y_{\mu+1}^{k+1} = G(y_{\mu}^{k+1}, \tau_{\mu}, \tau_{\mu+1}) + \delta$$

Продолжаем итерации, пока не получим необходимую точность. В данном алгоритме значения функций F на каждой подобласти на каждой итерации вычисляются параллельно. Между итерациями необходимо последовательно вычислять значения функции G.

3.2. Программная реализация

В качестве точного метода был выбран метод Рунге-Кутта 4-го порядка. В качестве менее затратного метод Рунге-Кутта 2-го порядка. С исходным кодом kernel - функции, а также самого алгоритма можно ознакомиться в приложении.

С теми же входными данными, что и в пункте 1 получился следующий результат:

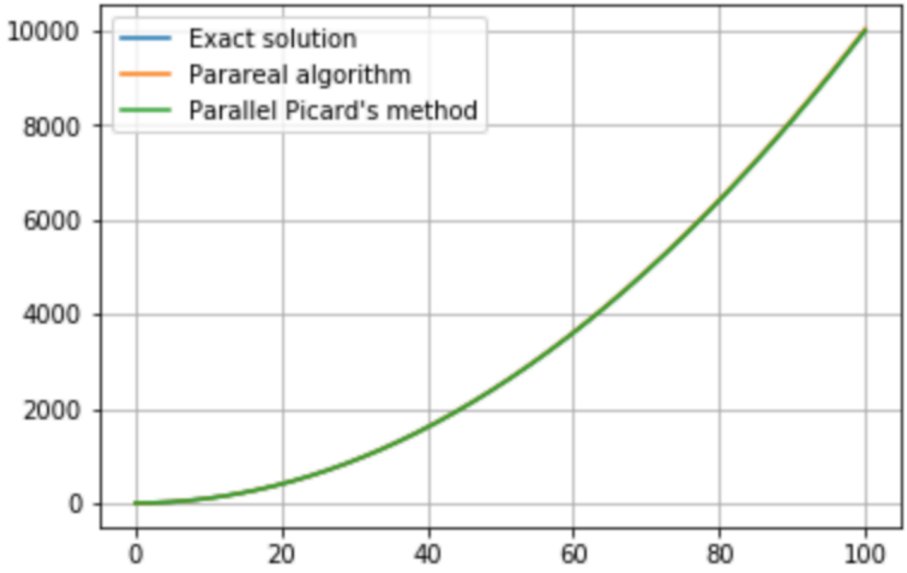
```
Parallel time: 2.545378
y[xN] = 101.093010
```

Получилось, что данный алгоритм быстрее в среднем быстрее в 2 раза, чем метод Пикара. Но его недостаток в том, что на некоторых входных данных могут возникать проблемы в сходимости. Решалось это следующим образом - итерации продолжались до тех пор, пока точность с каждой следующей итерацией повышалась. Как только наступала итерация, при которой точность начинала падать, решение останавливалось и предыдущее решение выдавалось как максимально точное. Понятно, что при таком подходе не всегда можно достичь заданной точности. Подробнее о вопросах сходимости метода можно прочесть в источниках 4-6.

