`cl.oquence`

A hybrid approach to productive high-performance programming

Scientific users

require **performance** and can often describe how they would utilize parallelism for their problems

...so why do they use **MATLAB** and **Python**?

Slow, dynamic languages with poor support for parallelism, yet they dominate.

Answer:

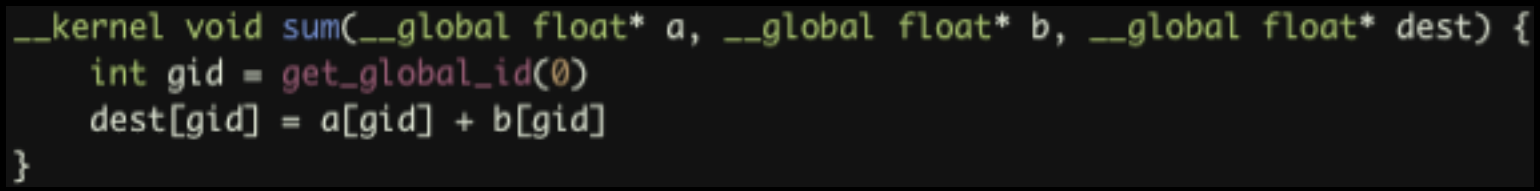
they focus on **productivity**

- Conceptually simple & convenient syntax
- Integrated programming environments
- Extensive library support for common, productivity-limited (rather than performance-limited) tasks like plotting

OpenCL?

Its fast but it is **not productive**.

Counterproductive!

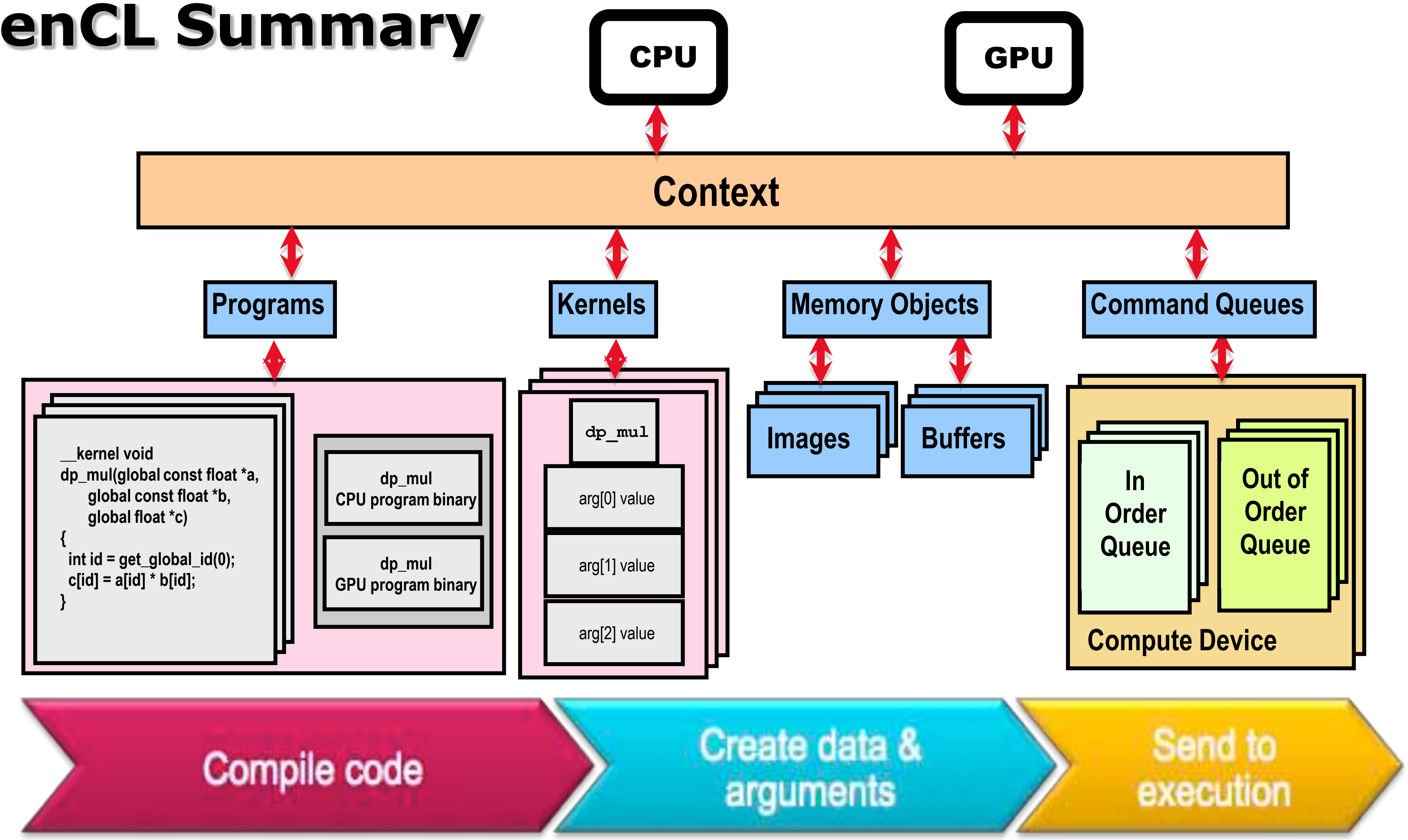


```
__kernel void sum(__global float* a, __global float* b, __global float* dest) {
    int gid = get_global_id(0)
    dest[gid] = a[gid] + b[gid]
}
```

The **OpenCL language** is based on **C**.

- **A subset of ISO C99**
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- **A superset of ISO C99 with additions for:**
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers
- **Also includes a large set of built-in functions**
  - Image manipulation
  - Work-item manipulation,
  - Specialized math routines, etc.

OpenCL Summary



CPU

GPU

Context

Programs

Kernels

Memory Objects

Command Queues

```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
 int id = get_global_id(0);
 c[id] = a[id] * b[id];
}
```

