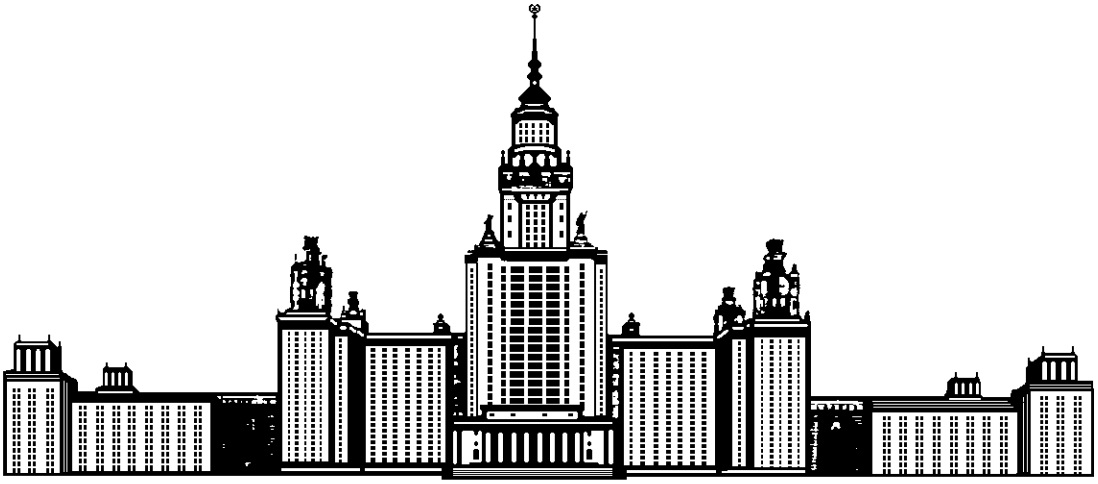
Московский государственный университет имени М. В. Ломоносова



факультет вычислительной математики и кибернетики
кафедра ВТМ

Конкурс стипендий CUDA Center of Excellence МГУ

# Научный проект "Исследование и развитие технологий автогенерации вычислительного кода на GPU в применении к задачам тензорной аппроксимации"

Кузнецов Максим Алексеевич
студент 4 курса

Москва, 2012

# 1 Введение

Для решения современных вычислительных задач необходимо использовать большие вычислительные мощности. Одним из наиболее эффективных вычислительных инструментов являются графические процессоры, однако, несмотря на развитие инструментов разработки, написание GPU-кода занимает достаточно много времени. Само по себе программирование на GPU — сложная задача, поэтому хотелось бы иметь технологии автоматического распараллеливания, однако они часто проигрывают "ручному" программированию. При этом можно выделить класс задач, которые допускают возможность автоматической генерации эффективного GPU-кода, а получающийся код можно использовать в динамических языках (Python — http://www.python.org/). До недавнего времени удобных инструментов такого рода не было, но они стали интенсивно развиваться. Мы возьмем за основу проект loopy (http://git.tiker.net/loopy.git), разработанный Андреасом Клекнером (http://mathema.tician.de/aboutme/), который позволяет генерировать OpenCL Python-модули для небольших, но трудоемких циклов.

# 2 Цель работы

**Целью работы является исследование и развитие технологий автогенерации вычислительного кода на GPU в применении к задачам тензорной аппроксимации.**Тензорные алгоритмы трудоемки и требуют большого числа вычислений, поэтому развитие параллельных версий стандартных алгоритмов играет большую роль, тем не менее не существует параллельных версий программ, реализующих эти алгоритмы. В качестве языка программирования используется язык Python, а в качестве средства автоматической генерации кода пакет loopy.

Выбор Python обусловлен тем,что Python обладает рядом достоинств перед стандартными языками (C, Fortran), в том числе: удобство разработки и написания нового кода, наличие стандартной библиотеки и многое другое. Язык Python достаточно медленный (в силу особенностей организации циклов, определения переменных), и идея использовать его для параллельных вычислений может показаться неудачной. Однако, мощь Python заключается в возможности подключения модулей, написанных на C, Fortran, а также пакетов автоматической генерации таких модулей, благодаря чему удается сохранить простоту Python и получить скорость исполнения C-кода. В частности в Python реализованы модули pyOpenCL (http://mathema.tician.de/software/pyopencl) и pyCUDA (http://mathema.tician.de/software/pycuda).

