МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

02.03.03 МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ И
АДМИНИСТРИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

БАКАЛАВРСКАЯ РАБОТА

на тему Разработка вычислительного ядра обработки данных с использованием технологии CUDA

Студент Д.А. Горбатюк

Руководитель А.А. Пупыкина

Допустить к защите
Заведующий кафедрой, к.т.н, доцент, А.В. Очеповский _____

«_____»_____20_____г.

Тольятти 2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

УТВЕРЖДАЮ
Завкафедрой «Прикладная
математика и информатика»
_____ А.В. Очеповский

«____»_____2016 г.

ЗАДАНИЕ
на выполнение бакалаврской работы

Студент Горбатюк Дмитрий Александрович
1. Тема Разработка вычислительного ядра обработки данных с использованием технологии CUDA
2. Срок сдачи студентом законченной выпускной квалификационной работы 24.06.2016
3. Исходные данные к выпускной квалификационной работе поддержка технологии CUDA
4. Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов, разделов) проанализировать научную и учебную литературу по тематике; рассмотреть основные принципы разработки программы с использованием технологии CUDA; создать структуру программы и назначение основных функций; обосновать выбор средств реализации программы; реализовать алгоритмы для вычислительного ядра и описать основные аспекты программной модели; протестировать реализованный продукт и оценить его эффективность
5. Ориентировочный перечень графического и иллюстративного материала схема структуры программы, презентация
6. Дата выдачи задания « 11 » января 2016 г.

Руководитель выпускной
квалификационной работы _____ А.А. Пупыкина

Задание принял к исполнению _____ Д.А. Горбатюк

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюдж
етное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

УТВЕРЖДАЮ
Завкафедрой «Прикладная
математика и информатика»
_____ А.В. Очеповский

«____»_____2016 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения бакалаврской работы

Студента Горбатюк Дмитрия Александровича
по теме Разработка вычислительного ядра обработки данных с использованием технологии CUDA

Наименование раздела работы	Плановый срок выполнения раздела	Фактический срок выполнения раздела	Отметка о выполнении	Подпись руководителя
Выбор и утверждение темы ВКР	11.01.2016	11.01.2016	выполнено	
Поиск и исследование литературы по теме ВКР	15.01.2016	15.01.2016	выполнено	
Подбор, изучение и проработка практических материалов	19.01.2016	19.01.2016	выполнено	
Разработка и согласование с руководителем первой главы ВКР	23.01.2016	23.01.2016	выполнено	
Разработка и согласование с руководителем второй главы ВКР	25.01.2016	25.01.2016	выполнено	
Разработка и согласование с руководителем третьей главы ВКР	28.01.2016	28.01.2016	выполнено	

Согласование с руководителем	05.02.2016	05.02.2016	выполнено	
Тестирование реализованного приложения	05.04.2016	05.04.2016	выполнено	
Оформление пояснительной записки ВКР	10.04.2016	10.04.2016	выполнено	
Подготовка доклада и графического материала к защите	10.05.2016	10.05.2016	выполнено	
Предварительная защита ВКР	20.05.2016	20.05.2016	выполнено	
Корректировка ВКР согласно сделанным замечаниям	21.05.2016	21.05.2016	выполнено	
Проверка ВКР в системе «Антиплагиат.ВУЗ»	16.06.2016	16.06.2016	выполнено	
Сдача пояснительной записки ВКР и реализованного продукта	24.06.2016	24.06.2016	выполнено	

Руководитель выпускной квалификационной работы _____ А.А. Пупыкина

Задание принял к исполнению _____ Д.А. Горбатюк

Аннотация

В данной выпускной квалификационной работе был выделен объект исследования, проанализирована предметная область, определены цели и задачи работы. Проведен обзор программных средств, позволяющих ускорить процесс обработки экспериментальных данных.

Анализ предметной области показал, что для достижения цели данной выпускной квалификационной работы требуется разработать приложение. В качестве инструмента для реализации был выбран высокоуровневый язык программирования Python и технология CUDA от NVidia.

Таким образом в рамках данной выпускной квалификационной работы был разработано приложение являющееся весьма актуальным на сегодняшний день, для широкого круга исследований в области науки.

В первой главе выпускной квалификационной работы описывается постановка задачи обработки экспериментальных данных.

Вторая глава описывает процесс проектирования вычислительного ядра.

Третья глава описывает процесс реализации программного продукта.

Выпускная квалификационная работа содержит пояснительную записку объемом 50 страниц, включая 27 иллюстраций, 4 таблицы, список использованной литературы из 20 наименований и одного приложения.

Оглавление

Введение ... 3
Глава 1 Постановка задачи обработки экспериментальных ... 6
1.1 Задача обработки экспериментальных данных ... 6
1.2 Дискретное преобразование Фурье ... 7
1.3 Архитектура графического процессора ... 12
1.4 Сравнение Central Processing Unit и Graphics Processing Unit ... 15
1.5 Программная модель Compute Unified Architecture и основные принципы ... 17
1.6 Атрибуты функций и переменных ... 22
1.7 Сравнение организации систем памяти ... 24
Глава 2 Проектирование вычислительного ядра ... 27
2.1 Анализ алгоритма Фурье ... 27
2.2 Структура программы и назначение основных компонентов ... 28
Глава 3 Реализация вычислительного ядра ... 32
3.1 Выбор средств реализации ... 32
3.2 Реализация вычислительного ядра ... 34
3.3 Описание установки и тестирования программного продукта. ... 38
3.4 Работа с Anaconda ... 40
3.5 Тестирование и анализ эффективности вычислительного ядра ... 43
Заключение ... 48
Список использованной литературы ... 49
Приложение А Исходный код ... 51

Введение

За последние десятилетия во всех областях науки значительно выросла потребность в обработке огромного количества данных. Для этого необходимы мощные вычислительные системы. Из-за того, что стало невозможно увеличивать тактовую частоту процессора, появились многоядерные архитектуры, которые потребовали навыков параллельного программирования. В связи с нехваткой высококвалифицированных специалистов стало необходимым создание эффективных высокоуровневых технологий программирования. В результате стало происходить бурное развитие суперкомпьютерных технологий по всему миру.

Графические ускорители стали использоваться для неграфических вычислений, что позволило достичь ускорения работы приложений в десятки, а то и в сотни раз. Для написания программ, которые предназначены для выполнения на графическом процессоре, были разработаны специальные языки программирования: PyCUDA, C–CUDA, Fortran–CUDA, OpenCL , OpenACC и другие.

Научный группой отдела физики плазмы Института физики РАН им. А.М. Прохорова была обоснована необходимостью разработки вычислительного ядра с использованием ускорения вычислений на базе GPU NVidia.

Под вычислениями понимается потоковая обработка экспериментальных данных по заранее известным алгоритмам. Данные представляют собой временные ряды, получаемые с различных диагностик, размещаемых на установке стелларатор Л-2М.

В настоящее время проблема обработки данных для расчета одной из основных функций, быстрого преобразования Фурье, и внедрение в целях предобработки сигналов очень актуальна. Так как количество данных на установке постоянно увеличивается, то необходимо использовать современные программно-аппаратные решения по ускорению вычислений.

Поэтому актуальность выпускной квалификационной работы заключается в том, что ускорение обработки экспериментальных данных по заранее известным алгоритмам значительно сокращает время на выполнение основных операций.

Для того, чтобы ускорить обработку данных, необходимо решить следующие задачи:

– провести анализ алгоритма быстрого преобразования Фурье;

– провести анализ входной и результатной информации;

– реализовать алгоритм быстрого преобразования Фурье;

– ускорить имеющийся алгоритм быстрого преобразования Фурье посредством технологии CUDA NVidia.

Объектом исследования выпускной квалификационной работы является процесс обработки экспериментальных данных.

Предметом исследования выпускной квалификационной работы является ускорение обработки экспериментальных данных посредством технологии CUDA от NVidia.

Целью данной выпускной квалификационной работы является разработка вычислительного ядра обработки данных с использованием технологии CUDA.

Для достижения необходимой цели нужно выполнить такие задачи как:

– проанализировать литературу по ускорению вычислений для обработки экспериментальных данных;

– проанализировать литературу по имеющимся алгоритмам быстрого преобразования Фурье;

– проанализировать литературу и документацию по технологии CUDA от NVidia;

– проанализировать литературу и документацию о высокоуровневом языке программирования Python;

– реализовать алгоритм быстрого преобразования Фурье, посредством высокоуровневого языка программирования Python;

– ускорить имеющуюся реализацию алгоритма быстрого преобразования Фурье, посредством внедрения технологии CUDA от NVidia.

В процессе выполнения выпускной квалификационной работы будет создано приложение, которое позволит:

– обрабатывать экспериментальные данные из файла с расширением .csv;

– производить расчет преобразования Фурье;

– выводить график.

В введении описывается актуальность проводимого исследования, формируется цель и ставятся задачи, которые необходимо решить для достижения цели.

В первой главе выпускной квалификационной работы описывается постановка обработки экспериментальных данных.

Вторая глава описывает процесс проектирования вычислительного ядра.

Третья глава описывает процесс реализации программного продукта.

В заключении приводятся основные выводы по выпускной квалификационной работе, достигнутые в ходе её выполнения.

Глава 1 Постановка задачи обработки экспериментальных

1.1 Задача обработки экспериментальных данных

Экспериментальные данные представляют собой временные ряды, получаемые с различных диагностик, размещаемых на установке стелларатор Л-2М. Исследования на данной установке носят в основном фундаментальный характер и нацелены на получение новых физических данных о поведении высокотемпературной плазмы при различных условиях удержания и нагрева. Физические диагностики имеют различное назначение и результатом их работы являются в общем виде абсолютно разные временные ряды, в том числе с различными параметрами оцифровки.

Из-за постоянного возрастания количества считываемых данных, необходимо использовать современные программно-аппаратные решения по ускорению вычислений. На данный момент предполагается использование только процедур обработки данных (data processing), но в дальнейшем планируется подключить в общую систему вычислений различное моделирование (plasma physics modeling). Характерное время между импульсами установки стелларатор Л-2М составляет 3-5 минут. После каждого импульса накапливается порядка 30 временных рядов (с высокой скоростью оцифровки), каждый из которых представляет интерес и требует пре- и постобработки. Есть необходимость производить робастный анализ слабоструктурированных данных в режиме обработки, близкой к реальному времени. На данный момент производится обработка одиночных импульсов, без учета устойчивости (нет группировки данных в момент работы) и не введены в эксплуатацию новые высокоскоростные АЦП.

Основные функции, которые проводятся над данными это:

- преобразование Фурье;
- Вейвлет преобразования;
- корреляционный и регрессионный анализ;
- вычисление различных энергетических характеристик;
- статический анализ, в том числе расчет стохастических моделей.

Учитывая все это предлагается начать разработку вычислительного ядра системы анализа слабоструктурированных данных на установке стелларатор Л-2М. Такое ядро должно быть совместимо с внешними управляющими программами (Python Windows). В данной выпускной квалификационной работе будет рассмотрен алгоритм преобразования Фурье и его дальнейшее ускорение посредством технологии CUDA NVidia.

1.2 Дискретное преобразование Фурье

Дискретное преобразование Фурье (ДПФ) трансформирует последовательность комплексных чисел x_n в последовательность комплексных чисел X_n. Формулы (1.1) и (1.2) определяют прямое и обратное преобразования Фурье:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \qquad k = 0, \dots, N-1, \tag{1.1}$$

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn} \qquad n = 0, \dots, N-1. \tag{1.2}$$

(1.2)

Формулы (1.1) и (1.2) имеют сложность $O(N^2)$ [6-8].

Быстрым преобразованием Фурье (БПФ) называют набор алгоритмов, предназначенных для быстрого вычисления ДПФ [6]. Основной принцип этих алгоритмов заключается в разложении операций вычисления ДПФ сигнала x(n) длиной N на вычисление ДПФ с меньшим числом точек. Разделив анализируемый набор отсчетов на части, вычисляют их ДПФ и объединяют результаты. Существует несколько реализаций БПФ, отличающихся организацией разбиения одного массива чисел. Разделение и объединение исходного сигнала может проводиться прореживанием по времени или по частоте. Кроме того, БПФ может проводиться по разному основанию, которое определяет на сколько фрагментов проводится разбиение исходной последовательности на каждом шаге.

Основная идея алгоритма прореживания по времени по базису 2 состоит в том, чтобы разбить исходный сигнал x(n) $N = 2^L$, $L \in R$ на два

более коротких сигнала, ДПФ которых могут быть объединены таким образом, чтобы получить ДПФ исходного сигнала [7].

Обозначим $W\frac{nk}{N} = e^{\frac{-jn2\pi k}{N}}$. Разобьем последовательность отсчетов сигнала на четные и нечетные:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot W\frac{nk}{N} = \sum_{n=0}^{\frac{N}{2-1}} x_{2n} \cdot W\frac{2nk}{N} + \sum_{n=0}^{\frac{N}{2-1}} x_{2n} + 1 \cdot W\frac{(2n+1)k}{N} = \\ = \sum_{n=0}^{\frac{N}{2-1}} x_{2n} \cdot W\frac{nk}{\frac{N}{2}} + W_N^k \sum_{n=0}^{\frac{N}{2-1}} x_{2n+1} \cdot W_{\frac{N}{2}}^{nk} = X_0(\omega_k) + W_N^k X_1(\omega), \quad (1.3)$$

разбиение спектра на 2 части:

$$X = \{X_k \vee k = \overline{0, N-1}\} = \{X_m \vee m = \overline{0, N/2-1}\} \cup \{X_r \vee r = \overline{N/2, N-1}\} = \\ \{X_m \vee m = \overline{0, N/2-1}\} \cup \{X_{r+N/2} \vee r = \overline{0, N/2-1}\}, \quad (1.4)$$

вычислим вторую половину спектра:

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n} \cdot W_{N/2}^{n(k+N/2)} + W_N^{k+N/2} \sum_{n=0}^{N/2} x_{2n+1} \cdot W_{N/2}^{n(k+N/2)}, \quad (1.5)$$

особенности коэффициента W_N^{nk}

$$W_{N/2}^{n(k+N/2)} = W_{N/2}^{nN/2} \cdot W_{N/2}^{nk}, \quad (1.6)$$

$$W_{N/2}^{nN/2} = e^{\frac{-j2\pi nN/2}{N/2}} = e^{-j2\pi n} = cos(-2\pi n) - jsin(-2\pi n) = 1, \quad (1.7)$$

получим:

$$W_{N/2}^{k+N/2} = W_{N/2}^{nk}, \quad (1.8)$$

алгоритм прореживания по времени имеет сложность $O(Nlog_2 N)$ [6-9].

Теория БПФ исходит из предположения о периодическом сигнале и для идеального преобразования необходимо выделить точно один или несколько периодов сигнала, что для реального сигнала сделать точно никогда не удается. Это приводит к разрывности исходной функции к искажению (расширению) расчетного частотного спектра.

Оконная функция БПФ используется для подавления краевых эффектов разрывности реальных функций путем введения весовых коэффициентов для выборки данных в окне, обеспечивающих снижение амплитуд краевых точек и улучшение результатов БПФ.

Недостаток анализа Фурье, обусловленный неспособностью осуществлять временную локализацию сингулярностей сигналов, может быть частично устранен путем введения в преобразование так называемого окна $w(t)$ – движущейся функции, имеющей компактный носитель [7]. Использование оконной функции позволяет представить результат анализа в виде функции двух переменных, частоты ω и времени τ положения окна:

$$F(\omega, t) = \int_{-\infty}^{\infty} w(t-\tau) f(t) e^{-j\omega t} dt, \qquad (1.9)$$

где $F(\omega, t)$ функция, которая даёт распределение частот части $f(t)$ в окрестности времени τ;

w некая оконная функция;

$f(t) -$ оригинальный сигнал;

$\tau -$ окрестность времени.

