В. А. Антонюк

GPU+Python

Параллельные вычисления в рамках языка Python

Москва
Физический факультет МГУ им. М. В. Ломоносова
2018

Антонюк Валерий Алексеевич

GPU+Python. Параллельные вычисления в рамках языка Python. –
М. : Физический факультет МГУ им. М. В. Ломоносова, 2018. – 48 с.

Пособие написано по материалам специального курса «Применение языка Python для параллельных вычислений», посвящённого написанию программ, из которых задействуется параллельная архитектура GPU с целью ускорения вычислений. Этот спецкурс впервые прочитан студентам отделения прикладной математики (ОПМ) физического факультета весной 2017 года (4-й семестр магистратуры).

Следует отметить, что в последние годы язык программирования Python — из вроде бы ещё одного из многих созданных универсальных языков — как-то незаметно стал чуть ли не самым распространённым. Этому способствуют и богатая «экосистема» (многочисленные созданные сообществом прикладные библиотеки из самых разных областей), и не слишком высокий пороговый «уровень входа» в программирование на этом языке.

В пособии излагаются: краткое введение в язык Python, его основные возможности, а также особенности языка, которые упрощают доступ к сторонним библиотекам, причём синтаксически элегантно. Рассматриваются способы доступа к вычислительным возможностям GPU через реализацию CUDA (пакеты PyCUDA, Theano, Numba), через реализацию OpenCL (пакеты PyOpenCL, PyViennaCL), посредством GLSL (написание т. н. шейдеров) либо с помощью различных библиотек-«обёрток» (ArrayFire Python Wrapper, gnumpy, cudamat, PyGPU).

Пособие рассчитано на студентов и аспирантов физического факультета.

Автор — сотрудник кафедры математического моделирования и информатики физического факультета МГУ.

Рецензенты: к. ф.-м. н. Д. А. Бикулов, м. н. с. В. В. Шишаков.

Подписано в печать 28.06.2018. Объем 3 п.л. Тираж 30 экз. Заказ № 102. Физический факультет им. М. В. Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д. 1, стр. 2.
Отпечатано в отделе оперативной печати физического факультета МГУ.

Оглавление

Предисловие

Для тех, кто знаком с программированием *GPU* в каком-либо виде (с помощью *CUDA*, *OpenCL* и т. п.) достаточно очевидно, что процесс такого программирования сравнительно сложен, тем более, если это делается ещё и на языке *C++*, который постепенно становится синтаксически всё более изощрённым.

Средства взаимодействия с *GPU* всегда сосредоточены в каких-то внешних разделяемых динамических библиотеках, поэтому тот язык, из которого хотелось бы воспользоваться возможностями *GPU*, должен позволять обращаться к таким библиотекам, передавать им (и принимать обратно) все необходимые данные. Кроме того, было бы удобно, если бы реализация языка хотя бы частично была интерпретирующей, поскольку в этом случае никакие «подготовительные» действия (вроде компиляции) не нужны, и можно «общаться» с *GPU*, используя не только полные программы, но и отдельные программные фрагменты.

Это позволяет прямо в среде интерпретатора буквально несколькими строками делать то, для чего в иных случаях надо было бы писать небольшую программу, например, опрос имеющихся *OpenCL*-реализаций с доступными в них устройствами с их возможностями:

```
>>> import pyopencl
>>> pyopencl.VERSION
(2015, 1)
>>> pyopencl.get_platforms()[0]
<pyopencl.Platform 'AMD Accelerated Parallel Processing' at 0x7f674630fa18>
>>> pyopencl.get_platforms()[0].get_devices()
[<pyopencl.Device 'Bonaire' on 'AMD Accelerated Parallel Processing' at 0x11d3a00>,
 <pyopencl.Device 'Intel(R) Core(TM) i3-4160 CPU @ 3.60GHz'
 on 'AMD Accelerated Parallel Processing' at 0x13d21f0>]
>>> pyopencl.get_platforms()[0].get_devices()[0].extensions
'cl_khr_fp64 cl_amd_fp64 cl_khr_global_int32_base_atomics
 cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics
 cl_khr_local_int32_extended_atomics cl_khr_int64_base_atomics
 cl_khr_int64_extended_atomics cl_khr_3d_image_writes cl_khr_byte_addressable_store
 cl_khr_gl_sharing cl_khr_gl_depth_images cl_ext_atomic_counters_32
 cl_amd_device_attribute_query cl_amd_vec3 cl_amd_printf cl_amd_media_ops
 cl_amd_media_ops2 cl_amd_popcnt cl_khr_image2d_from_buffer cl_khr_spir
 cl_khr_subgroups cl_khr_gl_event cl_khr_depth_images cl_khr_mipmap_image
 cl_khr_mipmap_image_writes '
>>> pyopencl.get_platforms()[0].get_devices()[0].version
'OpenCL 2.0 AMD-APP (1912.5)'
```

или выбор конкретного устройства для исполнения программы[1]. Сама программа (точнее, её хост-часть) также может быть существенно проще — за счёт «грамотного» оформления «надстройки» над необходимыми для работы библиотеками. И то, что она будет интерпретироваться, не очень повлияет на общую скорость исполнения, так как основные вычислительные действия всё равно будут производиться на *GPU*, а вот процесс разработки такой подход может ощутимо ускорить.

В данном пособии рассматривается использование для этой цели языка *Python* (хотя он и не является единственным языком, обеспечивающим возможность непосредственного «подключения» к библиотекам во время работы программы), — и не в последнюю очередь из-за растущей популярности языка и его синтаксической простоты.

[1] Опыт работы с реализациями *OpenCL* показал, что многие демонстрационные (да, впрочем, и любые, создаваемые нами) программы на *C/C++* почти наверняка потребуют модификации для возможности их запуска в другой системной конфигурации, где имеются другие реализации или же несколько доступных реализаций *OpenCL*. Оптимальной здесь будет возможность *указать* нужное устройство для исполнения, причём такой возможностью в идеале должна обладать *каждая* программа.

Кстати, в *PyOpenCL* эта проблема решена, причём несколькими способами (см. стр. 26); один из них — вызов в программе функции создания контекста `create_some_context()`, позволяющей пользователю уже запущенной программы самому выбрать необходимые платформу и устройство *OpenCL*.

1. Введение

Вероятно, имеет смысл чуть подробнее объяснить, почему «пропагандируется» именно язык *Python* в качестве средства написания программ с использованием *GPU*. Аргументы об относительной простоте языка для пользователя и его популярности, разумеется, весьма важны, однако только их явно недостаточно.

Итак, почему *Python*?

Модуль `ctypes` для интеграции кода *Python* и динамических библиотек (написанных чаще всего на *C/C++*) присутствует в стандартной поставке *Python* и никаких дополнительных средств для обращения к ним не нужно. Правда, при этом динамическая типизация в языке *Python* вступает в противоречие со статической типизацией библиотечного кода. Поэтому все типы в *Python*-программе должны быть «завёрнуты» в соответствующие типы `ctypes`, так что могут быть преобразованы в требуемые типы данных *C*.

Кроме того, как оказывается (см. далее *Декораторы*), язык *Python* позволяет элегантно оформить «подмену» одних своих функций другими (в т. ч. реализованными на других языках и исполняемыми вообще за рамками *Python*-программы).

Как это реализуется?

Поучительно проанализировать для этого `pycublas.py` (автор — ***Derek Anderson***) — файл[2], дающий доступ из языка *Python* к функциям библиотеки *cuBLAS* из состава ***CUDA*** (в системе *Windows* библиотека называется `cublas.dll`, в других системах — `libcublas.so` или `libcublas.dylib`). На этом сравнительно небольшом примере можно видеть, как реально осуществляется «интеграция» внешних библиотек в приложение *Python*.

Работа с декораторами на этом фоне, конечно, выглядит сложнее, тем более, если функции, сформулированные на языке *Python*, должны быть преобразованы в иной код. Тем не менее, здесь тоже можно изучить вполне обозримые (по объёму кода) примеры (см., скажем, ***ocl***)).

Как представлена информация в пособии

Хотя указанные примеры должны убедить читателя в способности почти всё осуществить самостоятельно, далее всё излагается более отстранённо — не с точки зрения разработчика, а, скорее, с точки зрения пользователя уже созданных программных пакетов.

Материал для каждого из рассматриваемых пакетов выстроен примерно по следующему плану: что это такое, для чего нужно, как устанавливается (или что необходимо для работы), как конфигурируется (если к пакету это применимо); приводятся простейшие тесты для проверки функционирования и правильности установки (скажем, получение версии и т. п.), кратко объясняется, как пользоваться пакетом. Могут быть даны какие-то демонстрации возможностей или тестовые программки из описаний пакетов, приведены ссылки на статьи, руководства и аналогичные ресурсы.

Подробнее о `ctypes`

`ctypes` – A foreign function library for Python
https://docs.python.org/2/library/ctypes.html

[2] Располагался по адресу http://kered.org/blog/2009-04-13/easy-python-numpy-cuda-cublas/, но в настоящее время уже недоступен; однако, его всё ещё можно найти кое-где в Сети по имени (`pycublas.py`).

2. Язык *Python* — краткий обзор

> Говоря «казённым» языком, ***Python*** является высокоуровневым языком программирования общего назначения и ориентирован на улучшение как производительности разработчика, так и читаемости кода. Он поддерживает несколько парадигм программирования: структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Его основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, удобные высокоуровневые структуры данных (кортеж, список, словарь, множество и др).

Поскольку *Python* был создан сравнительно поздно (1991), автор языка Гвидо ван Россум (***Guido van Rossum***) смог удачно аккумулировать в нём «находки» из других языков программирования: пакеты и модули, структуры данных высокого уровня, какие-то черты функционального программирования (`lambda`, `map`, `reduce`, `filter` и др.), комплексную арифметику, списочные выражения, генераторы, срезы массивов, именованные аргументы функций, использование `@` для декораторов, байт-компиляцию исходного кода — и всё это при явной синтаксической простоте...

Это привело к появлению языка программирования, который за более чем 25 лет своего существования успел понравиться, обрасти массой накопленного кода — и в результате стать одним из самых распространённых языков программирования.

> Поскольку к использованию языка *Python* мы часто приходим, уже имея опыт общения с другим языком программирования (нередко это *C/C++*), нам важно, на что из предшествующего опыта можно опереться, а на что — из-за того, что это будет непривычным для нас, — следует обратить пристальное внимание. Исходя из этого, в дальнейшем изложении проводится подобное неявное сравнение и акцентируется внимание на различиях.

Состав, синтаксис и приоритеты выполнения операций в языке *Python* будут привычны для всех, кто знаком с каким-нибудь одним из распространённых языков программирования, и призваны минимизировать употребление круглых скобок в выражениях.

Синтаксическая особенность языка — выделение блоков кода с помощью *отступов*, поэтому в нём отсутствуют разделители для этой цели (вроде операторных скобок `begin`/`end` языка *Pascal* или фигурных языка *C*). Это позволяет немного сократить в программах количество строк и символов, но вынуждает форматировать код «правильным» образом. Особенные неприятности это может доставить тем, кто только начинает осваивать язык *Python*, так как программа просто не будет запущена, если выравнивание не соблюдено или выполнено разными пробельными символами.

Исходный код транслируется в некоторое промежуточное представление, не являющееся машинным кодом какого-либо конкретного процессора, а потому легко переносимое на другие архитектуры, — в т. н. ***байт-код***, который затем интерпретируется (исполняется) ***виртуальной машиной***.

Наиболее распространённой (можно даже сказать — эталонной) реализацией языка *Python* является т. н. ***CPython***: написанный на языке *C* интерпретатор байт-кода. Кроме *CPython* имеются и другие реализации *Python* (**IronPython**, **PyPy**, **Jython**, **Stackless Python**).

Несмотря на то, что самой последней версией *Python* является версия *3.6*, и по сей день наравне с ней в ходу более старая версия *2.7*. Связано это с тем, что *Python 3* привнёс в язык некоторые изменения, несовместимые с уже существующим кодом на *Python 2*, что замедлило ожидавшийся переход с версии *2* (из-за накопленной массы кода) на версию *3*; поэтому в данный момент активно используются обе версии.

Код *Python* выполняется в определенной среде, включающей интерпретатор, стандартную библиотеку и некоторое количество установленных пакетов, что определяет и допустимые конструкции языка и синтаксис, и доступные возможности операционной системы, и то, какие пакеты можно использовать.

2.1. Виртуальные окружения (виртуальные среды)

Важно также понимать, что в случае языка *Python* можно не просто пользоваться различными его версиями, установленными «параллельно», но и выстроить целую систему независимых друг от друга «окружений» (***environment***) для изолированного исполнения программ — так называемых виртуальных сред или виртуальных окружений.

При этом пакеты виртуальной среды будут изолированы как от глобальной, так и от всех других виртуальных сред, поскольку каждое виртуальное окружение состоит из «своего» интерпретатора *Python*, «своей» библиотеки и «своего» набора пакетов. Такая изоляция позволяет устанавливать в виртуальных окружениях необходимые конкретные версии пакетов (скажем, для тестирования), но избегать при этом возможных конфликтов между несовместимыми пакетами или их различными версиями.

Создавать многочисленные виртуальные окружения — с различными версиями как самого *Python*, так и наборами пакетов — позволяет пакет `virtualenv`. Для того, чтобы начать работу в каком-либо конкретном виртуальном окружении, его необходимо сначала активировать, а по завершении работы — деактивировать.

> Установка (для пользователя без административных прав): `pip install --user virtualenv`[3]. Проверка версии возможна как с помощью самой программы (`virtualenv --version`), так и с помощью `pip` (`pip show virtualenv`). После установки `virtualenv` пакеты можно будет устанавливать как глобально, используя административные права (например, в *Linux*: `sudo pip install <ИмяПакета>`), так и в нужном окружении — как обыкновенный пользователь.

В виртуальных окружениях может использоваться другая версия *Python* и/или свой собственный набор модулей и приложений. Это удобно для тестирования новых версий модулей, поскольку не «засоряет» основную версию и (потенциально) не конфликтует с его набором модулей. Можно также создать виртуальное окружение специально для какого-то проекта с определённым набором модулей и далее не обновлять их (чтобы не нарушить работоспособность), в то время как основные модули могут обновляться.

> Для работы с виртуальными окружениями первоначально понадобится создать каталог, где будут находиться подкаталоги будущих виртуальных окружений. Лучше всего, если подобный каталог будет располагаться в «домашнем» пространстве пользователя; это позволит избежать проблем с правами доступа. Кроме того, удобно сделать его «скрытым» (для чего в *Linux* его имя должно начинаться с точки), поскольку едва ли понадобится что-то делать в нём вручную. Таким образом, создание нового (первого) окружения может выглядеть так:
>
> ```
> mkdir ~/.<ИмяКаталогаОкружений>/
> cd ~/.<ИмяКаталогаОкружений>/
> virtualenv --no-site-packages <ИмяОкружения>
> ```
>
> Если в системе установлено несколько версий *Python*, можно указать ту, что надо использовать
>
> ```
> virtualenv --no-site-packages -p python2.7 <ИмяОкружения>
> ```
>
> Наполнение окружения необходимыми модулями можно делать как без его активации:
>
> ```
> pip install -E <ИмяОкружения>/ <ИмяПакета>
> ```
>
> так и с активацией:
>
> ```
> source <ИмяОкружения>/bin/activate
> pip install <ИмяПакета>
> ```

[3] Менеджер *Python*-пакетов `pip` может быть не установлен в системе по умолчанию или иметь слишком старую версию. В этом случае самое правильное *Linux*-решение — `sudo apt-get install python-virtualenv` (для случая систем *Ubuntu* или *Debian*); при установке `virtualenv` автоматически установится и `pip`, который можно использовать для установки других пакетов в рамках конкретного окружения.

Покинуть активированное окружение можно командой `deactivate`.

Если эти манипуляции кажутся сложными, можно установить пакет `virtualenvwrapper` (для *Linux* — `sudo easy_install virtualenvwrapper`).

2.2. Записные книжки или блокноты (*notebooks*)

Ещё одним интересным инструментом, возникшим первоначально в рамках интерактивной оболочки для языка *Python* под названием ***IPython*** (улучшенного варианта программы ***IDLE*** — стандартной оболочки *Python*), является т. н. «блокнот» — своеобразная «рабочая тетрадь» программиста или исследователя.

IPython Notebook — интерактивное окружение в рамках обычного браузера, сочетающее в себе отображение кода, форматированного текста, изображений, видео и анимации, математических формул, графиков и иллюстраций в пределах одного документа, что позволяет эффективно сохранять результаты работы и/или распространять их.

«Блокноты» сохраняются как структурированные текстовые файлы (в т. н. формате *JSON*), из-за чего обмен ими чрезвычайно прост. Кроме того, в *IPython* имеется специальная утилита (`nbconvert`) для преобразования в другие популярные форматы (например, *PDF* и *HTML*). Ещё одна утилита (`nbviewer`) позволяет отображать содержимое любого «блокнота» прямо в браузере.