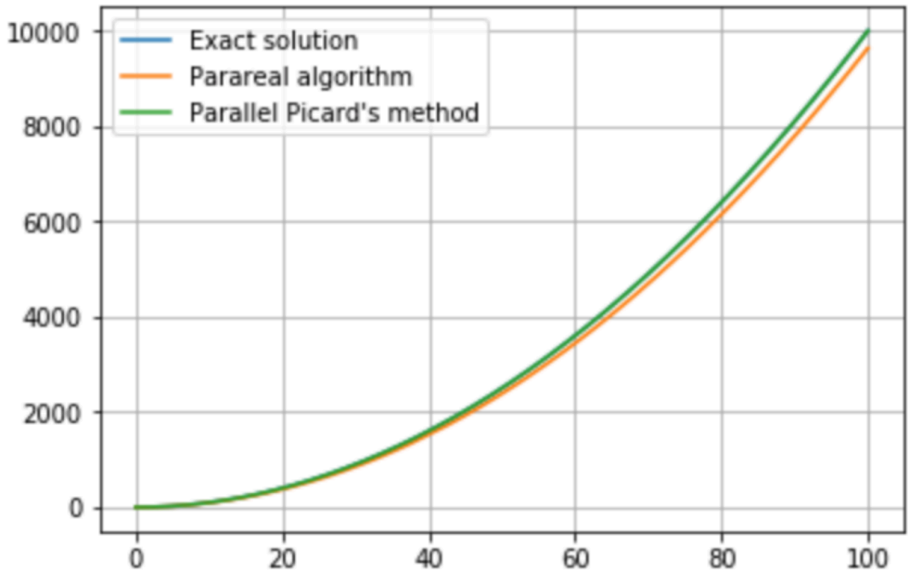
Заключение

В ходе работы были реализованы 2 параллельных метода решения задачи Коши с помощью GPU. В качестве наглядного сравнения с помощью языка Python и полученных данных были построены графики решений:

1. Интервал - [0, 100]. Количество точек - 1024 * 16. Количество потоков - 1024.



2. Интервал - [0, 100]. Количество точек - 1024. Количество потоков - 128.



Как видно из графиков, при достаточно малом шаге оба метода ведут себя хорошо. Но при увеличении шага и уменьшении количества потоков метод Parareal становится менее точным, чем метод Пикара. Такое поведение может наблюдаться при различных входных данных. Но его преимуществом является быстрота.

Список использованных источников

1. А. В. Старченко, В. Н. Берцун. Методы параллельных вычислений - Издательство Томского университета, 2013 - 224 с.

2. Официальная документация по Metal [электронный ресурс] - https://developer.apple.com/documentation/metal

3. Примеры программ с использованием Metal [электронный ресурс] - https://developer.apple.com/metal/sample-code/

4. Статья на википедии о алгоритме Parareal [электронный ресурс] - https://en.wikipedia.org/wiki/Parareal

5. Scott Field. Parareal methods [электронный ресурс] - http://cfm.brown.edu/people/jansh/page5/page10/page40/assets/Field_Talk.pdf

6. Christopherr Harden. Realtime Computing with the Parareal Algorithm [электронный ресурс] - http://diginole.lib.fsu.edu/islandora/object/fsu:182428/datastream/PDF/view

Приложение. Исходный код

Kernel и вспомогательные функции параллельного алгоритма Пикара:

```
float findIntegral(int i, device const float* x, device float* y) {
   return (x[i + 1] - x[i]) * (f(x[i], y[i]) + f(x[i + 1], y[i + 1])) / 2;
}

void setSums(
          device const float* x,
          device float* y,
          device float* currentThreadSums,
          device const int* numOfX,
          device const int* numOfThreads,
          device float* prevIntegrals,
          uint numOfCurrentThread) {
   float sum = 0;
   for (uint i = numOfCurrentThread * *numOfX / *numOfThreads;
           i < (numOfCurrentThread + 1) * *numOfX / *numOfThreads;
           i++) {
      prevIntegrals[i] = findIntegral(i, x, y);
      sum += prevIntegrals[i];
   }

   currentThreadSums[numOfCurrentThread] = sum;
}

kernel void solveODE(
                device const float* x,
                device float* y,
                device float* currentThreadSums,
                device float* totalSums,
                device float* prevIntegrals,
                device const int* numOfIteration,
                device const int* numOfX,
                device const int* numOfThreads,
                uint numOfCurrentThread [[thread_position_in_grid]]) {

   if(*numOfIteration != 0) {
      float sum = totalSums[numOfCurrentThread];
      y[numOfCurrentThread * *numOfX / *numOfThreads] = y[0] + sum;

      for (uint i = numOfCurrentThread * *numOfX / *numOfThreads + 1;
              i < (numOfCurrentThread + 1) * *numOfX / *numOfThreads;
               i++)
         y[i] = y[i - 1] + prevIntegrals[i - 1];
   }

   setSums(x, y, currentThreadSums, numOfX, numOfThreads, prevIntegrals, numOfCurrentThread);
}
```