Примерами оконных преобразований могут служить функции:

– прямоугольная;
– Хэмминга;
– Кайзера;
– Блэкмана.

Одной из самых простых и наиболее важных пар, сопряженных по Фурье, является прямоугольная функция, импульс которой записывается по формуле (1.10), спектр по формуле (1.11), а график изображен на рисунке 1.1.

$$t = \begin{cases} 1, |t| < \frac{1}{2} \\ 0, |t| > \frac{1}{2}, \end{cases} \qquad (1.10)$$

$$\sin cf = \frac{\sin \pi f}{\pi f}, \tag{1.11}$$

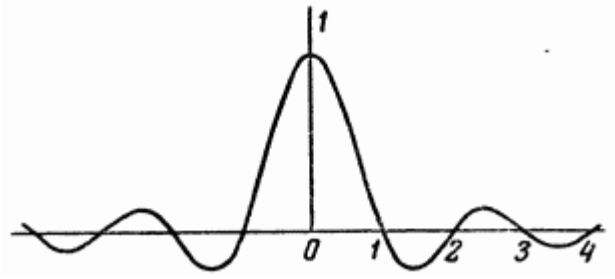

Рисунок 1.1 – График прямоугольной функции

Площадь, ограниченная кривой, равна единице (формула 1.12), так же, как и площадь, ограниченная $\sin c$ (формула 1.13). Прямоугольная функция входит в состав семейства сдвинутых друг относительно друга взаимно ортогональных функций.

$$\int_{-\infty}^{\infty} \sin cx \, dx = 1, \tag{1.12}$$

$$\int_{-\infty}^{\infty} \sin c(x-m) sin\, c(x-n) dx = \begin{cases} 1, m = n \\ 0, m \neq n \end{cases}, \tag{1.13}$$

где m и n – целые числа.

Функция Хэмминга определяется по формуле 1.14:

$$w(n) = 0{,}54 + 0{,}46 \cos\left(\frac{2\pi n}{N}\right) \begin{cases} -\frac{N-2}{2} \leq n \leq \frac{N-1}{2} (n - \text{нечетное}) \\ -\frac{N}{2} \leq n \leq \frac{N}{2} (N - \text{четное}) \end{cases}. \tag{1.14}$$

Во временной области функция Хэмминга более мягко выходит на нуль с обеих сторон, чем прямоугольная функция. В частной области амплитуда её главного лепестка шире, чем амплитуда прямоугольной функции, но её боковые лепестки меньше по сравнению с главным. Вследствие этого функция Хэмминга даст фильтр с большей плоскостью перехода, но и большим затуханием в полосе подавления [7-8]. Связь ширины полосы перехода фильтра, построенного на основе функции Хэмминга, с длинной фильтра, выражается формулой:

$$\Delta f = \frac{3{,}3}{n}, \tag{1.15}$$

где Δf – нормированная ширина полосы перехода;

n – длина фильтра.

Максимальное затухание в полосе подавления составляет 53 дБ, а минимальная амплитуда неравномерности в полосе пропускания составляет около 0,0194 дБ, это утверждение подходит для любых данных [6-8]. Для наглядного отображения на рисунке 1.2 слева, синим цветом изображен график сигнала, а справа, оранжевым цветом изображен спектр сигнала.

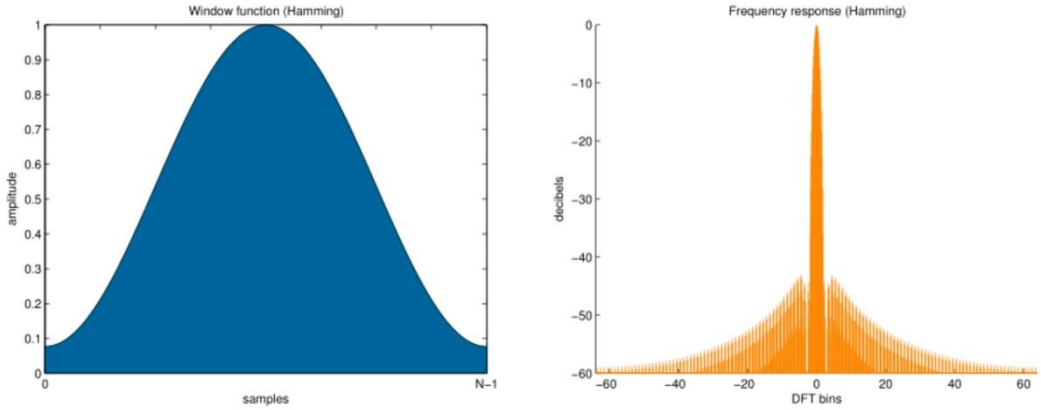

Рисунок 1.2 – Функция Хэмминга

Функция Кайзера имеет параметр β управляющий неравномерностью, что позволяет разработчику идти на компромиссы между шириной перехода и неравномерностью [6-9]. Функция задается уравнением по формуле:

$$w(n) = \frac{I_0\left(\beta\left[1-\left(\frac{2n}{N-1}\right)^2\right]^{\frac{1}{2}}\right)}{I_0(\beta)} - \frac{N-1}{2} \le n \le \frac{N-1}{2}, \qquad (1.16)$$

где $I_0(x)$ это модифицированная функция Бесселя первого рода нулевого порядка,

β - управляющий параметр, отвечающий за спад вырезающей функции на краях.

Для вычисления $I_0(x)$ используется разложение:

$$I_0(x) = 1 + \sum_{k=1}^{L}\left[\frac{\left(\frac{x}{2}\right)^k}{k!}\right]^2, \qquad (1.17)$$

где L < 25.

При β = 0 функция Кайзера соответствует прямоугольной весовой функции, а при β = 5.44 функция похожа на функцию Хэмминга. Для

наглядного отображения на рисунке 1.3 слева изображен график сигнала, а справа изображен спектр сигнала.

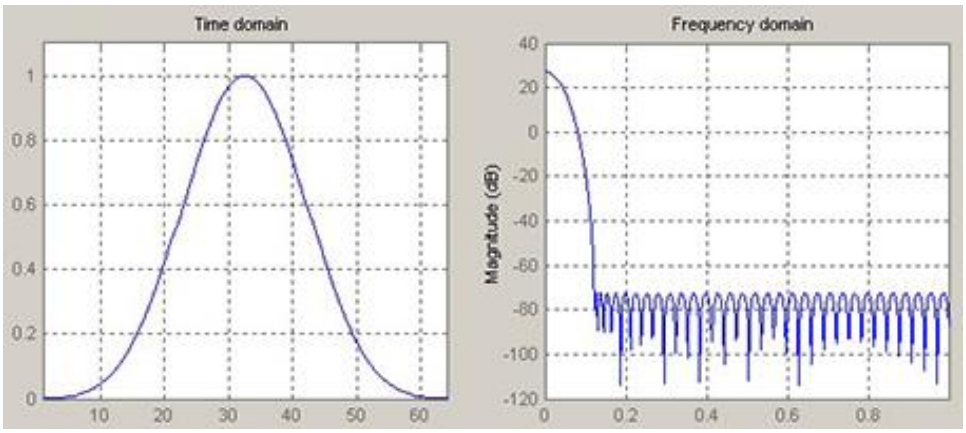

Рисунок 1.3 – Функция Кайзера

Функция Блэкмана формирует последовательность Y из входной последовательности X по соотношению:

$$w(n) = 0{,}42 - 0{,}5\cos\left(\frac{2\pi n}{L-1}\right) + 0{,}08\cos\left(\frac{4\pi n}{L-1}\right), \quad (1.18)$$

где n – количество элементов в последовательности X.

Для наглядного отображения на рисунке 1.4 слева синим цветом изображен график сигнала, а справа оранжевым цветом изображен спектр сигнала [6-9].

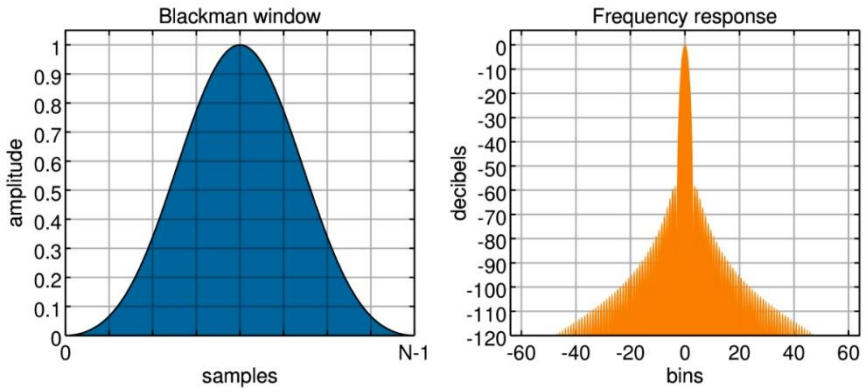

Рисунок 1.4 – Функция Блэкмана

1.3 Архитектура графического процессора

Одной из важнейших характеристик любого вычислительного устройства является производительность. Для математических расчетов она обычно измеряется в количестве операций над вещественными данными в

секунду. В зависимости от того учитывается ли только скорость расчета или также влияние других факторов, таких как скорость обмена данными, различают максимальную теоретически возможную пиковую и реальную производительность [1].

Производительность сильно зависит от тактовых частот центрального процессора и памяти. Процессоры с высокой частотой и большим количеством интегрированной памяти могли бы обладать превосходной производительностью, но не могут быть массовыми из-за слишком высокой цены. По этой причине основной объем памяти находится в отдельных от процессора внешних модулях, и частота их работы в несколько раз ниже частоты процессора. Таким образом, реальная производительность системы при обработке больших массивов данных в значительной мере характеризуется именно скоростью работы внешней памяти и поэтому может быть значительно ниже тактовой [1-4].

Процессоры архитектуры x86 с момента своего появления 1978 году увеличили свою тактовую частоту с 4,77 МГц до 3 ГГц, т.е. более чем в 600 раз, однако в последние несколько лет рост частоты более не наблюдается [1]. Это связано с ограничениями технологии производства микросхем, так и с тем, что энергопотребление пропорционально четвертой степени частоты. Таким образом, увеличение тактовой частоты всего в 2 раза приводит к увеличению тепловыделения в 16 раз. До сих пор с этим удавалось справляться за счет уменьшения размеров отдельных элементов микросхем [1-4]. Дальнейшая миниатюризация связана со значительными трудностями, поэтому в настоящее время рост производительности идет в основном за счет увеличения числа параллельно работающих ядер, т.е. за счет параллелизма.

Максимальное ускорение, которое можно получить, распределив выполнение программы на N параллельно работающих элементов, определяется законом Амдала:

$$S = \frac{1}{(1-P)+\frac{P}{N}}, \qquad (1.19)$$

где P это доля работы, которая может быть распределена.

Видно, что при увеличении числа процессоров N ускорение стремится к $\frac{1}{1-P}$. Таким образом, если возможно распараллелить $\frac{3}{4}$ всех вычислений в программе, то при большом числе доступных процессоров ускорение никогда не превысит четырех раз. Закон Амдала показывает, что возможная выгода от использования параллельных вычислений во многом предопределена свойствами применяемых в программе методов или алгоритмов.

Термин Graphics Processing Unit был впервые использован корпорацией NVIDIA для обозначения того факта, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным программируемым устройством, пригодным для решения значительно более широкого класса вычислительных задач [1].

Современные GPU представляют собой массивно-параллельные вычислительные устройства с производительностью порядка Терафлопса и большим объемом собственной памяти.

Термином SIMD (Single Instruction Multiple Data) называют метод обработки, при котором одна и та же операция применяется одновременно ко множеству независимых данных. SIMD процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая выходной поток. Для наглядности архитектура SIMD изображена на рисунке 1.5 [1].

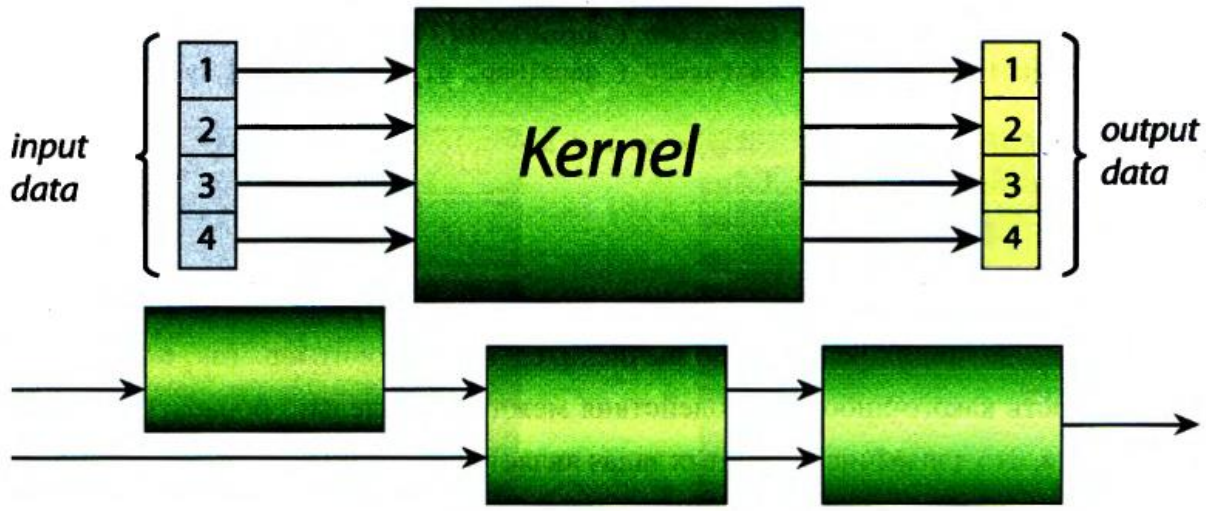

Рисунок 1.5 – SIMD архитектура

Программный модуль, описывающий подобное преобразование, называют вычислительным ядром (kernel). Отдельные ядра могут быть соединены между собой, образуя сложные составные схемы [1-5].

С введением поддержки текстур 16 и 32-х битных вещественных элементов появился простой и универсальный метод обмена большими массивам данных между GPU и памятью основной системы. Высокоуровневые языки, такие как Cg, GLSL и HLSL, значительно упростили процесс написания программ для GPU. К примеру программы шейдера, написанные на языке GLSL. Функциями OpenGLили Direct3D входные данные загружались в текстуры. Затем на графическом процессоре через операцию рендеринга запускалась программа обработки данных. Результат получался также в виде текстур, которые оставалось выгрузить обратно в системную память. Поэтому программа состояла из двух частей, написанных на разных языках. C/C++ применялся для подготовки и передачи данных и языка шейдеров, для вычислений на GPU [1].

1.4 Сравнение Central Processing Unit и Graphics Processing Unit

В основном отличия архитектур GPU от CPU состоят в том, что:

– CPU имеет лишь небольшое число ядер, работающих на высокой тактовой частоте независимо друг от друга. GPU работает на низкой

тактовой частоте и имеет сотни сильно упрощенных вычислительных элементов;

– значительная площадь CPU занята кэшем, в то время как почти весь GPU состоит из арифметико-логических блоков.

Сравнение внутренней архитектуры CPU и GPU изображено на рисунке 1.6 [14].