> Файл с расширением `.ipynb` (***IPython notebook***) — типичный `.json`-файл (т. е., он содержит описание/инициализацию некоторой структуры данных в синтаксисе языка *JavaScript*), в нём имеется хэш (словарь) лишь с одним ключом `cells`, а его значение — массив из отдельных ячеек (*cells*). Каждая ячейка — тоже хэш с обязательными ключами `cell_type` и `source`, а также некоторыми другими (например, `metadata`, `outputs` и т. д.).
>
> Как легко понять из названия, каждой ячейке приписан некий тип (`cell_type`) и сама она имеет какое-то содержимое (`source`). Типов, как правило, всего два: `markdown` и `code` — и они чередуются, представляя читающему возможность видеть описание программного кода вместе с результатами его исполнения. Учитывая то, что «блокноты» могут содержать также математические формулы и графические иллюстрации, они идеально подходят для хранения результатов экпериментирования в частности и документирования работы вообще.

В дальнейшем *IPython Notebook* эволюционировал в отдельную программу ***Jupyter*** (название образовано из имён трёх языков: `Ju(lia) + Pyt(hon) + (e)R`). Поддержка различных языков программирования в ней осуществляется с помощью модульных ядер (*kernels*); таковые существуют сейчас (помимо языков *Julia*, *Python* и *R*) для *JavaScript*, *Java*, *Scala*, *Go*, *C#*, *Ruby*, *Haskell* и др. — всего порядка полусотни языков программирования.

Некоторые ссылки

The Python Standard Library
https://docs.python.org/2/library/

Style Guide for Python Code
https://www.python.org/dev/peps/pep-0008/

Python Virtual Environments
https://www.python.org/dev/peps/pep-0405/

IPython: A System for Interactive Scientific Computing
Computing in Science and Engineering. — 2007. — Vol. 9, № 3. — P. 21-29.
https://dx.doi.org/10.1109/MCSE.2007.53

What is Jupyter?
https://www.oreilly.com/ideas/what-is-jupyter

3. Немного об интроспекции

Под интроспекцией (***type introspection***) в программировании понимается возможность в некоторых языках запросить во время исполнения программы сведения о типе и свойствах какого-либо программного объекта.

Например, в *Python* достаточно легко узнать, что доступно в рамках загруженного модуля[4]: для этого можно воспользоваться функцией `dir()`, передавая ей имя модуля в качестве параметра. Функция работает со всеми типами объектов, включая целые числа, строки, кортежи, списки, словари, функции, экземпляры и методы классов и классы, определенные пользователем. Если `dir()` вызывать без параметров, она возвратит имена из текущей области видимости, включая имена импортированных ранее модулей. Она — пример т. н. встроенной функции, которой можно пользоваться, не импортируя никаких модулей.

> Если необходимо узнать, что ещё «встроено» в интерпретатор *Python* помимо функции `dir()`, достаточно ввести ему команду `dir(__builtins__)`. В результате будут выведены атрибуты этого встроенного модуля, объекты ошибок и встроенные функции.
>
> Разумеется, в *Python* имеется интерактивная справочная утилита, вызываемая с помощью функции `help()`; после вызова приветствие сменится на `help>` и, вводя имя модуля, ключевого слова или темы, можно получить по ним справку; для получения списка доступных модулей, ключевых слов или же тем надо ввести строку `modules`, `keywords` или `topics` соответственно. Выход из утилиты осуществляется посредством команды `quit`.
>
> При получении списка доступных модулей (вводом `modules` в рамках справочной утилиты или `help('modules')` в интерпретаторе) информация, по-видимому, собирается путём импортирования всех находимых модулей, поэтому результат (особенно в системе с большим числом уже установленных пользователем модулей) способен удивить даже опытного *Python*-программиста своими «побочными эффектами»...

Почти всё в *Python* (кроме ключевых слов и специальных символов) является объектом и для каждого объекта можно узнать его тип (`type(<Объект>)`), имя (атрибут `__name__`), уникальный идентификатор (`id(<Объект>)`) и все атрибуты объекта (`dir(<Объект>)`).

Модули в состоянии анализировать «своё» значение `__name__`, чтобы выяснить, импортируются ли они или запущены как отдельное приложение (скажем, из командной строки), потому что в последнем случае локальной переменной `__name__` присваивается значение `'__main__'`. Поэтому в программах *Python* можно увидеть такую идиому:

```
if __name__ == '__main__':
    #
else:
    #
```

или весьма часто — её более простой вариант:

```
if __name__ == '__main__':
    #
```

где первая часть кода выполняется только при автономном запуске модуля. Использование её позволяет иметь в каждом модуле код для его тестирования; если модуль импортируется, этот код не исполняется, а вот при «отдельном» запуске тестирующий код будет исполнен и можно получить и проанализировать его результаты.

Ссылка по теме

Руководство по интроспекции на *Python*
https://www.ibm.com/developerworks/ru/library/l-pyint/

[4] Файла с кодом *Python*; он может содержать переменные, функции, классы и/или исполняемый код.

4. REPL — Read-Eval-Print Loop

Как уже говорилось, язык *Python* является интепретируемым. Мы можем либо разместить строки программы в текстовом файле, сохранив его на диск с расширением `.py`, либо вводить их в среде интерпретатора поочерёдно после подсказки *Python*, состоящей из трёх подряд символов «больше» (`>>>`) и иногда называемой «шеврон» (*chevron*).

Подсказка *Python* будет нам доступна, если мы запустим из консоли исполняемый файл интерпретатора или воспользуемся стандартным для *Python* приложением ***IDLE***. Наличие такой подсказки показывает пользователю, что интерпретатор ожидает от него ввода команд. Если же в командной строке консоли мы укажем после имени интерпретатора ещё один параметр — имя какого-либо `.py`-файла, то интерпретатор начнёт обрабатывать команды из этого файла и не перейдёт в интерактивный режим.

Последовательность действий интепретатора Python в интерактивном режиме именуется ***REPL*** (***R**ead-**E**val-**P**rint **L**oop*), поскольку интерпретатор считывает введённые команды (*read*), исполняет их (*evaluate*) и выводит полученный результат (*print*), ожидая новых команд от пользователя, т. е., по сути находится в процессе исполнения цикла (*loop*).

Базовые операции в языке, сразу доступные в интерактивном режиме: сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`) и возведение в степень (`**`); они работают ожидаемо, за исключением деления: оно — целочисленное для *Python 2*, точно так же, как и в *C/C++*. А вот в *Python 3* это уже не так, поскольку там результат будет вещественным.

Активизировать подобное поведение можно и в *Python 2* — с помощью (пока магической) команды

```
>>> from __future__ import division
```

После этого целочисленный вариант операции никуда не исчезает, мы тоже можем им пользоваться, употребляя новый символ деления `//` (работает это и в *Python 2*, и в *Python 3*).

Поскольку *Python* — язык программирования общего назначения, в нём доступны все стандартные математические функции, они располагаются в отдельном модуле с именем `math`. Для того, чтобы ими пользоваться, мы должны ***импортировать*** этот модуль:

```
>>> import math
>>> math.exp(1.0)
```

В отличие от большинства других распространённых сейчас языков, в *Python* мы можем сразу же узнать, какие функции доступны и как они правильно называются, поскольку имеется функция, позволяющая для каждого подгруженного модуля узнать, что в нём доступно:

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc',
 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi',
 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Будут перечислены объекты этого модуля; в данном случае можно видеть, что, в основном, это функции и некоторые важные переменные.

Аналогично, можно получить краткую справочную информацию о различных объектах модуля, используя функцию `help()`.

```
>>> help(math.exp)
```

5. Типы данных, величины и переменные

Встроенными типами в *Python* являются числовые типы, последовательности, отображения, классы, экземпляры (*instances*), исключения. Поскольку язык *Python* в данном пособии разбирается не слишком подробно, далее будут затронуты лишь первые две группы типов.

Python является языком с динамической типизацией. Переменные величины в *Python*-программе не объявляются (как, скажем, в *C/C++*) вместе с названием типа, т. к. их тип может измениться в процессе работы программы. Имя переменной *связывается* с какой-либо величиной (и типом этой величины) тогда, когда переменная «получает значение» в результате операции присваивания (для этого в *Python* — как и в *C/C++* — применяется знак равенства). Следует только иметь в виду, что присваивание в *Python* какого-либо значения переменной всегда создаёт лишь ссылку, а не копию значения. Величины, оставшиеся по какой-либо причине без ссылок на них, уничтожаются далее «сборщиком мусора».

Замечание. В отличие от *C/C++* присваивание в *Python* не является оператором, а потому не возвращает значения и не может быть использовано в выражениях.

Таким образом, понятие типа в *Python* применимо в первую очередь к значениям, а имена переменных — лишь «ярлыки» к этим значениям. Почти все значения в языке *Python* — это объекты, которые имеют три базовые характеристики: идентификатор, тип, значение. Идентификатор (возвращаемый функцией `id()`) можно считать адресом объекта в памяти, тип данных (получаемый с помощью функции `type()`) определяет возможности объектов (их свойства и поддерживаемые ими операции), а также допустимые значения для них.

Полезно поэкспериментировать с различными значениями в командной строке *Python*; мы сделаем это для наглядности сразу в двух основных версиях интерпретатора (слева — то, что получилось в *Python 2*, справа — в *Python 3*):

```
>>> type(4)
<type 'int'>
>>> type(2**75)
<type 'long'>
>>> type(1j)
<type 'complex'>
>>> type("3")
<type 'str'>
>>> type('1')
<type 'str'>
>>> type([6,9])
<type 'list'>
>>> type((2,))
<type 'tuple'>
>>> type({3,1,4})
<type 'set'>
```

```
>>> type(4)
<class 'int'>
>>> type(2**75)
<class 'int'>
>>> type(1j)
<class 'complex'>
>>> type("3")
<class 'str'>
>>> type('1')
<class 'str'>
>>> type([6,9])
<class 'list'>
>>> type((2,))
<class 'tuple'>
>>> type({3,1,4})
<class 'set'>
```

Интересно также присвоить эти или аналогичные значения каким-то переменным и после присваивания выводить эти значения, просто указывая имя переменной в строке команды. Можно заметить, что между версиями *Python* есть различия в трактовке типов: ранее они назывались просто типами, в новой версии типы — это классы. Изменилось также поведение типа `int`: теперь для «больших» величин уже не используется тип `long`, поскольку `int` может работать с целыми значениями, ограниченными лишь размером доступной памяти.

5.1. Числовые типы

К ним относятся `int` (целое число), `float` (вещественное) и `complex` (комплексное число). Роль мнимой единицы в записи комплексных констант выполняет символ `j` (или `J`; обратите

внимание на то, что ему обязательно должно предшествовать число!):

```
>>> (1+1j)*(2-3j)
(5-1j)
```

Математические функции, применимые к комплексным числам, находятся в модуле `cmath`:

```
>>> import cmath
>>> dir(cmath)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atanh', 'cos', 'cosh', 'e', 'exp', 'isinf', 'isnan', 'log', 'log10', 'phase',
 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

5.2. Последовательности

Типы (классы) `list` (список), `tuple` (кортеж) и `str` (строка) в *Python* — это последовательности, т. е. они являются коллекциями величин, для которых важен порядок. Наиболее общий тип из них — список: последовательность произвольных объектов, причём последовательность *изменяемая* (***mutable***). Кортеж (или набор) — неизменяемая версия списка. Строка — неизменяемая (***immutable***) последовательность символов, причём отдельно символа как типа в *Python* нет, он представляется строкой длины 1. Неудивительно поэтому, что одиночные и двойные кавычки в *Python* почти эквивалентны и взаимозаменяемы.

Списки конструируются с помощью квадратных скобок — с разделением в них отдельных элементов с помощью запятых. Кортежи образуются перечислением их элементов через запятую (могут быть в круглых скобках или без них[5]).

5.3. Ещё о переменных

Переменные в *Python* возникают тогда, когда они получают своё значение, но тут есть некоторая тонкость: присваивание значения переменной в рамках функции создаёт всегда *локальную* переменную — даже если имеется глобальная с тем же именем. В то же время воспользоваться в рамках функции значением этой глобальной переменной можно без труда — имя её там доступно. Как же изменить из функции глобальную переменную? Здесь поможет инструкция `global` *<СписокИдентификаторов>*, перечисляющая (через запятую) те идентификаторы, которые в текущем блоке будут ссылаться на глобальные переменные. Без этой инструкции присваивание значений глобальным переменным невозможно.

> Таким образом, в *Python*-программах могут возникать переменные с одинаковыми именами (глобальная и локальные) и даже иногда бесконфликтно сосуществовать — до тех пор, пока не понадобится изменять именно глобальную.
>
> С точки зрения возможности дублирования имён в языке *Python* интересно проанализировать соответствующую программу обхода квадратных досок ходом шахматного коня на странице http://rosettacode.org/wiki/Knight's_tour (посвящённой реализации таких алгоритмов на различных языках программирования), обращая внимание на имя `boardsize`...

Ссылки

Built-in Types
https://docs.python.org/2/library/stdtypes.html

Тонкости использования языка *Python*: Часть 2. Типы данных
https://www.ibm.com/developerworks/ru/library/l-python_details_02/

[5] Пустой кортеж обязан иметь круглые скобки, а кортеж из одного элемента — завершающую запятую.

6. Функции в языке *Python*

6.1. Синтаксис определения функций

Ключевое слово `def` начинает определение функции. Далее следует имя функции, а затем (в круглых скобках) — список формальных параметров, после чего ставится двоеточие. Инструкции, составляющие тело функции, записываются с отступом со следующей строки. Первой инструкцией в теле функции может быть и строка документации.

```
def <ИмяФункции>(<СписокПараметров>):
    '''<ДокументационнаяСтрока>'''
    <ТелоФункции>
```

Тип возвращаемого значения (в отличие от *C/C++*) здесь не присутствует, тем более, что в процессе исполнения функции он может изменяться. Рассмотрим простой пример: функцию, формирующую последовательность Люка (***Lucas***) и применяемую для поиска простых чисел Мерсенна (***Mersenne***) и совершенных чисел (использован *Python 2*).

```
def Lucas(K):
    S = 4
    for i in range(K):
        S = S*S - 2
    return S
```

```
>>> Lucas(0)
4
>>> Lucas(2)
194
>>> Lucas(6)
4023861667741036022825635656102100994L
```

6.2. Передача параметров в функции

Если сформулировать коротко, то функциям в языке *Python* при вызове всегда передаются ссылки на объекты, причём по значению (т. е., функции всегда работают с копиями ссылок).

К чему это может приводить на практике, можно продемонстрировать несколькими небольшими примерами (в них мы намеренно используем одно и то же имя в разных местах!).

> Каждый из этих примеров можно передавать интерпретатору копированием через буфер обмена «одним куском»; если последняя строка «куска» не является пустой, то для вывода результата последней команды понадобится ещё нажать клавишу *Enter*. Можно также создать несколько исходных файлов и запускать их (с командной строки, из *IDLE* или редактора вроде ***Geany***) по очереди.

Пример 1

Функции передаётся список, в котором она удваивает каждый элемент списка.

```
def make_double1(L):
    print("L = %s" % L)
    for i in range(len(L)):
        L[i] = 2 * L[i]
    print("L = %s" % L)

L = [1, 1, 2, 3, 5, 8]
print("L = %s" % L)
make_double1(L)
print("L = %s" % L)
```

```
>>> def make_double1(L):
...     print("L = %s" % L)
...     for i in range(len(L)):
...         L[i] = 2 * L[i]
...     print("L = %s" % L)
...
>>> L = [1, 1, 2, 3, 5, 8]
>>> print("L = %s" % L)
L = [1, 1, 2, 3, 5, 8]
>>> make_double1(L)
L = [1, 1, 2, 3, 5, 8]
L = [2, 2, 4, 6, 10, 16]
>>> print("L = %s" % L)
L = [2, 2, 4, 6, 10, 16]
```

Результат, полученный в этом примере, понятен: пользуясь передаваемой ссылкой, можно изменить объект по этой ссылке — хотя ссылка и была всего лишь копией исходной ссылки на объект.

Пример 2

Функции передаётся строка, заменяемая далее конкатенацией двух её копий.

```
def make_double2(L):
    print("L = %s" % L)
    L = L + L
    print("L = %s" % L)

L = "Hi"
print("L = %s" % L)
make_double2(L)
print("L = %s" % L)
```

```
>>> def make_double2(L):
...     print("L = %s" % L)
...     L = L + L
...     print("L = %s" % L)
...
>>> L = "Hi"
>>> print("L = %s" % L)
L = Hi
>>> make_double2(L)
L = Hi
L = HiHi
>>> print("L = %s" % L)
L = Hi
```

Здесь ситуация в чём-то интереснее, но и проще. Мы объект по ссылке не изменяли, но изменили саму ссылку. Точнее, мы создали в функции новый объект (конкатенированную строку), ссылка на который — то же самое имя L; переданная копия ссылки на глобальный объект при этом пропала и заместилась (локальной) ссылкой на новый объект. Понятно, что глобальный объект не подвергся никаким изменениям.

Пример 3

Функции передаётся число, а функция удваивает это число.