Следует отметить, что обычно автоматически сгенерированный код уступает "ручному", однако автоматическая генерация кода позволяет достичь высокой эффективности при распараллеливании циклов. Также уменьшается время разработки программы. **Основная задача — выяснить возможности ускорения программ, реализующих тензорные алгоритмы, написанных на Python с помощью средств автогенерации кода для GPU.**

# 3 Актуальность исследования

Привлекательность исследования обусловлена несколькими факторами:

1. Тензорные алгоритмы начали активно разрабатываться в последнее время
2. Написание GPU-кода — сложная задача, существует необходимость исследовать возможности автогенерации GPU-кода
3. Вычислительная мощность GPU превосходит многоядерные CPU, использование GPU эффективней

Ввиду того, что процесс написания GPU-кода вручную длительный и трудоемкий, хоть и эффективный, в вычислительных задачах хотелось бы использовать следующий "идеальный" способ его написания:

1. Использование в динамических языках (Python)
2. Автоматическое распараллеливание стандартных задач (циклов), генерация OpenCL/CUDA-кода
3. Высокая эффективность

## 4 Текущее состояние исследований

Пакет loopy очень свежий, однако уже сейчас обладает множеством возможностей. Loopy открытый проект, разрабатывающийся Андреасом Клекнером, активно добавляются недостающие функции. Пакет позволяет генерировать GPU-код из заданного Python-кода, а результат такой генерации возвращает в виде функции, имя которой определяется пользователем, а последующие вызовы можно осуществить, используя это имя, минуя процесс генерации кода. Приведем пример Python-кода с использованием пакета loopy:

```
def test_image_matrix_mul(ctx_factory):
    dtype = np.float32
    ctx = ctx_factory()
    order = "C"

    n = get_suitable_size(ctx)
# Создается вычислительное ядро GPU-кода
    knl = lp.make_kernel(ctx.devices[0],
            "{[i,j,k]: 0<=i,j,k<%d}" % n,
            [
                "c[i, j] = sum(k, a[i, k]*b[k, j])"
                ],
            [
                lp.ImageArg("a", dtype, shape=(n, n)),
                lp.ImageArg("b", dtype, shape=(n, n)),
                lp.GlobalArg("c", dtype, shape=(n, n), order=order),
                ],
            name="matmul") # Имя функции, заданное пользователем
 для дальнейшего использования

    seq_knl = knl
#Создается разбиение циклов
    knl = lp.split_iname(knl, "i", 16, outer_tag="g.0", inner_tag="l.1")
    knl = lp.split_iname(knl, "j", 16, outer_tag="g.1", inner_tag="l.0")
    knl = lp.split_iname(knl, "k", 32)
    # conflict-free
    knl = lp.add_prefetch(knl, 'a', ["i_inner", "k_inner"])
    knl = lp.add_prefetch(knl, 'b', ["j_inner", "k_inner"])
#Окончательное формирование кода
    kernel_gen = lp.generate_loop_schedules(knl)
    kernel_gen = lp.check_kernels(kernel_gen, dict(n=n))
```

Данный код реализует перемножение двух матриц размера $n \times n$. Сгенерированный loopy GPU-код достаточно велик (более 100 строк для этого примера), и писать его аналог вручную достаточно долго, в то время как длительность работы loopy измеряется в секундах. При этом код весьма эффективен и легко используется в Python.

Основная проблема — определение граничных условий в циклах.

В ходе исследования планируется использовать имеющееся оборудование (видеокарта NVIDIA для персонального компьютера, кластер GPU, установленный в ИВМ РАН)

# 5 Модельная задача

В качестве примера алгоритма аппроксимации тензора будем рассматривать алгоритм построения канонического разложения. Необходимо ввести следующие определения:

*Определение*

Тензором A размерности $d$ назовем многомерный массив, элементы которого $\mathrm{A}(\mathrm{i}_1,\mathrm{i}_2,\ldots,\mathrm{i}_d)$ имеют $d$ индексов. $1 \le i_k \le n_k$; $n_k$ называются модовыми размерами (размерами мод)

*Определение*

Каноническим разложением многомерного массива (*тензора*) называется представление вида

$$A(i_1, i_2, \ldots, i_d) = \sum_{\alpha=1}^{r} U_1(i_1, \alpha) U_2(i_2, \alpha) \ldots U_d(i_d, \alpha), \tag{1}$$

где $\mathrm{U}_k$ называются *факторами* канонического разложения, а $r$ — каноническим рангом.