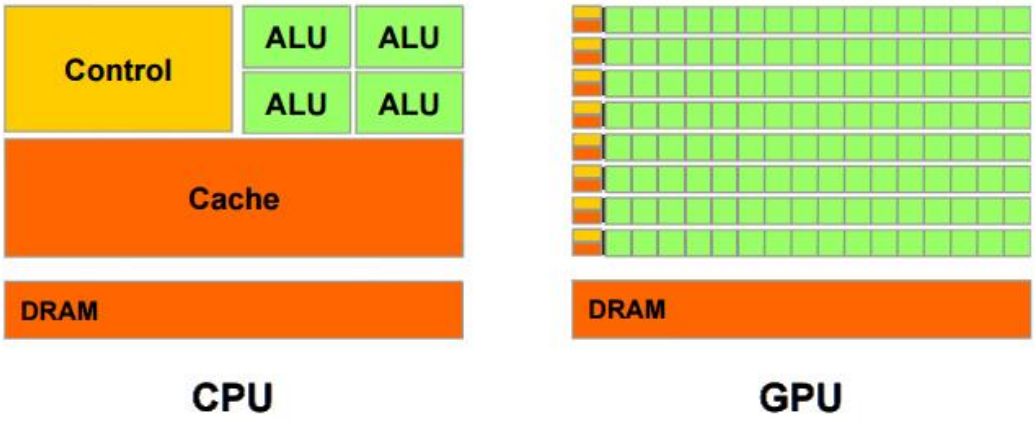

Рисунок 1.6 – Внутреннее устройство CPU и GPU

В архитектуре GPU кэш имеет меньшее значение, поскольку используется иная, от CPU, стратегия покрытия латентности памяти. За счет этих отличий производительность каждого нового поколения GPU быстро растет как в пиковом значении изображенном на рисунке 1.7 [1], так и на реальных приложениях, к примеру Linpack, рисунок 1.8 [1-4].

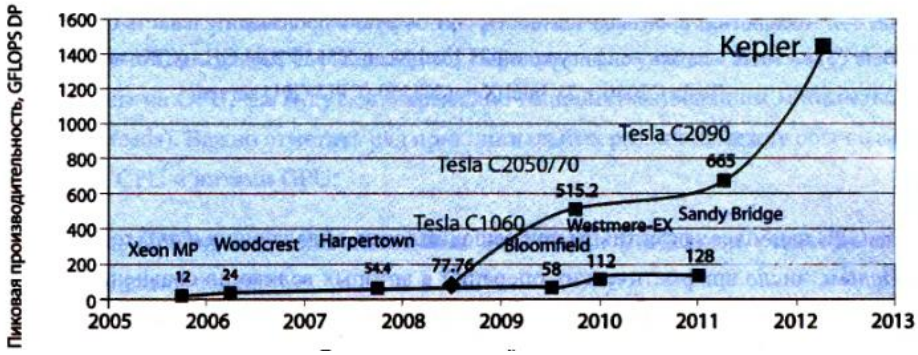

Рисунок 1.7 – Динамика роста пиковой производительности вычислений с двойной точностью CPU и GPU

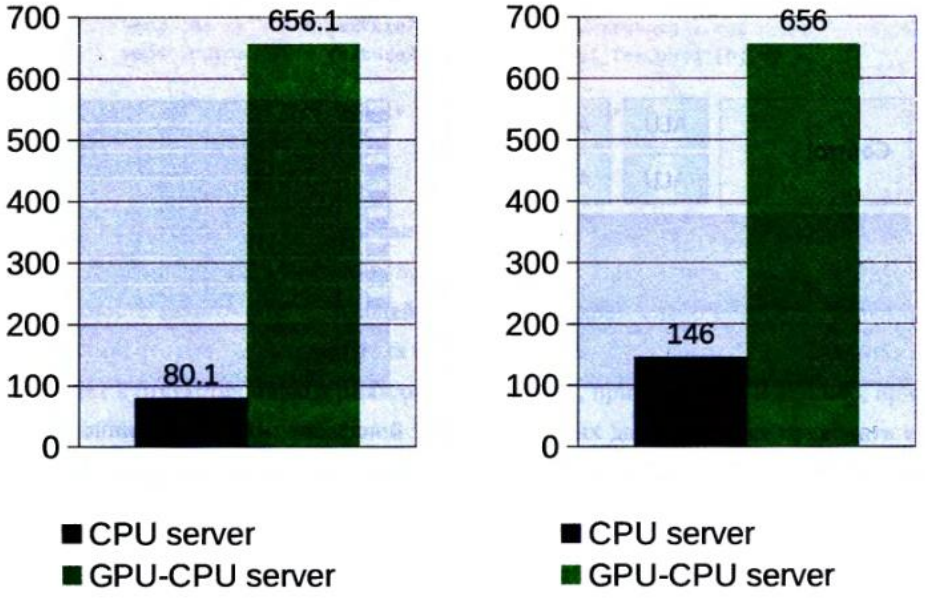

Рисунок 1.8 – Абсолютная производительность и производительность на Ватт, тест Linpack

GPU наиболее эффективны при решении задач, обладающих параллелизмом по данным, число арифметических операций в которых велико по сравнению с операциями над памятью. К примеру, в 3D-рендеринге параллелизм по данным выражается в распределении по потокам обработки отдельных вершин. Аналогично, обработка изображений, кодирование и декодирование видео, и распознавание образов легко делится на подзадачи над блоками изображений и пикселов. Более того, GPU может применяться для неграфических вычислений, к примеру обработка сигналов, физика, финансовый анализ, вычислительная биология.

1.5 Программная модель Compute Unified Architecture и основные принципы

Compute Unified Architecture (CUDA) это программная модель, включающая описание вычислительного параллелизма и иерархической структуры памяти в самом языке программирования. С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU. CUDA предназначена для разработки

GPGPU-приложений без привязки к графическим API и поддерживается всеми GPU NVIDIA, начиная с серии GeForce 8 [5].

Концепция CUDA отводит GPU роль массивно-параллельного сопроцессора. В литературе о CUDA основная система, к которой подключен GPU, для краткости называется термином хост (host), аналогично сам GPU по отношению к хосту часто называется просто устройством (device). CUDA-программа задействует как CPU, так и GPU, на CPU выполняется последовательная часть кода и подготовительные стадии для GPU-вычислений. Параллельные участки кода могут быть перенесены на GPU, где будут одновременно выполняться большим множеством нитей (threads) [1-5]. Главное ряд принципиальных различий между обычными потоками CPU и нитями GPU заключается в том, что:

– нить GPU чрезвычайно легковесна, ее контекст минимален, регистры распределены заранее;

– для эффективного использования ресурсов GPU программе необходимо задействовать тысячи отдельных нитей, в то время как на многоядерном CPU максимальная эффективность обычно достигается при числе потоков, равном или в несколько раз большем количестве ядер.

В целом работа нитей на GPU соответствует принципу SIMD, но есть существенное различие. Только нити в пределах одной группы, называемой варпом (warp) выполняются физически одновременно. Нити различных варпов могут находиться на разных стадиях выполнения программы. Такой метод обработки данных обозначается термином SIMT (Single Instruction – Multiple Threads). Управление работой варпов производится на аппаратном уровне. По ряду возможностей новых версий CUDA прослеживается тенденция к постепенному превращению GPU в самодостаточное устройство, полностью заменяющее обычный CPU за счет реализации некоторых системных вызовов, и добавления облегченного энергоэффективного CPU-ядра в сам GPU [1-5].

Важным преимуществом CUDA является использование для программирования GPU языков высокого уровня. В настоящее время существуют компиляторы C++, Fortran и Python. Эти языки расширяются небольшим множеством новых конструкций, такими как атрибуты функций и переменных, встроенные переменные и типы данных, оператор запуска ядра [5].

Нить GPU имеет координаты во вложенных трехмерных, декартовых, равномерных сетках, а также индексы блоков и индексы потоков внутри каждого блока. В контексте каждой нити значения координат и размерностей доступны через такие встроенные переменные, как:

– threadIdx;
– BlockIdx;
– blockDim;
– gridDim.

В общем случае соответствие нитей и частей задачи может быть любым, одна нить может обрабатывать не один элемент массива, а определенный диапазон. В течении времени работы ядра декартовы сетки нитей и блоков зафиксированы для отображения на аппаратный уровень вычислительных элементов GPU [1]. Иерархия нитей изображена на рисунке 1.9 [1].

При использовании глобальной памяти, расположенной физически на самом GPU, управлять ею можно с хоста. Общим приемом программирования для CUDA является группировка множества нитей в блоки. На это есть две причины:

– не для каждого параллельного алгоритма существует эффективная реализация на полностью независимых нитях. Результат одной нити может зависеть от результата некоторых других, исходные данные нити могут частично совпадать с данными соседних;

– размер одной выборки данных из глобальной памяти намного больше размера вещественного или целочисленного типа, одна выборка может покрыть запрос группы из нескольких нитей, работающих с подряд идущими в памяти элементами.

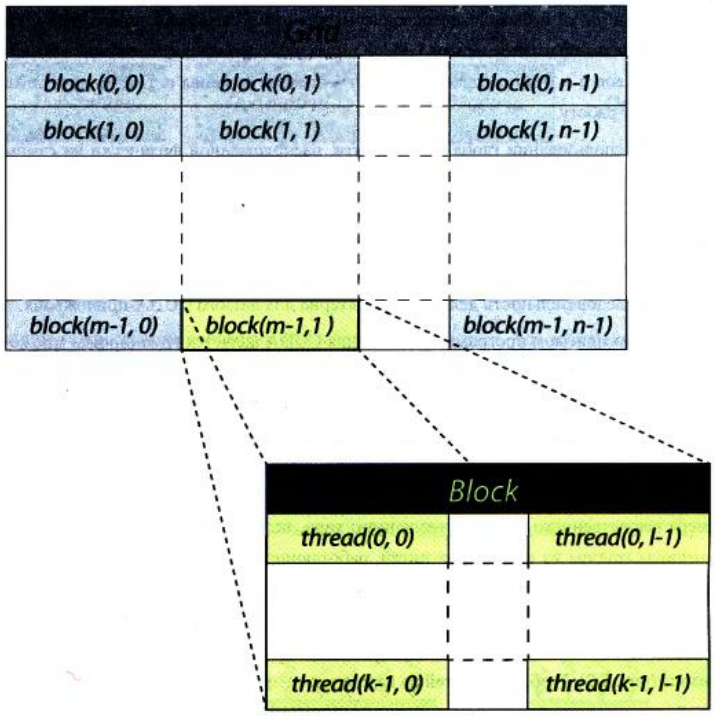

Рисунок 1.9 – Иерархия нитей в CUDA

В результате группировки нитей исходная задача распадается на независимые друг от друга подзадачи с возможностью взаимодействия нитей в рамках одного блока и объединения запросов в память в рамках одного варпа. Для наглядности разбиение исходной задачи на наборе независимо решаемых подзадач изображено на рисунке 1.10 [1].

Разбиение нитей на варпы также происходит отдельно для каждого блока. Объединение в блоки является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и возможностью сделать соответствующую аппаратную логику эффективной и дешевой [1-5].

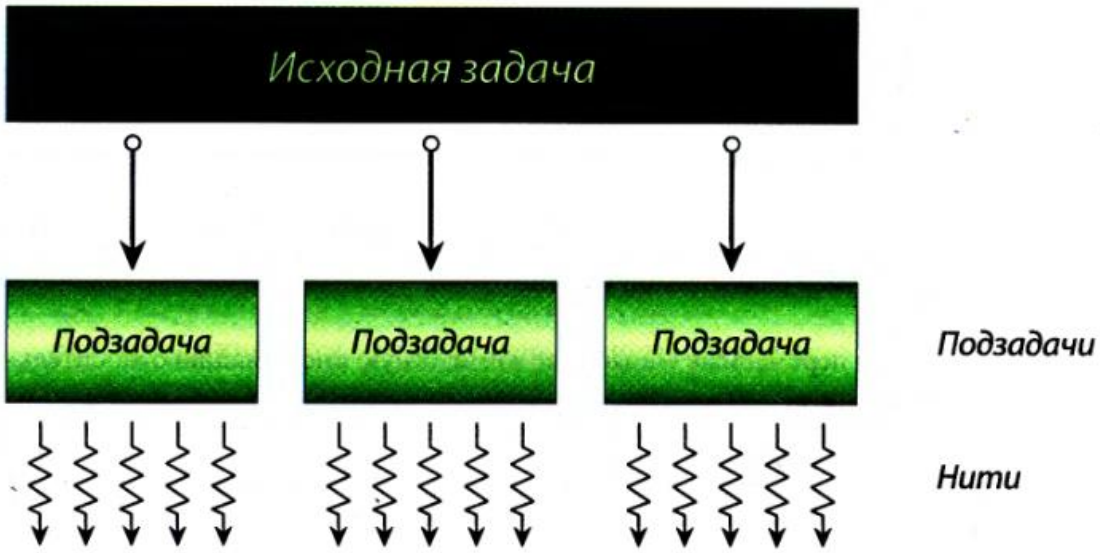

Рисунок 1.10 – Разбиение исходной задачи на набор независимо решаемых подзадач

На время выполнения ядра каждый блок получает в распоряжение часть быстрой разделяемой памяти, которую могут совместно использовать все нити блока. Поскольку не все нити блока выполняются физически одновременно, то для их взаимного согласования необходим механизм синхронизации. Для этой цели в CUDA предусмотрен вызов __syncthreads(), который блокирует дальнейшее исполнение кода ядра до тех пор, пока все нити блока не войдут в эту функцию. Для наглядности барьерная синхронизация изображена на рисунке 1.11 [1].

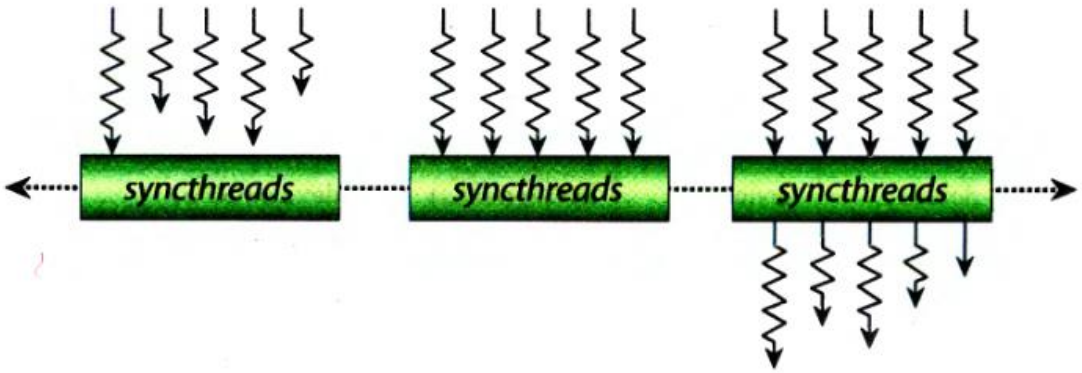

Рисунок 1.11 – Барьерная синхронизация

1.6 Атрибуты функций и переменных

Использующиеся на данный момент в CUDA атрибуты функций представлены в таблице 1.1 [1].

Таблица 1.1 Атрибуты функций и переменных в CUDA

Атрибут C	Атрибут Fortran	Функция выполняется на	Функция может вызываться из
__device__	device	device (GPU)	device (GPU)
__global__	global	device (GPU)	host (CPU)
__host__	host	host (CPU)	host (CPU)

Атрибуты __host__ и __device__ могут быть использованы вместе, то есть соответствующая функция может выполняться как на GPU, так и на CPU. Атрибуты __global__ и __host__ не могут быть использованы вместе. Атрибут __global__ обозначает ядро, и соответствующая функция CUDA должна возвращать значение типа void [1-2]. На функции, выполняемые GPU накладываются такие ограничения, как:

– не поддерживаются static переменные внутри функции;

– не поддерживается переменное число входных аргументов.