```
def make_double3(L):
    print("L = %s" % L)
    L = 2 * L
    print("L = %s" % L)

L = 10
print("L = %s" % L)
make_double3(L)
print("L = %s" % L)
```

```
>>> def make_double3(L):
...     print("L = %s" % L)
...     L = 2 * L
...     print("L = %s" % L)
...
>>> L = 10
>>> print("L = %s" % L)
L = 10
>>> make_double3(L)
L = 10
L = 20
>>> print("L = %s" % L)
L = 10
```

Ситуация аналогична предыдущей, только тип передаваемого объекта другой и операция над ним изменилась, но точно так же возник новый локальный объект (удвоенное число) и ссылка на переданный заместилась ссылкой на новый. Глобальный объект при этом тоже не изменился.

6.3. Дополнительные возможности в определении функций

В языке *Python* функции могут иметь переменное число передаваемых им параметров (аргументов). Реализуется это несколькими способами. Можно установить значения по умолчанию для одного или нескольких аргументов — как это делается в *C/C++*. Можно определить функцию так, что ей можно будет передать кортеж (с произвольным числом элементов). Можно также вызывать функцию с использованием *именованных аргументов*.

Значения аргументов по умолчанию

> **Замечание.** Значения по умолчанию вычисляются в месте определения функции в области видимости определения, причём только один раз. Это существенно тогда, когда аргумент со значением по умолчанию является изменяемым объектом.

Произвольный набор аргументов

Аргументы «произвольной» части передаются в виде кортежа, перед формальным именем которого стоит «звёздочка» (`*`). Перед переменным числом аргументов может присутствовать произвольное число обычных аргументов.

Именованные аргументы

Функцию можно вызывать с использованием ***именованных аргументов***, тогда количество и порядок аргументов могут отличаться, но обязательные параметры должны присутствовать. Кроме того, в списке аргументов именованные аргументы должны следовать после позиционных. Передавать дважды один и тот же параметр не допускается.

Если в определении функции присутствует формальный параметр вида `**<Имя>`, то его значением должен быть словарь, содержащий именованные аргументы, не совпадающие с другими формальными параметрами. Подобный способ передачи параметров можно комбинировать с упомянутой выше формой `*<Имя>`, когда передаётся кортеж (`tuple`) с позиционными аргументами (которые тоже не должны входить в список формальных параметров). При этом запись `**<Имя>` должна следовать после записи `*<Имя>`.

Однострочные функции

Если тело функции состоит из одной строки, её определение может быть написано проще:

```
def <ИмяФункции>(<СписокПараметров>): <ТелоФункции>
```

6.4. Анонимные функции

С помощью ключевого слова `lambda` в языке *Python* можно создать простую функцию вообще без имени. Эта короткая форма может быть использована везде, где требуется объект-функция, определяемая одним выражением. Так что можно сказать, что `lambda`-функция в *Python* — это способ создать «одноразовую» функцию без оформления её стандартным образом, особенно если вся функция состоит из одного выражения.

Синтаксис определения `lambda`-функции прост: ключевое слово `lambda`, после которого следуют параметры функции через запятую, а затем (после двоеточия) — выражение, вычисляемое при вызове функции и возвращаемое в качестве её результата.

```
lambda <Параметры> : <Выражение>
```

Исполнена `lambda`-функция будет там, где будет использоваться создаваемая при этом ссылка (скажем, в какой-нибудь из функций, принимающих как параметр функцию). Или, если присвоить определяемую `lambda`-функцию какой-то переменной, можно выполнить её как обычную функцию, используя имя этой переменной с параметрами.

Преимущества таких функций — более простая работа с функциями `filter()`, `map()`, `zip()`, `reduce()`, составляющими основу функционального способа программирования в языке.

7. Модули и пакеты в *Python*

Организация кода в языке *Python* может быть представлена такой иерархией: команды — функции — модули — пакеты. **Модуль** — это файл с кодом *Python*, в нём могут быть определены функции, классы, переменные; он может также включать в себя исполняемый код. Любой исходный файл *Python* можно использовать как модуль, исполняя инструкцию `import` с его именем (без расширения `.py`) в каком-то другом исходном файле *Python*. Модуль при этом подгружается лишь один раз, независимо от того, сколько раз он был импортирован.

> Способ обращения к содержимому модуля зависит от того, *как* модуль был импортирован. Если была использована простейшая форма (`import` *<ИмяМодуля>*), то понадобится имя модуля в качестве префикса. Например, после `import math` надо будет писать `math.cos(math.pi/4)`, а вот после `from math import *` или `from math import cos, pi` — можно просто `cos(pi/4)`; связано это с тем, что последние две формы позволяют импортировать либо все составляющие модуля, либо его отдельные части в *текущее пространство имён*, т. е., в глобальную символьную таблицу импортирующего модуля; при этом первая форма несколько «хуже» второй, поскольку «засоряет» глобальную таблицу излишними именами.
>
> Импортируемому объекту можно задать *псевдоним*, если завершить инструкцию импорта последовательностью `as` *<Псевдоним>*, например, `import math as m` или `import numpy as np`.
>
> В пределах одного ***пакета*** (определение см. ниже) возможен т. н. относительный импорт модулей, в котором указываются текущий каталог (`.`) или родительский (`..`) в качестве места расположения импортируемого модуля; подобный импорт всегда использует форму `from` *<Откуда>* `import` *<Что>*. Пример: `from .types import *` (файл `numba/__init__.py`[6]).

При (абсолютном) импорте модулей интерпретатор ищет их в такой последовательности:

- текущий каталог;
- каждый каталог, перечисленный в переменной окружения `PYTHONPATH`;
- путь по умолчанию (например, в *Linux* это `/usr/local/lib/python`*<Версия>*)

Пути поиска модулей хранятся в системном модуле `sys` (переменная `sys.path`).

Пакет в *Python* — это каталог, включающий в себя какие-то модули и, возможно, другие подобные каталоги (подпакеты), и при этом содержащий файл с именем `__init__.py` (может быть пустым), предназначенный для действий по инициализации пакета. В таком файле также может содержаться т. н. список `__all__`, определяющий, какие модули будут импортироваться из пакета инструкцией `from` *<ИмяПакета>* `import *`.

Установка дополнительных пакетов в *Python*

Большая часть функциональности языка *Python* сосредоточена в модулях, сопровождающих дистрибутив *Python*. То, чего недостаёт для решения каких-то конкретных задач, может быть установлено из разработанных сторонних программных пакетов.

Обычная установка (т. н. *distutils distribution*) включает распаковку пакета и исполнение команды `python setup.py install` с административными правами из *его* каталога; в результате пакет будет установлен в каталог `site-packages` дистрибутива *Python*. Для установки в заданное место надо указать это место: `python setup.py install` *<Место>*.

Если пакет имеется в ***Python Package Index***, он может быть установлен также с помощью команды `pip install` *<ИмяПакета>*[7] или команды `easy_install` *<ИмяПакета>*.

---

[6] Инициализирующий файл пакета ***Numba*** (см. стр. 28). Аналогичные формы импорта имеются также в `arrayfire/__init__.py` — инициализирущем файле пакета ***ArrayFire Python Wrapper*** (см. стр. 40).

[7] `pip` — инсталлятор ракетов *Python* (`http://pypi.python.org/pypi/pip`). Поскольку устанавливаются пакеты из исходного кода, то процесс инсталляции может не пройти, если не установлено соответствующее окружение (*C++*-компилятор).

8. Декораторы функций и методов в *Python*

Начиная с версии *2.4* в *Python* введена новая синтаксическая конструкция — ***декоратор*** функции/метода. С её помощью можно «декорировать» функции — дополнять какой-либо новой функциональностью без внесения изменений в сами функции или методы.

> Прежде всего, стоит отметить, что функции (как и всё в *Python*) — это полноценные объекты (используемая терминология — ***first-class objects***). На практике это означает, что они как объекты не имеют ограничений по использованию — они могут иметь атрибуты, могут быть присвоены переменной, переданы в функцию в качестве аргумента, могут быть возвращаемым значением из функции и т. п.
>
> При этом, объект функции подчиняется тем же правилам, которым подчиняется любой другой объект в *Python* — в частности, на него заводится счетчик ссылок, и сам объект не зависит от того первоначального имени, которое было ему дано при определении функции (этому имени может быть впоследствии сопоставлен другой объект, но объект функции будет доступен по другим ссылкам, если они есть).
>
> Ну и, конечно, функция может быть создана в процессе исполнения другой функции. Все эти особенности и используются для декорирования функций.

8.1. Декорирование функций

Декораторы в *Python* позволяют разработчику «завернуть» одну функцию в другую, так что результаты работы внутренней функции будут «декорированы» внешней функцией. Определение декоратора похоже на определение любой функции *Python*, которой передаётся функция, только здесь это — функция, которую необходимо «декорировать». Внутри функции-декоратора создаётся другая функция, в которой описывается, как нужно вызывать переданную для декорирования функцию. Можно также написать функцию, которая возвращает функцию-декоратор (т. е., функцию, которая может декорировать функции). Такая функция называется «фабрикой» декораторов.

> В принципе, декоратором может служить любой объект, который можно вызвать на исполнение. Такой объект можно сконструировать, определив в его классе метод `__call__`. Таким образом, декоратор в *Python* — это вызываемый объект, принимающий какой-то вызываемый объект как параметр и возвращающий тоже вызываемый объект.

Рассмотрим несложный пример декорирования (пока даже без использования общепринятого синтаксиса), располагая определение декорирующей функции *перед* определением декорируемой, — чтобы подчеркнуть, что декорировать каким-либо образом можно любую определённую впоследствии функцию.

```
def decorate(fn):
  if not hasattr(fn, '__call__'):
    raise TypeError('The argument should be a callable')
  def wrapper(arg):
    print "(calling function %s with arg %s)" % (fn.__name__, str(arg))
    return fn(arg)
  return wrapper

def func(arg):
  print arg

func = decorate(func)
# func(1)
```

Функция `func()` просто выводит переданный ей аргумент. Функция `decorate()` принимает в качестве аргумента функцию (или объект, который может быть вызван на исполнение).

Если будет передан какой-нибудь неподходящий объект, то возникнет исключительная ситуация `TypeError`. Внутри функции `decorate()` объявляется функция `wrapper()`, которая «оборачивает» вызов переданной в `decorate()` функции, предваряя его выводом имени вызываемой функции и её аргумента. Функция `wrapper()` — фактически шаблон для создания новой функции, которая возникнет только в процессе исполнения `decorate()`.

Эта новая функция и есть результат вызова `decorate()` и ей снова присвоено имя `func`, которое теперь ссылается уже не на оригинальную функцию `func()`, а на эту новую функцию, которая была создана из определения `wrapper()` в процессе исполнения `decorate(func)`. На саму оригинальную `func()` осталась ссылка только внутри этой созданной функции.

> Здесь также использован такой механизм языка, как вложенные области видимости (*nested scopes*): если переменная доступна в локальной области видимости какой-то функции, то она будет доступна также в локальной области видимости внутри блока другой функции, находящегося в блоке этой функции (если эта переменная там не будет переопределена). Т. о., аргумент `fn` функции `decorate()` доступен внутри функции `wrapper()`.

Основное достоинство такого приёма в том, что функцию `decorate()` можно применять к любым функциям, не внося при этом изменений в эти функции.

8.2. Синтаксис декораторов функций

Синтаксис декорирования был заимствован из синтаксиса аннотаций *Java*. Для того, чтобы декорировать функцию, нужно перед её объявлением указать имя функции-декоратора с символом `@` впереди. В нашем случае, выражение `func = decorate(func)` с помощью нового синтаксиса можно записать (вместе с определением `func()`) так:

```
@decorate
def func(arg):
  print arg
```

```
>>> func(1)
(calling function func with arg 1)
1
```

Теперь любой вызов `func()` будет предваряться выводом имени функции и её параметра.

Комбинирование декораторов

Допустим, нам надо проверять, чтобы аргумент был целым числом, однако при этом — положительным целым. Мы можем записать это так (несущественные детали опущены):

```
def must_be_positive(fn):
  def wrapper(arg):
    if arg <= 0:
      raise ValueError('The argument should be positive')
    return fn(arg)
  return wrapper

def must_be_int(fn):
  def wrapper(arg):
    if type(arg) != int:
      raise TypeError('The argument should be an integer')
    return fn(arg)
  return wrapper

def f(arg):
  print arg

func = must_be_positive(func)
func = must_be_int(func)
```

А можем и так (предполагая, что декорирующие функции уже ранее определены):