Уравнение (1) является основным. Подробнее о тензорах и их разложениях можно узнать в обзоре [3]

## 5.1 Алгоритм ALS

Пусть задан тензор $A$ с элементами $A_{i_1 \ldots i_d}$. Задача состоит в том, чтобы найти его каноническое приближение, а именно найти такие матрицы $U_1, \ldots, U_d$

$$A_{i_1,\ldots,i_d} \approx \sum_{\alpha=1}^{r} U_1(i_1, \alpha) U_2(i_2, \alpha) \ldots U_d(i_d, \alpha). \tag{2}$$

Математическая постановка задачи состоит в том, чтобы решить задачу (2) в смысле наименьших квадратов

$$\sum_{i_1,\ldots,i_d} \left( A(i_1, \ldots, i_d) - \sum_{\alpha=1}^{r} U_1(i_1, \alpha) U_2(i_2, \alpha) \ldots U_d(i_d, \alpha) \right)^2 \longrightarrow \min. \tag{3}$$

Будем решать вариационную задачу поиска аппроксимации тензора с помощью алгоритма ALS (Alternating Least Squares), подробное изложение которого можно найти в статье [1]. Основная идея алгоритма, состоит в том, чтобы фиксировать все факторы канонического разложения, кроме одного, и искать минимум функционала только по нему. Путем циклических перестановок, используя уже полученные факторы, строятся последующие, до тех пор, пока не будет достигнута требуемая точность аппроксимации или, пока не сработают другие критерии остановки алгоритма (превышение максимального количества итераций, превышение времени выполнения программы). Подробный обзорр алгоритма можно найти в статье [1].

### 5.1.1 Оценка сложности алгоритма ALS и возможности его параллельной реализации

Предположим, что заданный тензор A имеет размеры мод $n$ и ранг $r$.

Простейшая программа для вычисления каждого фактора $\mathrm{U}_{\mathrm{i}\alpha}$ может быть написана с помощью помощью вложенных циклов. Тогда сложности вычисления правой и левой частей системы соответственно:

1. Сложность вычисления левой части системы для одной матрицы U пропорциональна $O(nr^2)$;

2. Сложность вычисления правой части $O(n^3 r)$;

Что уже при $n = 512$ требует большого количества времени для вычисления.Сравнительную характеристику алгоритма ALS можно найти в статье [2]

Можно сформулировать **основную задачу программирования:**

1. **Выделить наиболее трудоемкий цикл**

2. **Распараллелить его, используя пакет loopy**

## 6 План исследований

План исследований состоит из следующих пунктов:

1. Анализ эффективности loopy на стандартных примерах, сравнение различных аппаратных платформ

2. Реализация модельных примеров (матрично-матричное перемножение)

3. Реализация модельной задачи

4. Сравнение быстродействие программ, написанных на Python+loopy и только на Python, анализ результатов

## 7 Заключение

Средствами автоматической генерации кода удобно воспользоваться для распараллеливания тензорных алгоритмов с использованием вычислительного потенциала графических процессоров. Ускорение работы программы ожидается существенным, как в силу высокой производительности GPU, так и благодаря структуре самого алгоритма. Тензорные алгоритмы широко востребованы, а создание их эффективной и быстрой реализации является одной из приоритетных задач, в то время как возможности автоматической генерации кода на GPU позволяют создать такую реализацию быстро.

## Список литературы

[1] J.D. Carroll and J.J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[2] N.K.M. Faber, R. Bro, and P.K. Hopke. Recent developments in candecomp/parafac algorithms: a critical review. *Chemometrics and Intelligent Laboratory Systems*, 65(1):119–137, 2003.

[3] T.G. Kolda and B.W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455, 2009.