Для задания способа размещения переменных в памяти GPU используются следующие атрибуты: __device__, __constant__ и __shared__. На их использование накладываются ограничения:

– атрибуты не могут быть применены к полям структуры;

– соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять, как extern;

– запись в переменные типа __constant__ может осуществляться только CPU при помощи специальных функций;

– __shared__ переменные не могут инициализироваться при объявлении.

Пока существующие компиляторы для CUDA не реализуют поддержку модульной сборки GPU кода, каждая __global__ функция должна находиться

в одном исходном файле вместе со всеми __device__ функциями и переменными, которая она использует [5].

В язык CUDA добавлены специальные переменные:

– gridDim это размер сетки;

– blockDim это размер блока;

– blockIdx это индекс текущего блока в сетке;

– threadIdx индекс текущей нити в блоке;

– warpSize это размер варпа.

Многие функции CUDA являются асинхронными, к числу асинхронных операций относятся:

– запуск ядра;

– функции копирования и инициализации памяти, имена которых оканчиваются на Async;

– функции копирования памяти device внутри устройства и между устройствами.

С асинхронностью связаны объекты CUDA streams (потоки исполнения), позволяющие группировать последовательности операций, которые необходимо выполнять в строго определенном порядке. При этом порядок выполнения операций между разными CUDA streams не является строго определенным и может изменяться. По умолчанию все асинхронные операции выполняются в нулевом CUDA streams, это означает, что потоки последовательны между собой и асинхронны по отношению к остальным вызовам программы. Для явной синхронизации всех CUDA streams текущего устройства с текущим потоком CPU используется функция cudaDeviceSynchronize. Во многих случаях синхронизация происходит неявно, к примеру, при копировании результатов с GPU по адресу, ранее использованному при запуске ядра, основной поток будет заблокирован на копировании до завершения работы ядра. Неявная синхронизация происходит и при использовании CUDA Events [1].

1.7 Сравнение организации систем памяти

GPU и CPU существенно отличаются организацией систем памяти и методами работы с ней. В CPU большую долю площади схемы обычно занимают кэши различных уровней. Основная же часть GPU занята вычислительными блоками. CPU обычно не предоставляет прямой доступ к управлению кэшами, ограничиваясь в основном инструкцией prefetch. В GPU также присутствует не контролируемый явно кэш, но существует еще несколько видов памяти, которые всегда должны управляться программно [1-4]. Типы памяти в Compute Unified Architecture представлены в таблице 1.2 [1].

Таблица 1.2 Типы памяти в CUDA

Тип памяти	Расположение	Кэш	Доступ	Видимость	Время жизни
Регистры	Мультипроц.	Нет	R/W	Нить	Нить
Локальная	DRAM GPU	Нет	R/W	Нить	Нить
Разделяемая	Мультипроц.	Нет	R/W	Блок	Блок
Глобальная	DRAM GPU	Нет	R/W		
Константная	DRAM GPU	Есть	R/O		
Текстурная	DRAM GPU	Есть	R/O		
Общее адресное пространство (UVA)	DRAM хоста	Нет	R/W	Везде	Динамическое

Одни виды памяти расположены непосредственно в каждой из потоковых мультипроцессоров, другие размещены в DRAM.

Наиболее простым видом памяти является регистровый файл. Каждый потоковый мультипроцессор в GPU с compute capability 1.0, 1.1, 1.2, 1.3 и 2.x, содержит 8192, 16384 или 32768 32-битных регистров соответственно. Регистры распределяются между нитями блока на этапе компиляции, что

влияет на максимальное количество блоков, которые может выполнять один мультипроцессор. Каждая нить получает в использование для чтения и записи некоторое количество регистров на все время исполнения ядра. Доступ к регистрам других нитей запрещен. Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, они обладают минимальной латентностью [1].

Если регистров не хватает, то для размещения локальных данных нити дополнительно используется локальной памятью. Так как она размещена в DRAM, то латентность доступа к ней составляет от 400 до 800 тактов. Также в локальную память всегда попадают союзы (unions), а также больше или нетривиально индексируемые автоматические массивы.

Важным типом памяти в CUDA является разделяемая память (shared memory). Эта память расположена непосредственно в потоковом мультипроцессоре, но в отличие от регистров, выделяется не на уровне нитей, а на уровне блоков. Каждый блок получает в свое расположение одно и то же количество разделяемой памяти. Каждый мультипроцессор устройств с compute capability 2.x содержит 48 Кбайт разделяемой памяти. От размеров разделяемой памяти, требуемой каждому блоку, зависит максимальное количество блоков, которое может быть запущено на одном мультипроцессоре. Разделяемая память обладает очень небольшой латентностью, сравнимой со временем доступа к регистрам и может быть использована всеми нитями блока для чтения и записи [1-5].

Глобальная память, это обычная память DRAM, расположенная на плате GPU или разделяемая с памятью мобильного устройства. Начиная с архитектуры Fermi глобальная память кэшируется. Влияние присутствия кэша не стоит переоценивать, в отличии от CPU, его размер в пересчете на одну нить очень мал. Глобальная память может выделяться как с CPU функциями CUDA API, так и нитями CUDA ядра с помощью вызова malloc. Все нити ядра могут читать и писать в глобальную память. Как и локальная память, глобальная обладает высокой латентностью. Минимизация доступа к

глобальной памяти, это основной метод получения высокоэффективных CUDA приложений.

Константная и текстурная память также расположены в DRAM, но обладают независимым кэшем, обеспечивающим высокую скорость доступа. Оба типа памяти доступны всем нитям GPU, но только для чтения. Запись в них может производить CPU при помощи функций CUDA API. Общий объем памяти ограничен 64 Кбайтами. Текстурная память также предоставляет специальные возможности по работе с текстурами.

Также любой CUDA вызов, связанный с выделением и освобождением памяти на GPU, всегда является синхронным и поэтому будет выполнен только после завершения всех активных асинхронных вызовов [1-5].

В результате анализа предметной области и выявления недостатков было решено, что в настоящее время ускорение процесса обработки экспериментальных данных необходимо.

Определены требования необходимые для реализации приложения. Проанализированы теоретические основы разработки приложения с ускорением на базе GPU NVIDIA.

Глава 2 Проектирование вычислительного ядра

2.1 Анализ алгоритма Фурье

Эффективное распараллеливание алгоритма Фурье позволит ускорить обработку и вычисление данных. Вычисления Фурье с использованием потоков на CPU обеспечиваются на уровне итерационного цикла. Вычислительная эффективность многопроцессорного устройства со специальной системой связей находится в прямой зависимости от соответствия его структуры информационному графу, при помощи которого описывается решаемая задача или класс решаемых задач. Информационный граф стандартного алгоритма быстрого преобразования Фурье (БПФ) характеризуется наличием перекрестных связей между вершинами графа, причем вид связей зависит от номера итерации алгоритма БПФ. При реализации стандартного алгоритма БПФ вычислительной системой, состоящей из нескольких процессоров, должна быть предусмотрена коммутационная структура, обеспечивающая в соответствии с алгоритмом БПФ переключение связей между процессорами [6-9].

При реализации алгоритма Фурье на параллельных вычислительных системах, стоит знать о некоторых особенностях. Когда преобразование Фурье адаптируется для параллельных вычислений логическая сложность алгоритма, а также сложность деления данных по частям становится существенным проблемой, из-за которой реализация значительно замедляется. Большинство многопроцессорных вычислительных систем с раздельной памятью отдает максимальное число узлов, которые являются степенью двойки. Также нелинейная схема вызовов элементов массива сказывается на скорости различных алгоритмов Фурье [6-7].

2.2 Структура программы и назначение основных компонентов

Была разработана структура программы и представлена на рисунке 2.1.

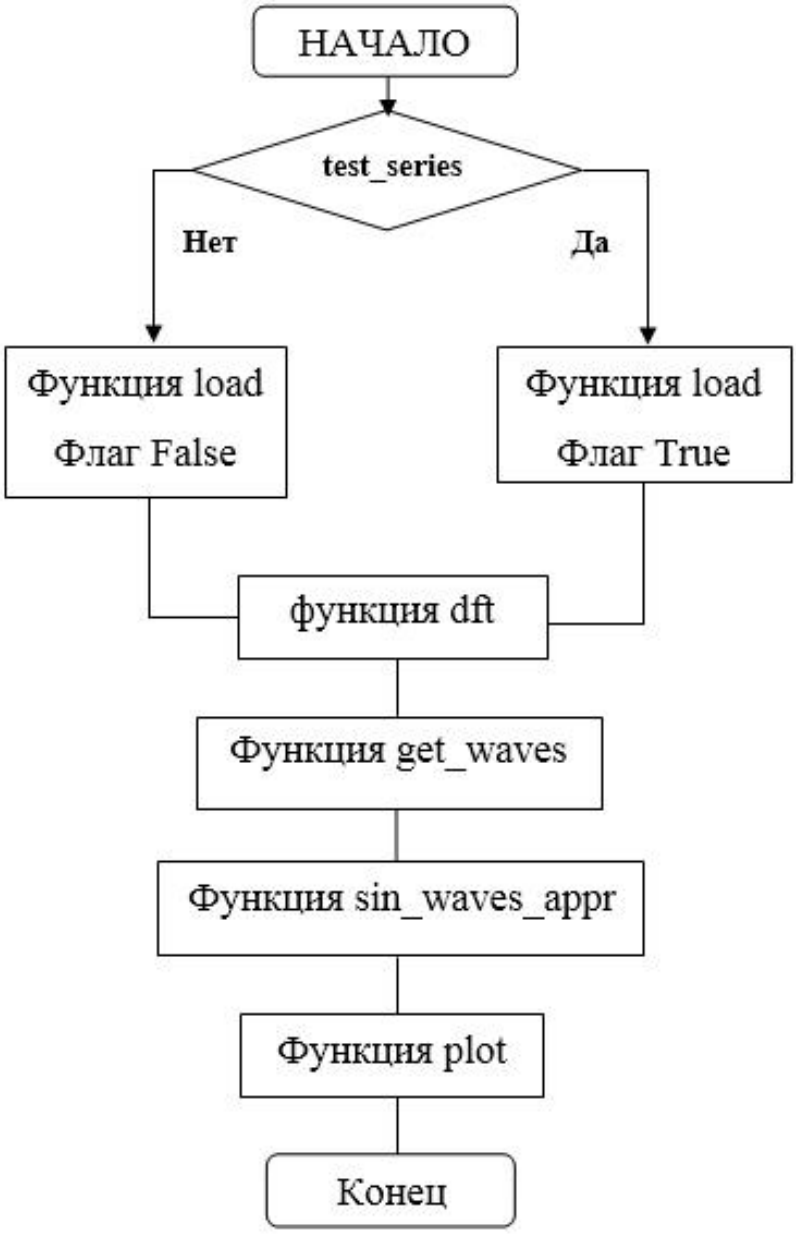

Рисунок 2.1 – Структура разрабатываемого программного продукта

Функция fft(series: list), на вход функция получает серию из N значений. А на выходе функция выдает M гармоник разложения (амплитуда, фаза). По алгоритму считается, что все N значений сигнала были замерены с определенной и фиксированной частотой v_d – частотой дискретизации. Это значит, что временной шаг измерения уровня сигнала должен быть фиксирован и равен периоду дискретизации $T_d = t_{i+1} - t_i = const$. Комплексная амплитуда гармоники X_w вычисляется по формуле:

$$X_w = \sum_{t=0}^{N-1} S(t) * e^{-\frac{2\pi i}{N} wt}, \quad (2.1)$$

где $S(t)$ - серия значений;

$t = 0, T_d, 2T_d, 3T_d \ldots (N-1)T_d$;

w = 0,1,2, … M – 1;

M - количество компонент разложения (количество получаемых синусоидальных гармоник сигнала).

В программе функция fft возвращает $M = \frac{N}{2}$ гармоник в виде [амплитуда, фаза]. X_w в функции fft вычисляется по реальной и мнимой компонентам отдельно, так как функция cos четная реальная часть вычисляется по формуле 2.2, а мнимая часть по формуле 2.3, так как sin нечетная.

$$Re(X_w) = \sum_{t=0}^{N-1} S(t) * \cos\left(-\frac{2\pi}{N} wt\right) = \sum_{t=0}^{N-1} S(t) * \cos\left(\frac{2\pi}{N} wt\right), \quad (2.2)$$

$$Im(X_w) = \sum_{t=0}^{N-1} S(t) * \sin\left(-\frac{2\pi}{N} wt\right) = \sum_{t=0}^{N-1} (-1) * S(t) * \sin\left(\frac{2\pi}{N} wt\right) =$$

$$= (-1) * \sum_{t=0}^{N-1} S(t) * \sin\left(\frac{2\pi}{N} wt\right), \quad (2.3)$$

Такое покомпонентное вычисление комплексного числа возможно благодаря известной формуле Эйлера:

$$e^{\varphi i} = \cos(\varphi) + i * \sin(\varphi) \text{ или } e^{-\varphi i} = \cos(\varphi) - i * \sin(\varphi), \quad (2.4)$$

из комплексной амплитуды X_w получаем вещественную амплитуду (формула 2.5) и фазу (формула 2.6):

$$A_w = \frac{|X_w|}{M}, \quad (2.5)$$

$$\varphi_w = arg(X_w) = \operatorname{atan2}\left(Im(X_w), Re(X_w)\right), \quad (2.6)$$

Функция get_waves(series: list, num), на вход функция получает серию из N значений, и параметр num это ограничитель количества гармоник. На

выходе возвращает num компонентов разложения (гармоник). После вызывается функция fft, которая выдаёт M гармоник в виде [амплитуда, фаза]**.** В цикле проходит по ним и создаёт список объектов Wave (амплитуда, частота, фаза). Частота определяется из амплитуды и фазы по следующей формуле**:**

$$f_w = \frac{2\pi w}{(2M)}, \tag{2.7}$$

где w – номер волны.

Далее список объектов волн сортируется в убывающем порядке либо по амплитуде (freq_sort = False), либо по частоте (freq_sort = True). И из полученного упорядоченного списка выбирает первые num волн.

Функция sin_waves_appr(series: list, num), Аппроксимирует серию из N значений используя num старших гармоник разложения. На выходе возвращает серию такой же размерности из точек Y_k. Аппроксимация происходит по формуле:

$$Y_k = \sum_{j=0}^{M-1} A_j * cos\left(k * f_j + \varphi_j\right), \tag{2.8}$$

где Y_k это аппроксимированное значение;

k = 0, 1, 2, … N-1.

Рассмотрим использование функции на примере для наглядности ведем значения N = 500 и num = 2. Сначала генерируется тестовая серия согласно формуле:

$$S(t) = 2 * \sin(0.05 * t + 0.5) + 0.5 * \sin(0.2 * t + 1.5), \tag{2.9}$$

где t=0, 1, 2, … N-1.

Аппроксимированная модель сигнала вычисляется по формуле 2.10, спектр сигнала по формуле 2.11.

$$\mathrm{Y(t)} = \mathrm{sin_waves_appr(S(t),\ num)}, \tag{2.10}$$

$$\mathrm{A} = \mathrm{fft(\ S(t)\ )}, \tag{2.11}$$

Список из M амплитуд разложений сигнала, и рисуется на нижнем графике без учёта фазы согласно масштабу по оси X:

$$x_k = \frac{2\pi k}{N}, \tag{2.12}$$

На рисунке 2.2 представлен вывод исходной серии и аппроксимированной на верхнем графике и спектра исходной серии на нижнем графике.