Параллельный метод Пикара (PicardsParallelSolver.m):

```
#import "PicardsParallelSolver.h"
#include <time.h>

float x0;
float xN;
float yInit;

unsigned long numOfXs;
unsigned long bufferXsSize;
const unsigned long numOfThreads = 512;
const unsigned long numOfThreadsPerThreadgroup = 32;
const unsigned int bufferGroupsSize = numOfThreads * sizeof(float);
const unsigned int bufferNumOfXsSize = sizeof(long int);
const unsigned int bufferNumOfGroupsSize = sizeof(long int);
const unsigned int bufferNumOfIterationSize = sizeof(int);

@implementation PicardsMetalSolver {
    id<MTLDevice> _mDevice;
    id<MTLComputePipelineState> _mSolveFunctionPSO;
    id<MTLCommandQueue> _mCommandQueue;
    id<MTLBuffer> _mBufferXs;
    id<MTLBuffer> _mBufferYs;
    id<MTLBuffer> _mBufferCurrentThreadSums;
    id<MTLBuffer> _mBufferTotalSums;
    id<MTLBuffer> _mBufferPrevIntegrals;
    id<MTLBuffer> _mBufferNumOfThreads;
    id<MTLBuffer> _mBufferNumOfXs;
    id<MTLBuffer> _mBufferNumOfIteration;
}

- (instancetype) initWithDevice: (id<MTLDevice>) device {
    self = [super init];
    if (self) {
        _mDevice = device;
        NSError* error = nil;
        id<MTLLibrary> defaultLibrary = [_mDevice newDefaultLibrary];
        id<MTLFunction> solveFunction = [defaultLibrary newFunctionWithName:@"solveODE"];
        _mSolveFunctionPSO = [_mDevice newComputePipelineStateWithFunction:
                              solveFunction error:&error];
        _mCommandQueue = [_mDevice newCommandQueue];
    }
    return self;
}

- (void) setX0: (float) initX0
         setXN: (float) initXN
         setY0: (float) initY0
         setNumOfX:(unsigned long int) initNumX {
    x0 = initX0;
    xN = initXN;
    yInit = initY0;
    numOfXs = initNumX;
    bufferXsSize = numOfXs * sizeof(float);

    _mBufferXs = [_mDevice newBufferWithLength:bufferXsSize
options:MTLResourceStorageModeShared];
    _mBufferYs = [_mDevice newBufferWithLength:bufferXsSize
options:MTLResourceStorageModeShared];
```

```
    _mBufferPrevIntegrals = [_mDevice newBufferWithLength:bufferXsSize
options:MTLResourceStorageModeShared];
    _mBufferCurrentThreadSums = [_mDevice newBufferWithLength:bufferGroupsSize
options:MTLResourceStorageModeShared];
    _mBufferTotalSums = [_mDevice newBufferWithLength:bufferGroupsSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfXs = [_mDevice newBufferWithLength:bufferNumOfXsSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfThreads = [_mDevice newBufferWithLength:bufferNumOfGroupsSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfIteration = [_mDevice newBufferWithLength:bufferNumOfIterationSize
options:MTLResourceStorageModeShared];

    generateXs(_mBufferXs);
    generateYs(_mBufferYs);
    generateNums(_mBufferNumOfXs, _mBufferNumOfThreads, _mBufferNumOfIteration);
}

void generateXs(id<MTLBuffer> buffer) {
    float* dataPtr = buffer.contents;
    float h = (xN - x0) / (numOfXs - 1);

    dataPtr[0] = x0;
    dataPtr[numOfXs - 1] = xN;

    for (unsigned long index = 1; index < numOfXs - 1; index++)
        dataPtr[index] = dataPtr[index - 1] + h;
}

void generateYs(id<MTLBuffer> buffer) {
    float* dataPtr = buffer.contents;
    for (unsigned long index = 0; index < numOfXs; index++)
        dataPtr[index] = yInit;
}

void generateNums(id<MTLBuffer> bufferNumOfXs, id<MTLBuffer> bufferNumOfGroups,
                  id<MTLBuffer> bufferNumOfIteration) {
    long int* numXs = bufferNumOfXs.contents;
    *numXs = numOfXs;
    long int* numGroups = bufferNumOfGroups.contents;
    *numGroups = numOfThreads;