```
@must_be_int
@must_be_positive
def func(arg):
  print arg
```

```
>>> func(1)
1
>>> func(-1)
. . .
ValueError: The argument should be positive
>>> func(-0.1)
. . .
TypeError: The argument should be an integer
```

Видно, что декораторы вызываются здесь в том порядке, в котором они упомянуты...

Стоит обратить внимание на порядок записи декораторов: это выражение эквивалентно `func = must_be_int(must_be_positive(func))`. Это важно, так как порядок применения декораторов часто имеет значение. В нашем случае сначала должна быть проверена целочисленность аргумента, т. е. функция, созданная в `must_be_int()` должна быть вызвана прежде, чем функция, созданная в `must_be_positive()` проверит, положительное ли число мы получили.

Видно, что при декорировании без использования специального синтаксиса выражения пишутся в обратном порядке — `func` будет ссылаться на результат последнего из них, т. е. `must_be_int()` отработает первой. Таким образом, специальный синтаксис также более удобен, т. к. декораторы перечисляются в прямом порядке. Кроме того, он позволяет проще записывать передачу параметров декоратору.

Декораторы с параметрами

Иногда бывает полезно, чтобы декораторы принимали какие-либо параметры — помимо декорируемой функции. В этом случае в декораторе появится дополнительный уровень «косвенности»: декоратор с параметрами должен возвратить реальный декоратор, а тот, в свою очередь, уже декорирует функцию. Таким образом, декораторы с параметрами — что-то вроде «фабрики» декораторов: функции, возвращающие функцию, которая может принимать функцию как параметр и возвращать функцию. В них, как в матрёшках будут определены друг в друге новые функции: сначала функция, в которой (или во вложенной в неё функции) обрабатываются параметры такого декоратора с параметрами, а возвращается декоратор (без параметров), затем собственно функция декоратора, принимающая декорируемую функцию, а возвращающая новую функцию, определяемую в ней.

Декораторы без параметров записываются без круглых скобок, а вот с параметрами — всегда со скобками, даже если параметры не передаются (параметры по умолчанию), т. к. функции, возвращаемые этими декораторами, существенно различаются.

Поскольку декорирование какой-либо функции — по сути вызов декорирующей функции с передачей ей ссылки на декорируемую, то любые действия, предусмотренные в рамках декоратора за пределами новой функции (определяемой для вызова вместо декорируемой), будут выполнены лишь один раз: во время вызова декорирующей функции, производимого при определении декорируемой.

Здесь не обсуждаются декораторы классов — поскольку сами классы почти не упоминаются.

8.3. Применение декораторов

Декораторы применяются достаточно широко: для проверки допустимости аргументов, для перехвата и изменения аргументов функций, для перехвата и изменения возвращаемых функциями значений, для проверки дополнительных условий и т. п. Рассмотрим несколько примеров.

Неприятное последствие декорирования — некоторые сведения об исходной функции будут утеряны, если не предпринимать специальных мер по их сохранению. В этом легко убедиться с помощью функции `inspect.getargspec()`, возвращающей четвёрку (кортеж из четырёх элементов) с описанием параметров, принимаемых задаваемой функцией. Первый элемент — список параметров, второй — `None` или имя `*`-параметра, третий — `None` или имя `**`-параметра, четвёртый — `None` или список параметров с величинами по умолчанию.

Модуль *Python* `functools` содержит функцию `wraps()`, которая возвратит декоратор с учётом сведений об исходной функции.

Декораторы для методов *Python*: `classmethod`, `staticmethod`

При объявлении методов классов поведением по умолчанию будет оборачивание функций в методы объектов; они будут вызывать исходные функции при своём вызове, передавая им в качестве первого аргумента экземпляр класса.

Если объявление метода предварить декоратором `classmethod`, то будет создан объект-метод, который в первом аргументе функции будет передавать не экземпляр класса, а сам класс.

Если объявление метода предварить декоратором `staticmethod`, то будет создан объект-метод, который никак не будет изменять список аргументов, переданный методу. Метод будет вызван как будто это обычная функция, не имеющая представления о состоянии класса или экземпляра класса.

Декораторы «на службе» *GPU*

В тех случаях, которые интересуют нас, декоратор может вызвать компиляцию (или какое-либо иное преобразование) помеченной им функции. В качестве параметров декоратора можно указывать необходимые величины для процесса преобразования (например, типы параметров, поскольку они нужны для компиляции).

Как мы увидим далее, такая техника используется в ***Numba*** (см. стр. 28), когда — помимо компиляции функций для работы на *GPU* или различных потоках *CPU* — указываются целевое устройство для них (на каком устройстве функции должны будут исполняться), специальные флаги, изменяющие ход процесса компиляции и тип компиляции (во время исполнения или до исполнения), альтернативные имена и сигнатуры для функций (типы передаваемых и возвращаемых параметров). В случае пакета ***ocl*** (см. стр. 47) декоратор просто задаёт тип компиляции, но для заранее выбранных «целей», причём как связанных с *GPU*, так и не связанных (*JavaScript*).

Ещё о декораторах

Decorators With Arguments in Python
http://scottlobdell.me/2015/04/decorators-arguments-python/
Обсуждение декораторов без параметров и с параметрами, сопровождаемое простыми примерами и эквивалентными выражениями без синтаксиса декорирования.

3 decorator examples & other awesome things about Python
https://mrcoles.com/blog/3-decorator-examples-and-awesome-python/
Некоторые подробности о функции `functools.wraps()`; применение функции `functools.partial()` (создаёт функцию из существующей — с несколькими предопределёнными параметрами).

Python decorators with optional arguments
https://blogs.it.ox.ac.uk/inapickle/2012/01/05/python-decorators-with-optional-arguments/
Пример оформления декоратора, который можно будет использовать как без параметров, так и с параметрами по умолчанию; использование функции `functools.partial()`.

9. Как *Python* помогает в работе с *GPU*?

Теперь уже понятно, что средствами *Python* можно получить доступ к любым библиотекам, а также создать код, который может быть исполнен на специализированных устройствах типа *GPU* (или на других ускорителях), — хотя он и может изначально выглядеть как код на языке *Python*. При этом вполне возможно реализовать взаимодействие с *GPU* самостоятельно — для этого в библиотеках изготовителей *GPU* имеются все необходимые функции. Однако интересно (и полезно) познакомиться с различными существующими решениями в этой области, посмотреть, как эти решения различаются по способам организации своего взаимодействия со средствами, на базе которых они функционируют (*CUDA*, *OpenCL* и т. п.), узнать, как при этом придётся формулировать алгоритмы для *GPU*. Краткая информация об этом приведена в таблице.

Таблица 1: Программные решения для работы с *GPU* в рамках *Python*.

Программный пакет	Что использует	Пользовательский язык для действий на *GPU*
PyCUDA	*CUDA*	*C/C++*
PyOpenCL	*OpenCL*	*C99*
Numba	*CUDA*, *HSA*, *CPU*	*Python*
Theano	*CUDA*, *CPU* *OpenCL?*	*Python*
ArrayFire *Python Wrapper*	*ArrayFire* (*CUDA*, *OpenCL*, *CPU*)	*Python*
PyViennaCL	*ViennaCL* (*CUDA*, *OpenCL*, *OpenMP*)	*Python*
cudamat	*CUDA*+*cuBLAS*	*Python*
gnumpy	*cudamat*	*Python*
PyGPU	*Cg*	*Python*
PyStream	*GLSL*	*Python*
ocl	*PyOpenCL*	*Python*

PyCUDA и ***PyOpenCL*** — хотя и в более удобном для пользователя виде — фактически переносят в *Python* программный интерфейс соответствующих решений низкого уровня (*CUDA* и *OpenCL*), а потому в них надо будет позаботиться о т. н. ядрах для исполнения на *GPU*, исходный код которых будет выглядеть так же, как и в случае *CUDA* (*C/C++*) и *OpenCL* (*C99*).

Напротив, последующие решения (***Numba***, ***Theano***, ***ArrayFire Python Wrapper*** и др.) дают возможность пользователю формулировать программу для *GPU* уже средствами *Python*[8]. В одних случаях оформление *GPU*-кода использует декораторы (***Numba***, ***PyGPU***, ***PyStream***, ***ocl***), в других *GPU*-код является частью реализации различных специальных типов данных. Скажем, в ***ArrayFire*** — это матричный объект `array`, в ***PyCUDA*** — `GPUarray`, в ***Theano*** — `libgpuarray`, в ***cudamat*** — `CUDAmatrix`.

Далее эти решения рассмотрены немного подробнее — в соответствии с принятым ранее планом: назначение, зависимости, процедура установки, примеры использования.

[8] Это, впрочем, не означает, что программы-ядра не будут больше нужны, просто в этих случаях ядра будут создаваться не пользователем, а программными средствами.

10. *PyCUDA*

Пакет ***PyCUDA*** (автор — ***Andreas Klöckner***) обеспечивает доступ к программному интерфейсу параллельных вычислений *CUDA* для графических процессоров фирмы *NVidia* (предполагается, что т. н. *CUDA Toolkit* уже установлен; если же это не так, имеет смысл воспользоваться инсталляционным пакетом с сайта *NVidia*).

Установка *PyCUDA* в *Windows*

Простейший способ здесь — воспользоваться готовым дистрибутивом (***Christoph Gohlke***). На своём сайте (`https://www.lfd.uci.edu/~gohlke/pythonlibs/`) упомянутый автор поддерживает в актуальном состоянии дистрибутивы для большинства пакетов *Python*, причём исключительно для установки в системе *Windows*, поскольку её пользователи «избалованы» готовыми инсталляционными пакетами и редко любят читать какие-либо инструкции по установке (и уж тем более — компилировать что-то самостоятельно).

Более «изощрённый» и правильный способ установки (рекомендуется только упорным) — откомпилировать исходный код (см. инструкции на сайте *PyCUDA*).

Установка *PyCUDA* в *Linux* — на примере *Ubuntu 14.04*

В *Ubuntu 16.04* и новее пакет `python-pycuda` уже присутствует в репозитории и потому устанавливается тривиально (`sudo apt-get install python-pycuda`). Если же *CUDA* на компьютере ещё нет, в репозитории имеются также необходимые `nvidia-cuda-toolkit` и `nvidia-cuda-dev`.

Для установки в *Ubuntu 14.04* надо учесть т. н. зависимости *PyCUDA* (модули, которые должны присутствовать перед установкой[9]); можно видеть, например, что дистрибутив `pycuda-2017.1.1.tar.gz` требует наличия 6 модулей *Python*:

```
numpy >= 1.6
appdirs >= 1.4.0
decorator >= 3.2.0
mako
pytest >= 2
pytools >= 2011.2
```

Сопровождающие эти имена номера означают ожидаемые номера версий (не ниже). Ничего особенного, в принципе, в этом списке модулей нет, так как все они имеются в репозитории *Ubuntu 14.04*[10]. Вот только один из них (`appdirs`) в репозитории имеет неприемлемо низкую версию (*1.2*) для инсталлятора, а потому должен быть установлен как-то иначе...

Проверка работоспособности

Установленную версию пакета `pycuda` легко проверить прямо в интерпретаторе:

```
>>> import pycuda.autoinit
>>> from pycuda.tools import make_default_context
>>> make_default_context().get_device().name()
```

[9] В файле `setup.py` дистрибутива `pycuda-X.Y.Z.tar.gz` (где `X`, `Y`, `Z` — это конкретные цифры версии), в самом его конце следует обратить внимание на поименованные параметры, передаваемые функции `setup()` при вызове: `setup_requires` и `install_requires`; их значения и есть списки зависимостей.

[10] Соответствующие модули имеют имена с префиксом '`python-`' (`python-numpy` и т. д.) и могут быть установлены с помощью команды `sudo apt-get install` *<ИмяМодуля>*. Можно также до установки проверить их наличие в системе (`dpkg -l | grep` *<ИмяМодуля>*) и в репозитории (`apt-cache search` *<ИмяМодуля>*).

Работа с *PyCUDA*

Прежде чем начать работать с *PyCUDA*, пакет необходимо импортировать и инициализировать; типичный вариант «пролога» может выглядеть примерно так:

```
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import pycuda.autoinit
```

Здесь использованы три разных формы импорта: импорт модуля `driver` из пакета `pycuda` под именем `cuda` (через него осуществляется работа с *GPU*), импорт описания класса `SourceModule`, во время создания объекта которого компилируется исходный код функций-ядер *CUDA* и возникает бинарный исполняемый модуль (называемый в *CUDA* ***cubin***), из которого затем можно «извлечь» для исполнения нужную функцию-ядро, и (как бы[11]) импорт модуля `autoinit` из пакета `pycuda`; во время этого импорта производится необходимая инициализация *CUDA*, создаётся т. н. контекст для работы с устройством *CUDA* и оно активизируется для работы, определяется функция для завершения всей работы с *GPU* (освобождение ресурсов) и эта функция регистрируется как принудительно вызываемая в самом конце работы программы.

Конечно, необязательно использовать инициализацию с помощью `import pycuda.autoinit`, можно самостоятельно создать контекст для работы с реализацией *CUDA* и освобождать все использованные ресурсы в конце работы, но указанная строка импорта в реальности делает это всё за нас, а потому весьма удобна и часто встречается в *PyCUDA*-программах.

«Традиционный» вид кода — выделение памяти на устройстве и копирование данных туда, а после расчётов — обратно (нужное обрамление показано слева, работа с ядром — справа):

```
import numpy as np
a = np.random.randn(512)
a = a.astype(np.float32)
r = np.zeros_like(a)
r_gpu = cuda.mem_alloc(r.nbytes)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)
#    .    .    .    (sqr)
sqr(r_gpu, a_gpu, block=(512,1,1))
cuda.memcpy_dtoh(r, r_gpu)
```

```
mod = SourceModule("""
  __global__ void sqr(float *r,
                      float *a)
  {
    const int i = threadIdx.x;
    r[i] = a[i] * a[i];
  }
  """, options=['-use_fast_math'],
       arch='sm_30')
sqr = mod.get_function("sqr")
```

Для исполнения кода ядер надо получить (вызываемый) объект, который возвращает метод `get_function()` объекта исходного модуля (т. е., экземпляра класса `SourceModule`). После этого можно инициировать исполнение ядер в нужном количестве и в необходимой конфигурации, вызывая этот объект как функцию, которой передаются сначала все параметры каждого ядра, а затем — через параметры размера блока (`block`) и сетки (`grid`), задаваемые кортежами из трёх целых значений — размерности «пространства исполнения».

> Оказывается, код, передаваемый конструктору `SourceModule`, в *PyCUDA* будет «обёрнут» в `extern "C" {...}`; нужен поименованный параметр `no_extern_c` со значением `True` в этом конструкторе, чтобы избежать этого (например, для включения какого-то специфического заголовочного файла). Правда, после этого все имена функций будут подвергнуты преобразованию (т. н. ***mangling***) и должны извлекаться с помощью метода `get_function()` по уже преобразованному имени (скажем, ядро с именем `test` и двумя параметрами типов `float*` и `int` будет называться `"_Z4testPfi"`); можно также снова «обёрнуть» функции в `extern "C" {...}`.
>
> Параметры командной строки для компилятора `nvcc` (если они понадобятся) можно передать в массиве строк через поименованный параметр конструктора `options`; целевая архитектура для формируемого кода указывается в поименованном параметре `arch`.

[11] Этот модуль содержит не набор функций (или описаний объектов), а лишь функцию освобождения ресурсов и исполняемый во время загрузки модуля код.

Можно также воспользоваться обработчиками параметров `In`, `Out`, `InOut` из `pycuda.driver`; тогда переносы между памятью хоста и памятью устройства будут выполняться неявно:

```
sqr(cuda.Out(r), cuda.In(a), block=(512,1,1), grid=(1,1,1))
```

Особенно просто всё будет выглядеть в случае одного параметра, содержимое которого можно перезаписать для возвращения результата (`sqr(cuda.InOut(a), block=(512,1,1))`).

Ещё один способ работать с данными на *GPU* — использовать массивы из `pycuda.gpuarray`:

```
import pycuda.gpuarray as gpuarray
import pycuda.cumath as cumath
import pycuda.autoinit
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X) # 1. transfer -> gpu
Y_gpu = cumath.sin(X_gpu)  # 2. execute kernel
Y = Y_gpu.get()            # 3. retrieve result
```

> В `pycuda.cumath` располагаются математические функции для работы на *GPU*. Выражение с переменными, расположенными в *GPU*, и использующее стандартные операции и функции *GPU*, приводит к созданию соответствующего ядра и его запуску в нужной конфигурации, после чего результаты можно извлечь из памяти *GPU* с помощью метода `get()`. Понятно, что такой способ годится лишь для некоторых алгоритмов.

Ядра *CUDA* и система *Windows*

Следует заметить, что в системе *Windows* предусмотрено принудительное завершение работы программ, которые загружают *GPU* дисплея дольше заданного промежутка времени (как правило, это несколько секунд). У каждого *CUDA*-устройства имеется свойство `kernelExecTimeoutEnabled`; оно показывает, что существует ограничение на время исполнения ядер, т. е., начиная с некоторого момента их исполнение может быть остановлено. Способов избежать этого — по сути, два: либо запускать «недолго» исполняемые ядра, либо использовать выделенный *GPU*, не участвующий ни в каком отображении в системе.

Ссылки

PyCuda – Andreas Klöckner's wiki
https://wiki.tiker.net/PyCuda

PyCUDA – Andreas Klöckner's web page
https://mathema.tician.de/software/pycuda/

Installing PyCuda on Windows
http://wiki.tiker.net/PyCuda/Installation/Windows

Installing PyCUDA on Linux
https://wiki.tiker.net/PyCuda/Installation/Linux

Programming GPUs with PyCuda
http://conference.scipy.org/static/wiki/scipy09-pycuda-tut.pdf
Обучающий доклад (**Nicolas Pinto**, **Andreas Klöckner**). ***SciPy 2009***, *Pasadena, California.*

PyCUDA 2017.1.1 documentation
https://documen.tician.de/pycuda/

Examples of PyCuda usage
https://wiki.tiker.net/PyCuda/Examples

11. *PyOpenCL*

PyOpenCL позволяет получить доступ к *GPU* (или другим параллельным вычислителям) из языка *Python*, используя *OpenCL*-интерфейс. Автор пакета (***Andreas Klöckner***) — тот же, что и у *PyCUDA*.

Для использования *PyOpenCL* необходимо иметь реализацию *OpenCL* и установленный *Python*-пакет `numpy`. В рамках *PyOpenCL* остаётся доступным весь программный интерфейс *OpenCL* — если это необходимо. Ошибки, возникающие в процессе вызова функций *OpenCL*, автоматически транслируются в исключительные ситуации в языке *Python*. Базовый слой *PyOpenCL* реализован на языке *C++*.

Установка *PyOpenCL* в *Windows*

Всё, что было сказано про установку *PyCUDA* в *Windows* в предыдущем разделе, будет справедливо здесь и для *PyOpenCL*: наиболее простой способ — воспользоваться готовым дистрибутивом. Для самостоятельной компиляции исходного кода пакета полезно изучить инструкции на сайте *PyOpenCL*.

Установка *PyOpenCL* в *Linux* — на примере *Ubuntu 14.04* и выше

Пакет `python-pyopencl` есть в репозиториях *Ubuntu*, начиная с версии *14.04*[12], только везде он присутствует с солидным «запаздыванием»: в *14.04* — 2013 года, в *16.04* — 2015 года, в *17.10* — 2016 года. Установка из репозитория тривиальна (`sudo apt-get install python-pyopencl`); для установки последних дистрибутивов (`pyopencl-2017.2.2.tar.gz` или `pyopencl-2018.1.1.tar.gz`) потребуются такие модули (т. н. зависимости):

```
numpy
pytools >= 2017.6
pytest >= 2
decorator >= 3.2.0
cffi >= 1.1.0
appdirs >= 1.4.0
six >= 1.9.0
```

Проверка работоспособности

Одна из наиболее простых проверочных программ с использованием *PyOpenCL* обнаружилась в документации достаточно экзотической реализации *OpenCL* от *Texas Instruments*:

```
import pyopencl as cl
kcode = """kernel void test() { printf("Hi! (%d)\\n", get_group_id(0)); }"""
ctx   = cl.create_some_context()
Q     = cl.CommandQueue(ctx)
prg   = cl.Program(ctx, kcode).build(options="")
prg.test(Q, [8], [1]).wait()
```

Эта тестовая программка должна исполняться на реализации *OpenCL*, поддерживающей использование функции `printf()` в коде ядер, иначе результаты её работы увидеть не удастся!

Простота её достигнута тем, что никакие данные не передаются в *GPU* и не возвращаются оттуда; только работающие ядра «сообщают» информацию о себе. Но компиляция кода ядра здесь присутствует, контекст устройства *OpenCL* и очередь команд — создаются, а потому эта проверка работоспособности связки *Python*+*OpenCL* — вполне честная.

[12] Однако версия для процессоров *ARM* появляется лишь в репозитории *Ubuntu 17.10*.

Опрос и выбор платформ и устройств *OpenCL*

Реальной программе может потребоваться выбор среди имеющихся платформ и устройств; вот как можно узнать, что будет доступно программе с командной строки интерпретатора:

```
>>> import pyopencl
>>> from pyopencl.tools import get_test_platforms_and_devices
>>> get_test_platforms_and_devices()
```

Зная нумерацию платформ и устройств в системе, можно указывать устройства, необходимые для работы программы, значениями в переменных окружения в самой *Python*-программе (хотя это и весьма непредусмотрительно с точки зрения её переносимости):

```
import pyopencl as cl
import os

os.environ['PYOPENCL_COMPILER_OUTPUT'] = '1'
os.environ['PYOPENCL_CTX'] = '0:0'
        .         .         .
```

В данном примере первая переменная отвечает за выдачу сообщений при компиляции ядер, вторая указывает конкретное *OpenCL*-устройство.

> **Замечание.** Тонкость использования второй переменной окружения в том, что точный вид строки зависит от конкретного состава выбранной первой цифрой *OpenCL*-платформы: если в этой платформе одно устройство, то строка должна выглядеть как `'0'`, если больше, то надо указывать и вторую цифру (после двоеточия) — номер устройства.

Кроме того, эти (и другие) переменные окружения можно указывать прямо в командной строке при запуске скрипта (перед именем запускаемой программы, т. е., до `python`).