dp_mul
CPU program binary

dp_mul
GPU program binary

`dp_mul`

arg[0] value

arg[1] value

arg[2] value

Images

Buffers

In Order Queue

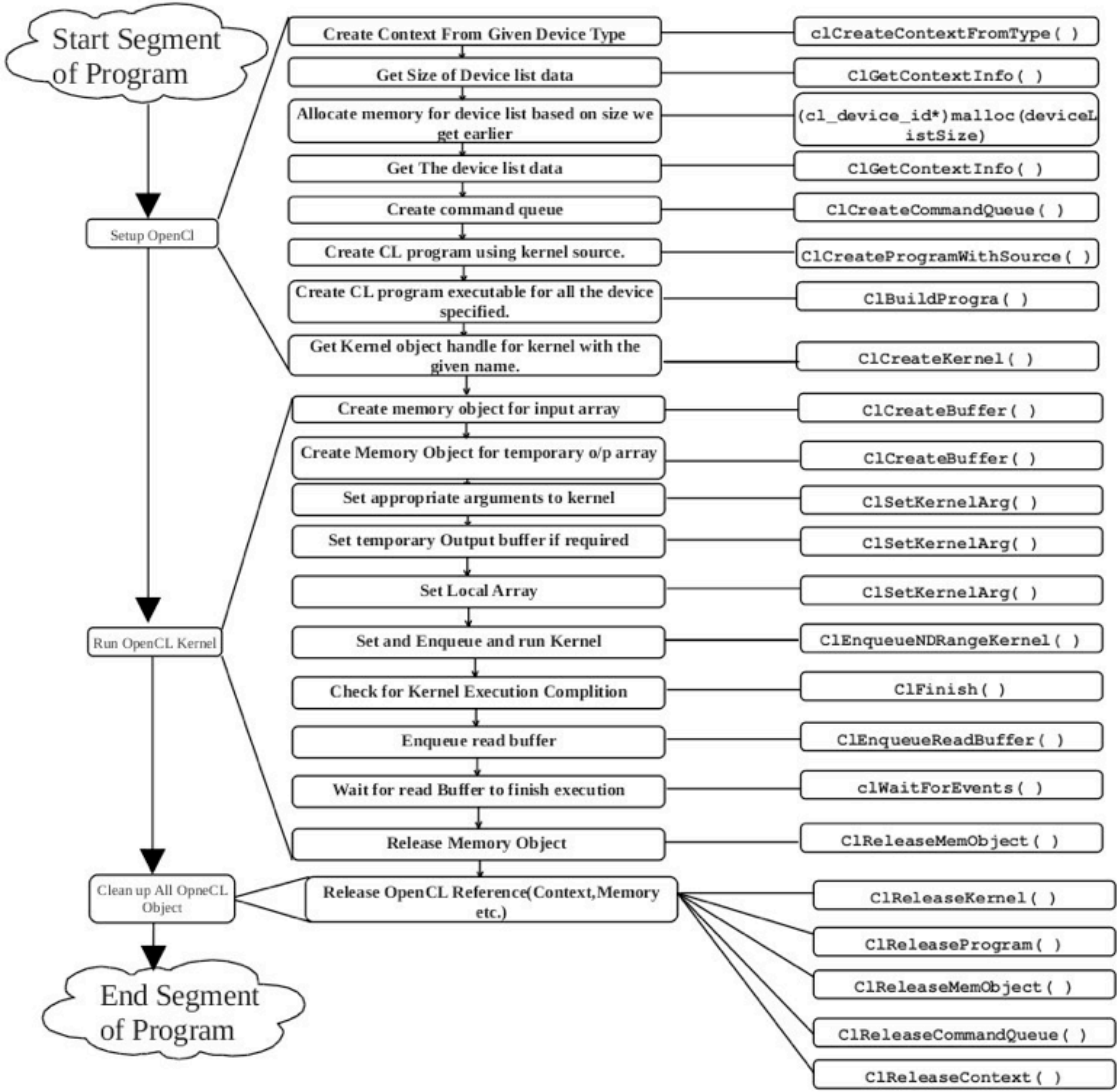
Out of Order Queue

Compute Device

Compile code

Create data & arguments

Send to execution



Start Segment of Program

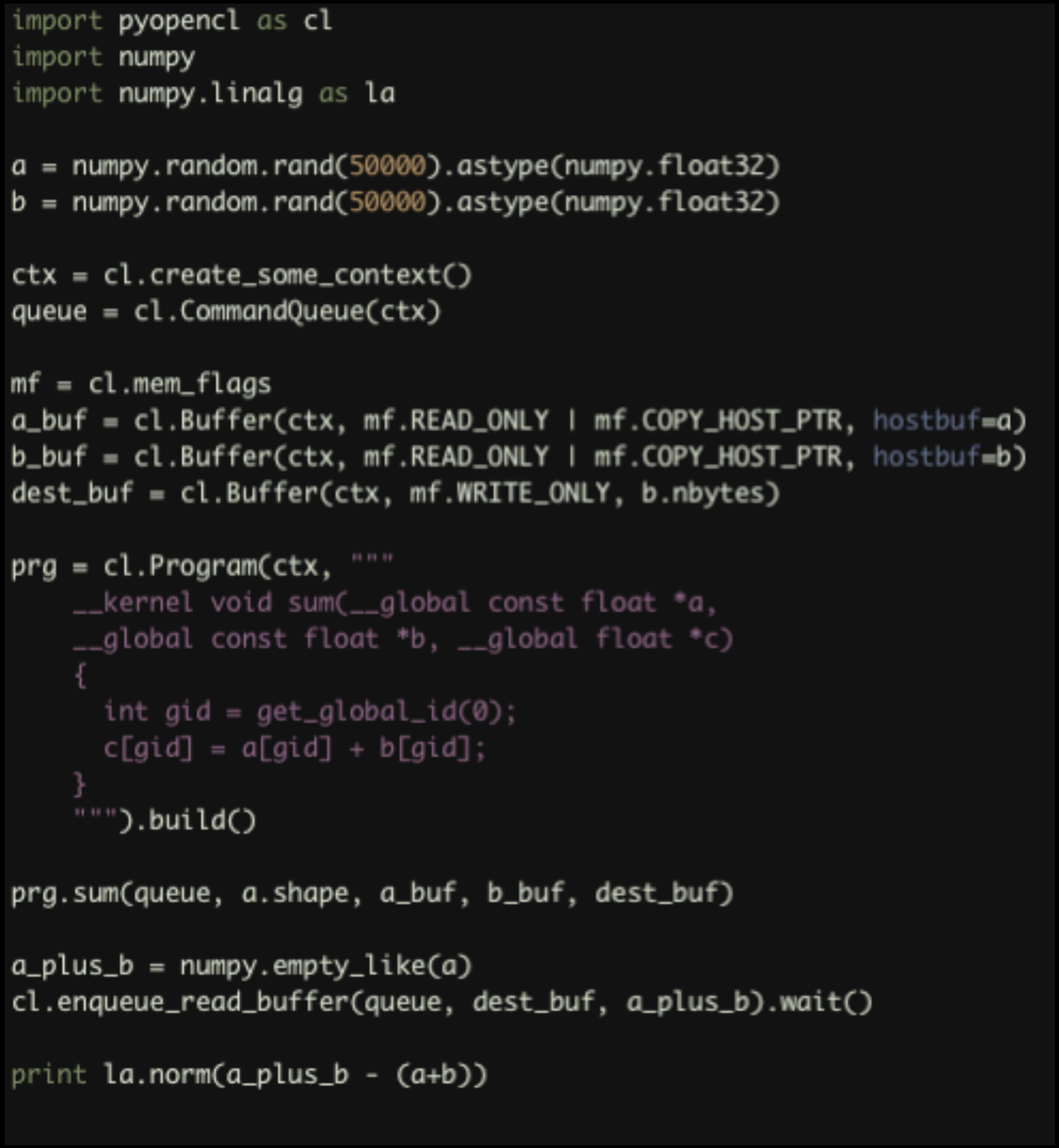
- Setup OpenCl
  - Create Context From Given Device Type — `clCreateContextFromType ( )`
  - Get Size of Device list data — `ClGetContextInfo ( )`
  - Allocate memory for device list based on size we get earlier — `(cl_device_id*)malloc(deviceListSize)`
  - Get The device list data — `ClGetContextInfo ( )`
  - Create command queue — `ClCreateCommandQueue ( )`
  - Create CL program using kernel source. — `ClCreateProgramWithSource ( )`
  - Create CL program executable for all the device specified. — `ClBuildProgra ( )`
  - Get Kernel object handle for kernel with the given name. — `ClCreateKernel ( )`
- Run OpenCL Kernel
  - Create memory object for input array — `ClCreateBuffer ( )`
  - Create Memory Object for temporary o/p array — `ClCreateBuffer ( )`
  - Set appropriate arguments to kernel — `ClSetKernelArg ( )`
  - Set temporary Output buffer if required — `ClSetKernelArg ( )`
  - Set Local Array — `ClSetKernelArg ( )`
  - Set and Enqueue and run Kernel — `ClEnqueueNDRangeKernel ( )`
  - Check for Kernel Execution Complition — `ClFinish ( )`
  - Enqueue read buffer — `ClEnqueueReadBuffer ( )`
  - Wait for read Buffer to finish execution — `clWaitForEvents ( )`
  - Release Memory Object — `ClReleaseMemObject ( )`
- Clean up All OpneCL Object
  - Release OpenCL Reference(Context,Memory etc.)
    - `ClReleaseKernel ( )`
    - `ClReleaseProgram ( )`
    - `ClReleaseMemObject ( )`
    - `ClReleaseCommandQueue ( )`
    - `ClReleaseContext ( )`

