Applied Parallel Computing

parallel-computing.pro

PyCUDA

Программирование GPU на языке Python

PyCUDA: задачи

- Адаптеры для указателей
- Вызов ядер
- Обработка ошибок
- «Обёртки» для runtime-библиотеки
- «Обёртки» для прикладных библиотек
- Взаимодействие с GL
- Высокоуровневые конструкции
- Метапрограммирование

Hello, PyCUDA!

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

import numpy

mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

a = numpy.random.randn(4,4)
a = a.astype(numpy.float32)

a_gpu = cuda.mem_alloc(a.nbytes)

cuda.memcpy_htod(a_gpu, a)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)

print a; print; print a_doubled
```

Запуск

```
[mitya@localhost pycuda]$ PYCUDA_DEFAULT_NVCC_FLAGS="-w -D__GNUC_MINOR__=6" python doublify_short.py
[[-1.77416289  1.87755203 -1.20945001  0.06791665]
 [-0.77498513  0.92289156 -0.15961683 -0.78835815]
 [-0.87043971  0.02797419  0.11622938 -0.57282966]
 [ 1.00931525  0.68197769 -0.40039062 -0.26544631]]

[[-3.54832578  3.75510406 -2.41890001  0.13583329]
 [-1.54997027  1.84578311 -0.31923366 -1.5767163 ]
 [-1.74087942  0.05594838  0.23245876 -1.14565933]
 [ 2.0186305   1.36395538 -0.80078125 -0.53089261]]
```

- pycuda.compiler.DEFAULT_NVCC_FLAGS ← $PYCUDA_DEFAULT_NVCC_FLAGS

Работа с памятью

```
# GPU memory allocation
pycuda.driver.mem_alloc(bytes)

# Memory initialization
pycuda.driver.memset_d{8,16,32}(dest, data, count)

# Host-to-device copy
pycuda.driver.memcpy_htod(dest, src)
pycuda.driver.memcpy_htod_async(dest, src, stream=None)

# Device-to-host copy
pycuda.driver.memcpy_dtoh(dest, src)
pycuda.driver.memcpy_dtoh_async(dest, src, stream=None)

# Device-to-device copy (same device)
pycuda.driver.memcpy_dtod(dest, src, size)
pycuda.driver.memcpy_dtod_async(dest, src, size, stream=None)

# Device-to-device copy (different devices)
pycuda.driver.memcpy_peer(dest, src, size, dest_context=None, src_context=None)
pycuda.driver.memcpy_peer_async(dest, src, size, dest_context=None, src_context=None, stream=None)

# Return type for mem_alloc
class pycuda.driver.DeviceAllocation:
  free()
```

Краткая форма

```
class pycuda.driver.In(array)
class pycuda.driver.Out(array)
class pycuda.driver.InOut(array)
```

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

import numpy

mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

a = numpy.random.randn(4,4).astype(numpy.float32)

func = mod.get_function("doublify")

print a; print

func(cuda.InOut(a), block=(4, 4, 1))

print a
```

GPUArray

```
class pycuda.gpuarray.GPUArray(shape, dtype, *, allocator=None, order="C")
```

- Конструкторы
- Взаимодействие с `ndarray`
- Арифметические операции:
  - +, -, *, **, /, mul_add
- Редукции:
  - sum, dot, max, min
- Поэлементные операции:
  - округление и модуль
  - показательная функция, логарифмы, корни
  - тригонометрические функции

GPUArray: конструкторы

```
# Return a GPUArray that is an exact copy of the numpy.ndarray instance ary.
pycuda.gpuarray.to_gpu(ary, allocator=None)

# The same done asynchronously, optionally sequenced into stream.
pycuda.gpuarray.to_gpu_async(ary, allocator=None, stream=None)

# A synonym for the GPUArray constructor.
pycuda.gpuarray.empty(shape, dtype, *, allocator=None, order="C")

# Same as empty(), but the GPUArray is zero-initialized before being returned.
pycuda.gpuarray.zeros(shape, dtype, *, allocator=None, order="C")

# Make a new, uninitialized GPUArray having the same properties as other_ary.
pycuda.gpuarray.empty_like(other_ary)

# Make a new, zero-initialized GPUArray having the same properties as other_ary.
pycuda.gpuarray.zeros_like(other_ary)
```

GPUArray: редукции

```
pycuda.gpuarray.sum(a, dtype=None, stream=None)

pycuda.gpuarray.dot(a, b, dtype=None, stream=None)

pycuda.gpuarray.subset_dot(subset, a, b, dtype=None, stream=None)

pycuda.gpuarray.max(a, stream=None)

pycuda.gpuarray.min(a, stream=None)

pycuda.gpuarray.subset_max(subset, a, stream=None)

pycuda.gpuarray.subset_min(subset, a, stream=None)
```