    int* numIteration = bufferNumOfIteration.contents;
    *numIteration = 0;
}

- (void) nextIteration {
    int* numIteration = _mBufferNumOfIteration.contents;
    *numIteration += 1;
}

- (void) sendComputeCommand {
    id<MTLCommandBuffer> commandBuffer = [_mCommandQueue commandBuffer];
    assert(commandBuffer != nil);

    id<MTLComputeCommandEncoder> computeEncoder = [commandBuffer
computeCommandEncoder];
    assert(computeEncoder != nil);

    [self encodeSolveCommand:computeEncoder];
```

```
    [computeEncoder endEncoding];
    [commandBuffer commit];
    [commandBuffer waitUntilCompleted];
}

- (void) encodeSolveCommand:(id<MTLComputeCommandEncoder>)computeEncoder {
    [computeEncoder setComputePipelineState:_mSolveFunctionPSO];
    [computeEncoder setBuffer:_mBufferXs offset:0 atIndex:0];
    [computeEncoder setBuffer:_mBufferYs offset:0 atIndex:1];
    [computeEncoder setBuffer:_mBufferCurrentThreadSums offset:0 atIndex:2];
    [computeEncoder setBuffer:_mBufferTotalSums offset:0 atIndex:3];
    [computeEncoder setBuffer:_mBufferPrevIntegrals offset:0 atIndex:4];
    [computeEncoder setBuffer:_mBufferNumOfIteration offset:0 atIndex:5];
    [computeEncoder setBuffer:_mBufferNumOfXs offset:0 atIndex:6];
    [computeEncoder setBuffer:_mBufferNumOfThreads offset:0 atIndex:7];

    MTLSize threadsPerThreadgroup = MTLSizeMake(numOfThreadsPerThreadgroup, 1, 1);
    MTLSize threadsPerGrid = MTLSizeMake(numOfThreads, 1, 1);
    [computeEncoder dispatchThreads: threadsPerGrid
                threadsPerThreadgroup: threadsPerThreadgroup];
}

-(float*) getResult {
    float* yCheck = _mBufferYs.contents;
    return yCheck;
}

- (void) setTotalSums {
    setTotalSums(_mBufferTotalSums, _mBufferCurrentThreadSums);
}

void setTotalSums(id<MTLBuffer> totalSums, id<MTLBuffer> currentSums) {
    float* tSums = totalSums.contents;
    float* cSums = currentSums.contents;

    for (uint i = 0 ; i < numOfThreads; i++) {
        tSums[i] = 0;
    }

    for (uint i = 0; i < numOfThreads; i++) {
        for (uint j = i + 1; j < numOfThreads; j++) {
            tSums[j] += cSums[i];
        }
    }
}

@end

float getMaxDifference(float* answer, float* nextAnswer, unsigned long numX) {
    float maxDifference = FLT_MIN;

    for (int i = 0; i < numX; i ++) {
        float diff = fabsf(nextAnswer[i] - answer[i]);
        if (diff > maxDifference)
            maxDifference = diff;
    }

    return maxDifference;
}
```

```
float* parallelPicardsMethod(float x0, float xN, float y0, unsigned long numX, double* time) {
    const float error = 0.0001;
    float* answer = (float*)malloc(numX * sizeof(int));

    for (int i = 0; i < numX; i++)
        answer[i] = FLT_MAX;

    id<MTLDevice> device = MTLCreateSystemDefaultDevice();

    PicardsMetalSolver* solver = [[PicardsMetalSolver alloc] initWithDevice:device];

    [solver setX0: x0
          setXN: xN
          setY0: y0
          setNumOfX: numX];

    NSDate *start = [NSDate date];
    NSTimeInterval sumTime = 0;
    [solver sendComputeCommand];
    float* nextAnswer = [solver getResult];

    sumTime += [start timeIntervalSinceNow];

    while (getMaxDifference(answer, nextAnswer, numX) > error) {
        for (int i = 0; i < numX; i ++)
            answer[i] = nextAnswer[i];

        NSDate *start = [NSDate date];

        [solver nextIteration];
        [solver setTotalSums];
        [solver sendComputeCommand];
        nextAnswer = [solver getResult];

        sumTime += [start timeIntervalSinceNow];
    }

    *time += (double)fabs(sumTime);

    return nextAnswer;
}
```