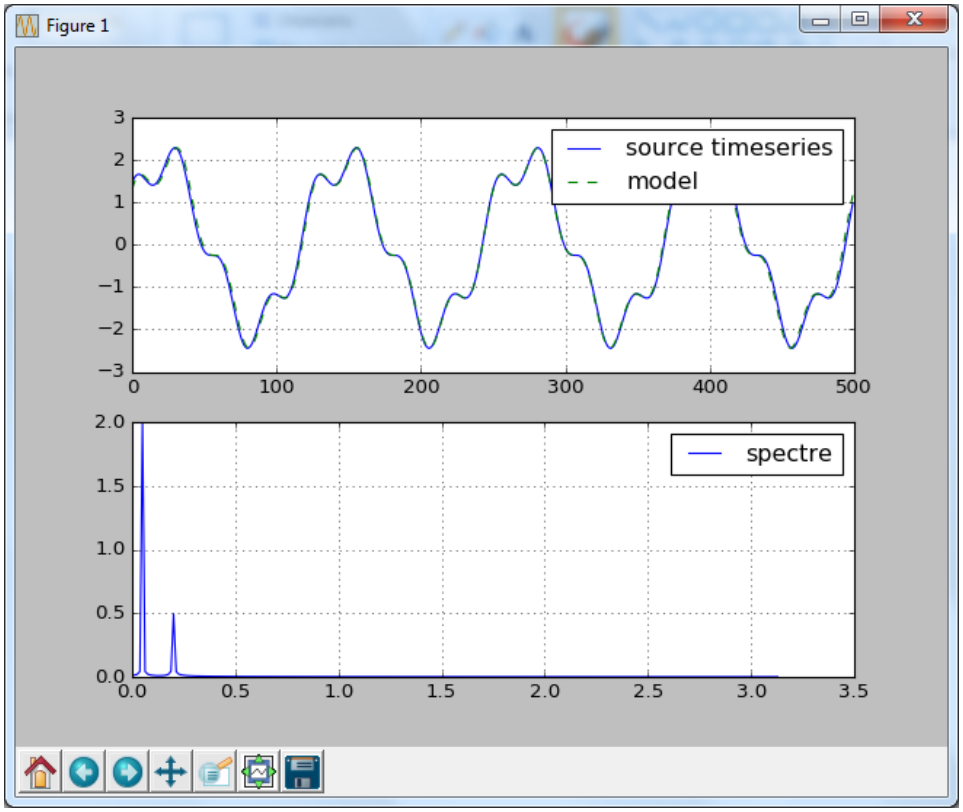

Рисунок 2.2 – Результат выполнения функции в тестовом примере

В данной главе были описаны основные функции необходимые для реализации приложения. А также проведен анализ алгоритма Фурье на предмет параллельности процессов.

Глава 3 Реализация вычислительного ядра

3.1 Выбор средств реализации

Исследовав популярные технологии разработки программных продуктов совместимых с технологией Compute Unified Device Architecture от NVidia, сформировались критерии характеризующие технологию с целью разработки на основе которых, методика выбора оптимального средства создания программного продукта в зависимости от задач [5].

Разработка каждого программного продукта требует конкретные критерии в зависимости от конкретных задач, а также от характеристик технологии. Критериями языка для программирования в данной выпускной квалификационной работе являются:

- поддержка ООП;
- поддержка динамической типизации;
- поддержка многопоточности;
- удобные высокоуровневые структуры данных.

Исходя из этих критериев, были выделены следующие высокоуровневые языки программирования C++ и Python.

C++ имеет два основных аспекта, это набор алгоритмов и набор данных, которыми оперируют. В процедурной парадигме программирования задача непосредственно моделируется набором алгоритмов. Так для примера можно представить систему выдачи книг в библиотеке. В ней реализуются две главные процедуры, процедура выдачи книг и процедура приема книг. Данные хранятся отдельно и передаются этим процедурам как параметры. К наиболее известным процедурным языкам программирования относятся FORTRAN, C и Pascal. C++ также поддерживает процедурное программирование. Отдельные процедуры носят в этом языке название функций.

В 70-е годы процедурную парадигму стала вытеснять парадигма абстрактных типов данных. В рамках этой парадигмы задача моделируется

набором абстракций данных. В языке C++ эти абстракции получили название классов. Алгоритмы, реализуемые каждым классом, называются открытым интерфейсом класса. Данные скрыты внутри объектов класса.

Python один из наиболее популярных современных языков программирования. Он пригоден для решения разнообразных задач и предлагает те же возможности, что и другие языки программирования:

– поддержка ООП;
– динамичность;
– кросс-платформенность.

Разработку Python начал Гвидо Ван Россум (Guido Van Rossum) еще в середине 1990-х годов [10-12; 16].

Python поддерживает несколько парадигм программирования, такие как:

– структурная;
– объектно-ориентированная;
– функциональная;
– императивная;
– аспектно-ориентированная.

Основные архитектурные черты, это динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули.

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях, включая проприетарные. Есть реализации интерпретаторов для JVM, MSIL, LLVM и других. Проект PyPy предлагает реализацию Python

на самом Python, что уменьшает затраты на изменения языка и постановку экспериментов над новыми возможностями [16-20; 15].

Python — активно развивающийся язык программирования, новые версии выходят примерно раз в два с половиной года. Вследствие этого и некоторых других причин на Python отсутствуют стандарт ANSI, ISO или другие официальные стандарты, их роль выполняет CPython. Синтаксис ядра Python минималистичен, в то же время стандартная библиотека включает большой объём полезных функций [16-20].

Оба высокоуровневых языка программирования соответствуют критериям, но Python имеет небольшое преимущество перед C++. К таким преимуществам относятся как, большое количество модулей расширения, наличие удобных графических библиотек.

В результате анализа технологий создания программного продукта был выбран язык программирования Python.

3.2 Реализация вычислительного ядра

На рисунке 3.1 представлены библиотеки, использованные для реализации вычислительного ядра, такие как NumPy, Matplotlib.pyplot, Math, Time, а также использована инструкция from с помощью которой подключены атрибуты библиотеки Numbapro, это cuda, jit, void, float32, int32.

```
# -*- encoding: utf-8 -*-

#%matplotlib inline

import math
import matplotlib.pyplot as plt

import time
import numpy as np
from numbapro import cuda, jit, void, float32, int32
```

Рисунок 3.1 – Листинг подключенных библиотек и их атрибутов

Numpy это одна из базовых библиотек, которая позволяет проводить вычисления с многомерными массивами. Включает в себя реализацию

основных процедур линейной алгебры, работу со случайными числами, а также преобразование Фурье [16-20; 15].

Math это одна из важнейших библиотек Python для работы с числами. Включает в себя округление до ближайшего числа, факториал числа, остаток от деления, логарифмирование числа, квадратный корень, вычисление гиперболических синуса, косинуса и тангенса, а также многие другие операции [10-12; 15; 16-20].

Matplotlib это библиотека для работы с двумерными графиками для языка программирования Python, с помощью которой можно создавать рисунки различных форматов. В свою очередь Pyplot это интерфейс, позволяющий использовать готовые решения и настройки базовых параметров рисунка [10-12; 16-20].

Numbapro это расширенная версия библиотеки Numba которая добавляет дополнительные возможности, которые позволяют позволяют создавать оптимизированный код хорошо интегрируемый с Numpy. В работе с GPU Numbapro позволяет оптимизировать код и максимально упростить работу с аппаратными средствами [19].

Для реализации вычислительного ядра были написаны такие основные функции как get_waves, sin_waves_appr, fftCUDA.

Функция get_waves(series: list, num), на вход функция получает серию из N значений, и параметр num это ограничитель количества гармоник. На выходе возвращает num компонентов разложения (гармоник). После вызывается функция fft, которая выдаёт M гармоник в виде [амплитуда, фаза]**.** В цикле проходит по ним и создаёт список объектов Wave (амплитуда, частота, фаза). Частота определяется из амплитуды и фазы по следующей формуле**:**

$$f_w = \frac{2\pi w}{(2M)}, \tag{3.1}$$

где w – номер волны.

Далее список объектов волн сортируется в убывающем порядке либо по амплитуде (freq_sort = False), либо по частоте (freq_sort = True). И из полученного упорядоченного списка выбирает первые num волн. Листинг функции get_waves представлен на рисунке 3.2.

```
def get_waves(series : list, num):
    # получаем m волн в виде амплитуда-фаза
    amp, phase = fft(series)
    # построить Wave объекты, дополнив частотой
    m = len(amp)
    waves = []
    for i in range(m):
        waves.append(Wave(amp[i], 2 * math.pi * i / (m * 2), phase[i]))
    # отсортировать и выбрать num старших гармоник
    #(с наибольшими амплитудами или частотами, см. freq_sort)
    ts = TEstimate()
    ts.start("Время на сортировку объектов волн в убывающем порядке: ")
    waves.sort()
    waves.reverse()
    ts.diff()
    waves = waves[0:num] #num самых больших гармоник
    print("the program found the next %s sin waves:"% len(waves))
    #print(waves)
    return waves
```

Рисунок 3.2 – Листинг функция get_waves

Функция sin_waves_appr(series: list, num), Аппроксимирует серию из N значений используя num старших гармоник разложения. На выходе возвращает серию такой же размерности из точек Y_k. Аппроксимация происходит по формуле:

$$Y_k = \sum_{j=0}^{M-1} A_j * cos\left(k * f_j + \varphi_j\right), \quad (3.2)$$

где Y_k это аппроксимированное значение,

$k = 0, 1, 2, \ldots$ N-1.

Для примера листинг функции sin_waves_appr изображен на рисунке 3.3

```
def sin_waves_appr(series : list, num):
    # секундомер наш
    ts = TEstimate("Time execution sin_waves_appr: ")
    ts.start()
    #
    n = len(series)
    freq = get_waves(series, num)
    m = len(freq)
    # тут будут накапливаться новые значения точек, аппроксимированных через пит гармоник
    model = []
    for i in range(n):
        # чтобы получить каждую аппроксимированную точку модели,
        #нужно найти взвешенную сумму по всем компонентам разложения
        summ = 0
        for j in range(m):
            summ += freq[j].amp * math.cos(freq[j].freq * i + freq[j].phase)
        model.append(summ)
    # показываем время, сколько прошло с вызова ts.start
    ts.diff()
    return model
```

Рисунок 3.3 – Листинг функция sin_waves_appr

Функция fftCUDA(series: list) оптимизированная под GPU (CUDA). На вход функция получает серию из N значений. На выходе выдаёт M гармоник разложения (амплитуда, фаза). Каждый поток вычисляет одну компоненту разложения. M потоков разделяются на блоки фиксированной ширины. Каждый блок потоков обязан пройти по всем N точкам исходной серии и получить взвешенную сумму косинусов (для реальной части комплексного числа) и синусов (для мнимой части комплексного числа).

Каждый блок в цикле используя свои потоки за раз считывает следующую порцию данных в быструю память, синхронизируется со всеми потоками. Далее каждый поток в своём цикле проходит по совместному участку быстрой памяти и считает свою часть взвешенной суммы косинусов и синусов и аккумулирует. После потоки синхронизируются, чтобы все потоки из блока смогли успеть вычислить свою сумму и переходят к чтению следующей порции точек из серии (следующая итерация). Так пока не пройдут все блоки по всем N точкам серии. После того как реальная и мнимая часть комплексной компоненты разложения готовы, каждый поток преобразует их в амплитуду по формуле (3.3) и фазу волны согласно формуле (3.4):

$$A_w = \frac{|X_w|}{M}, \tag{3.3}$$

$$\varphi_w = arg(X_w) = \operatorname{atan2}\big(\mathrm{Im}(X_w), \mathrm{Re}(X_w)\big). \quad (3.4)$$

Листинг функции fftCUDA оптимизированной под GPU (CUDA) изображен на рисунке 3.4.

```
# Запуск kernel-функции, принимает список из точек серии
def fftCUDA(series : list):
    # Convert to numpy array
    series = np.asarray(series, dtype='float32')
    #
    n = series.shape[0] #длина серии
    m = int(n / 2) #максимальное количество компонент разложения
    print ("FFT CUDA:", n, m)
    # Секундомер
    ts = TEstimate("Time execution FFT CUDA: ")
    ts.start()
    # Future result - сюда будут собираться амплитуды и фазы гармоник разложения
    amplitude = np.zeros(shape=m, dtype=series.dtype)
    phase = np.zeros(shape=m, dtype=series.dtype)
    # Run kernel with data - отсылаем данные в память GPU
    gpu_in = cuda.to_device(series)
    gpu_amp = cuda.to_device(amplitude)
    gpu_phase = cuda.to_device(phase)
    # определяем количество задействованных блоков для задачи
    numBlocks = int((m+BLOCK_SIZE-1)/BLOCK_SIZE)
    if numBlocks == 0:
        numBlocks = 1
    print ("Задача разделена на %d блоков по %d потоков" % (numBlocks, BLOCK_SIZE))
    fftKernel[numBlocks, BLOCK_SIZE](gpu_in, gpu_amp, gpu_phase, m, n)
    gpu_amp.to_host()   #копируем результат из памяти девайса GPU в переменную amplitude
    gpu_phase.to_host()  #копируем результат из памяти девайса GPU в переменную phase
    #
    ts.diff()
    return amplitude.tolist(), phase.tolist() #return as lists - переводим в обычные списки
```

Рисунок 3.4 – Листинг функции fftCUDA

Полный листинг приведен в приложении.

3.3 Описание установки и тестирования программного продукта.

Качество программного продукта характеризуется набором свойств, определяющих насколько программный продукт актуален на сегодняшний день. Каждый человек может иметь разное представление о продукте и о том, насколько он качественно сделан. Таким образом, постановка задачи обеспечения качества программного продукта, переходит в задачу нахождения оптимального решения. Тестирование является одним из наиболее устоявшихся способов обеспечения качества разработки программного обеспечения и входит в набор эффективных средств современной системы обеспечения качества программного продукта.

С технической точки зрения тестирование заключается в выполнении приложения на некотором множестве исходных данных и сверке получаемых

результатов с заранее известными, с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам. Тестирование является самой популярной методикой повышения качества программного продукта, подкрепленная многими исследованиями и богатым опытом разработки коммерческих приложений. Существует множество видов тестирования, одни обычно выполняют сами разработчики, а другие – специализированные группы. Виды тестирования перечислены ниже:

– блочное тестирование, это тестирование полного класса, метода или небольшого приложения, написанного одним программистом или группой, выполняемое отдельно от прочих частей системы;

– тестирование компонента, это тестирование класса, пакета, небольшого приложения или другого элемента системы, разработанного несколькими программистами или группами, выполняемое в изоляции от остальных частей системы;

– интеграционное тестирование, это совместное выполнение двух или более классов, пакетов, компонентов или подсистем, созданных несколькими программистами или группами;

– регрессионное тестирование, это повторное выполнение тестов, направленное на обнаружение дефектов в программе, уже прошедшей этот набор тестов;

– тестирование системы, это выполнение программного обеспечения в его окончательной конфигурации, интегрированного с другими программными и аппаратными системами.

Реализация тестирования программного продукта была проведена в два этапа. А именно создание тестового набора путем ручной разработки или автоматической генерации для конкретной среды тестирования и оценка результатов выполнения программы на наборе тестов с целью принятия решения о продолжении или остановке тестирования.

Для запуска программного продукта необходимо иметь установленный Anaconda, а также менеджер внутренних пакетов Conda. В данной работе для тестирования функционала использовался IPython Notebook.