```
PYOPENCL_CTX = '0:0' python <ИмяСкрипта>.py
```

Если же создавать контекст с помощью функции `create_some_context()` (как это сделано в предыдущем примере, см. стр. 25), то пользователю будет предоставлена возможность выбора платформы и устройства прямо во время работы программы.

Оформление кода ядер в *PyOpenCL*

Обычный способ оформления пользовательского кода ядра в *PyOpenCL* реализуется в соответствии с такой схемой (здесь и далее предполагается, что модуль `pyopencl` включён под псевдонимом `cl`):

```
program = cl.Program(<Контекст>, <ЯдроКакТекстоваяСтрока>).build()
```

или просто с включением текста ядра в код *Python*-программы:

```
program = cl.Program(<Контекст>, """
__kernel void <ИмяЯдра>(<СписокФиктивныхПараметров>)
{
  <КодЯдра>
}""").build()
```

т. е., функция ядра «упакована» в т. н. *документирующую строку*[13] (`docstring`), которая является параметром вызова метода `Program()`, к возвращаемому результату применяется

[13] Тройные кавычки позволяют в этой документирующей строке иметь многострочный текст.

метод `build()` для «построения» программы, а в результате получается программный объект, который можно вызвать для исполнения кода ядра (или точнее — поставить в очередь на исполнение):

```
program.<ИмяЯдра>(<Очередь>,<Размерность>,None,<СписокПередаваемыхПараметров>)
```

Здесь *<ИмяЯдра>* — это имя функции-ядра в его тексте, *<Очередь>* — та очередь, в которую на исполнение передаётся объект ядра, *<Размерность>* — кортеж из одного, двух или трёх целых чисел, показывающий, в какой «глобальной» конфигурации должны быть запущены копии ядер («локальную» конфигурацию можно при этом не указывать, см. параметр `None`, так что об этом позаботится сама *OpenCL*-реализация). *<СписокПередаваемыхПараметров>* тут относится к параметрам исполняемого ядра.

Ядра для поэлементных алгоритмов

Ещё один вариант оформления ядер в *PyOpenCL* — т. н. ядра для поэлементных алгоритмов в модуле `pyopencl.elementwise`, когда выражения для обработки одного, двух или более массивов становятся сложнее; ядра при этом задаются своими отдельными фрагментами:

```
from pyopencl.elementwise import ElementwiseKernel
        .       .       .
<ОбъектЯдра> = ElementwiseKernel(<Контекст>,"<СписокФиктивныхПараметров>","<КодЯдра>",
                                 name="<ИмяЯдра>",preamble="<ПредварительныеДействия>")
```

Параметры ядра даёт *<СписокФиктивныхПараметров>*, *<КодЯдра>* содержит вычисляемое выражение, значение именованного параметра `name` — имя ядра для последующего исполнения, значение именованного параметра `preamble` — скажем, дополнительный заголовок.

Ссылки и примеры программ

PyOpenCL – Andreas Klöckner's wiki
https://wiki.tiker.net/PyOpenCL

PyOpenCL – Andreas Klöckner's web page
https://mathema.tician.de/software/pyopencl/

OpenCL integration for Python
https://github.com/inducer/pyopencl

Installing PyOpenCL on Windows
https://wiki.tiker.net/PyOpenCL/Installation/Windows

Installing PyOpenCL on Linux
https://wiki.tiker.net/PyOpenCL/Installation/Linux

Scripting GPUs with PyOpenCL
http://conference.scipy.org/scipy2010/slides/tutorials/andreas_kloeckner_pyopencl.pdf
Обучающий доклад (**Andreas Klöckner**). ***SciPy 2010****, Austin, Texas.*

PyOpenCL 2018.1.1 documentation
https://documen.tician.de/pyopencl/

Examples of PyOpenCL usage
https://wiki.tiker.net/PyOpenCL/Examples
https://github.com/inducer/pyopencl/tree/master/examples

https://github.com/tmramalho/easy-pyopencl
Полтора десятка примеров от ***Tiago Ramalho***. Интересно посмотреть у него и другие разделы.

12. *Numba*

Пакет ***Numba*** даёт возможность ускорить программы при помощи высокопроизводительных функций, написанных непосредственно на языке *Python*. Использование специальных аннотаций (декораторов с различными параметрами) при функциях, перегруженных вычислениями и/или обрабатывающих массивы, позволяет компилировать код прямо во время исполнения (***just-in-time***) в машинные инструкции, приближая производительность такого кода к производительности программ, написанных на языках *C/C++* и *Fortran*. Создан компанией ***Anaconda, Inc.*** (ранее — ***Continuum Analytics***).

Пакет *Numba* генерирует оптимизированный машинный код с помощью компилятора *LLVM*. Поддерживается компиляция кода *Python* для последующего исполнения как на *CPU*, так и на *GPU*: в виде ядер и функций для устройств *CUDA* от *NVidia* или ядер и функций для устройств *HSA* (*Heterogenous System Architecture*) от *AMD*.

Установка *Numba*

Пакет *Numba* совместим с *Python 2.7* и *3.4*, а также версиями *Numpy* от *1.7* до *1.13*. К настоящему времени выпущена стабильная версия *Numba 0.36.1* (декабрь 2017). Исходный код располагается по адресу `https://github.com/numba/numba`, документация доступна здесь: `http://numba.pydata.org/numba-doc/latest/`. Для инсталляции пакета из исходного кода понадобится *C*-компилятор, соответствующий версии *Python*, а также пакеты `numpy` и `llvmlite`, но это не самый простой путь. Намного проще для работы с *Numba* установить ***Anaconda Distribution*** или хотя бы ***Miniconda*** (содержит пакетный менеджер *Conda* и дистрибутив *Python*). Проверить результаты установки можно с помощью команд `python -c 'import numba; print(numba.__version__)'` и `pycc --help`.

Пример оформления кода в *Numba*

Простой пример оформления кода с помощью *Numba* (взят из текста документации):

```
from numba import jit
from numpy import arange

@jit
def Sum2D(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(Sum2D(a))
```

Декоратор `@jit` перед функцией предписывает *Numba* компилировать её; типы параметров станут известны *Numba* при вызове функции.

Создание модулей расширения

Помимо компиляции кода во время исполнения (*just-in-time*) возможна также и компиляция до исполнения (*ahead of time*, *AOT*), в результате которой создаётся модуль расширения, уже не зависящий от *Numba*, который можно распространять, не предполагая

у пользователя наличия *Numba* (хотя присутствие у него *NumPy*, разумеется, обязательно). Простой самодостаточный пример из руководства (обратите внимание, что здесь надо явно указывать сигнатуры — имена и типы параметров — определяемых функций!):

```
from numba.pycc import CC

cc = CC('my_module')
#cc.verbose = True

@cc.export('multf', 'f8(f8, f8)')
@cc.export('multi', 'i4(i4, i4)')
def mult(a, b):
    return a * b

@cc.export('square', 'f8(f8)')
def square(a):
    return a ** 2

if __name__ == "__main__":
    cc.compile()
```

Если запустить эту программу, будет сгенерирован модуль расширения с именем `my_module` (для вывода информации при компиляции строку `cc.verbose = True` можно раскомментировать). В зависимости от используемой платформы реальное имя модуля может быть `my_module.so`, `my_module.pyd`, `my_module.cpython-34m.so` и т. п. В этом модуле будут присутствовать три функции: `multi()`, `multf()` и `square()`. Первая работает с 32-битными целыми значениями, остальные — с вещественными двойной точности. Задействуются эти функции путём импорта созданного модуля (`import my_module; my_module.multi(3,4)`).

Примеры использования *Numba* от изготовителя

Фирма ***Continuum Analytics*** имеет на сайте GitHub раздел с примерами, называемый исторически `numbapro-examples` (https://github.com/ContinuumIO/numbapro-examples), хотя сейчас все они действительны и для *Numba*. Там имеет смысл посмотреть примеры использования декоратора `cuda.jit` (каталог `cudajit`, файлы: `matmul.py`, `matmul_smem.py`, `max.py`, `sum.py`), использования декоратора `vectorize` (каталог `vectorize`, содержащий файлы `cuda_polynomial.py`, `cuda_vectorize.py`, `polynomial.py`, `scalar_broadcasting.py`, `sum.py`), использования декоратора `guvectorize` (каталог `guvectorize`, `sumrows.py`), а также варианты формирования множества Мандельброта с применением декораторов `jit` и `vectorize` (каталог `mandel`, файлы `mandel_autojit.py`, `mandel_vectorize.py`) и варианты решения двумерного уравнения Лапласа (каталог `laplace2d`, файлы `laplace2d-numba.py`, `laplace2d-numba-gpu.py`, `laplace2d-numba-gpu-improve.py`); получить полезные сведения по работе с «закреплённой» памятью в *CUDA* (каталог `cuda_memory`, файл `pinned.py`) и замеру времени работы ядер (каталог `cuevents`, файл `kernel_timing.py`).

12.1. Декораторы в *Numba*

Декоратор `jit`

Простейший способ работы с *Numba* — это применять декорирование с помощью `@jit` (как в приведённом ранее простом примере), предписывающее *Numba* компилировать помеченную функцию, используя заданные при её вызове типы параметров. Можно также указать заранее, для каких типов параметров компилировать функцию, — с помощью строки, именуемой сигнатурой функции и передаваемой декоратору, в которой ключевыми словами типов (или их сокращениями) указаны типы возвращаемого и передаваемых параметров.

Декорирование функций с помощью `@jit` обеспечит работу через *CUDA*-устройство только при соответствующем целевом устройстве (`target="cuda"`). Декоратор же `cuda.jit` предназначен в первую очередь для оформления функций-ядер, поскольку только тогда поддерживаются встроенные переменные вроде `threadIdx` и спецификаторы типа `__shared__`. Поэтому, если необходимо написать аналог *CUDA*-ядра, надо использовать декорирование `@cuda.jit`. В остальных случаях — для ускорения существующего кода на *GPU* — следует применять `@jit` и `target="cuda"`.

Важная характеристика *Numba* — это поддержка массивов *NumPy*. Приведённый ниже пример показывает, как указать, что компилировать надо функцию, принимающую в качестве параметра *NumPy*-массив вещественных чисел двойной точности:

```
from numba import jit

@jit("f8(f8[:])")
def Sum1D(Array):
    .    .    .
```

Здесь входной параметр определён как строка `"f8[:]"`, что означает одномерный массив восьмибайтовых вещественных чисел. Двумерный массив был бы задан как `"f8[:,:]"`, трёхмерный — как `"f8[:,:,:]"` и т.д.

Встроенные в *Numba* элементарные типы и их сокращения перечислены в таблице:

Имя типа	Сокращение	Результирующий тип
boolean	b1	uint8 (char)
bool_	b1	uint8 (char)
byte	u1	unsigned char
uint8	u1	uint8 (char)
uint16	u2	uint16
uint32	u4	uint32
uint64	u8	uint64
char	i1	signed char
int8	i1	int8 (char)
int16	i2	int16
int32	i4	int32
int64	i8	int64
float_	f4	float32
float32	f4	float32
double	f8	float64
float64	f8	float64
complex64	c8	float complex
complex128	c16	double complex

Сигнатуры функций могут быть также выражены объектами типов и непосредственно, а не в составе строки описания, если они соответствующим образом импортированы:

```
from numba import jit, f8

@jit(f8(f8[:]))
def Sum1D(Array):
    .    .    .
```

Если *Numba* не удаётся понять тип переменной или какой-то тип не поддерживается, то просто используются объекты языка *Python*, что, конечно, снижает быстродействие. Однако, можно и запретить использование объектов *Python* с помощью специального флага `nopython` в декораторе:

```
@jit(nopython=True)
def Sum1D(Array):
    .    .    .
```

Декоратор с таким значением этого флага уже некоторое время существует под именем `@njit`. Его использование замечено в опубликованных фрагментах кода одного из разработчиков *Numba*: ***Siu Kwan Lam*** (https://gist.github.com/sklam, программа `raytracing.py`).

Декоратор `cuda.jit`

С некоторых пор *Numba* может взаимодействовать с устройствами *CUDA* и загружать в них для исполнения т. н. ***PTX***-код. Большая часть программного интерфейса *CUDA* доступна через модуль `numba.cuda`, подключаемый с помощью импорта вида `from numba import cuda`.

Кроме того, ядра *CUDA* и функции, исполняемые на устройстве *GPU*, могут быть откомпилированы с помощью декораторов из этого модуля. Например, для ядра с двумя параметрами типа одномерных массивов можно использовать декорирование с помощью `@cuda.jit`:

```
@cuda.jit("void(int32[:], int32[:])")
def kfunc(ArrayA, ArrayB):
    .    .    .
```

Запуск на исполнение этого ядра осуществляется так:

```
GrDim = 100, 200
BlkDim = 16, 16
kfunc[GrDim, BlkDim](ArrA, ArrB)
```

что соответствует такому коду на *CUDA C*:

```
dim3 GrDim(100, 200);
dim3 BlkDim(16, 16);
kfunc<<<GrDim,BlkDim>>>(ArrA, ArrB);
```

Если необходимо скомпилировать функцию для работы на устройстве, в декоратор добавляется параметр `device=True`:

```
@cuda.jit("int32(int32[:], int32[:])", device=True)
def somefunc(Array1, Array2):
    .    .    .
```

Некоторое время существовало (а сейчас не рекомендуется к использованию) «упрощённое» декорирование с помощью `@cuda.autojit` для компиляции ядер и сигнатура ядра там отсутствовала, поскольку определялась во время вызова ядра на исполнение.

Декоратор `hsa.jit`

Для исполнения ядер *HSA* на устройствах *GPU* от *AMD* используется декоратор `hsa.jit`. Синтаксис их запуска на исполнение аналогичен синтаксису запуска ядер *CUDA*.

```
from numba import hsa

@hsa.jit
def matmulfast(A, B, C):
    .    .    .
```

Декоратор `vectorize`

Декорирование вида `@vectorize` создаёт объект `ufunc` универсальной функции *NumPy* (***NumPy Universal Function object***) из функции *Python*. Такая `ufunc` может быть перегружена, чтобы воспринимать многочисленные комбинации типов своих параметров. Пользователю необходимо указать список «типов» функций первым параметром декоратора.

```
from numba import vectorize

@vectorize(['int8(int8,int8)',
            'int16(int16,int16)',
            'int32(int32,int32)',
            'int64(int64,int64)',
            'f4(f4,f4)',
            'f8(f8,f8)'])
def add(x, y):
    return x + y
```

Декоратор `guvectorize`

Декорирование вида `@guvectorize` создаёт объект `gufunc` т. н. обобщённой универсальной функции *NumPy* (***NumPy Generalized Universal Function object***) из функции *Python*. Здесь будет необходим дополнительный параметр, описывающий размерности входных и выходных величин, поскольку `@guvectorize` (в отличие от `@vectorize`) работает не со скалярными параметрами, а с аргументами-массивами. Вот как может начинаться определение функции перемножения двух матриц:

```
from numba import guvectorize

@guvectorize(['void(float64[:,:], float64[:,:], float64[:,:])'],
             '(m,n),(n,p)->(m,p)')
def matmul(A, B, C):
    m, n = A.shape
    n, p = B.shape
    .    .    .
```

Целевое устройство

«Общие» декораторы `jit`, `autojit`, `vectorize`, `guvectorize` могут иметь поименованный параметр `target="..."` для выбора целевого устройства. Первоначально в *Numba* поддерживалось лишь одно целевое устройство: `cpu`. Впоследствии были добавлены `parallel` и `cuda`. Первое используется только совместно с декоратором `vectorize`, тогда работа распределяется между различными потоками *CPU*. Второе заставляет вынести вычисления на *NVidia GPU*.

Ещё один пример оформления функции с помощью декоратора `guvectorize` (сигнатуры здесь указаны уже не с помощью текстовой строки, а импортированными объектами типов; присутствует и параметр целевого устройства):

```
from numba import guvectorize, complex64, int32

@guvectorize([(complex64[:], int32[:], int32[:])],
             '(n),(n)->(n)', target='cuda')
def mandelbrot_numpy(c, maxit, output):
    .    .    .
```

13. *Theano*

Theano — библиотека *Python* с лицензией BSD, позволяющая определять, оптимизировать и вычислять математические выражения, в том числе и с многомерными массивами `numpy.ndarray`. Она сочетает в себе свойства системы компьютерной алгебры и оптимизирующего компилятора. Создана в университете Монреаля (группа ***Yoshua Bengio***).

В данный момент у *Theano* официально имеется только *CUDA backend* (это означает, что для работы на *GPU* понадобится графическая карта с поддержкой *CUDA*), работающий лишь для тензоров с типом элементов `float32`. Ведётся также работа по созданию *OpenCL backend*, но пока она не завершена[14].

Установка *Theano* (`http://deeplearning.net/software/theano/install.html`) подробно описана в документации и для *Windows*, и для *Mac OS*, и для некоторых вариантов *Linux* (*Ubuntu* и *CentOS 6*), но если говорить кратко, то может быть сделана средствами *Python* (`pip install theano`) — с административными правами или без — при условии, что уже как минимум установлены ***NumPy***, ***SciPy*** и ***BLAS*** (для работы с *GPU* нужен и ***g++***).

Простейшие примеры

Определяется функция суммирования двух величин и затем используется для вычислений.