Kernel и вспомогательные функции алгоритма Parareal:

```
float rungeKutta4(float prevX, float prevY, float dt) {
    float k1 = f(prevX, prevY);
    float k2 = f(prevX + dt / 2, prevY + dt / 2 * k1);
    float k3 = f(prevX + dt / 2, prevY + dt / 2 * k2);
    float k4 = f(prevX + dt, prevY + dt * k3);

    return prevY + dt / 6 * (k1 + 2 * k2 + 2 * k3 + k4);
}



kernel void solveODE(
                device const float* x,
                device float* y,
```

```
                device float* coarseGridValues,
                device const int* numOfIteration,
                device const int* numOfX,
                device const float* dt,
                device const int* numOfThreads,
                uint numOfCurrentThread [[thread_position_in_grid]]) {
    int startIndex = numOfCurrentThread * (*numOfX - 1) / *numOfThreads;

    y[startIndex + 1] = rungeKutta4(x[startIndex], coarseGridValues[numOfCurrentThread], *dt);

    for (uint i = startIndex + 2;
        i < (numOfCurrentThread + 1) * (*numOfX - 1) / *numOfThreads + 1; i++)
        y[i] = rungeKutta4(x[i - 1], y[i - 1], *dt);
}
```

Алгоритм Parareal (PararealSolver.m):

```
#import "PararealSolver.h"
#import "rightSide.h"
#include <time.h>

float x0;
float xN;
float yInit;

unsigned long numOfXs;
unsigned long bufferXsSize;
const unsigned long numOfThreads = 1024;
const unsigned long numOfThreadsPerThreadgroup = 32;

const unsigned int bufferGroupsSize = numOfThreads * sizeof(float);
const unsigned int bufferCoarseYsSize = (numOfThreads + 1) * sizeof(float);
const unsigned int bufferNumOfXsSize = sizeof(long int);
const unsigned int bufferDTSize = sizeof(float);
const unsigned int bufferNumOfGroupsSize = sizeof(long int);
const unsigned int bufferNumOfIterationSize = sizeof(int);

@implementation PararealSolver {
    id<MTLDevice> _mDevice;
    id<MTLComputePipelineState> _mSolveFunctionPSO;
    id<MTLCommandQueue> _mCommandQueue;
    id<MTLBuffer> _mBufferXs;
    id<MTLBuffer> _mBufferYs;
    id<MTLBuffer> _mBufferCoarseYs;
    id<MTLBuffer> _mBufferDT;
    id<MTLBuffer> _mBufferNumOfThreads;
    id<MTLBuffer> _mBufferNumOfXs;
    id<MTLBuffer> _mBufferNumOfIteration;
}

- (instancetype) initWithDevice: (id<MTLDevice>) device {
    self = [super init];
    if (self) {
        _mDevice = device;
        NSError* error = nil;
        id<MTLLibrary> defaultLibrary = [_mDevice newDefaultLibrary];
        id<MTLFunction> solveFunction = [defaultLibrary newFunctionWithName:@"solveODE"];
```

```
        _mSolveFunctionPSO = [_mDevice newComputePipelineStateWithFunction: solveFunction
error:&error];
        _mCommandQueue = [_mDevice newCommandQueue];
    }

    return self;
}

- (void) setX0: (float) initX0
         setXN: (float) initXN
         setY0: (float) initY0
     setNumOfX:(unsigned long int) initNumX {
    x0 = initX0;
    xN = initXN;
    yInit = initY0;
    numOfXs = initNumX;
    bufferXsSize = numOfXs * sizeof(float);

    _mBufferXs = [_mDevice newBufferWithLength:bufferXsSize
options:MTLResourceStorageModeShared];
    _mBufferYs = [_mDevice newBufferWithLength:bufferXsSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfXs = [_mDevice newBufferWithLength:bufferNumOfXsSize
