

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://mathematician.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://mathematician.de/software/pyopencl>) и pyCUDA (<http://mathematician.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, разрабатываемый 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.

## 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  $A(i_1, i_2, \dots, i_d)$  имеют  $d$  индексов.  $1 \leq i_k \leq n_k$ ;  $n_k$  называются модовыми размерами (размерами мод)

*Определение*

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

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

где  $U_k$  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  $A_{i_1 \dots i_d}$ . Задача состоит в том, чтобы найти его каноническое приближение, а именно найти такие матрицы  $U_1, \dots, U_d$

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

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

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

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).

#### 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  $U_{i\alpha}$  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  $O(n^3r)$ ;

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 быстродействия программ, написанных на Python+loopy и только на Python, анализ результатов

## 7 О пакете Loopy

### 7.1 Установка

Пакет loopy в настоящее время имеет несколько зависимостей. Эти пакеты нужно установить перед началом установки loopy:

- gmpy <https://code.google.com/p/gmpy/>
- pyopencl <http://github.com/inducer/pyopencl>
- pymbolic <http://github.com/inducer/pymbolic>
- islpy <http://github.com/inducer/islpy>
- cgen <http://github.com/inducer/cgen>

Практически все из них можно скачать воспользовавшись помощью git. После установки вышеперечисленных пакетов можно переходить к установке самого пакета loopy. В настоящее время Андреас Клекнер (разработчик пакета) перенес актуальную версию в закрытый репозиторий, однако стабильную версию можно найти здесь: <http://git.tiker.net/loopy.git>. Как только все пакеты будут установлены на компьютер, можно приступать к работе с loopy.

### 7.2 Предназначение и синтаксис loopy

Пакет loopy предназначен для автоматической генерации OpenCL кода, который можно использовать на GPU. Для использования приема автоматической генерации кода (с помощью loopy) алгоритм изначально должен быть приведен к алгоритму со вложенными циклами (последовательности вложенных циклов). Основная задача данного модуля “разворачивать” вложенные циклы, причем пакет в состоянии преобразовать циклы различной степени вложенности. В процессе работы loopy генерирует вычислительное ядро, которое впоследствии и нужно запускать на GPU. Приведем пример ядра, в котором используются основные функции пакета:

```
def LU_solver(ctx):
    order='C'
    dtype = np.float32
    knl = lp.make_kernel(ctx.devices[0],
    [
```

```

    "{[l,k,i,j,m]: 0<=l<r and 0<=k<n-1 and k+1<=i<n and 0<=j<n-1 and 0<=m<n-1-j}",

],
[
    "bcopy[i,l] = bcopy[i,l]-bcopy[k,l]*LU[i,k] {id=lab1}",
    "bcopy[n-1-j,l]=bcopy[n-j-1,l]/LU[n-j-1,n-1-j] {id=l2, dep=lab1}",
    "bcopy[m,l]= bcopy[m,l]-bcopy[n-j-1,l]*LU[m,n-1-j] {id=l3, dep =l2}",
    "bcopy[0,l]=bcopy[0,l]/LU[0,0]{id=l4, dep=l2}",
],
[
    lp.GlobalArg("LU", dtype, shape = "n, n" , order=order),
    lp.GlobalArg("bcopy", dtype, shape = "n, r" , order=order),
    lp.ValueArg("n", np.int64),
    lp.ValueArg("r", np.int64),
],
assumptions="n>=1")
knl = lp.split_iname(knl, "k", 1)
knl = lp.split_iname(knl, "i", 32)
knl = lp.split_iname(knl, "j", 32)
knl = lp.split_iname(knl, "l", 32, outer_tag="g.0", inner_tag="l.0")

print knl
print lp.CompiledKernel(ctx, knl).get_highlighted_code()
return knl

```

Данный код реализует решение системы, поданной в стандартном виде LU-разложения, алгоритм подается в специальном виде с использованием синтаксиса loo.py.

### 7.3 Входной параметр ядра

На вход функции реализующей ядро подается контекст выполнения программы. Стандартным является следующий способ получения контекста:

```

plt = cl.get_platforms()
nvidia_plat = plt[1]
ctx = cl.Context(nvidia_plat.get_devices())

```

После выполнения кода в переменную ctx будет подан контекст, соответствующий графической плате (в данном случае NVIDIA)

### 7.4 Внутренние элементы ядра

Генерацией ядра в переменную knl занимается функция make<sub>kernel</sub> на вход которой подается:

- Домен, иными словами имена переменных-счетчиков цикла и границы изменения переменных в виде строки. Loo.py поддерживает циклы с заранее неизвестными граничными условиями,

переменными условиями

```

"{[l,k,i,j,m]: 0<=l<r and 0<=k<n-1 and k+1<=i<n and 0<=j<n-1 and 0<=m<n-1-j}",

```

В примере переменными цикла являются  $l, k, i, j, m$ , где  $l \in [0, r)$  причем  $r$  до этого нигде не объявлена и будет определена в процессе выполнения из входных параметров.  $k$  являющаяся переменной цикла, объемлющего вложенный в него цикл по  $i$  определена в переменных пределах. Таким образом можно сконструировать широкий класс алгоритмов, допускающих подобную реализацию.