End Segment of Program

A Solution:

allow performance bottlenecks to be written in OpenCL *from within Python*

pyopencl



```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

wraps the OpenCL C API and language nearly directly

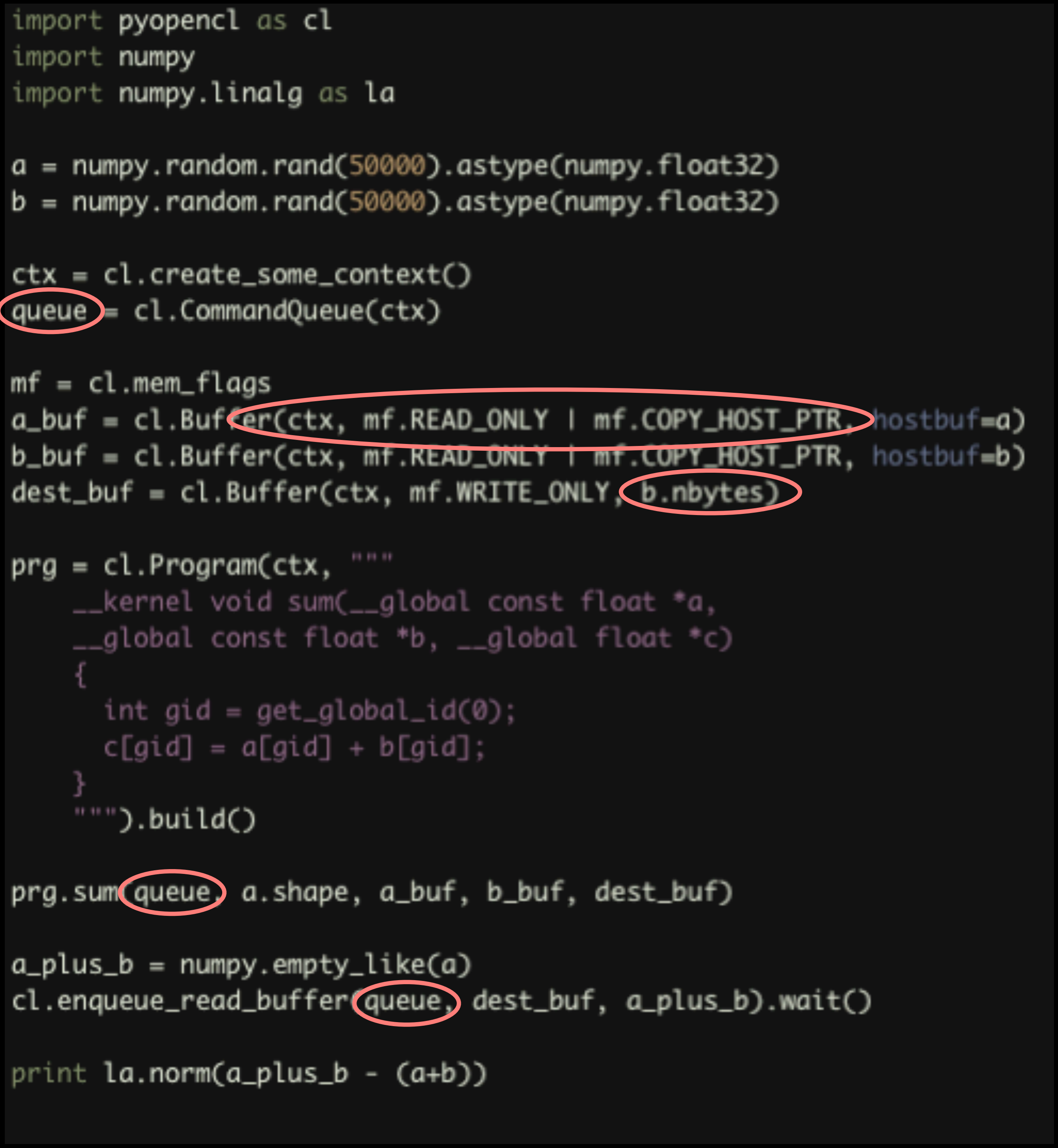
after your bottleneck code runs, you can use numpy and matplotlib as usual.

BIG BOOST IN PRODUCTIVITY

&

NEGLIGIBLE LOSS IN PERFORMANCE

pyopencl



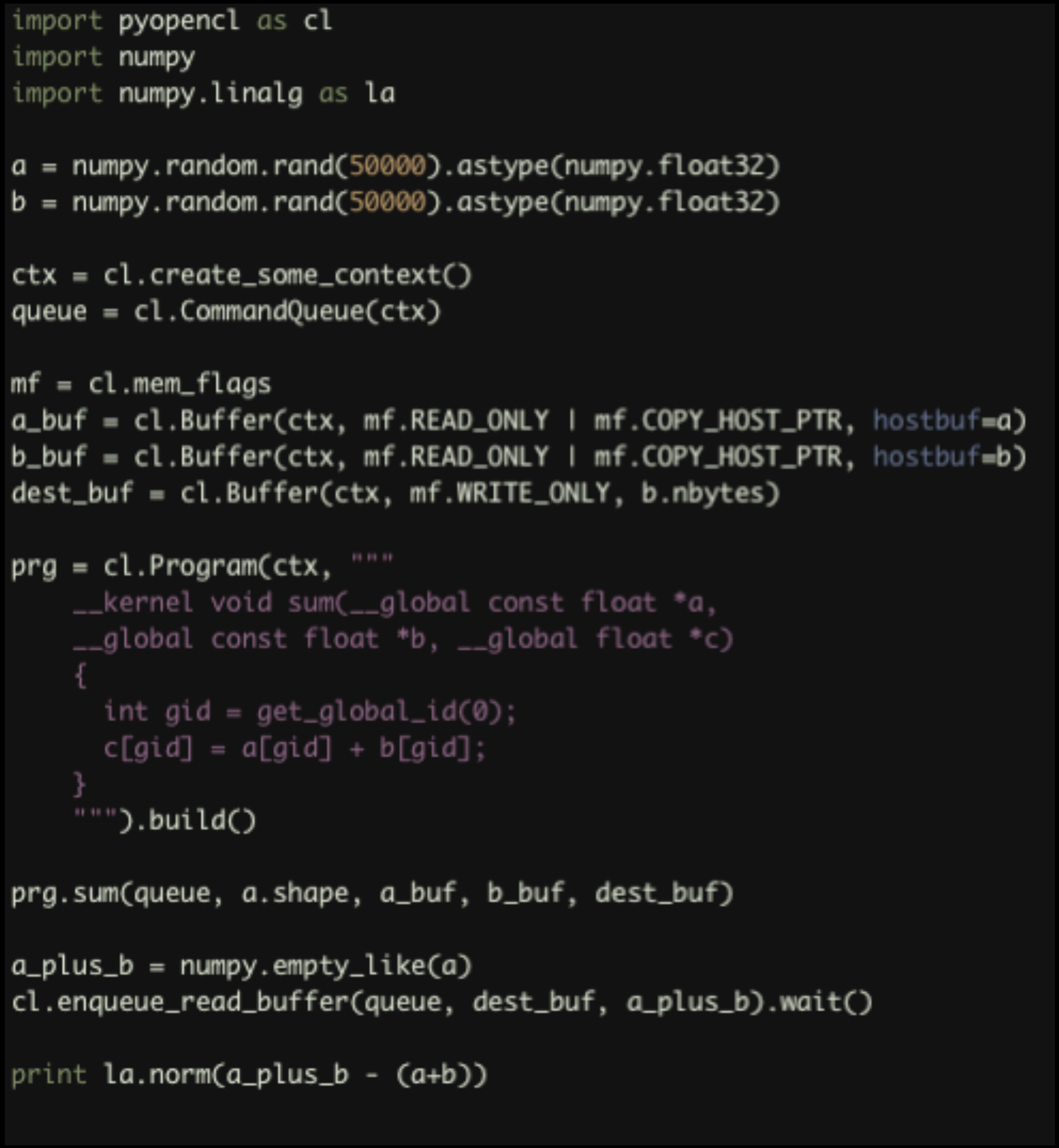
```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

Problem 1: The OpenCL host API is itself not very usable.