options:MTLResourceStorageModeShared];
    _mBufferCoarseYs = [_mDevice newBufferWithLength:bufferCoarseYsSize
options:MTLResourceStorageModeShared];
    _mBufferDT= [_mDevice newBufferWithLength:bufferDTSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfThreads = [_mDevice newBufferWithLength:bufferNumOfGroupsSize
options:MTLResourceStorageModeShared];
    _mBufferNumOfIteration = [_mDevice newBufferWithLength:bufferNumOfIterationSize
options:MTLResourceStorageModeShared];

    generateXs(_mBufferXs, _mBufferDT);
    generateYs(_mBufferYs);
    generateNums(_mBufferNumOfXs,
                 _mBufferNumOfThreads,
                 _mBufferNumOfIteration);
}

- (void) setCoarseValues {
    generateCoarseValues(_mBufferCoarseYs, _mBufferXs);
}

- (void) updateCoarseValues {
    correctCoarseGridValues(_mBufferCoarseYs, _mBufferYs, _mBufferXs);
}

void generateXs(id<MTLBuffer> bufferXs, id<MTLBuffer> bufferDT) {
    float* dataPtr = bufferXs.contents;
    float h = (xN - x0) / (numOfXs - 1);

    dataPtr[0] = x0;
    dataPtr[numOfXs - 1] = xN;

    for (unsigned long index = 1; index < numOfXs - 1; index++)
        dataPtr[index] = dataPtr[index - 1] + h;

    float* dt = bufferDT.contents;
    *dt = h;
```

```
}

void generateYs(id<MTLBuffer> buffer) {
   float* dataPtr = buffer.contents;

   for (unsigned long index = 0; index < numOfXs; index++)
      dataPtr[index] = yInit;
}

void generateNums(id<MTLBuffer> bufferNumOfXs,
                  id<MTLBuffer> bufferNumOfGroups,
                  id<MTLBuffer> bufferNumOfIteration) {
   long int* numXs = bufferNumOfXs.contents;
   *numXs = numOfXs;

   long int* numGroups = bufferNumOfGroups.contents;
   *numGroups = numOfThreads;

   int* numIteration = bufferNumOfIteration.contents;
   *numIteration = 0;
}

float rungeKutta2 (float prevX, float prevY, float dT) {
   return prevY + dT * f(prevX + dT / 2, prevY + dT / 2 * f(prevX, prevY));
}

void generateCoarseValues(id<MTLBuffer> bufferCoarse, id<MTLBuffer> bufferXs) {
   float* coarseGridValues = bufferCoarse.contents;

   coarseGridValues[0] = yInit;

   float dT = (xN - x0) / (numOfThreads - 1);
   float* xs = bufferXs.contents;

   for (int i = 1; i < numOfThreads + 1; i++)
      coarseGridValues[i] = rungeKutta2(xs[(i - 1) * (numOfXs - 1) /  numOfThreads],
                                        coarseGridValues[i - 1], dT);
}

void correctCoarseGridValues(id<MTLBuffer> bufferCoarse,
                             id<MTLBuffer> bufferYs,
                             id<MTLBuffer> bufferXs) {
   float* coarseGridValues = bufferCoarse.contents;
   float* ys = bufferYs.contents;
   float* xs = bufferXs.contents;
   float dT = (xN - x0) / (numOfThreads - 1);

   float* defect = (float*)malloc((numOfThreads + 1) * sizeof(float));

   for (int i = 0; i < numOfThreads + 1; i++)
      defect[i] = ys[i * (numOfXs - 1) /  numOfThreads] - coarseGridValues[i];

   coarseGridValues[0] = yInit;

   for (int i = 1; i < numOfThreads + 1; i++)
      coarseGridValues[i] = rungeKutta2(xs[(i - 1) * (numOfXs - 1) /  numOfThreads],
                                        coarseGridValues[i - 1], dT) + defect[i - 1];

   free (defect);
}
```

```
- (void) nextIteration {
    int* numIteration = _mBufferNumOfIteration.contents;