- Инструкции для исполнения (не менее одной), каждой из которой можно присвоить метку с помощью переменной *id* и зависимости *dep*. Инструкция *id = lab1* зависит от инструкции *id = lab2*, если она должна быть выполнена после инструкции *lab2*.

Инструкции из примера:

```
"bcopy[i,1] = bcopy[i,1]-bcopy[k,1]*LU[i,k] {id=lab1}",
"bcopy[n-1-j,1]=bcopy[n-j-1,1]/LU[n-j-1,n-1-j] {id=l2, dep=lab1}",
```

- Аргументов, в которые входят входные параметры, константы, выходные параметры.

Каждый параметр должен иметь тип, размер (возможно указание в символьном “неявном” виде, так и в явном численном или в виде переменной (которая должна быть до этого определена)) Пример аргумента:

```
lp.GlobalArg("LU", dtype, shape = "n, n" , order=order),
```

- Дополнительных параметров, как допущение, приблизительная размерность или величина.

Примеры можно найти в директории *test* пакета *loo.py*

## 7.5 Задание разбиения вычислительной сетки

После того как ядро написано, необходимым является указать каким образом нужно разбить вычислительную сетку для этого ядра (как разбить циклы). Этим занимается функция “*split\_iname*”:

```
kn1 = lp.split_iname(kn1, "l", 32, outer_tag="g.0", inner_tag="l.0")
```

Первый параметр — ядро, переменные которого нужно разбить. Следующий — имя переменной счетчика, далее указывается размер по сколько нужно разбить цикл (обычно 16 или 32, является рекомендуемым разбиением, однако возможно и любое другое). В конце указываются опциональные параметры внешних и внутренних рабочих групп.

### 7.5.1 О выборе параметров разбиения

К сожалению невозможно придумать универсальный алгоритм, по которому следует выбирать разбиение. Однако несмотря на это, очень сильно качество распараллеливания зависит от выбора “*outer\_tag*” и “*inner\_tag*”. Есть несколько базовых правил, как например “всегда выбирать для оси 0 разбиение 1”, однако попытка подобрать “идеальные” параметры ведет к чрезмерному усложнению операций с памятью, взаимодействию между частями, что не позволяет создать устойчивую и надежную автоматическую реализацию. Для пользователя пакета *loo.py* это значит, что лучше воспользоваться стандартными разбиениями и попробовать на основе затрачиваемого на выполнение времени подобрать наиболее выгодные параметры разбиения. Напрямую с разбиением связан и доступ к памяти. В *loo.py* есть специальная функция `addprefetch(kn1, “a”, [“iinner”, “jinner”], fetchboundingbox=True)`, однако Андреас Клекнер сейчас работает над ее усовершенствованием, использовать ее пока сложно (все движется к автоматизации распределения памяти, однако пока код еще не готов)

## 8 О вызове ядра

### 8.1 Расположение массивов

После того, как написано ядро, расставлено разбиение это ядро можно начинать использовать. Однако перед этим необходимо выполнить некоторые приготовления. Как было сказано выше, необходимо определить контекст. После определения контекста, желательно (в силу серьезной эконогмии времени) все параметры (массивы, тензоры) поместить на устройство. Для этого нужно выполнить серию команд.

## 1. Создать очередь

```
queue = cl.CommandQueue(ctx, properties=cl.command_queue_properties.PROFILING_ENABLE)
```

1. Специальной командой `cl.array_ to device(queue, variable)` послать объект `variable` на устройство

```
u2=cl.array.to_device(queue,u)
```

Привести `u` к обычному массиву (`numpy.array`) можно с помощью метода `get()`

```
numpy_array_u2 = u2.get()
```

### Важно чтобы все массивы имели определенный явно тип

Вызов ядра напоминает вызов функции или процедуры. Однако перед самым вызовом нужно выполнить несколько команд

- Создать очередь “queue”. **Очередь в программе должна быть единственной!**
- Создать словарь параметров “parameters”. При этом выходные параметры могут подаваться в словаре или нет.
- Скомпилировать ядро. Ядро может быть скомпилировано единожды и запомнено в специальной переменной для дальнейшего использования.
- Вызвать скомпилированное ядро с параметрами очереди “queue” и “parameters”

Приведем пример вызова ядра:

```
cknl_r_U = lp.CompiledKernel(ctx, knl_r_U)
parameters={"a":a2,"v":v2,"w":w2,"n":n,"r":r,"f":prav}
evt=cknl_r_U(queue, **parameters)[0]
#evt,(f)=cknl_r_U(queue, **parameters) этот способ с пересылкой и поэтому не очень хорош
evt.wait()
```

## 9 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.