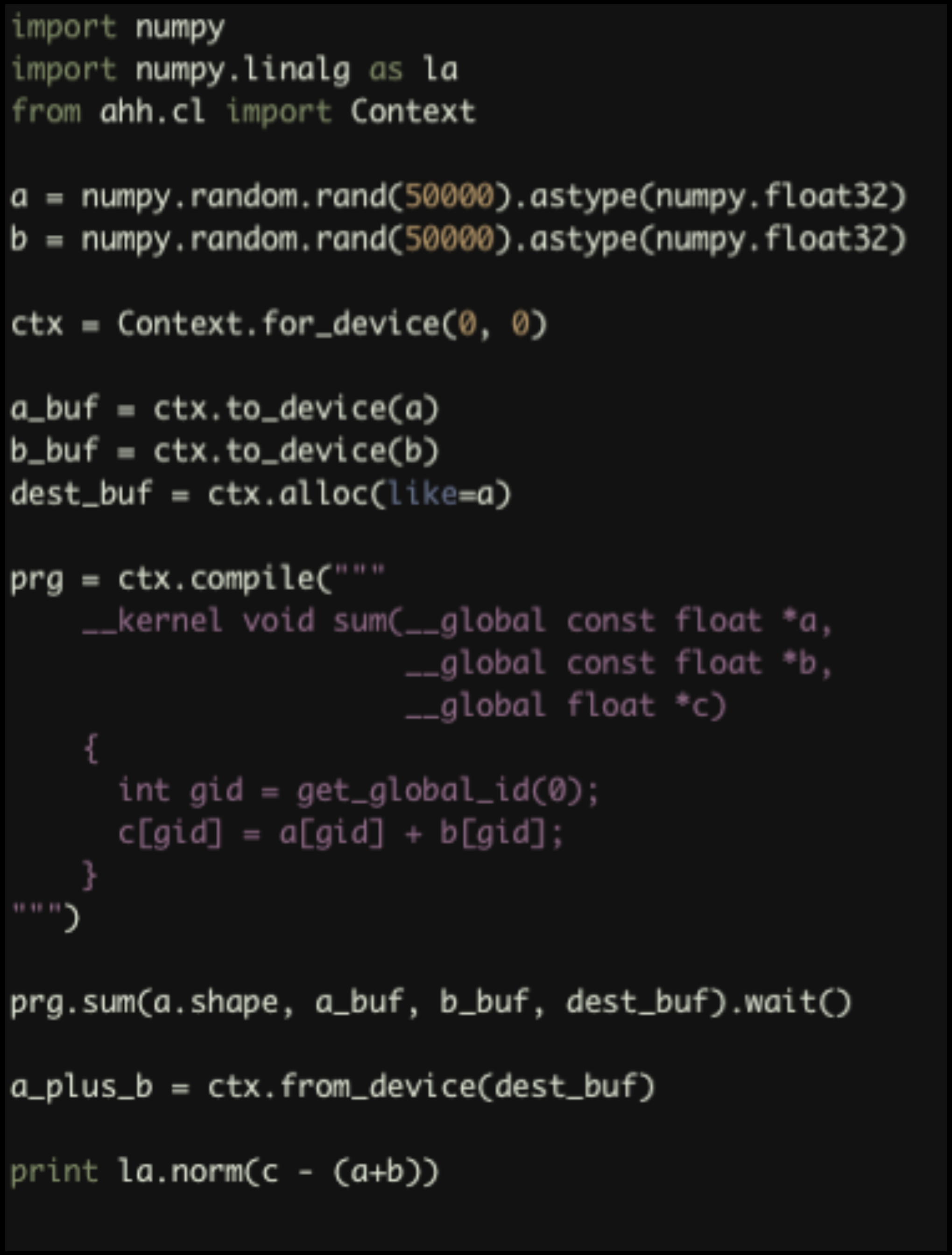
verbose: *queues and flags everywhere*

opaque: *buffers are just a bunch of bytes*

pyopencl ⊂ ahh.cl



```python
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

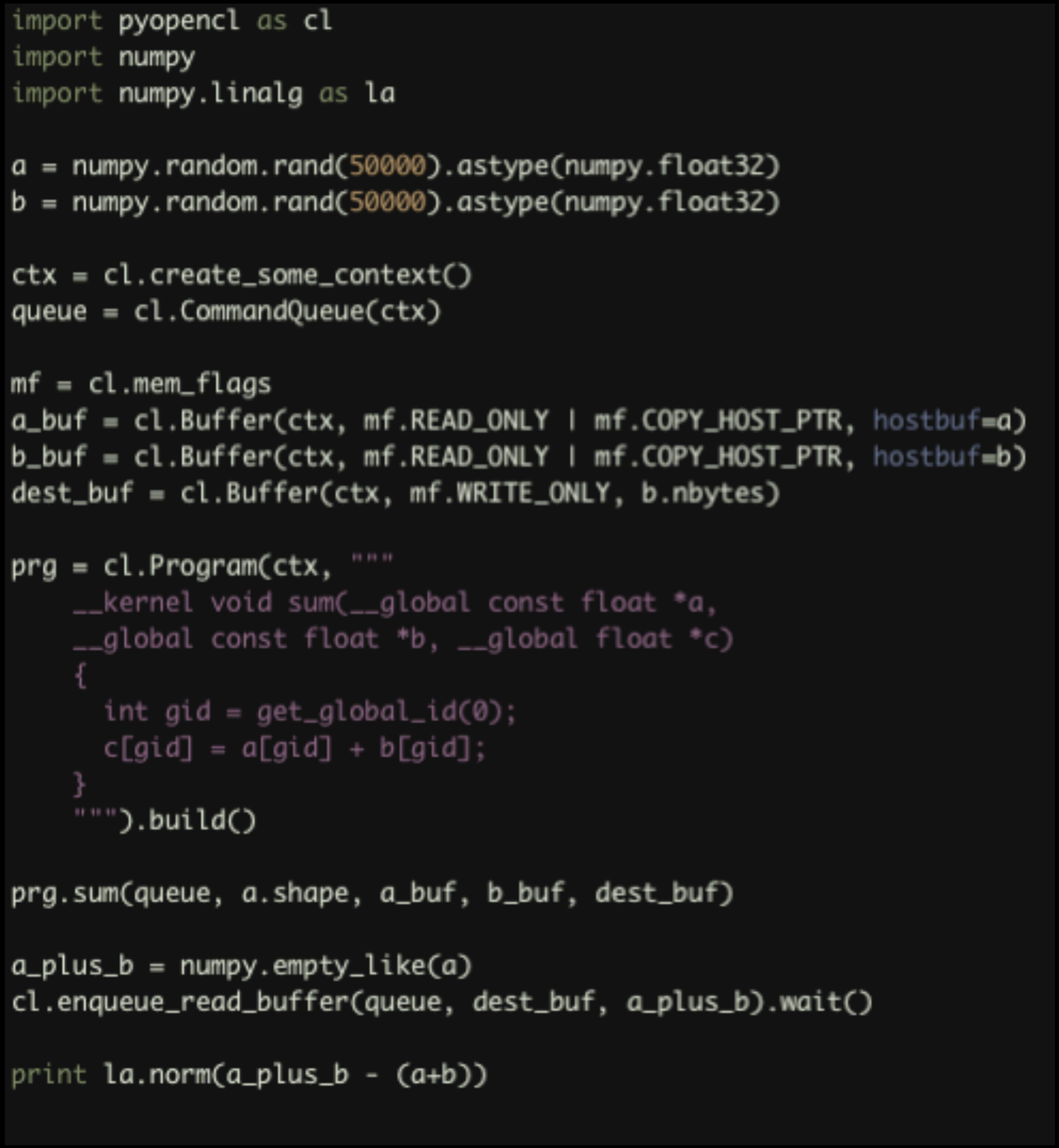