```
import theano

a = theano.tensor.dscalar()
b = theano.tensor.dscalar()
c = a + b
f = theano.function([a,b],c)
```

```
>>> print(f(1.414,2.236))
3.65
>>>
```

В принципе, *Theano* — не язык программирования в обычном смысле слова, поскольку построением выражения для *Theano* занимается программа на языке *Python*. Тем не менее, некоторые элементы нового «языка» здесь имеются:

- «объявляются» переменные (`a`,`b`) и указываются их типы;
- строятся выражения, показывающие, как комбинируются переменные;
- графы выражений компилируются и далее их можно использовать для вычислений.

Можно представлять себе `theano.function()` как интерфейс к компилятору, создающему вызываемый объект из чисто символьного графа. Главная особенность *Theano* — этот граф оптимизируется и преобразуется в машинные инструкции.

Функция «обращения» последовательности. Предыдущий пример предполагал создание отдельного файла с кодом и эксперименты с ним после подгрузки файла через `import`, однако можно экспериментировать и прямо в интерактивной сессии интерпретатора (обратите внимание, что для удобства здесь импортируется чуть больше и немного по-другому; многоточиями здесь заменён вывод *Theano* о компиляции кода)

```
>>> from theano import tensor as T
. . .
>>> from theano import function
>>> z = T.vector()
>>> Flip = function([z],z[::-1])
. . .
>>> Flip([1,3,5,7,9])
array([ 9.,  7.,  5.,  3.,  1.], dtype=float32)
>>>
```

[14] 28.09.2017 стало известно, что после выпуска версии *1.0* дальнейшая разработка *Theano* прекращается.

Итак, из модуля `theano` импортируются: подмодуль `tensor` под именем `T` и подмодуль `function`. Величина с именем `z` является «тензорным» вектором, а `Flip()` — функцией, принимающей такой вектор и возвращающей его расположенным в обратном порядке. После определения такой функции вызывается компилятор *Theano* (строки его сообщений заменены выше многоточиями) и превращает её в код для *GPU*, которым теперь можно пользоваться: функция именуется `Flip()` и принимает в качестве единственного входного параметра вектор. Видно, что результат её вызова соответствует назначению функции. Что ещё интереснее — эта функция теперь часть сессии интерпретатора и будет доступна далее для обработки любых передаваемых ей векторов (попробуйте в качестве упражнения что-нибудь вроде `Flip([N for N in range(1000)])`).

Разумеется, возможно определение и более сложных функций, например, вычисляющих более одной величины, имеющих параметры с величинами по умолчанию, а также обрабатывающих более «сложные» входные величины.

Пример поэлементной обработки массивов

Определена функция, возвращающая два значения различных функций класса сигмоид с одним параметром `A` и обрабатывающая поэлементно произвольную матрицу `x`. Скалярный параметр `A` является необязательным и имеет значение `1` по умолчанию. Обратите внимание, что функция вычисления абсолютной величины имеет имя `abs_()`!

```
import theano
import theano.tensor as T
from theano import In
from theano import function

x = T.dmatrix('x')
A = T.dscalar('A')
s1 = 1 / (1 + T.exp(-A*x))
s2 = x / (T.abs_(x) + A)
S = function([x, In(A, value=1)],
             [s1, s2])
```

```
>>> S([[0,1,2],[1,2,3],[2,3,4]])
[array([[ 0.5       ,  0.73105858,  0.88079708],
       [ 0.73105858,  0.88079708,  0.95257413],
       [ 0.88079708,  0.95257413,  0.98201379]]),
 array([[ 0.        ,  0.5       ,  0.66666667],
       [ 0.5       ,  0.66666667,  0.75      ],
       [ 0.66666667,  0.75      ,  0.8       ]])]
>>> S([[0,1,2],[1,2,3],[2,3,4]],2)
[array([[ 0.5       ,  0.88079708,  0.98201379],
       [ 0.88079708,  0.98201379,  0.99752738],
       [ 0.98201379,  0.99752738,  0.99966465]]),
 array([[ 0.        ,  0.33333333,  0.5       ],
       [ 0.33333333,  0.5       ,  0.6       ],
       [ 0.5       ,  0.6       ,  0.66666667]])]
>>>
```

Циклы в *Theano* — `scan`-функции

Эти функции обеспечивают базовую функциональность, необходимую для организации циклов в *Theano*. Для более комфортного «погружения» в эту тему полезно будет начать с простого примера.

Предположим, что для данного `k` надо получить в цикле из тензора `A` его степень `A**k`. Код на языке *Python* мог бы выглядеть так:

```
result = 1
for i in range(k):
  result = result * A
```

Здесь необходимы: переменная `result`, где сначала хранится первоначальная величина и потом осуществляется накапливание вычисляемого значения, и неизменная величина `A`. Неизменяемые величины передаются `scan` как `non_sequences`. Инициализация происходит в `outputs_info`, а накопление осуществляется автоматически.

Эквивалентный код *Theano*[15] (с дополнительным выводом результатов):

[15] Здесь и в примерах далее, приводя фрагменты кода *Theano*, мы будем считать, что необходимые команды импорта — как правило, это `import theano; import theano.tensor as T` — уже выполнены.

```
k = T.iscalar("k")
A = T.vector("A")

result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)

final_result = result[-1]

power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)

print(power(range(10),2))
print(power(range(10),4))
```

Символическое описание результата производится при задании параметров `scan`. Конструируется (с помощью лямбда-выражения) функция, которой передаются `prior_result` и `A`, а возвращается `prior_result * A`. Порядок параметров фиксирован: выход предшествующего вызова `fn` (или первоначально — инициализирующее значение), затем все `non_sequences`.

Затем инициализируется выход как тензор той же формы и типа, что и `A`, заполненный единицами. `A` передаётся `scan` как не параметр последовательности (*non sequence*), задаётся также число шагов `k` для итераций лямбда-выражения.

Возвращает `scan` кортеж (`tuple`), содержащий результат `result` и словарь обновлений (в данном случае — пустой). Заметим, что результат — не матрица, а $3D$-тензор, содержащий величину `A**k` для каждого шага. Нам нужна последняя величина (после `k` шагов), так что мы компилируем функцию, чтобы она возвращала именно это. Существующая оптимизация позволит обнаружить, что необходима только последняя величина, и промежуточные результаты после использования сохраняться не будут. Таким образом, можно не беспокоиться, если `A` и `k` будут большими.

Итерирование по первому измерению тензора

В дополнение к повторению фиксированное число раз `scan` «способен» итерировать по ведущему измерению тензора (аналогично конструкции `for x in a_list`).

Завершение scan по условию

Можно также использовать `scan` как блок типа `repeat-until`. В таких ситуациях `scan` остановится тогда, когда либо будет достигнуто максимальное число итераций, или когда заданное условие будет выполнено.

```
def power_of_2(previous_power, max_value):
    return previous_power*2, theano.scan_module.until(previous_power*2 >
                                                      max_value)

max_value = T.scalar()
values, _ = theano.scan(power_of_2,
                        outputs_info = T.constant(1.),
                        non_sequences = max_value,
                        n_steps = 1024)

f = theano.function([max_value], values)

print(f(45))
```

Как можно видеть, для остановки по условию внутренней функции необходимо вернуть также условие, «обёрнутое» `theano.scan_module.until`. Это условие должно выражаться в терминах параметров внутренней функции (в данном случае — `previous_power` и `max_value`).

Конфигурирование *Theano*

Модуль `theano.config` содержит много величин, способных изменять поведение *Theano*. Как правило, эти величины не должны модифицироваться пользовательским кодом. Они имеют значения по умолчанию, но их можно изменить с помощью `.theanorc`-файла, а заданные в нём значения в свою очередь могут быть переопределены с помощью переменной окружения `THEANO_FLAGS`. Порядок приоритета конфигурационных величин таков:

1. Присваивание `theano.config.<`*`Свойство`*`>`
2. Указание значения в `THEANO_FLAGS`
3. Указание значения в файле `.theanorc`[16] (или другом, заданном в `THEANORC`)

Узнать значения параметров текущей конфигурации можно с помощью такой командной строки: `python -c "import theano; print theano.config"`.

> Если *Theano* использует *GPU*, то после импорта пакета (т.е. исполнения строки `import theano`) выводится сообщение «*Using gpu device ...*». Если сообщений такого вида нет, это значит, что *Theano* использует *CPU* (поведение по умолчанию).

Минимально необходимое содержимое конфигурационного файла `.theanorc` таково:

```
[global]
device = gpu
floatX = float32
```

Здесь указано, что надо использовать *GPU*, а вещественные числа — одинарной точности. Без этих параметров *Theano* будет использовать *CPU*.

Кроме этого, *Theano* необходим компилятор из *Visual Studio* (а если его *C++*-возможности не соответствуют современным требованиям, то и отдельный *C++*-компилятор, скажем, ***g++*** версии 4.2 или выше).

Тензорные типы в *Theano*

Theano «сфокусирован» на поддержке матричных символьных выражений. Запись вида `x = T.fmatrix()` создаёт экземпляр тензорной переменной (`TensorVariable`), а объект `T.fmatrix` — экземпляр тензорного типа (`TensorType`).

Создание тензорных переменных может осуществляться с помощью разных конструкторов, однако все они имеют необязательный поименованный параметр `name` — для облегчения отладки, а тип (`dtype`) по умолчанию будет взят из конфигурационных установок:

```
theano.tensor.<ТипТензора>(name=None , dtype=config.floatX )
```

Список типов тензоров (возможных конструкторов) с типом данных, заданным по умолчанию: `scalar` (нульмерный `ndarray`), `vector` (одномерный `ndarray`), `row` (двумерный `ndarray` с числом строк, равным 1), `col` (двумерный `ndarray` с числом столбцов, равным 1), `matrix` (двумерный `ndarray`), `tensor3` (трёхмерный `ndarray`), `tensor4` (четырёхмерный `ndarray`), `tensor5` (пятимерный `ndarray`).

[16] Конфигурационный файл `.theanorc` должен располагаться в домашнем каталоге пользователя (т. е., `~` — в *Linux*, `C:\Documents and Settings\<`*`ИмяПользователя`*`>` или аналогичном — в *Windows*).

Кроме этого, конструктор может включать в имя (в самом его начале) букву типа данных: `b` (соответствует типу `int8`), `w` (`int16`), `i` (`int32`), `l` (`int64`), `d` (`float64`), `f` (`float32`), `c` (`complex64`), `z` (`complex128`) — так, как это было сделано в приводимом выше примере записи создания тензорной переменной.

Имеется некоторое количество множественных конструкторов, создающих сразу несколько переменных; в основном, они используются для экономичной записи (мы перечислим их вот так: `{i,l,f,d}{scalars,vectors,rows,cols,matrices}`, подразумевая, что возможно произвольное сочетание буквы типа из первой фигурной скобки с именем множественного типа из второй фигурной скобки для получения в результате имени конструктора).

> Всегда, когда `ndarray` или число из *Python* используются совместно с экземплярами тензорных переменных в арифметических выражениях, результат тоже будет тензорной переменной. Это происходит потому, что *Theano* автоматически создаёт «свою» копию `ndarray` и «оборачивает» константу в `TensorConstant`.
>
> Из-за копирования массивов *NumPy* для использования в компилируемых выражениях последующие изменения в *NumPy*-версиях массивов никак не скажутся на выражении *Theano*.

Методы, применимые к тензорной переменной чаще всего эквивалентны методам из *NumPy*:

`reshape(shape, ndim=None)` Возвращает вид тензора изменённой формы сообразно значению параметра `shape`. Если он задан переменной величиной, возможно, понадобится указать значение параметра `ndim`.

`dimshuffle(*pattern)` Возвращает вид тензора с переставленными измерениями согласно указанному шаблону. Обычно такой шаблон включает целые от `0` до `ndim-1` и любое число символов `'x'` в тех измерениях, где данный тензор может быть расширен.

Несколько примеров шаблонов и их эффект:

`('x')`	*создаёт скаляр в одномерном векторе* (?)
`(0,1)`	*тождественное преобразование для одномерного вектора*
`(1,0)`	*переставляет первое и второе измерения*
`('x',0)`	*создаёт строку из одномерного вектора*
`(0,'x')`	*создаёт столбец из одномерного вектора*
`(2,0,1)`	A×B×C *превращается в* C×A×B
`(0,'x',1)`	A×B *превращается в* A×1×B
`(1,'x',0)`	A×B *превращается в* B×1×A
`(1,)`	*удаляет измерение* 0 *(оно должно быть расширяемым)*

`flatten(ndim=1)` Возвращает вид тензора с заданным количеством измерений.

`ravel()` Для совместимости с *NumPy* возвращает `self.flatten()`.

`T` Транспонирование тензора

> **Замечание.** Следует иметь в виду, что в *NumPy* и *Theano* транспонирование вектора даёт тот же самый вектор. Надо использовать `.reshape()` или `.dimshuffle()` для превращения этого вектора в строку или столбец.

Разделяемые переменные

Разделяемые переменные (***shared variables***) позволяют определять функции, имеющие внутренние состояния. Таким образом различные функции могут совместно использовать какие-то данные. Разделяемые переменные могут использоваться в символьных выражениях, но имеют также и значение. Это значение можно получить с помощью `.get_value()` и изменить с помощью `.set_value()`.

Рассмотрим простой пример — аккумулятор. Первоначальное его состояние равно нулю, но после каждого вызова функции оно увеличивается на заданную при вызове величину:

```
import theano
import theano.tensor as T
from theano import function

state = theano.shared(0)
inc = T.iscalar('inc')
accumulator = function([inc],state,
      updates=[(state,state+inc)])
```

```
>>> state.get_value()
0
>>> accumulator(1)
array(0)
>>> state.get_value()
1
>>> accumulator(100)
array(1)
>>> state.get_value()
101
```

Параметр `updates` получает список пар *параметр — новое выражение* в качестве значения. Во время исполнения функции величина каждой разделяемой переменной заменяется в соответствии с результатом вычисления «своего» выражения.

Компиляция символьного графа — `theano.function()`

Взаимодействие пользователя с компилятором символьных графов задаваемых выражений осуществляется через `theano.function()`, которая создаёт (и возвращает) вызываемый объект, используемый далее для вычислений.