    *numIteration += 1;
}

- (void) sendComputeCommand {
    id<MTLCommandBuffer> commandBuffer = [_mCommandQueue commandBuffer];
    assert(commandBuffer != nil);

    id<MTLComputeCommandEncoder> computeEncoder = [commandBuffer
computeCommandEncoder];
    assert(computeEncoder != nil);

    [self encodeSolveCommand:computeEncoder];

    [computeEncoder endEncoding];
    [commandBuffer commit];
    [commandBuffer waitUntilCompleted];
}

- (void)encodeSolveCommand:(id<MTLComputeCommandEncoder>)computeEncoder {
    [computeEncoder setComputePipelineState:_mSolveFunctionPSO];
    [computeEncoder setBuffer:_mBufferXs offset:0 atIndex:0];
    [computeEncoder setBuffer:_mBufferYs offset:0 atIndex:1];
    [computeEncoder setBuffer:_mBufferCoarseYs offset:0 atIndex:2];
    [computeEncoder setBuffer:_mBufferNumOfIteration offset:0 atIndex:3];
    [computeEncoder setBuffer:_mBufferNumOfXs offset:0 atIndex:4];
    [computeEncoder setBuffer:_mBufferDT offset:0 atIndex:5];
    [computeEncoder setBuffer:_mBufferNumOfThreads offset:0 atIndex:6];

    MTLSize threadsPerThreadgroup = MTLSizeMake(numOfThreadsPerThreadgroup, 1, 1);
    MTLSize threadsPerGrid = MTLSizeMake(numOfThreads, 1, 1);
    [computeEncoder dispatchThreads: threadsPerGrid
            threadsPerThreadgroup: threadsPerThreadgroup];
}

-(float*) getResult {
    float* yCheck = _mBufferYs.contents;
    return yCheck;
}

@end

float getMaxDifference(float* answer, float* nextAnswer, unsigned long numX) {
    float maxDifference = FLT_MIN;

    for (int i = 0; i < numOfThreads + 1; i ++) {
        float diff = fabsf(nextAnswer[i * (numOfXs - 1) /  numOfThreads] -
                    answer[i * (numOfXs - 1) /  numOfThreads]);

        if (diff > maxDifference)
            maxDifference = diff;
    }

    return maxDifference;
}


float* pararealMethod(float x0, float xN, float y0,
```

```
                unsigned long numX, double* time) {
    const float error = 0.0001;

    numX ++;
    float* answer = (float*)malloc(numX * sizeof(int));

    for (int i = 0; i < numX; i++)
        answer[i] = FLT_MAX;

    float prevDifference = FLT_MAX;

    id<MTLDevice> device = MTLCreateSystemDefaultDevice();

    PararealSolver* solver = [[PararealSolver alloc] initWithDevice:device];

    [solver setX0: x0
          setXN: xN
          setY0: y0
       setNumOfX: numX];

    NSTimeInterval sumTime = 0;

    NSDate *start = [NSDate date];
    [solver setCoarseValues];
    [solver sendComputeCommand];
    float* nextAnswer = [solver getResult];

    sumTime += [start timeIntervalSinceNow];

    float difference = getMaxDifference(answer, nextAnswer, numX);
    while (difference > error && difference <= prevDifference) {
        for (int i = 0; i < numX; i ++)
            answer[i] = nextAnswer[i];

        NSDate *start = [NSDate date];

        [solver updateCoarseValues];
        [solver nextIteration];
        [solver sendComputeCommand];
        nextAnswer = [solver getResult];

        sumTime += [start timeIntervalSinceNow];

        prevDifference = difference;
        difference = getMaxDifference(answer, nextAnswer, numX);
    }

    *time += (double)fabs(sumTime);

    return answer;
}
```