```python
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

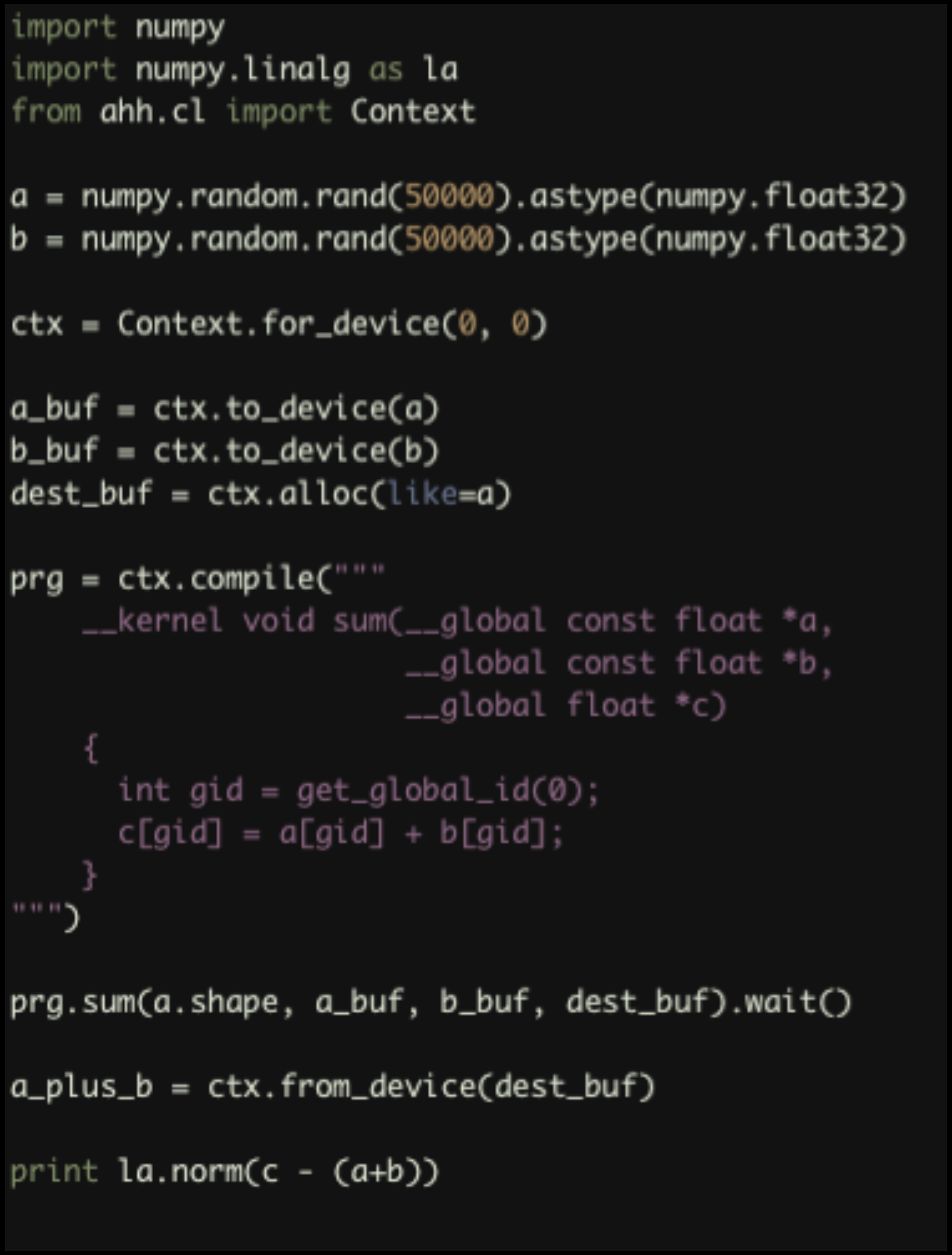
ahh.cl is a superset of pyopencl

(`from pyopencl import *` at the top of ahh.cl)

pyopencl ⊂ ahh.cl



```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

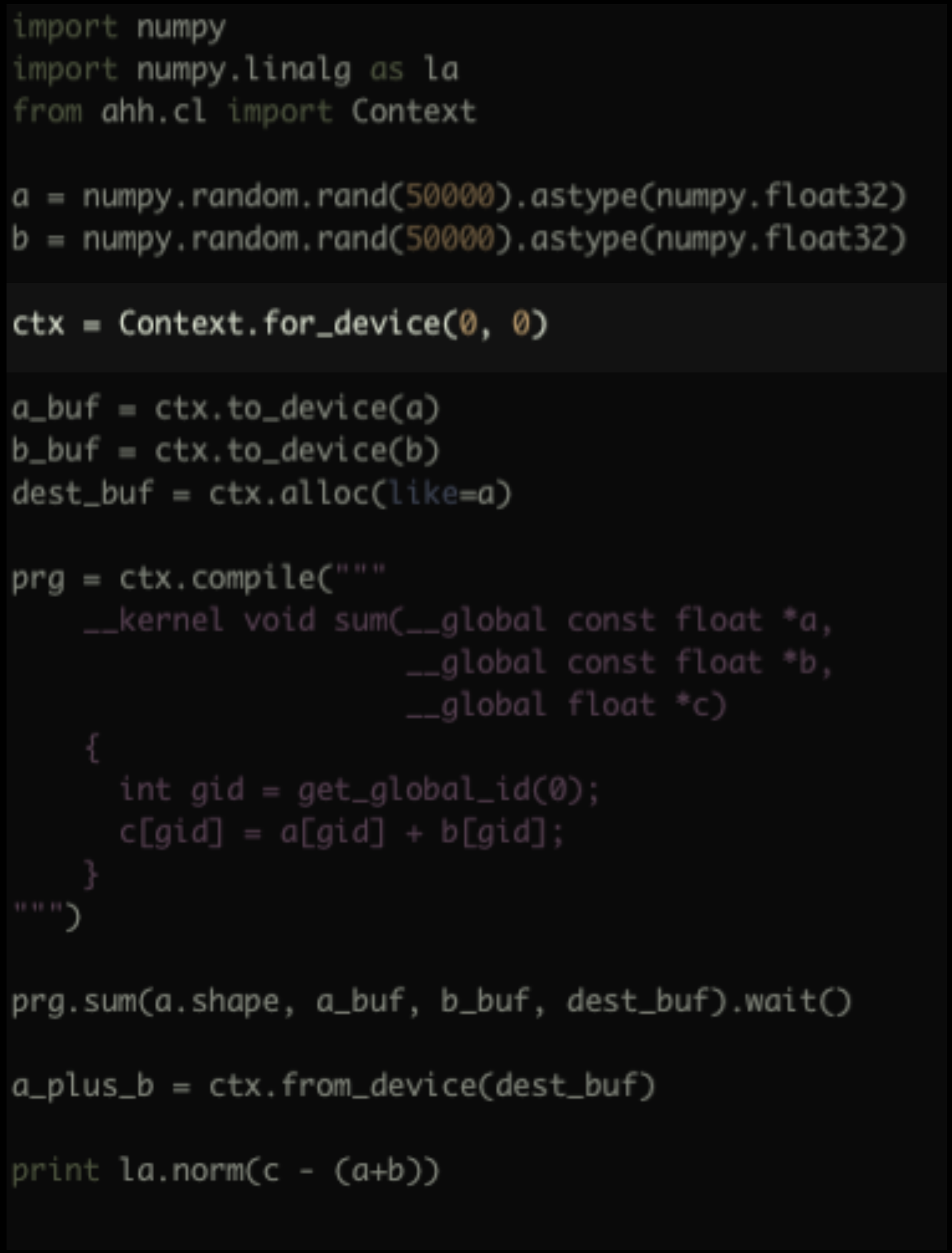