```
theano.function(inputs, outputs, mode=None, updates=None, givens=None,
                no_default_updates=False, accept_inplace=False, name=None,
                rebuild_strict=True, allow_input_downcast=None, profile=None,
                on_unused_input='raise')
```

Функция возвращает вызываемый объект, вычисляющий `outputs` на основе `inputs`, а также обновляет неявные параметры функции, соответствующие `updates`.

Смысл и возможные значения параметров этой функции:

`params` (список переменных или экземпляров объекта `In`[17] (ранее — `Param`), без разделяемых переменных) — у функции будут параметры из указанного списка.

`outputs` (список переменных или экземпляры объекта `Out`[18]) выражения для вычислений.

`mode` (`None`, строка или экземпляр объекта `Mode`) — режим компиляции.

`updates` (что-нибудь, итерируемое по парам (*РазделяемаяПеременная*,*НовоеВыражение*): список, кортеж или словарь) — выражения для новых величин разделяемых переменных.

`givens` (что-нибудь, итерируемое по парам (*Var1*,*Var2*) переменных: список, кортеж или словарь) — специфические подстановки, которые необходимо сделать в вычислительном графе (величина *Var2* замещает величину *Var1*). Не рекомендуется делать подстановки взаимозависимыми!

`no_default_updates` (булевская величина или список переменных) — если `True`, то автоматическое обновление переменных не осуществляется. Если `False` (значение по умолчанию), то обновляются все переменные. В остальных случаях производится обновление всех переменных, не входящих в списки `updates` и `no_default_updates`.

`name` — необязательное имя для функции. Режим профилирования будет выводить время, проведённое в этой функции.

[17] `In` (`Param`) — класс для добавления параметров к входным величинам.
[18] `Out` — класс для добавления параметров к входным величинам.

`rebuild_strict` — значение по умолчанию `True`, оно является наиболее безопасным и протестированным.

`allow_input_downcast` (булевская величина или `None`) — значение `True` означает, что величины, передаваемые как `inputs`, могут быть без уведомления преобразованы к типу соответствующей переменной, что часто приводит к потере точности. Значение `False` означает, что приведение типа будет производиться к более общему (более точному) типу. `None` (значение по умолчаниию) почти не отличается от `False`, но разрешается преобразование скаляров *Python* к типу `floatX`.

`profile` (`None`, `True` или экземпляр `ProfileStats`) — накапливается профилирующая информация.

`on_unused_input` — показывает, что будет сделано, если какая-либо переменная из списка не используется в графе. Возможные значения: `"raise"` (значение по умолчанию), `"warn"`, `"ignore"`.

После очередного исполнения функции механизм `updates` замещает величину каждой разделяемой переменной (неявного входного значения) новыми значениями, вычисленными на основании выражений в списке `updates`.

Рекомендуемые примеры программ

Bistable.py,
http://www.nehalemlabs.net/prototype/blog/2013/10/17/
Примеры решения стохастических дифференциальных уравнений с помощью *Theano*.

> В программе, названной здесь **Bistable.py**, имеется небольшая неточность, которая проявилась при проверке программы на *Mac*: `downsample_factor_t`, вычисляемый там как частное `0.1/dt0` (где значение `dt0` тоже инициализировано величиной 0.1), используется далее для индексации массива, что, конечно, не очень хорошо и справедливо «не нравится» версии *Python* для *Mac*; проще всего сразу привести это частное к целой величине: `int(0.1/dt0)`.

https://pastebin.com/x6WREp7D
Подобный пример фильтрации границ, написанный для *Theano*, можно найти в нескольких вариантах, но эта ссылка — одна из первых, хотя, возможно, и не первоисточник (можно искать **LeCun Local Contrast Normalization**). Требует наличия изображения для обработки (используется применяемая во многих исследованиях классическая иллюстрация, часто именуемая `Lena.jpg` или аналогично); в тексте, возможно, придётся исправить местоположение и имя файла.

Интересные статьи и руководства

http://www.marekrei.com/blog/theano-tutorial/
Здесь не только много простых примеров програм для *Theano*, это также отличный вводный курс.

THEANO: NUMERICAL COMPUTATION IN PYTHON
http://xcorr.net/2014/01/26/theano-numerical-computation-in-python/

Basic Tensor Functionality
http://deeplearning.net/software/theano/library/tensor/basic.html

scan — Looping in Theano
https://theano.readthedocs.io/en/rel-0.6rc3/library/scan.html

function — defines theano.function
http://deeplearning.net/software/theano/library/compile/function.html

More Examples
http://deeplearning.net/software/theano/tutorial/examples.html

14. Привязка *Python* к *ArrayFire*

ArrayFire — высокопроизводительная библиотека для научных вычислений с простым программным интерфейсом. Программы, написанные с её помощью, могут работать через реализации *CUDA*, *OpenCL* или применяя векторные операции современных *CPU* (т. е., будут использовать соответствующие устройства как т. н. ***backend***[19]). Функциональность *ArrayFire*: векторные алгоритмы, обработка изображений/сигналов, компьютерное зрение, линейная алгебра, статистика. Создана и развивается компанией *AccelerEyes* практически со времён возникновения *GPU*. Сначала для работы с библиотекой предполагалось получение пользовательской лицензии (без которой она была неработоспособна), но в 2014 году было решено сделать её исходный код открытым — и вот в июне 2015 года появилась версия *3.0.0* (актуальная версия на момент написания пособия — *3.6.0*).

В библиотеке *ArrayFire* всё «построено» вокруг единственного объекта-массива `array`: контейнера, который может хранить величины комплексные или вещественные (одинарной и двойной точности), а также целочисленные (со знаком или без знака) и булевские. Массивы `array` являются многомерными, их данные хранятся *на обрабатывающем устройстве*.

ArrayFire распространяется вместе с графической библиотекой, именуемой *Forge*, которая спроектирована так, чтобы проще отображать данные, находящиеся на *GPU* (графические примитивы воспроизводятся из памяти *GPU*). Тем самым и вычисления, и визуализация могут использовать *GPU* — без необходимости лишний раз копировать данные между *CPU* и графическим устройством. Для большей переносимости *Forge* использует кроссплатформенные зависимости (***GLEW***, ***GLFW***, ***Freetype***, ***fontconfig***, ***OpenGL 3.3***).

Привязка *Python* к *ArrayFire* состоит из некоторого набора модулей, соответствующих исходным `.py`-файлам: `algorithm` — векторные алгоритмы (`sum`, `min`, `sort` и т. п.); `arith` — математические функции (`sin`, `sqrt`, `exp` и т. д.); `array` — класс `Array` и вспомогательные функции; `base` — реализация класса `BaseArray`; `bcast` — функция выполнения «расширительных» операций (над массивами различающихся размеров); `blas` — *BLAS*-функции (`matmul`, `dot` и др.); `cuda` — функции *CUDA*-бэкенда; `data` — функции создания массивов и работы с ними; `device` — функции работы с устройствами выбранного бэкенда; `features` — класс `Features`, используемый для алгоритмов «компьютерного зрения»; `graphics` — графические функции (`plot`, `image` и др.); `image` — функции обработки изображений; `index` — классы `Index` и `Seq`, используемые в операциях индексации; `interop` — взаимодействие с другими пакетами *Python*; `lapack` — функции линейной алгебры для плотных матриц (`solve`, `inverse` и т. д.). `library` — модуль, содержащий перечисления и другие константы; `opencl` — функции *OpenCL*-бэкенда; `random` — функции генерации случайных чисел; `sparse` — функции для работы с разреженными матрицами; `signal` — функции обработки сигналов (`fft`, `convolve` и пр.); `statistics` — статистические алгоритмы (`mean`, `var`, `stdev` и т. д.); `timer` — таймерные функции; `util` — вспомогательные функции работы с метаданными `Array`; `vision` — функции «компьютерного зрения» (*FAST*, *ORB* и т. п.).

Установка и проверка

Так как пока нельзя установить *Python*-привязку к библиотеке без неё самой, необходимо сначала установить *C-C++*-библиотеку *ArrayFire*, а только потом — выполнить команду `pip install arrayfire` для установки этой привязки из репозитория ***PyPI*** или загрузить исходный код с ***GitHub*** и запустить `python setup.py install` в каталоге пакета `arrayfire-python` после распаковки.

[19] По умолчанию *backend* для работы ***ArrayFire*** выбирается (по имеющимся библиотекам) в порядке такого приоритета: *CUDA*, *OpenCL*, *CPU*; его можно также указать, вызывая функцию `set_backend(<ТипУстройства>)`, где в качестве типа устройства указано `'cuda'`, `'opencl'` или `'cpu'`.

Общая схема установки этой библиотеки (как, впрочем, и любого другого готового продукта) такова: установить необходимые зависимости для используемой системы (если их установка в дистрибутиве не предусмотрена; в данном случае это так), скачать нужный дистрибутив, установить его и протестировать полученную инсталляцию. Возможен также и вариант её самостоятельной компиляции из исходного кода, но это обычно сложнее и требует некоторого опыта — чтобы понять, что следует делать в случае неудачи.

Наиболее просто установить библиотеку *ArrayFire*, если воспользоваться готовым дистрибутивом с сайта `https://arrayfire.com/download/`; там они имеются для *Windows*, *Linux*, *OSX*, а также для *ARM*-версии *Linux* для процессоров ***Tegra*** (от *NVidia*): ***K1*** (поддерживается более старыми версиями *ArrayFire*), ***X1***, ***X2***. Преимущество использования уже откомпилированного варианта: код будет включать *Intel Math Kernel Library (MKL)*; она ускоряет функции линейной алгебры.

Если при работе с ***ArrayFire*** предполагается использовать *CUDA* или *OpenCL*, то соответствующие средства должны быть уже установлены (включая во втором случае т. н. ***ICD Loader***; в *Ubuntu* или *Debian* пакет называется `ocl-icd-libopencl1`).

В системах *Windows* перед установкой надо убедиться, что имеются библиотеки времени исполнения от *Visual Studio 2015*; после установки *ArrayFire* каталог `%AF_PATH%\lib` (где располагаются её динамические библиотеки) следует добавить в переменную окружения `PATH`.

Поддержка высокопроизводительной визуализации через *Forge* влечёт зависимости, которые указаны выше; пакеты для *Ubuntu*/*Debian* называются `libfreeimage-dev`, `libatlas3gf-base`, `libfftw3-dev`, `libglew-dev`, `libglewmx-dev`, `libglfw3-dev`, в других *Linux*-системах пакеты могут называться по-другому, скажем, `freeimage`, `atlas`, `fftw`, `libGLEW`, `libGLEWmx`, `glfw`.

В *Ubuntu 14.04* и ранее пакета `libglfw3-dev` нет в репозитории, в этих случаях его придётся устанавливать из *PPA* (возможный вариант здесь — `ppa:keithw/glfw3`) или исходного кода.

ARM-версиям *Linux* для *Tegra* дополнительно понадобятся `libatlas-dev` и `liblapacke-dev`.

В системах *Linux* рекомендуется устанавливать *ArrayFire* в каталог `/usr/local` — чтобы все подключаемые файлы и разделяемые библиотеки находились в стандартных путях их поиска:

```
./arrayfire_<Версия>_Linux_x86_64.sh --exclude-subdir --prefix=/usr/local
```

Если же установка произведена в другое место, то для того, чтобы загрузчик разделяемых библиотек находил библиотеки *ArrayFire*, нужно добавить их местоположение через `ldconfig` либо разместить его в переменной окружения `LD_LIBRARY_PATH`.

Начать эксперименты с привязкой можно и в командной строке интерпретатора, задавая разные величины пакета или вызывая «справочные» функции (ответы могут отличаться!):

Импорт привязки с псевдонимом.
```
>>> import arrayfire as af
```
«Старшая» версия библиотеки?
```
>>> af.AF_VER_MAJOR
```
3
```
'3'
```
Найден ли пакет *Numba*?
```
>>> af.AF_NUMBA_FOUND
```
Нет
```
False
```
Найден ли пакет *NumPy*?
```
>>> af.AF_NUMPY_FOUND
```
Да
```
True
```
Найден ли пакет *PyCUDA*?
```
>>> af.AF_PYCUDA_FOUND
```
Да
```
True
```
Найден ли пакет *PyOpenCL*?
```
>>> af.AF_PYOPENCL_FOUND
```
Нет
```
False
```
Какие доступны бэкенды?
```
>>> af.util.get_available_backends()
```
CPU, CUDA
```
('cpu', 'cuda')
```
Какой бэкенд активен?
```
>>> af.get_active_backend()
```
CUDA
```
'cuda'
```
Каково количество устройств?
```
>>> af.get_device_count()
```
1
```
1
```
Поддерживается ли двойная точность?
```
>>> af.is_dbl_supported()
```
Да
```
True
```

Приводимые команды выполнены автором на установленной привязке *Python* к библиотеке *ArrayFire v3.4.2*, использующей *CUDA 6.5* в *ARM*-версии *Linux* (плата ***Jetson TK1***).

Можно также создать файл `calc_pi.py` (слева) и воспользоваться им (справа):

```
import arrayfire as af

def calc_pi(samples):
    x = af.randu(samples)
    y = af.randu(samples)
    InC = (x*x + y*y) < 1
    return 4*af.count(InC)/samples
```

```
>>> from calc_pi import *
>>> print(calc_pi(100000))
3.13956
>>> print(calc_pi(1000000))
3.14032
>>> print(calc_pi(10000000))
3.1408308
>>> print(calc_pi(100000000))
3.14173528
```

Наличие *Forge* — интересная особенность библиотеки *ArrayFire*: от пользователя скрыты сложности взаимодействия между вычислительной и графической частями; это позволяет получать весьма простые графические программы даже на *C/C++* (пример из статьи ***Conway's Game of Life using ArrayFire***).

```
#include <arrayfire.h>

int main()
{
  static const float h_kernel[] = { 1,1,1,1,0,1,1,1,1 };
  static const af::array kernel(3, 3, h_kernel, afHost);
  af::Window w(512, 512, "Conway Life using ArrayFire");
  af::array state = (af::randu(256,256,f32) > 0.5).as(f32);
  while (!w.close()) {
     af::array nHood = convolve(state, kernel);
     af::array C0 = (nHood == 2);
     af::array C1 = (nHood == 3);
     state = state * C0 + C1;
     w.image(state);
  }
  return 0;
}
```

Создаётся графическое окно с размерами 512×512, в котором постоянно воспроизводится содержимое двумерного массива `state`, сначала заполненного случайным образом единицами и нулями, а затем перезаписываемого по правилам игры «Жизнь» (***John Conway***). Цикл воспроизведения продолжается до закрытия окна или нажатия клавиши *Esc*. С программной точки зрения здесь создаются массивы `kernel`, `state`, `C0`, `C1`, которые располагаются на *GPU* (а библиотека *Forge* может работать с ними!); пока графическое окно не закрыто, в нём всё время отображается результат свёртки состояния массива `state` с ядром `kernel` и этот результат «подправляется» перевычисляемыми массивами `C0` и `C1`, содержащими «стабильные» и «возникающие» значения. Другие графические примеры можно найти в каталоге `examples` дистрибутива библиотеки *ArrayFire* или в приводимых далее ссылках.

Полезные ссылки

ArrayFire: a general purpose GPU library
https://github.com/arrayfire/arrayfire

ArrayFire Forge: A high-performance visualization library

Python bindings for ArrayFire: A general purpose GPU library
https://github.com/arrayfire/arrayfire-python

Image editing using ArrayFire
http://arrayfire.com/image-editing-using-arrayfire/
Материал из трёх частей (см. также часть 2, часть 3) с примерами обработки изображений.

https://gist.github.com/9prady9/0a3167f5c57ea3ad83e6cad00d4a7bde
Исходный код `perlin_noise.cpp` с примером формирования и отображения т. н. шума Перлина.

15. *PyViennaCL*

ViennaCL (http://viennacl.sourceforge.net) — это бесплатная открытая библиотека линейной алгебры для вычислений на многоядерных архитектурах (имеются в виду *GPU* или *CPU* с несколькими ядрами), написанная на *C++* и поддерживающая *CUDA*, *OpenCL* и *OpenMP* (в качестве т. н. *backend*) и допускающая их переключение при исполнении. Разработана небольшой группой из ***Vienna University of Technology*** (***Karl Rupp*** и др.).

В библиотеку входят функции первого, второго и третьего уровней библиотеки *BLAS*, быстрые умножения разреженных матриц на матрицы и на векторы (поддержка разреженных матриц пока является ограниченной), итерационные методы решения систем линейных уравнений, алгоритмы *БПФ*. Имеется также *Python*-интерфейс к библиотеке *ViennaCL* — «обёртка», именуемая ***PyViennaCL*** (автор — ***Toby St Clere Smithe***).

PyViennaCL разделяется на пять подмодулей: `pycore` (классы для основных объектов библиотеки *ViennaCL*: `Vector` и `Matrix`), `linalg` (интерфейс к функциям линейной алгебры), `vclmath` (математические функции; они доступны также как `pyviennacl.math`), `util` (вспомогательные функции для создания объектов *ViennaCL* из массивов `ndarray` пакета *NumPy*) и `_viennacl` (*C++*-интерфейс непосредственно к *ViennaCL*).

Обычно будет достаточно команды `pip install pyviennacl` для установки *PyViennaCL*. Однако, если исполняемый файл *Python* не находится в пути поиска (так по умолчанию бывает в *Windows*), то может понадобиться что-нибудь вроде

```
C:\Python2.7\python -m pip install pyviennacl
```

Для компиляции *PyViennaCL* из исходного кода необходимы: *C++*-компилятор, *Python 2.7* и *NumPy* (*1.7+*). Рекомендуется также иметь реализацию *OpenCL*. Для компиляции на *Windows*-системах будет нужна также версия *Windows SDK*, соответствующая имеющейся установке *Python*. После конфигурирования (`./configure --<Опции>`[20]) и построения (`python setup.py build`) «обёртка» устанавливается (`sudo python setup.py install` в *Linux* или `python setup.py install` с административными правами в *Windows*).

Простой пример её использования приводит ***Karl Rupp*** в заметке о *PyViennaCL* :

```
import pyviennacl
import numpy

x = [1.0, 2.0, 3.0]
A = numpy.array([[1.0, 2.0, 3.0],
                 [0.0, 3.0, 4.0],
                 [0.0, 0.0, 5.0]])
gpu_x = pyviennacl.Vector(x)
gpu_A = pyviennacl.Matrix(A)
gpu_y = gpu_A * gpu_x
```

```
>>> gpu_y
array([ 14.,  18.,  15.])