Для установки и запуска необходимо выполнить следующие действия:

– если «Anaconda» не установлен, то устанавливаем его в директорию по умолчанию;

– запускаем консоль «Anaconda», посредством двойного нажатия на «Anaconda Prompt»;

– устанавливаем пакеты «Numbapro», «Accelerate» используя менеджер внутренних пакетов «Conda»(т.е. в данном случае команда будет выглядеть так «conda install accelerate» и после этого «conda install numbapro»);

– запускаем интерактивную оболочку командой «ipython notebook» (все команды должны выполняться в консоли «Anaconda», а не в консоли Windows);

– после выполнения предыдущей команды в браузере по умолчанию должна открыться страница главного окна программы;

– для создания нового проекта нажимаем на кнопку «New» в выпадающем списке выбираем предустановленную версию python (для написания данного приложения использовалась версия python 3.5);

– если приложение уже было создано и сохранено, то находим его в списке главного окна;

– главное окно будет доступно по ссылке http://localhost:8888/.

В результате тестирования были устранены ошибки при запуске «IPython Notebook».

3.4 Работа с Anaconda

Для использования программного продукта, необходимы средства описанные выше, а также наличие браузера. Для запуска компилятора необходимо запустить «Anaconda Prompt» и выполнить команду «ipython

notebook». Для наглядности менеджер пакетов Anaconda изображен на рисунке 3.5.

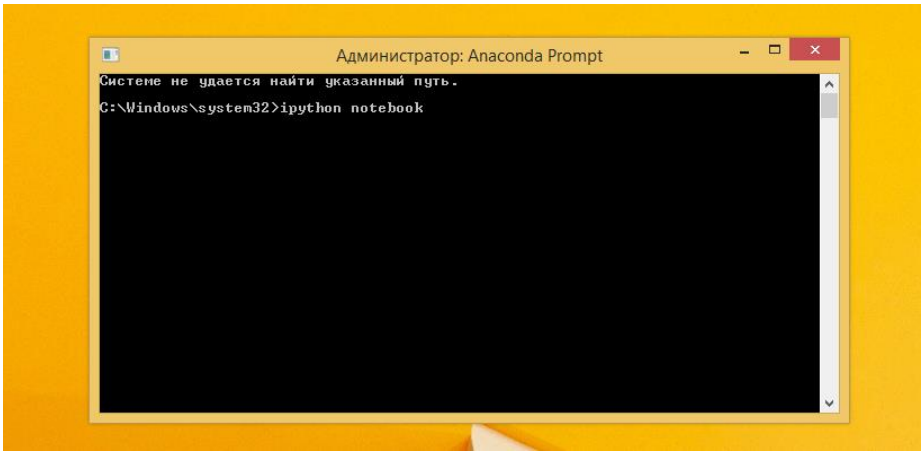

Рисунок 3.5 – Менеджер пакетов Anaconda

После чего запустится браузер по умолчанию, если браузер по умолчанию не запустился, то вручную в адресной строке переходим по ссылке http://localhost:8888/. Для наглядности оболочка для работы с Python изображена на рисунке 3.6.

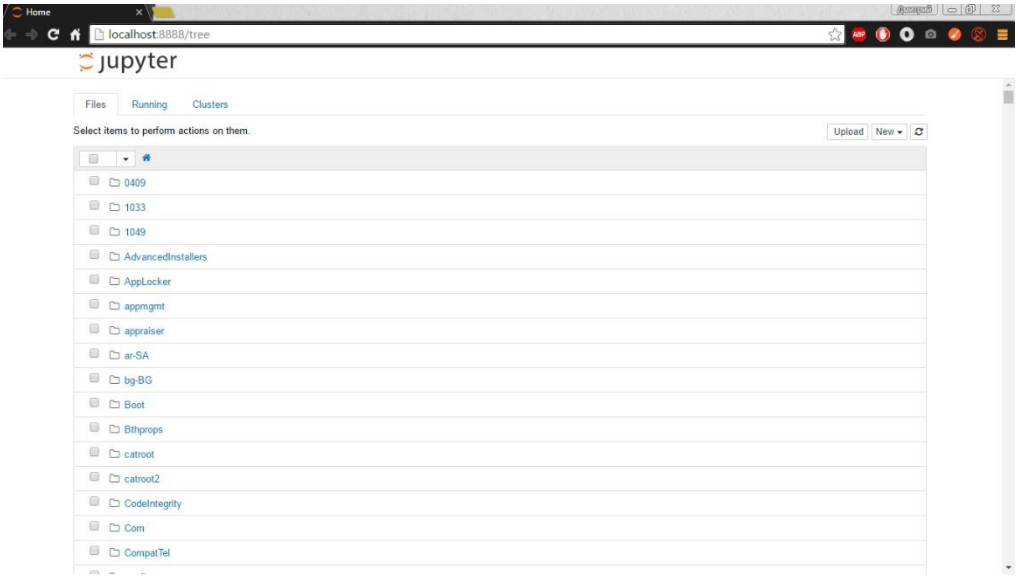

Рисунок 3.6 – Оболочка для работы с Python

Чтобы загрузить файлы необходимые для работы программного продукта, необходимо нажать кнопку «Upload» и в появившемся окне выбрать файл, который вам нужен. После чего нажать кнопку «Upload» находящуюся непосредственно напротив вашего файла. Для наглядности Загрузка файлов изображена на рисунке 3.7.

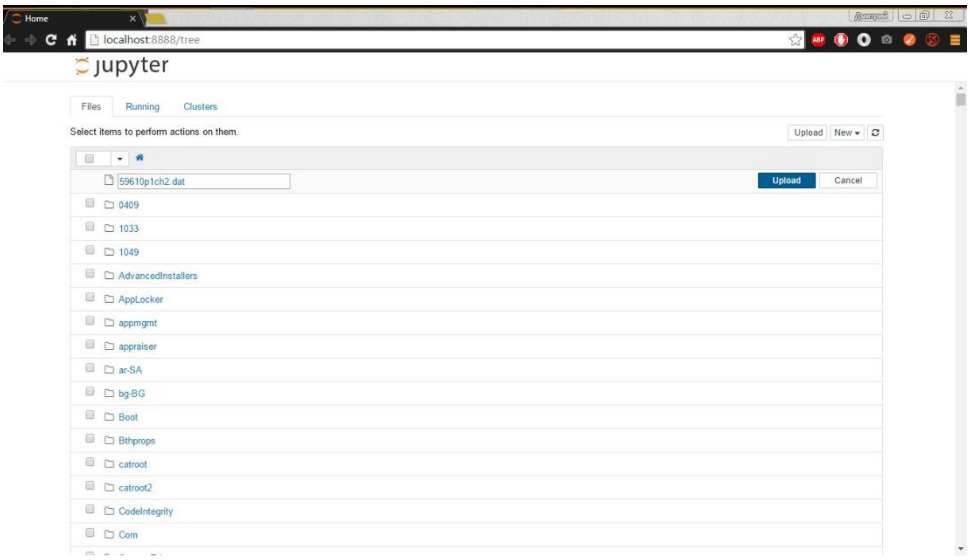

Рисунок 3.7 – Загрузка файлов

Для создания нового решения необходимо нажать на кнопку «New», в выпадающем списке выбираем предустановленную версию python (для написания данного программного продукта использовалась версия python 3.5). Для наглядности создание нового проекта изображено на рисунке 3.8.

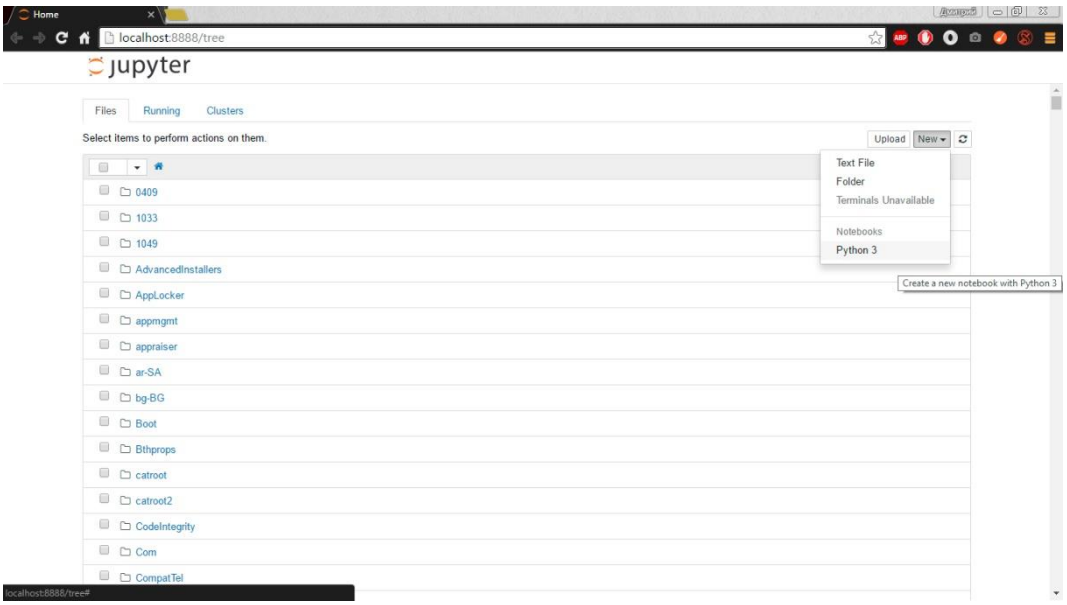

Рисунок 3.8 – Создание нового проекта

После нажатия на кнопку «Python 3» как показано на рисунке 3.8, открывается новое окно с компилятором. И уже непосредственно в нем и был разработано приложение. Для наглядности компилятор Jupyter изображен на рисунке 3.9.

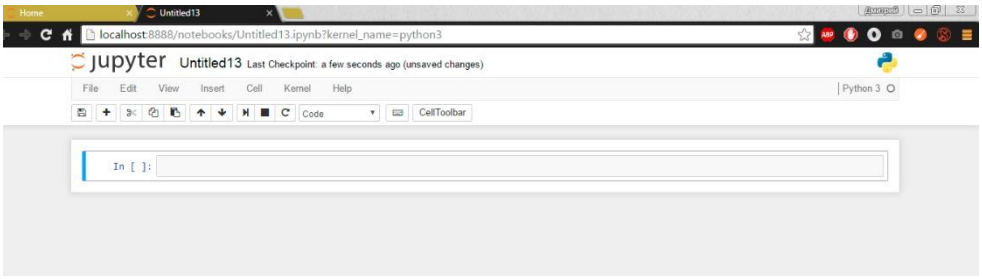

Рисунок 3.9 – Компилятор Jupyter

Для выхода из «jupyter» нужно закрыть браузер. Но также нужно остановить сервер нажатием клавиш «Ctrl + C». Для наглядности прекращение работы ipython notebook изображено на рисунке 3.10.

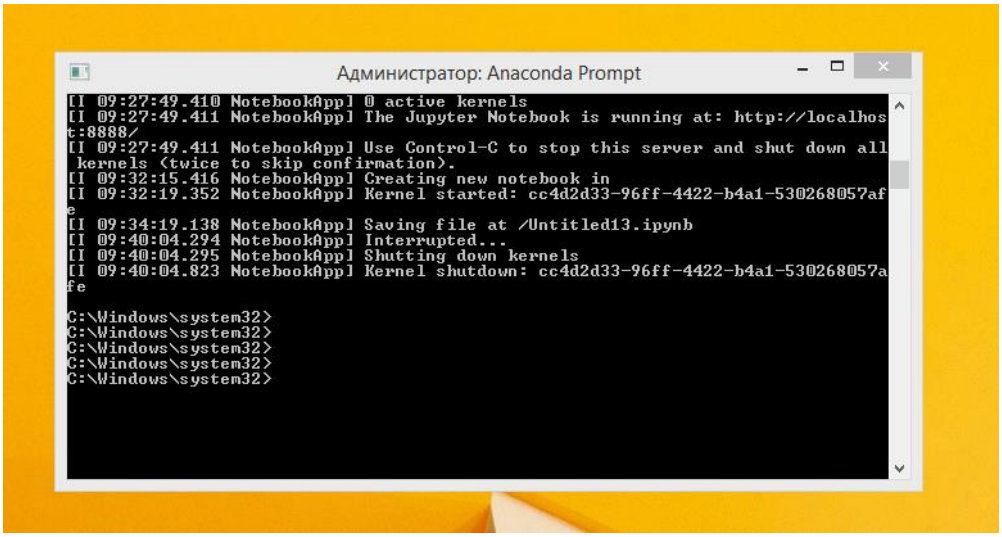

Рисунок 3.10 – Прекращение работы ipython notebook

3.5 Тестирование и анализ эффективности вычислительного ядра

Для запуска программного продукта необходимо запустить Ipython компилятор, как это сделать описано в пункте 3.2

Для сравнения будем использовать алгоритм преобразования Фурье, написанный на языке программирования Python без ускорения вычислений при помощи технологии CUDA от NVidia и его аналог, но уже ускоренный.

Сначала удостоверимся что все вычисления производятся правильно и на графике отображены верные данные. Тест будет проводиться с количеством точек равным 1000 и количеством гармоник для аппроксимации равным 2. На рисунке 3.10 для наглядности сверху изображен сигнал и его аппроксимированная модель, ниже график спектра.

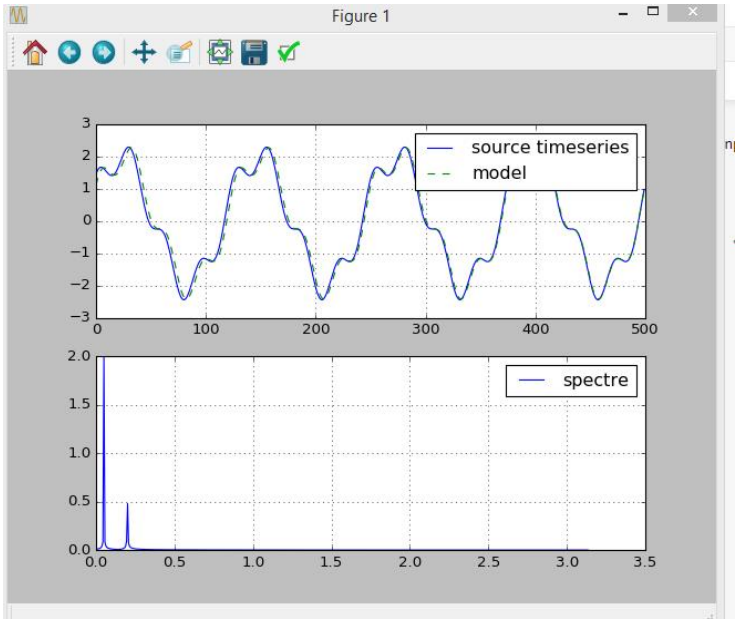

Рисунок 3.10 – Результат выполнения программы без использования CUDA

Для наглядности на рисунке 3.11 изображен результат выполнения программы с использованием CUDA.