```
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

+ extensions for convenience and clarity

ahh.cl

various utility functions for creating
a Context bound to the device you want
and a **default implicit queue**



```python
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

simple, intuitive memory management

four flexible functions:

`alloc`
allocates empty buffer

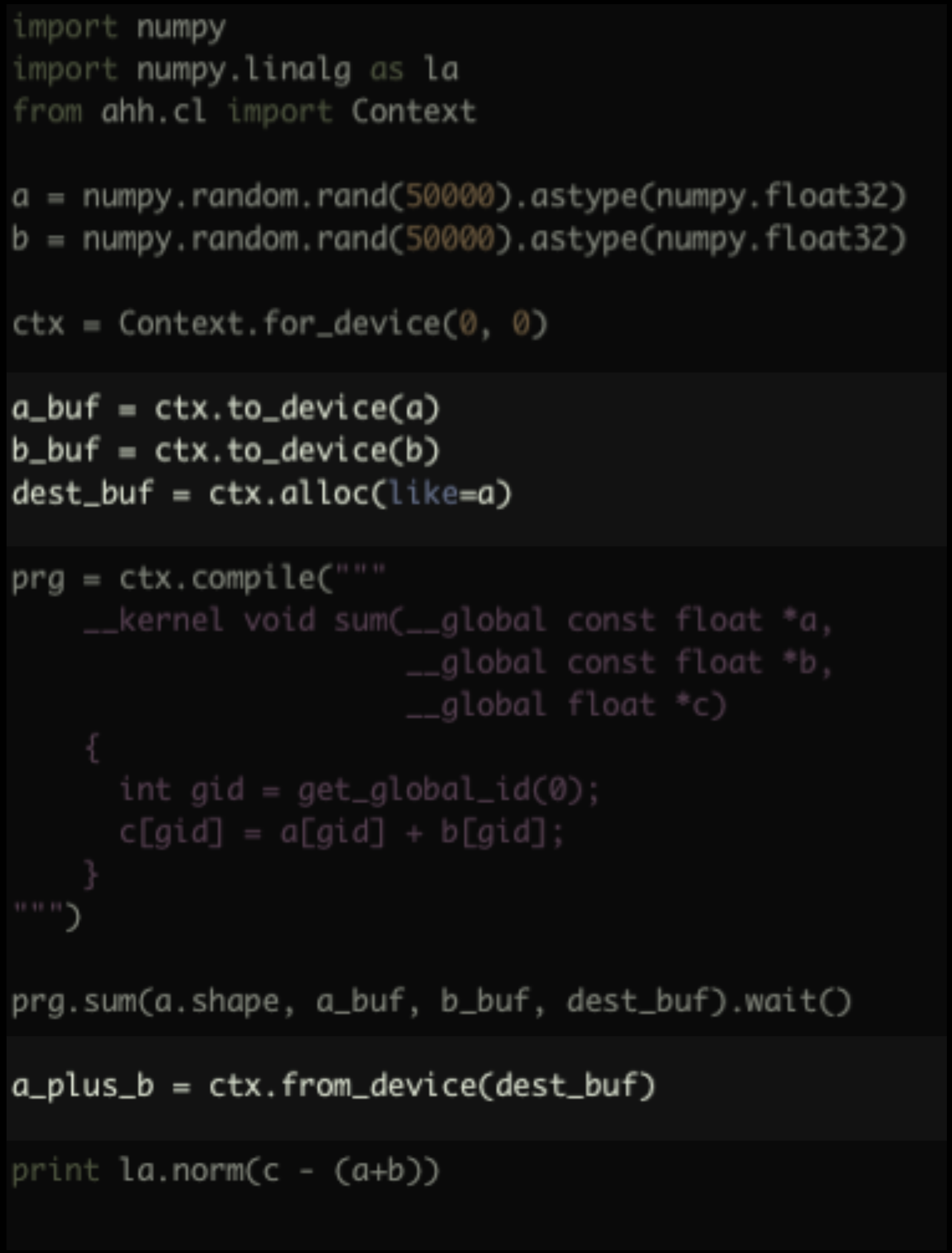
`to_device`
numpy array => new buffer

`from_device`
buffer => new numpy array

`memcpy`
copies between existing buffers or arrays

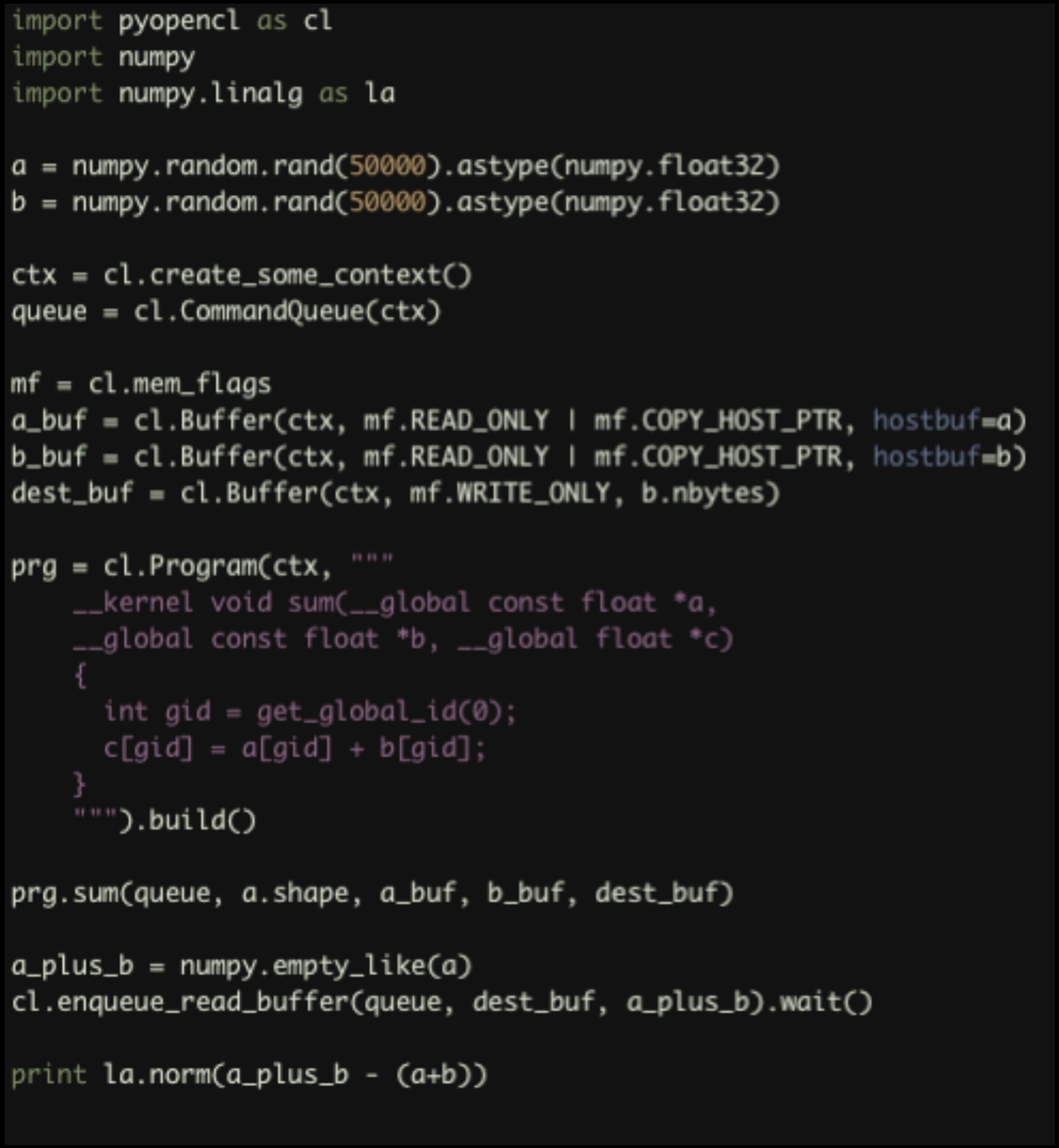
buffers save metadata (type, shape, order), so you never have to repeat yourself (this will be useful later)

ahh.cl



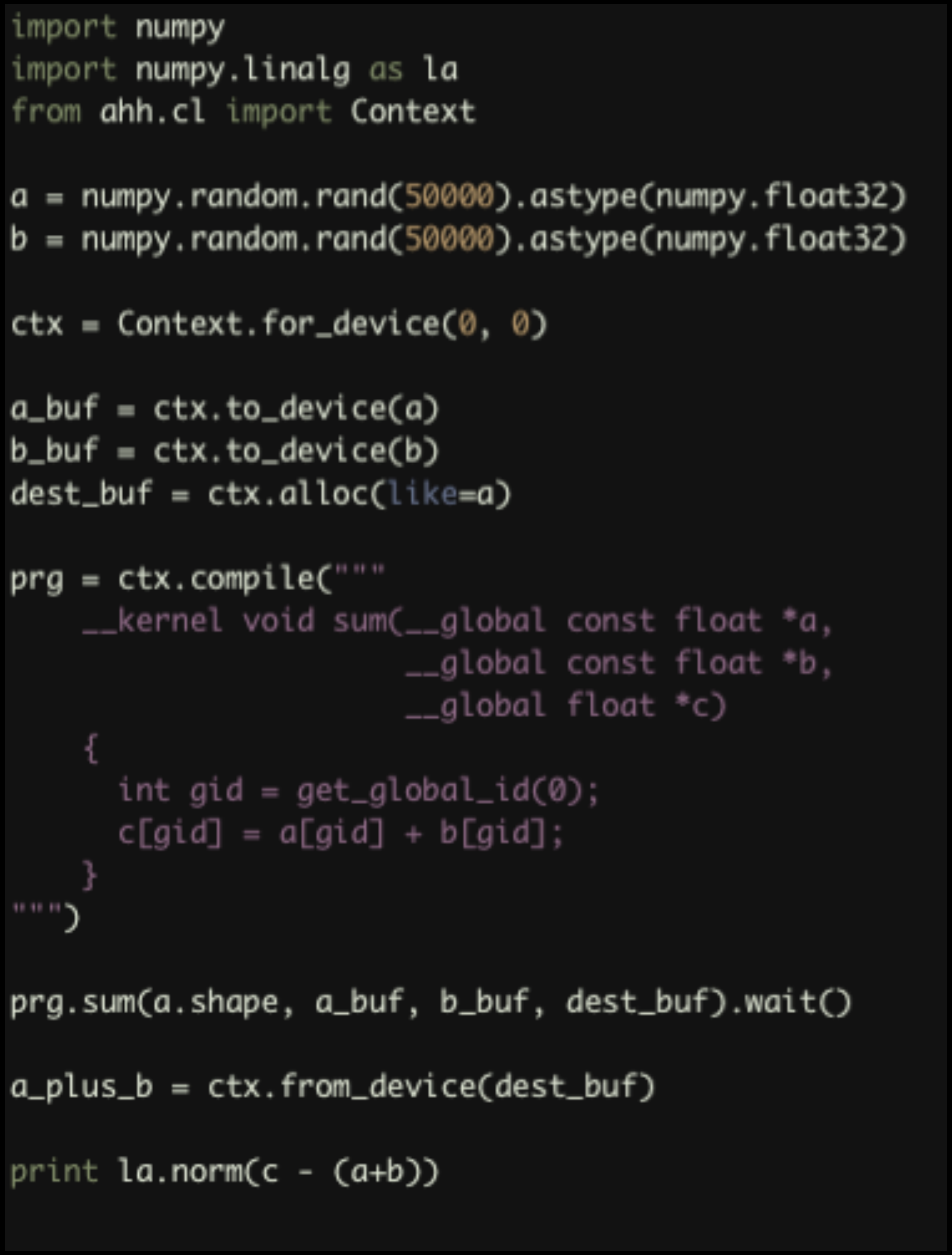
```
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