>>> type(gpu_y)
<class 'pyviennacl.pycore.Mul'>

>>> y = gpu_y.value
>>> type(y)
<type 'numpy.ndarray'>
>>> y
array([ 14.,  18.,  15.])
```

Результат кажется получаемым немедленно, однако, он будет задержан (т. н. ***lazy***), т. е., выражение вычисляется только тогда, когда выводится или копируется на хост.

Ссылки

http://tsmithe.net/pyviennacl/ , http://viennacl.sourceforge.net/pyviennacl/doc/

PyViennaCL: GPU-accelerated Linear Algebra for Python
https://www.karlrupp.net/2014/02/pyviennacl-gpu-accelerated-linear-algebra-for-python/

[20] Понадобится опция `--cl-lib-dir` при конфигурировании, чтобы использовать *OpenCL*.

16. *cudamat* и *gnumpy*

Эти два пакета сравнительно невелики, а потому вполне обозримы для изучения и повторения тех приёмов, что в них применены (т. е., интересны методически). Оба они появились в *University of Toronto*, оба связаны между собой (второй использует в качестве основы первый). Они предполагают, что в распоряжении пользователя имеется видеокарта с *GPU* от *NVidia* и установлен пакет *CUDA*.

16.1. *cudamat*

Пакет ***cudamat*** состоит из четырёх исходных файлов на *C*: `cudamat.cu`, `cudamat.cuh`, `cudamat_kernels.cu`, `cudamat_kernels.cuh`, — а также «оборачивающего» файла *Python* (`cudamat.py`). Из первых четырёх файлов компилируется библиотека `libcudamat.so` (в более поздних версиях для *Windows* использовалось также расширение `.dll`). Для работы надо расположить библиотеку и тестирующие файлы в каком-либо каталоге, а `cudamat.py` и сопутствующий `__init__.py` — в подкаталоге `cudamat`. Тогда использовать всё это можно после импорта этого модуля (`import cudamat` или `import cudamat as cm`).

> Для глобального доступа к модулю он должен быть размещён в каталоге `site-packages` из установки *Python*: библиотека прямо в нём, а оба `.py`-файла — в подкаталоге `cudamat`.

C-файлы компилируются с помощью компилятора `nvcc` из установки *CUDA* (`Makefile` прилагается, но фактически он состоит из одной строки):

```
nvcc -O --ptxas-options=-v -o libcudamat.so --shared cudamat.cu cudamat_kernels.cu -lcublas
```

> Опция `--shared` указывает, что во время сборки должна быть создана разделяемая библиотека (в зависимости от системы — `.so` или `.dll`), но с именем, указанным после ключа `-o` (т. е., `libcudamat.so`). Применение параметра `--ptxas-options=-v` приводит к выводу информационных сообщений об использовании регистров и памяти при ассемблировании функций ядер. Флаг `-O` предписывает оптимизацию хост-кода, но указан здесь без какого-либо её уровня, а потому неизвестно, даёт ли он что-то вообще. При сборке к программе прилинковывается библиотека ***cuBLAS***.

Версия 2010 года с сайта GoogleCode компилируется под *WinXP* с *CUDA 4.1* без ошибок, но после сборки оказывается, что в библиотеке не экспортирована ни одна из тех функций, которые потом пытается «подхватить» `cudamat.py`! Видимо, автор программы (***Volodymyr Mnich***) в то время компилировал её только под *Linux*, поскольку экспортируемые функции помечены в коде как `extern`; но в *Windows* между `nvcc` и компилятором *Visual Studio* нет полного взаимопонимания: первый мог бы сообщить второму (точнее, его линкеру), какие функции помечены как `extern`, но по какой-то причине этого не делает.

> Прямолинейный способ указания экспортируемых функций прямо в командной строке (т. е., с помощью опции `-Xlinker="..."`, где вместо многоточия надо подставить список величин вида `/EXPORT:<`*`ИмяФункции`*`>` через пробел) работает, но не слишком удобен при разработке; попытки же использовать список экспортируемых функций из файла (скажем, `cudamat.rsp`) наталкиваются на проблему указания местоположения файла, поскольку `nvcc` не слишком аккуратен с текущим каталогом и линкер к моменту этапа линкования не в состоянии найти этот файл (а писать полный путь к нему как-то глупо).
>
> Второй способ — включить все опции командной строки `nvcc` в отдельный файл (скажем, `cudamat.opt`) и потом запустить компиляцию командой `nvcc --options-file cudamat.opt` — тоже вроде бы работоспособен, но в результате находит ошибку в каком-то уж совсем неожиданном месте, что наталкивает на мысль, что длина одной опции в файле `.opt` также ограничена... В результате помогает разнесение опций по разным строкам, а для самой длинной (`-Xlinker="..."`) — разбиение на отдельные строки с этой опцией, но с разными списками в них. Похоже, что критическая длина строки с данной опцией для `nvcc` из *CUDA 4.1* — около 1000 символов.

В последующих версиях *cudamat* и, в частности, в версии, размещённой на ***GitHub*** вместо пометки функций как `extern` используется `EXPORT`, определённая так:

```
#if defined(_WIN32) || defined(__CYGWIN__)
  #define EXPORT __declspec(dllexport)
#else
  #define EXPORT __attribute__ ((visibility("default")))
#endif
```

Теперь и при компиляции *cudamat* в *Windows* помеченные словом `EXPORT` функции будут экспортироваться без дополнительных усилий.

Пример простой программы

```
import numpy as np
import cudamat as cm

cm.cublas_init()

# create two random matrices and copy them to the GPU
a = cm.CUDAMatrix(np.random.rand(32, 256))
b = cm.CUDAMatrix(np.random.rand(256, 32))

# perform calculations on the GPU
c = cm.dot(a, b)
d = c.sum(axis = 0)

# copy d back to the host (CPU) and print
print(d.asarray())
```

Ссылки

CUDAMat: a CUDA-based matrix class for Python

https://github.com/cudamat/cudamat , https://github.com/surban/cudamat
Актуальный исходный код библиотеки и ответвление исходного кода для *Windows*.

16.2. *gnumpy*

gnumpy по выражению автора (***Tijmen Tieleman***) — это простой модуль *Python* с интерфейсом в стиле *NumPy*, выполняющий вычисления на *GPU* и работающий «поверх» (а, следовательно, и требующий) рассмотренной ранее библиотеки *cudamat*. Этот модуль может работать и в режиме симуляции, исполняя всё на *CPU*, — что полезно, если нужно писать программы на устройстве без *GPU*, а использовать на устройствах, которые его имеют. Режим симуляции требует дополнительного модуля ***npmat***.

Файл `gnumpy.py` форматирован по меркам *Python* довольно агрессивно (отступ — один пробел), но, тем не менее, проходит синтаксический анализ нормально.

Ссылки

Gnumpy: an easy way to use GPU boards with Python

http://www.cs.toronto.edu/~tijmen/gnumpy.py

17. ... и другие: *PyGPU*, *PyStream*, *ocl*

В данную группу попали сравнительно малоизвестные программы. Во-первых, это те, что сейчас кажутся слегка устаревшими, поскольку написаны относительно давно и с тех пор не обновлялись. Это касается ***PyGPU*** и ***PyStream***, появившихся на свет в результате «диссертационных» усилий. Во-вторых, это программы вообще малораспространённые, как, например, ***ocl***, однако интересные по замыслу и потенциальным возможностям. Но так как код этих программ доступен, они могут быть отправной точкой и наших программ.

17.1. *PyGPU*

PyGPU (автор — ***Calle Lejdfors***, *Lund University*) — типичный язык программирования для прикладной области (***domain-specific language***), предназначенный для решения задач обработки изображений с помощью компилятора, генерирующего код, исполняемый на *GPU*. Реализован этот язык как встроенный в *Python*, а потому использующий его синтаксис и функциональность. Пакеты `numpy`, `pygame`, `pyglew`, `Cg` и `PIL.Image` — это те зависимости, без которых нормальная работа ***PyGPU*** невозможна.

Поскольку возник он в тот период, когда *GPU* ещё не стали универсальными вычислителями, он опирался на один из специализированных языков того периода: ***Cg*** от *NVidia* (имелись тогда и альтернативные языки — ***HLSL*** от *Microsoft* и ***GLSL*** из *OpenGL*).

Фундаментальная абстракция *PyGPU* — это модель изображения, рассматриваемая как функция на двумерной дискретной сетке со значениями в некотором цветовом пространстве (*RGB*, *YUV*, *CMYK* и т. д.).

Декорирование с помощью `@gpu` какой-либо функции — это, по сути, директива компилятору *PyGPU* сгенерировать код функции для графического процессора (*GPU*). Значения параметров по умолчанию у декорированной функции являются объявлениями типов и необходимы при компиляции (в основном это `Position` и `RGBImage`). В остальном подобная функция выглядит совсем как обыкновенная функция *Python*.

Так как в языке *Python* имеется возможность доступа к байткоду функции (такое свойство называется, напомним, *introspection*) и можно извлечь из него информацию о циклах и условиях, *PyGPU* — вместо использования исходного кода — разбирает байткод *Python*, транслируя его в граф, передаваемый генератору промежуточного кода.

> В дальнейшем у создателей были планы расширить компилятор *PyGPU* для использования всех возможностей *GPU*, включая общие численные алгоритмы, однако, эти планы, похоже, остались не реализованными.

Ссылки

High-Level GPU Programming
Calle Lejdfors. Doctoral dissertation, 2008, Lund Universty, Sweden.

PyGPU
http://fileadmin.cs.lth.se/cs/Personal/Calle_Lejdfors/pygpu/

17.2. *PyStream*

PyStream (автор — ***Nicholas C. Bray***) — компилятор, преобразующий код *Python* в код ***GLSL*** для отрисовки графики в реальном времени. Генерирует также необходимый связующий код для вызова созданного *GLSL* из *Python*.

Ссылки

PyStream: Python Shaders Running on the GPU
Nicholas C. Bray. Dissertation, 2010, University of Illinois.

https://github.com/ncbray/pystream

17.3. *ocl*

Автор ***ocl***, https://github.com/mdipierro/ocl (***Massimo Di Pierro***) характеризует свой пакет как минималистическую библиотеку (лицензия — 2-clause BSD), способную динамически (т. е., в процессе исполнения) превращать декорированные функции *Python* в код на *C99*, *OpenCL* или *JavaScript*. Состоит библиотека из одного файла `ocl.py`; для работы с ней дополнительно необходимы: пакет `meta` (всегда) и пакеты `numpy` и `pyopencl`, если предполагается работа с *GPU*.

Как справедливо замечено им в статье ***«OpenCL programming using Python syntax»*** (*осторожно, много опечаток*; предпочтительнее — журнальный вариант статьи), основная сложность в написании программ в архитектурах вроде *CUDA* и *OpenCL* заключается в том, что приходится писать «программу в программе», поскольку хост-программа управляет исполнением программ-ядер. Некоторое упрощение процесса будет возможно, если для хост-программирования использовать не *C/C++*, а «дружелюбный» язык вроде *Python*. Поэтому он предлагает библиотеку ***ocl***, «надстроенную» над *PyOpenCL* и позволяющую писать и хост-программу, и ядра на одном языке — *Python*. По сути она будет состоять из двух частей: тонкой «обёртки» вокруг *PyOpenCL* и декоратора, призванного во время исполнения декорированной функции преобразовать инструкции *Python* в код *C99* и откомпилировать получаемый *OpenCL*-код «на лету» (***JIT***, ***just-in-time***) для его последующего исполнения. Декоратор производит анализ декорированной функции во время исполнения, преобразуя тело этой функции в абстрактное синтаксическое дерево (***Abstract Syntax Tree***, ***AST***) для последующего превращения в код *OpenCL*. Поддерживаются управляющие конструкции и команды языка: `def`, `if ... elif ... else`, `for ... range`, `while`, `break`, `continue`. Возвращаемый тип функции определяется автоматически, надо только возвращать не выражения, а переменные. Могут быть определены и другие типы переменных и указателей, соответствующий синтаксис приведён в таблице:

ocl	**C99/OpenCL**
`x = new_type(...)`	`type x = ... ;`
`x = new_ptr_type(...)`	`type *x = ... ;`
`x = new_ptr_ptr_type(...)`	`type **x = ... ;`
`ADDR(x)`	`&x`
`REFD(x)`	`*x`
`CAST(ptr_type,x)`	`(type*)x`

Для вывода графических результатов этот же автор создал «обёртку» вокруг `matplotlib` под названием ***Canvas***.

Ссылки

OpenCL programming using Python syntax
http://airccj.org/CSCP/vol3/csit3506.pdf

https://github.com/mdipierro/ocl

https://github.com/mdipierro/canvas

18. Вместо заключения: *Intel Python 2017, 2018*

> ***Intel Distribution for Python*** — это бинарный дистрибутив интерпретатора *Python* и часто используемых пакетов для научных и инженерных вычислений. Он поддерживает *Python 2* и *3* для систем *Windows*, *Linux* и *OS X* и упрощает инсталляцию *Python* тем, что содержит необходимые пакеты в двоичном виде (вместе с зависимостями для работы на перечисленных платформах) и уже сконфигурированными, так что никакие компилирующие утилиты не нужны. Пакеты «ускорены» использованием таких библиотек от *Intel*, как ***MKL*** (Math Kernel Library), ***TBB*** (Threading Building Blocks), ***DAAL*** (Data Analytics Acceleration Library).

Обсуждаемый здесь вариант дистрибутива *Python* интересен тем, что это — один из тех редких случаев, когда язык, развиваемый самим его автором и заинтересованным сообществом, «взят на вооружение» и поддержан крупным коммерческим игроком индустрии[21].

В результате пользователю предлагается интегрированный кроссплатформенный продукт, обеспечивающий более высокую производительность приложений *Python* на современных платформах *Intel* (от процессоров **Intel Atom** или **Intel Core i3** и выше). Он включает в себя такие инструменты, как *Conda* и *Jupyter Notebook*, пакеты *NumPy*, *SciPy*, *Matplotlib*, *Numba*, *Theano* и другие.

Так, например, в составе дистрибутива ***Intel Python 2017 U2***, выпущенного 13.02.2017, имелись (среди прочих) такие пакеты: `conda` *4.2.12*, `intelpython` *2017.0.2*, `jupyter_console` *5.0.0*, `mkl` *2017.0.2*, `numba` *0.30.1*, `matplotlib` *1.5.3*, `numpy` *1.11*, `notebook` *4.2.3*, `pip` *8.1.2*, `scipy` *0.18.1*, `sympy` *1.0*, `boost` *1.61.0* (т. е., там уже имеется `compute`!), `caffe` *1.0.0*, `networkx` *1.11*, `pillow` *3.4.2*, `theano` *1.0* (впрочем, версия последнего немного «завышена»...).

Все пакеты дистрибутива будут установлены в виртуальное окружение с именем `root`.

> Хотя в этом дистрибутиве и присутствуют два пакета из обсуждавшихся в пособии (*Numba* и *Theano*), использовать их для работы с *GPU* не так легко: первому пакету требуется для этого либо *CUDA*, либо *AMD HSA*, второму тоже необходима *CUDA*, а это предполагает наличие соответствующей видеокарты (поскольку дистрибутив ориентирован на *CPU* фирмы *Intel*, встроенная видеокарта — несмотря на присутствие *GPU* — тут ничем не поможет).
>
> Автор пособия, который экспериментировал с данным дистрибутивом на микро-ПК **MeLE PCG02U**, оказался именно в такой ситуации, только без возможности добавить видекарту...

В сентябре 2017 года был анонсирован ***Intel Distribution for Python 2018***, в котором версия *Python* изменилась (с *3.5* на *3.6*). Поэтому для того, чтобы обновить выпущенный ранее *Intel Python 2017* с помощью `conda`, следует использовать `conda install` (а не `conda update`, поскольку последняя команда не обновляет версию *Python*):

```
conda install python=3.6 -c intel
```

Ссылки

Intel® Distribution for Python*
https://software.intel.com/en-us/distribution-for-python

The Intel® Distribution for Python* 2018 has officially been released!
https://software.intel.com/en-us/forums/intel-distribution-for-python/topic/743830

Intel® Optimized Packages for the Intel® Distribution for Python*
https://software.intel.com/en-us/articles/intel-optimized-packages-for-the-intel-distribution-for-python

Complete List of Packages for the Intel® Distribution for Python*
https://software.intel.com/en-us/articles/complete-list-of-packages-for-the-intel-distribution-for-python

[21] Впрочем, справедливости ради, следует отметить, что поддержана всё-таки конкретная реализация *Python* — *Anaconda Python distribution* от компании, выпустившей ранее пакет *Numba*.