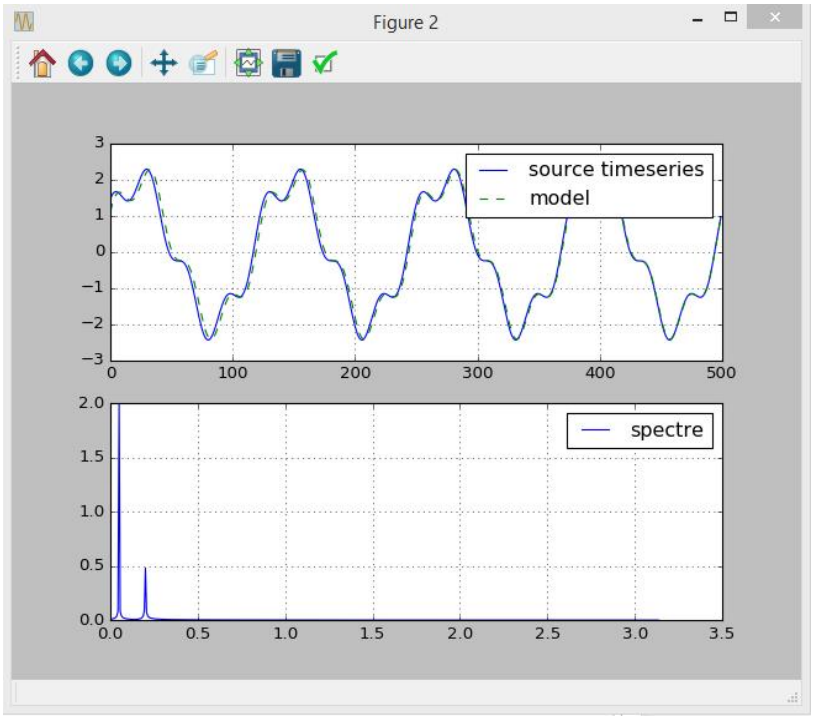

Рисунок 3.11 – Результат выполнения программы с использованием CUDA

На рисунке 3.11 отображены аналогичные данные, исходя из этого делаем вывод что тестирование прошло успешно и ускоренный алгоритм

соответствует своему аналогу. Для наглядности на рисунке 3.12 изображено сравнение затраченного времени на вычисления.

```
DFT SImple: 1000 500
Time execution DFT Simle:  4.611064195632935
the program found the next 2 sin waves:
Time execution of `sin_waves_appr`:  4.62907862663269
DFT SImple: 1000 500
Time execution DFT Simle:  4.608064413070679
====================
DFT CUDA: 1000 500
Time execution DFT CUDA:  0.008006811141967773
the program found the next 2 sin waves:
Time execution of `sin_waves_appr`:  0.03101968765258789
DFT CUDA: 1000 500
Time execution DFT CUDA:  0.006003618240356445
====================
```

Рисунок 3.12 – Сравнение затраченного времени на вычисления

Проведя вычисления на примере тестовой серии, которая составляет 1000 входных значений, можно увидеть, что ускоренная программа эффективна как на малой серии из 1000 входных данных, так и на более крупной.

Для того чтобы убедиться в эффективности и целесообразности разработки данного программного продукта, проведем замер времени, которое тратится на выполнение вычислений. Результат данного замера времени представлен в таблице 3.3 и таблице 3.4.

Таблица 3.1 – Результат тестовой серии без использования CUDA

Количество входных точек	Время выполнения алгоритма Фурье, сек	Время аппроксимации, сек	Время выполнения функции fft, сек
500	1.29475736618042	1.31038498878479	1.311521053314209
1000	5.133431911468506	5.149059057235718	5.146613597869873
1500	11.709134340286255	11.740386486053467	11.64614415168762
10000	62.79357886314392	76.14642143249512	62.65648913383484
20000	269.7803509235382	300.60776352882385	263.9385752677917

Продолжение таблицы 3.2

Количество входных точек	Время выполнения алгоритма Фурье, сек	Время аппроксимации, сек	Время выполнения функции fft, сек
30000	572.1509819030762	614.4200332164764	576.5186364650726
40000	1022.1889519691467	1087.658136844635	1011.568767070770

Таблица 3.2 – Результат тестовой серии с использованием CUDA

Количество входных точек	Время выполнения алгоритма Фурье, сек	Время аппроксимации, сек	Время выполнения функции fft, сек
500	0.0156235694885253	0.0156235694885253	0.000932345544120
1000	0.0156242847442626	0.0312502384185791	0.001334535510102
1500	0.0913434028625531	0.0512519073486328	0.015625476837158
10000	0.1871237754821777	0.2711801528930664	0.143095731735229
20000	0.5703794956207275	0.6994638442993164	0.536355495452880
30000	1.2978589534759521	1.5920519828796387	1.279845714569091
40000	2.1624324321746826	2.503657817840576	2.134418487548828

На рисунке 3.13 приведен график зависимости времени выполнения вычислений алгоритма Фурье на GPU и на CPU от количества точек. По графику видно, что время вычислений на CPU сильно зависит от количества точек и значительно возрастает при обработке больших серий данных. С другой стороны время обработки на GPU возрастает незначительно.

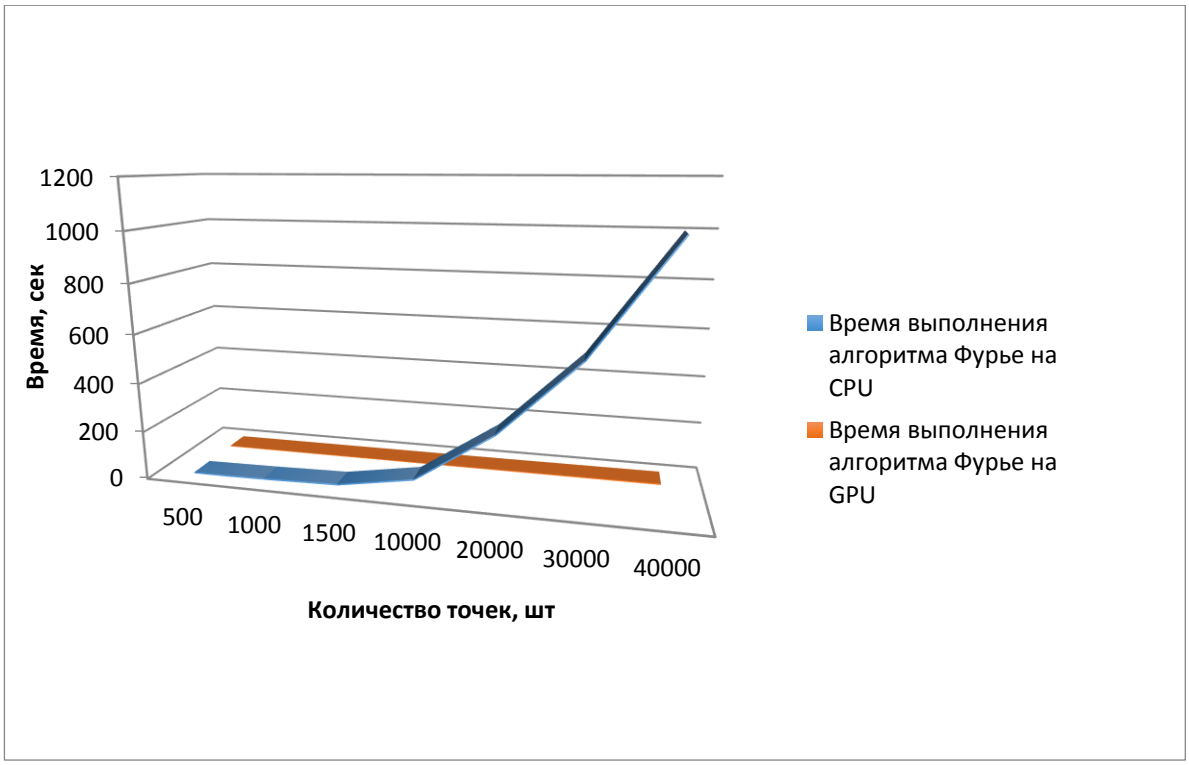

Рисунок 3.13 – График зависимости времени выполнения вычислений алгоритма Фурье на GPU и на CPU от количества точек

В результате анализа были выбраны средства для реализации вычислительного ядра. В результате выполнения действий с интерактивной оболочкой и языка программирования Python было реализовано вычислительное ядро.

Были проведены тесты, которые показали, что технология CUDA позволяет сократить время обработки экспериментальных данных как на малой серии, так и на большой.

Заключение

Использование графического процессора для неграфических вычислений становится неотъемлемой частью деятельности современного человека и функционирования преуспевающих организаций в области науки.

На основе результатов сравнительных тестовых серий можно сказать что применение технологии CUDA наиболее эффективно при входных данных превышающих 1000 точек, ускоренная программа проводит вычисления в 100 раз быстрее. Для малого же количества входных точек применение технологии CUDA не дает большого отрыва по скорости выполнения. В дальнейшем есть возможность улучшения данной реализации. К примеру задействование ресурсов двух и более видеокарт поддерживающих технологию CUDA, для этого необходимо провести ряд исследований. Цели выпускной квалификационной работы были достигнуты и ускорение вычислительного ядра значительно сократило время обработки экспериментальных данных.

Список использованной литературы

Учебники и учебные пособия

1. Боресков А. В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие. / А. В. Боресков, В. А. Садовский. –М.: ИМУ, 2012. –336 с.

2. Перепёлкин Е. Е. Вычисления на графических процессорах (GPU) в задачах математической и теоретической физики. / Е. Е. Перепёлкин, Б. И. Садовников, Н. Г. Иноземцев. –М.: URSS, 2014. –176 с.

3. Иноземцева Н. Г. Оптимизация алгоритмов задач математической физики для графических процессоров. / Н. Г. Иноземцева, Е. Е. Перепёлкин, Б. И. Садовников. –М.: ИМУ, 2012. –240 с.

4. Романенко А. А. Особенности адаптации программ под GPU с использованием технологии OpenACC: учеб. Пособие. –Новосиб.: РИЦ НГУ, 2016. –33 с

5. Джейсон С. Технология CUDA в примерах. Введение в программирование графических процессоров. / С. Джейсон, К. Эдвард –М.: ДМК Пресс, 2011. –232 с.

6. Мирошниченко Г. П. Численные методы. Учебное пособие. / Г. П. Мирошниченко, А. Г. Петрашень. –СПБ.: СПБГУИТМО, 2007. –120 с.

7. Привалов И. И. Ряды Фурье. –М.: Либроком, 2008. –168 с.

8. Харди Г. Г. Ряды Фурье. / Г. Г. Харди, В. В. Рогозинский. –М.: Либроком, 2009. –152 с.

9. Привалов И. И. Ряды Фурье. Учебник. –М.: Юрайт, 2016. –166 с.

10. Лутц М. Программирование на Python. Том 1. –СПБ.: Символ-Плюс, 2011. –992 с.

11. Макграт М. Программирование на Python для начинающих. –М.: Эксмо, 2015. –192 с.

12. Саммерфилд М. Python на практике. –М.: ДМК-Пресс, 2014. –338 с.

Электронные ресурсы

13. NVIDIA Corporation. NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110/210, 2014 //официальный доклад [Электронный ресурс]: сайт NVidia: http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

14. CUDA toolkit Documentation, 2015 // официальная документация [Электронный ресурс]: сайт NVidia: http://docs.nvidia.com/cuda/index.html

15. PyCUDA’s documentation, 2008 // официальная документация [Электронный ресурс]: сайт о документации: https://documen.tician.de/pycuda/