pyopencl



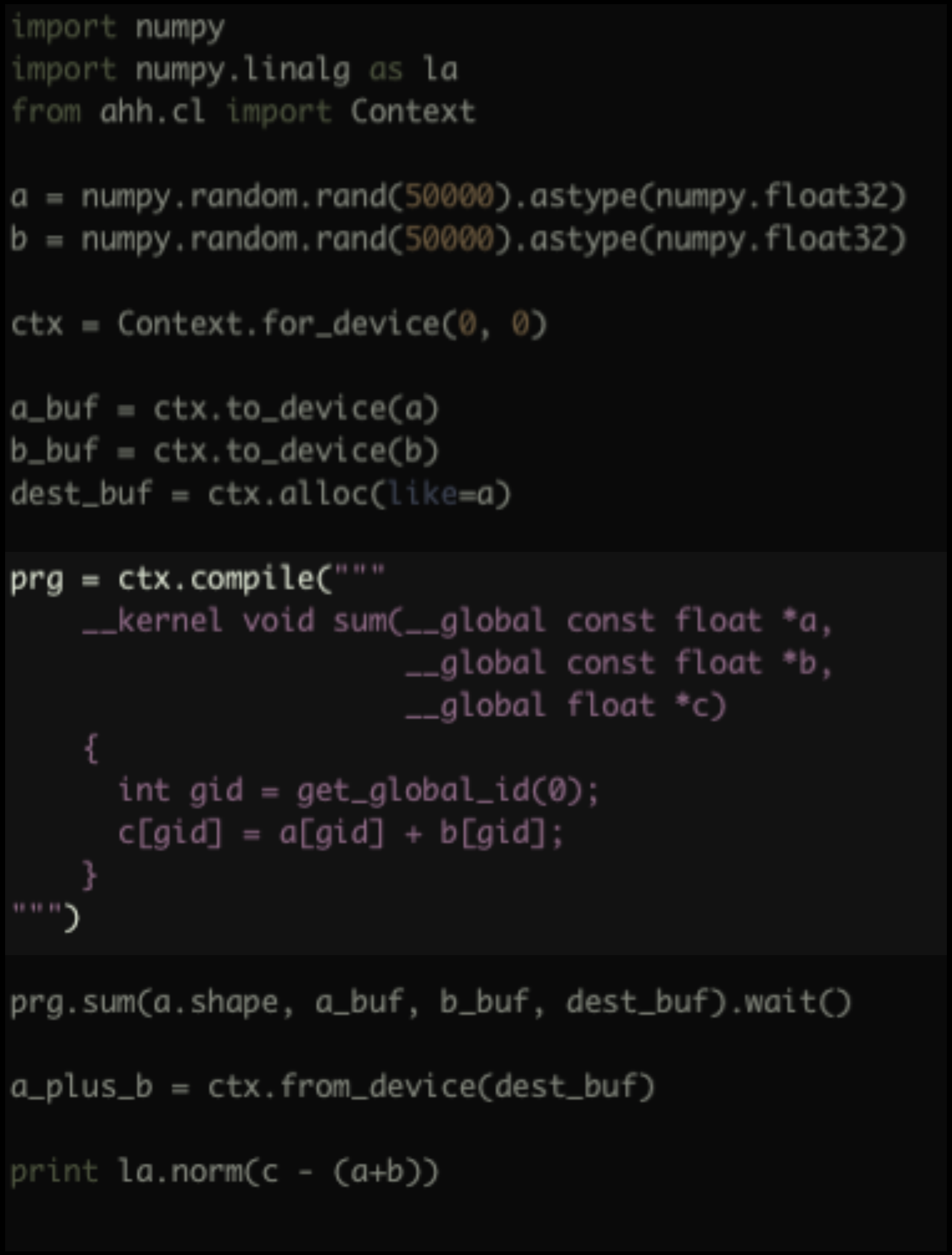
```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

ahh.cl