Вычисление произвольных выражений

```
# Generate a kernel that takes a number of scalar or vector arguments and performs the scalar operation on ⟵
      each entry of its arguments, if that argument is a vector.
class pycuda.elementwise.ElementwiseKernel(arguments, operation, name="kernel", keep=False, options=[], ⟵
      preamble="")
```

- *arguments* – объявление аргументов на языке Си
- *operation* – операция в виде присваивания на языке Си (без ";")
- *name* – имя сгенерированной функции-ядра
- *keep* и *options* передаются в `pycuda.compiler.SourceModule`
- *preamble* задаёт код, который будет предшествовать функции-ядру
  - может быть использовано для включения файлов и/или определения дополнительных функций

ElementwiseKernel: пример

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand

a_gpu = curand((50,))
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]",
        "linear_combination")

c_gpu = gpuarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

import numpy.linalg as la
assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

Программируемые редукции

```
# Generate a kernel that takes a number of scalar or vector arguments (at least one vector argument)
class pycuda.reduction.ReductionKernel(dtype_out, neutral, reduce_expr, map_expr=None, arguments=None, name="reduce_kernel", keep=False, options=[], preamble="")
```

- применяет *map_expr* к каждому элементу вектора-аргумента
- затем применяет *reduce_expr*
- *neutral* задаёт начальное значение
- *name* – имя сгенерированной функции-ядра
- *keep* и *options* передаются в `pycuda.compiler.SourceModule`
- *preamble* задаёт код, который будет предшествовать функции-ядру

ReductionKernel: пример

```
a = gpuarray.arange(400, dtype=numpy.float32)
b = gpuarray.arange(400, dtype=numpy.float32)

krnl = ReductionKernel(numpy.float32, neutral="0",
        reduce_expr="a+b", map_expr="x[i]*y[i]",
        arguments="float *x, float *y")

my_dot_prod = krnl(a, b).get()
```

Прочие функции

- Информация о версиях
- Управление устройствами
- Управление потоками
- Обработка ошибок

Метапрограммирование с использованием PyCUDA

Предпосылки

- Автоматическая подстройка параметров
  - Предугадать сложнее, чем попробовать
- Типы данных
  - Генерация кода для float и double
- Специализация кода
  - Не следует делать код слишком общим (и, как следствие, медленным)

Метапрограммирование с использованием шаблонизатора

```
from jinja2 import Template

tpl = Template("""
    __global__ void add(
            {{ type_name }} *tgt,
            {{ type_name }} *op1,
            {{ type_name }} *op2)
    {
      int idx = threadIdx.x +
        {{ thread_block_size }} * {{block_size}}
        * blockIdx.x;

      {% for i in range(block_size) %}
          {% set offset = i*thread_block_size %}
          tgt[idx + {{ offset }}] =
            op1[idx + {{ offset }}]
            + op2[idx + {{ offset }}];
      {% endfor %}
    }""")

rendered_tpl = tpl.render(
    type_name="float", block_size=block_size,
    thread_block_size=thread_block_size)

mod = SourceModule(rendered_tpl)
```

Метапрограммирование с использованием codepy

```
from codepy.cgen import FunctionBody, \
        FunctionDeclaration, Typedef, POD, Value, \
        Pointer, Module, Block, Initializer, Assign
from codepy.cgen.cuda import CudaGlobal

mod = Module([
    FunctionBody(
        CudaGlobal(FunctionDeclaration(
            Value("void", "add"),
            arg_decls=[Pointer(POD(dtype, name))
                for name in ["tgt", "op1", "op2"]])),
        Block([
            Initializer(
                POD(numpy.int32, "idx"),
                "threadIdx.x + %d*blockIdx.x"
                % (thread_block_size*block_size)),
            ]+[
            Assign(
                "tgt[idx+%d]" % (o*thread_block_size),
                "op1[idx+%d] + op2[idx+%d]" % (
                    o*thread_block_size,
                    o*thread_block_size))
            for o in range(block_size)]))])

mod = SourceModule(mod)
```

Использование библиотеки CURAND с PyCUDA

```
# Return an array of shape filled with random values of dtype in the range [0,1)
pycuda.curandom.rand(shape, dtype=numpy.float32, stream=None)

# Provides pseudorandom numbers. Generates sequences with period at least (2^190)
class pycuda.curandom.XORWOWRandomNumberGenerator(seed_getter=None, offset=0):
  fill_uniform(data, stream=None)
  fill_normal(data, stream=None)

# Provides quasirandom numbers. Generates sequences with period of \(2^32\)
class pycuda.curandom.Sobol32RandomNumberGenerator(dir_vector=None, offset=0)
  fill_uniform(data, stream=None)
  fill_normal(data, stream=None)
```

Ресурсы

- Автор: Andreas Klöckner
- Веб-сайт: `http://mathema.tician.de/software/pycuda`
- Загрузка: `https://pypi.python.org/pypi/pycuda`
- Документация: `http://documen.tician.de/pycuda/`