Литература на иностранном языке

16. Solem J. E. Programming Computer Vision with Python. – first edition, O’Reilly Media INC, 2012.

17. Pirnat M. How to Make Mistakes in Python. – first edition, O’Reilly Media INC, 2015.

18. Mertz D. Functional Programming in Python. – first edition O’Reilly Media INC, 2015.

19. Ginther J. S. Start Here: Python Programming Made Fun and Easier. – first edition, Alien Cat Studio, 2013.

20. Mertz D. Picking a Python Version: A Manifesto. – first edition, O’Reilly Media INC, 2015.

Приложение А

Исходный код

```
# -*- encoding: utf-8 -*-
#%matplotlib inline

import os
import math
import matplotlib.pyplot as plt
import time
import numpy as np
from numbapro import cuda, jit, void, float32, int32

# Вспомогательный класс для измерения времени выполнения куска кода
class TEstimate:
    def __init__(self, label=""):
        self.c = None
        self.label = label

    def start(self, label=None):
        self.a = time.time()
        if label:
            self.label = label

    def diff(self):
        self.b = time.time()
        self.c = self.b - self.a
        print (self.label,self.c)
# описание классов
# класс Wave - (амплитуда,частота,фаза)
# сделан для удобной сортировки:
# по амплитуде (freq_sort=False)
# и по частоте (freq_sort=True)

class Wave:
    # флаг сортировки freq_sort:
    # False-сортировка по амплитуде
```

```
# True-сортировка по частоте
freq_sort = False
# конструктор: присвоение переменным значения
def __init__(self, amp, freq, phase):
    self.freq = freq
    self.amp = amp
    self.phase = phase

# Следующие 4 функции используются при сортировки объектов Wave
# вызывается оператором "< other"
def __lt__(self, other):
    if self.freq_sort:
        return self.freq < other.freq
    else:
        return self.amp < other.amp

# вызывается оператором "> other"
def __gt__(self, other):
    if self.freq_sort:
        return self.freq > other.freq
    else:
        return self.amp > other.amp

# вызывается оператором "<= other"
def __le__(self, other):
    if self.freq_sort:
        return self.freq <= other.freq
    else:
        return self.amp <= other.amp

# вызывается оператором ">= other"
def __ge__(self, other):
    if self.freq_sort:
        return self.freq >= other.fre
    else:
```

```
            return self.amp >= other.amp


        # вызывается оператором "str()"
        def __str__(self):
            s = "(amp=" + str(self.amp) + ",frq=" + str(self.freq) + ",phase=" + str(self.phase) + ")"
            return s


        def __repr__(self):
            return self.__str__()
    ###################################################################
    # Старые алгоритмы на чистых списках, без использования ускорения на GPU
    ###################################################################
    # оконное преобразование фурье, алгоритм FFT:
    def fft(series : list):
        # Мой секундомер
        ts = TEstimate()
        ts.start("Time execution FFT Simple: ")
        #
        n = len(series) #длина исходной серии точек
        m = int(n / 2) #максимальное количество компонент разложения (гармоник)
        print ("FFT Simple:", n, m)
        real = [0 for _ in range(n)] #в питоне это можно записать ещё проще: [0]*n
        imag = [0 for _ in range(n)] #в питоне это можно записать ещё проще: [0]*n
        # Амплитуда и фаза для m компонентов разложения (они же гармоники, они же волны, они же синусоиды)
        amplitude = []
        phase = []
        angle_const = 2 * math.pi / n
        for w in range(m):
            a = w * angle_const
            for t in range(n):
                real[w] += series[t] * math.cos(a * t)
                imag[w] += -1 * series[t] * math.sin(a * t)
            amplitude.append(math.sqrt(real[w] * real[w] + imag[w] * imag[w]) / m)
            phase.append(math.atan2(imag[w], real[w]))
```

```
    # показываем время, сколько прошло с вызова ts.start
    ts.diff()
    return amplitude, phase
# извлечение волн из временного ряда
# series - временной ряд, сигнал
# num - количество волн

def get_waves(series : list, num):
    # получаем m волн в виде амплитуда-фаза
    amp, phase = fft(series)
    # построить Wave объекты, дополнив частотой
    m = len(amp)
    waves = []
    for i in range(m):
        waves.append(Wave(amp[i], 2 * math.pi * i / (m * 2), phase[i]))
    # отсортировать и выбрать num старших гармоник (с наибольшими амплитудами или частотами, см. freq_sort)
    ts = TEstimate()
    ts.start("Время на сортировку объектов волн в убывающем порядке: ")
    waves.sort()
    waves.reverse()
    ts.diff()
    waves = waves[0:num] #num самых больших гармоник
    print("the program found the next %s sin waves:"% len(waves))
    #print(waves)
    return waves

#аппроксимирует сигнал в num гармоник
#series - наш временной ряд

def sin_waves_appr(series : list, num):
    # секундомер наш
    ts = TEstimate("Time execution sin_waves_appr: ")
    ts.start()
    #
```

```
    n = len(series)
    freq = get_waves(series, num)
    m = len(freq)
    # тут будут накапливаться новые значения точек, аппроксимированных через num гармоник
    model = []
    for i in range(n):
      # чтобы получить каждую аппроксимированную точку модели, нужно найти взвешенную сумму по всем компонентам разложения
      summ = 0
      for j in range(m):
        summ += freq[j].amp * math.cos(freq[j].freq * i + freq[j].phase)
      model.append(summ)
    # показываем время, сколько прошло с вызова ts.start
    ts.diff()
    return model
  ###################################################################
  # размер блока для задачи на GPU
  # размер может быть от 1 до 1024 на большинствах картах и означает сколько потоков будет выделено на каждом блоке
  # количество блоков тоже может быть от 1 до 1024
  # поэтому максимальное общее количество потоков на карте может быть 1024x1024
  BLOCK_SIZE = 512
  # kernel-функция оконного преобразование фурье (исполнение на GPU)
  @jit(void(float32[:], float32[:], float32[:], int32, int32), target='gpu')
  def fftKernel(S, amp, phase, M, N): #N - длина серии; amp, phase - выходные данные разложения
    # окно быстрой памяти, сюда параллельно будет копироваться (кэшироваться) BLOCK_SIZE точек для последующего чтения
    T = cuda.shared.array(shape=(BLOCK_SIZE,), dtype=float32)
    #
    bx = cuda.blockIdx.x  #i-block
    tx = cuda.threadIdx.x #i-thread
    gw = cuda.gridDim.x   #num blocks
    bw = cuda.blockDim.x  #BLOCK_SIZE
    # Start
```

```
    idx = bx*bw + tx      #w - same as cuda.grid(1)
    Re = 0.0              #local var within each thread
    Im = 0.0              #local var within each thread
    a = 2 * math.pi * idx / N #const_angle within each thread


    # Calculate
    k = 0
    # шагаем от 0 до N блоками
    for i in range(0, N, BLOCK_SIZE):
        # скопировать следующий блок значений в быструю память, каждый поток из блока копирует в свою ячейку
        k = i + tx
        # количество N может быть не кратным размеру блока BLOCK_SIZE
        if k < N:
            T[tx] = S[k]
        # подождать пока все потоки дойдут до этого места
        cuda.syncthreads()
        # вычислить часть взвешенной суммы для Re и Im, используя кэшированные значения из быстрой памяти
        k = i
        for t in range(BLOCK_SIZE):
            Re += T[t]*math.cos(a * k)
            Im += T[t]*math.sin(a * k)
            k += 1
            # количество N может быть не кратным размеру блока BLOCK_SIZE
            if k >= N:
                break
        # подождать пока все потоки дойдут до этого места
        cuda.syncthreads()
    # Output results
    # количество M может быть не кратным размеру блока BLOCK_SIZE
    # лишние потоки из блока, для которых idx >= M пройдут мимо, так как нам не нужно больше компонент, чем M
    if idx < M:
        amp[idx] = math.sqrt(Re*Re + Im*Im) / M
        if Im != 0:
```

```
            Im = -Im
        phase[idx] = math.atan2(Im, Re)


# Запуск kernel-функции, принимает список из точек серии
def fftCUDA(series : list):
    # Convert to numpy array
    series = np.asarray(series, dtype='float32')
    #
    n = series.shape[0] #длина серии
    m = int(n / 2) #максимальное количество компонент разложения
    print ("FFT CUDA:", n, m)
    # Секундомер
    ts = TEstimate("Time execution FFT CUDA: ")
    ts.start()
    # Future result - сюда будут собираться амплитуды и фазы гармоник разложения
    amplitude = np.zeros(shape=m, dtype=series.dtype)
    phase = np.zeros(shape=m, dtype=series.dtype)
    # Run kernel with data - отсылаем данные в память GPU
    gpu_in = cuda.to_device(series)
    gpu_amp = cuda.to_device(amplitude)
    gpu_phase = cuda.to_device(phase)
    # определяем количество задействованных блоков для задачи
    numBlocks = int((m+BLOCK_SIZE-1)/BLOCK_SIZE)
    if numBlocks == 0:
        numBlocks = 1
    print ("Задача разделена на %d блоков по %d потоков" % (numBlocks, BLOCK_SIZE))
    fftKernel[numBlocks, BLOCK_SIZE](gpu_in, gpu_amp, gpu_phase, m, n)
    gpu_amp.to_host()   #копируем результат из памяти девайса GPU в переменную amplitude
    gpu_phase.to_host()  #копируем результат из памяти девайса GPU в переменную phase
    #
    ts.diff()
    return amplitude.tolist(), phase.tolist() #return as lists - переводим в обычные списки


@jit(void(float32[:], float32[:], float32[:], float32[:], int32, int32), target='gpu')
def sinapprKernel(model, amp, phase, freq, N, M):
```

```
    # Memory window to fetch - окно быстрой памяти для кэширования блока значений из amp, phase, freq
    T1 = cuda.shared.array(shape=(BLOCK_SIZE,), dtype=float32)
    T2 = cuda.shared.array(shape=(BLOCK_SIZE,), dtype=float32)
    T3 = cuda.shared.array(shape=(BLOCK_SIZE,), dtype=float32)
    #
    bx = cuda.blockIdx.x  #i-block
    tx = cuda.threadIdx.x #i-thread
    gw = cuda.gridDim.x   #num blocks
    bw = cuda.blockDim.x  #BLOCK_SIZE
    # Start
    idx = bx*bw + tx      #i - same as cuda.grid(1)
    summ = 0.0            #local var within each thread

    # Calculate
    k = 0
    for i in range(0, M, BLOCK_SIZE):
      # Copy block of data
      # скопировать следующий блок значений в быструю память, каждый поток из блока копирует в свою ячейку
      k = i + tx
      if k < M:
        T1[tx] = amp[k]
        T2[tx] = phase[k]
        T3[tx] = freq[k]
      # подождать пока все потоки дойдут до этого места
      cuda.syncthreads()
      # Calculate sum from prepared data block
      # вычислить часть взвешенной суммы для summ, используя кэшированные значения из быстрой памяти
      k = i
      for t in range(BLOCK_SIZE):
        summ += T1[t] * math.cos(T3[t] * idx + T2[t])
        k += 1
        # количество M может быть не кратным размеру блока BLOCK_SIZE
        if k >= M:
```

```
                break
            # подождать пока все потоки дойдут до этого места
            cuda.syncthreads()

        # Output results
        # количество N может быть не кратным размеру блока BLOCK_SIZE
        # лишние потоки из блока, для которых idx >= N пройдут мимо
        if idx < N:
            model[idx] = summ

    def sinapprCUDA(series : list, num):
        # получить num волн (компонент разложения)
        waves = get_wavesCUDA(series, num)
        n = len(series)
        m = len(waves)
        # Секундомер
        ts = TEstimate("Time execution sinapprCUDA: ")
        ts.start()
        # Convert to numpy array - сюда будут собираться аппроксимированные точки
        model = np.asarray(series, dtype='float32')

        # Unpack waves by each property
        # распаковываем объекты волн по свойствам, создвая 3 отдельных списка для амплитуд, фаз и частот
        ampIter = (waves[i].amp for i in range(m))
        phaseIter = (waves[i].phase for i in range(m))
        freqIter = (waves[i].freq for i in range(m))
        amplitude = np.fromiter(ampIter, count=m, dtype=model.dtype)
        phase = np.fromiter(phaseIter, count=m, dtype=model.dtype)
        freq = np.fromiter(freqIter, count=m, dtype=model.dtype)
        del waves #удалить уже ненужную переменную

        # Run kernel with data
        # копируем в память девайса GPU
        gpu_model = cuda.to_device(model)
```

```
    gpu_in1 = cuda.to_device(amplitude)
    gpu_in2 = cuda.to_device(phase)
    gpu_in3 = cuda.to_device(freq)
    # определяем количество задействованных блоков для задачи
    numBlocks = int((n+BLOCK_SIZE-1)/BLOCK_SIZE)
    if numBlocks == 0:
        numBlocks = 1
    print ("Задача разделена на %d блоков по %d потоков" % (numBlocks, BLOCK_SIZE))
    sinapprKernel[numBlocks, BLOCK_SIZE](gpu_model, gpu_in1, gpu_in2, gpu_in3, n, m)
    gpu_model.to_host() #копируем результат из памяти девайса GPU в переменную model
    ts.diff()
    return model.tolist() #возвращаем как обычный список, вместо numpy array
# извлечение волн из временного ряда
# series - временной ряд, сигнал
# num - количество волн

def get_wavesCUDA(series : list, num):
    # Получаем списки: amp, phase
    amp, phase = fftCUDA(series)
    # Построить Wave объекты, дополнив частотой
    m = len(amp)
    waves = []
    for i in range(m):
        waves.append(Wave(amp[i], 2 * math.pi * i / (m * 2), phase[i]))
    # Отсортировать и выбрать num старших гармоник
    ts = TEstimate()
    ts.start("Время на сортировку объектов волн в убывающем порядке: ")
    waves.sort()
    waves.reverse()
    ts.diff()
    waves = waves[0:num] #num самых больших частот
    print("the program found the next %s sin waves:"% len(waves))
    #print(waves)
    return waves
#################################################################
```

```
# функция используется для сравнения 2-х списков значений, когда нужно сравнить результаты, полученные обычным способом и новым
def allclose(a, b, rtol):
    n = len(a)
    if n != len(b):
        print ("No same size!")
        return False

    r = True
    ip = -1; p = 1e-8
    ipm = -1; pm = 1
    with open("cmp.log", 'w') as f:
        for i in range(n):
            f1 = float(a[i])
            f2 = float(b[i])
            if abs(f1-f2) > rtol:
                f.write(">>>")
                r = False
            k = abs(f1-f2)
            if k > p:
                ip = i
                p = k
            if k < pm:
                ipm = i
                pm = k
            f.write(str(f1) + "\t" + str(f2))
            f.write("\n")
    print ("Max on %d: %f" % (ip, p))
    print ("Min on %d: %f" % (ipm, pm))
    return r

# функция определяет если npR, cuR можно считать одинаковыми с заданной точностью rtol
def compareResults(name, npR, cuR):
    #print (name, "- can be considered the same [np]:", np.allclose(npR, cuR, rtol=0.1))
```

```
        print (name, "- can be considered the same:", allclose(npR, cuR, rtol=0.001))
        print ("npR=", np.asarray(npR))
        print ("cuR=", np.asarray(cuR))


    # функция для сравнения результатов выполнения fft обычным методом и методом через CUDA
    def testCUDA_fft():
        # Загрузить нужное количество точек - 500, 2 - количество гармоник для апроксимации (здесь не используется)
        p = PlotfFT(500, 2)
        p.load()
        y = p.x
        num = p.num
        # Проверка по старому способу
        ts = TEstimate()
        ts.start("Time Simple: ")
        amp, phase = fft(y)
        ts.diff()
        # Проверка новым способом
        ts.start("Time CUDA: ")
        cu_amp, cu_phase = fftCUDA(y)
        ts.diff()
        # Сравнить результаты
        compareResults("Amplitude", amp, cu_amp)
        compareResults("Phase", phase, cu_phase)
        print ('='*20)


    # функция для сравнения результатов выполнения аппроксимации обычным методом и методом через CUDA
    def testCUDA_approximate():
        # Загрузить нужное количество точек - 1000, 2 - количество гармоник для апроксимации (здесь не используется)
        p = PlotFFT(5000, 200)
        p.load()
        y = p.x
        num = p.num
```

```
        # Проверка по старому способу
        ts = TEstimate()
        ts.start("Time Simple: ")
        model = sin_waves_appr(y, num)
        ts.diff()
        # Проверка новым способом
        ts.start("Time CUDA: ")
        cu_model = sinapprCUDA(y, num)
        ts.diff()
        # Сравнить результаты
        compareResults("Model", model, cu_model)
        print ('='*20)

    # PlotFFT #

    class PlotFFT:
        pwd = "C:/Users/Дмитрий/Downloads/PS"  #откуда будет читаться входной файл
        # True - генерировать тестовую серию известной функции, иначе читает из файла
        test_series = False
        # сколько точек должны отрисовать на графике
        max_display_points = 500
        def __init__(self, N, num, accelerate=0):
            self.N = N
            self.num = num
            self.t = [] #первый столбец - не используется (так как интервалы времени измерения можно считать одинаковыми)
            self.x = [] #второй столбец - основной
            self.setAccelerationLevel(accelerate)

        # установка режима ускорения или без него
        def setAccelerationLevel(self, accelerate):
            self.accelerate = accelerate
            self.approximate_func = sin_waves_appr
            self.fft_func = fft
            # FFT & sin_waves_appr на GPU
```

```
    if (accelerate > 0):
        self.approximate_func = sinapprCUDA
        self.fft_func = fftCUDA


# загрузка серии значений
def load(self, fname="ferst.csv"):
    if len(self.t) > 0 or len(self.x) > 0:
        self.t = []
        self.x = []
    # True, если использовать тестовую серию
    if self.test_series:
        # Test series
        for t in range(self.N):
            self.t.append(t)
            self.x.append(2 * math.sin(0.05 * t + 0.5) + 0.5 * math.sin(0.2 * t + 1.5))
    else:
        # File series
        k = self.N
        i = 0
        fname = os.path.join(self.pwd, fname)
        with open(fname, "r") as f:
            for line in f:
                if i >= k:
                    break
                s1, s2 = line.replace(',', '.').split('\t')
                s1 = float(s1.strip())
                s2 = float(s2.strip())
                self.t.append(s1)
                self.x.append(s2)
                i += 1
        self.N = i
    return self.N


# нарисовать исходную кривую, аппроксимирванную модель, и спектр
def plot(self, fig, block):
```

```
        # Секундомер
        ts = TEstimate()
        #
        y = self.x
        num = self.num
        N = self.N
        # генерируем аппроксимированную модель
        ts.start("Время выполнения аппроксимации: ")
        model = self.approximate_func(y, num)
        ts.diff()
        # создать окно для вывода графиков
        plt.figure(fig)
        # Отрисовываем только первую часть графика, для большей детализации
        k = int(self.max_display_points)
        suby = y[:k]            #оригинальный сигнал (серия)
        submodel = model[:k]    #аппроксимированный сигнал (серия)
        # рисуем сигнал и его аппроксимированную модель на первом графике
        plt.subplot(211)
        h_signal, = plt.plot(suby, label='source timeseries')
        h_model, = plt.plot(submodel, label='model', linestyle='--')
        plt.legend(handles=[h_signal, h_model])
        plt.grid()
        # рисуем спектр на втором графике
        amp, _ = self.fft_func(y)
        xaxis = [2*math.pi*i / N for i in range(len(amp))]
        plt.subplot(212)
        h_freq, = plt.plot(xaxis[:k], amp[:k], label='spectre')
        plt.legend(handles=[h_freq])
        plt.grid()
        print ('='*20)
        #
        plt.show(block=block)
def plot():
    p = PlotFFT(2000, 500)
    p.load()
```

```
    # Без ускорения
    p.setAccelerationLevel(0)
    p.plot(1, False)
    # С ускорением на GPU
    p.setAccelerationLevel(1)
    p.plot(2, True)
# отсюда начинает выполняться программа
if __name__ == '__main__':
    #testCUDA_fft()
    #testCUDA_approximate()
    plot()
```