```
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

OpenCL

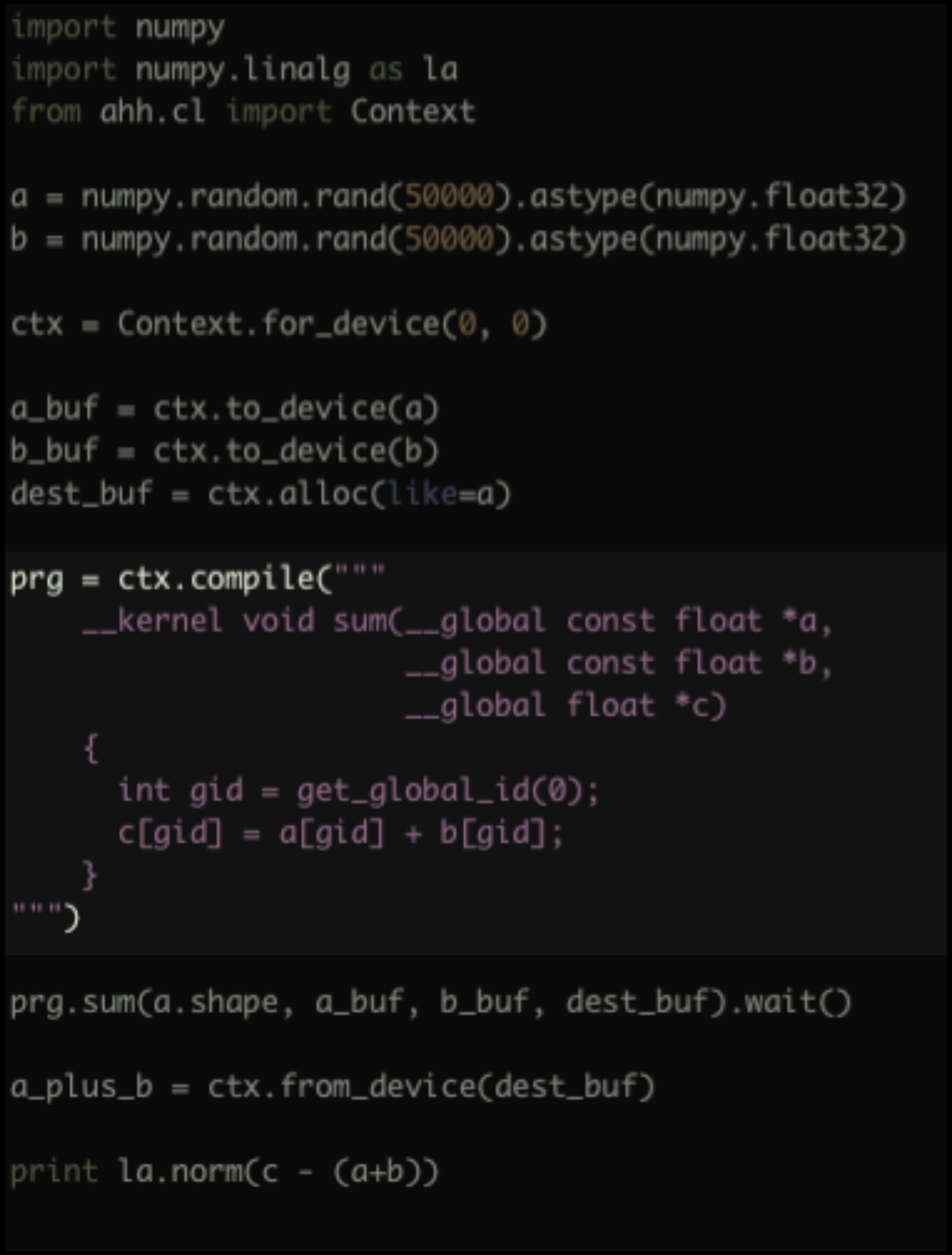


```
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

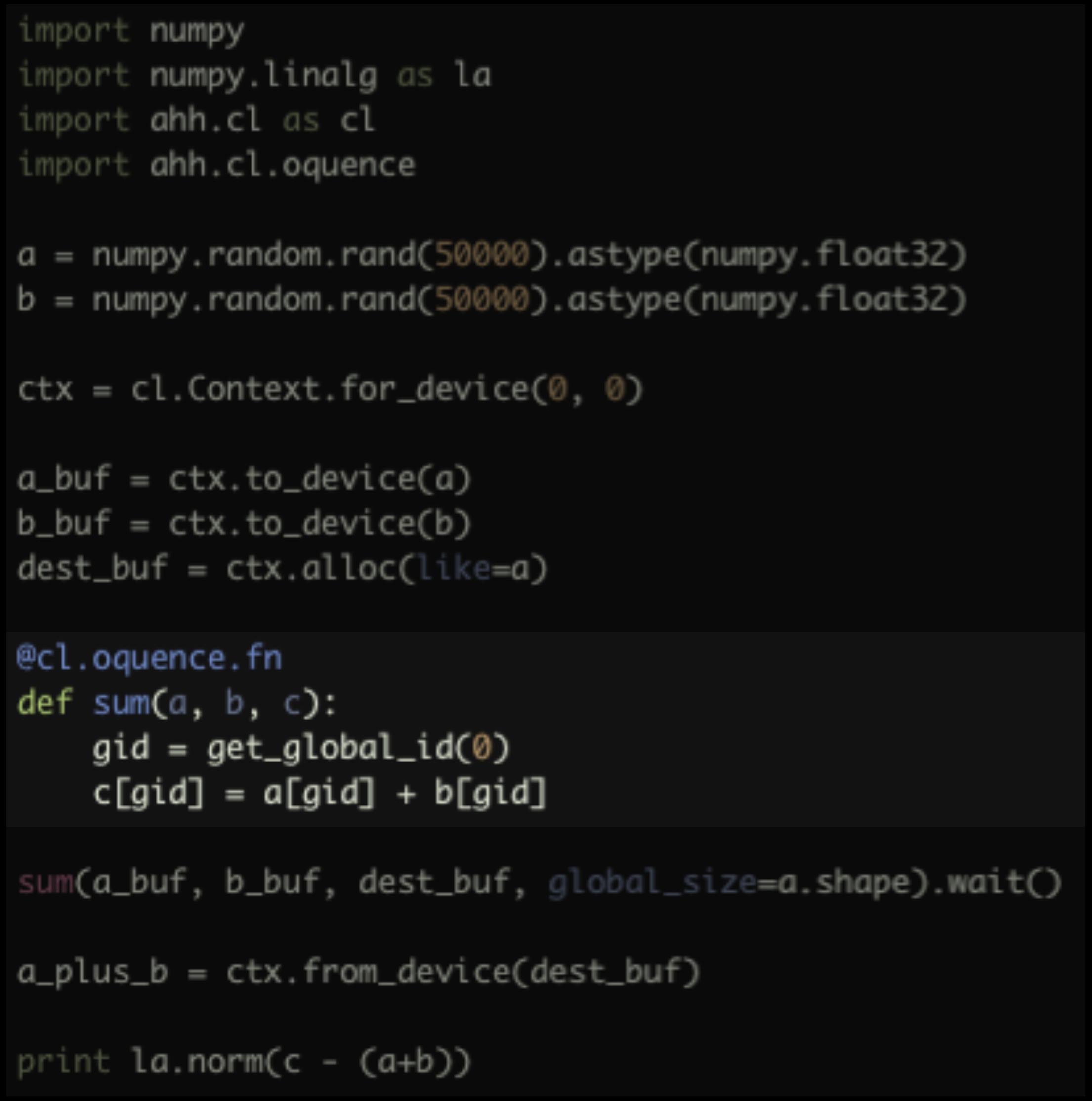
Problem 2: The OpenCL **language** is itself not very usable.

Solution:
cl.oquence

OpenCL



```
import numpy
import numpy.linalg as la
from ahh.cl import Context

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

prg = ctx.compile("""
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
""")

prg.sum(a.shape, a_buf, b_buf, dest_buf).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

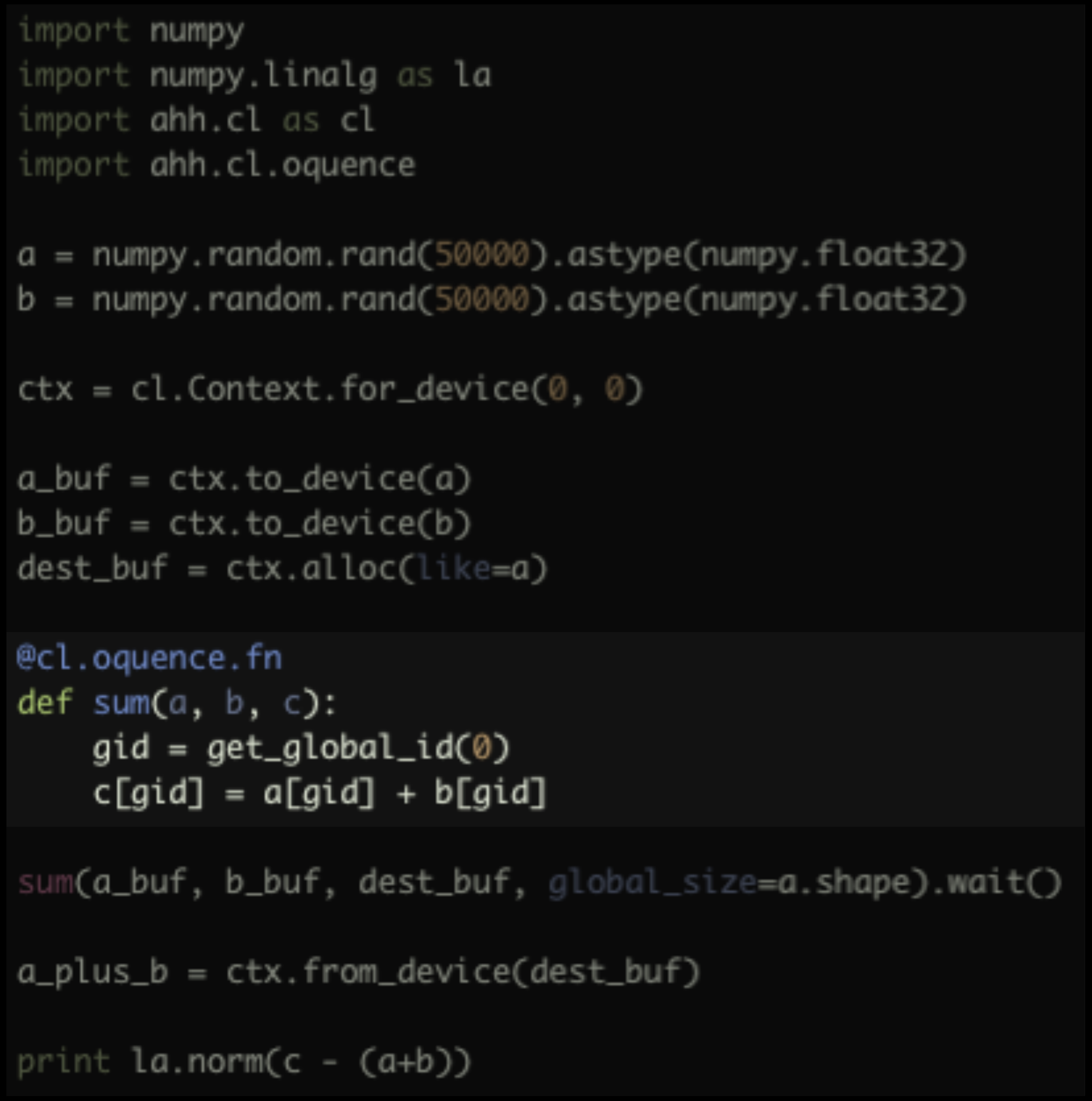


```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

is a
minimal, Pythonic
syntax
for OpenCL
(+extensions)

Solution: `cl.oquence`

> “Everything should be made as simple as possible, but not simpler.”
>
> **Albert Einstein**



```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

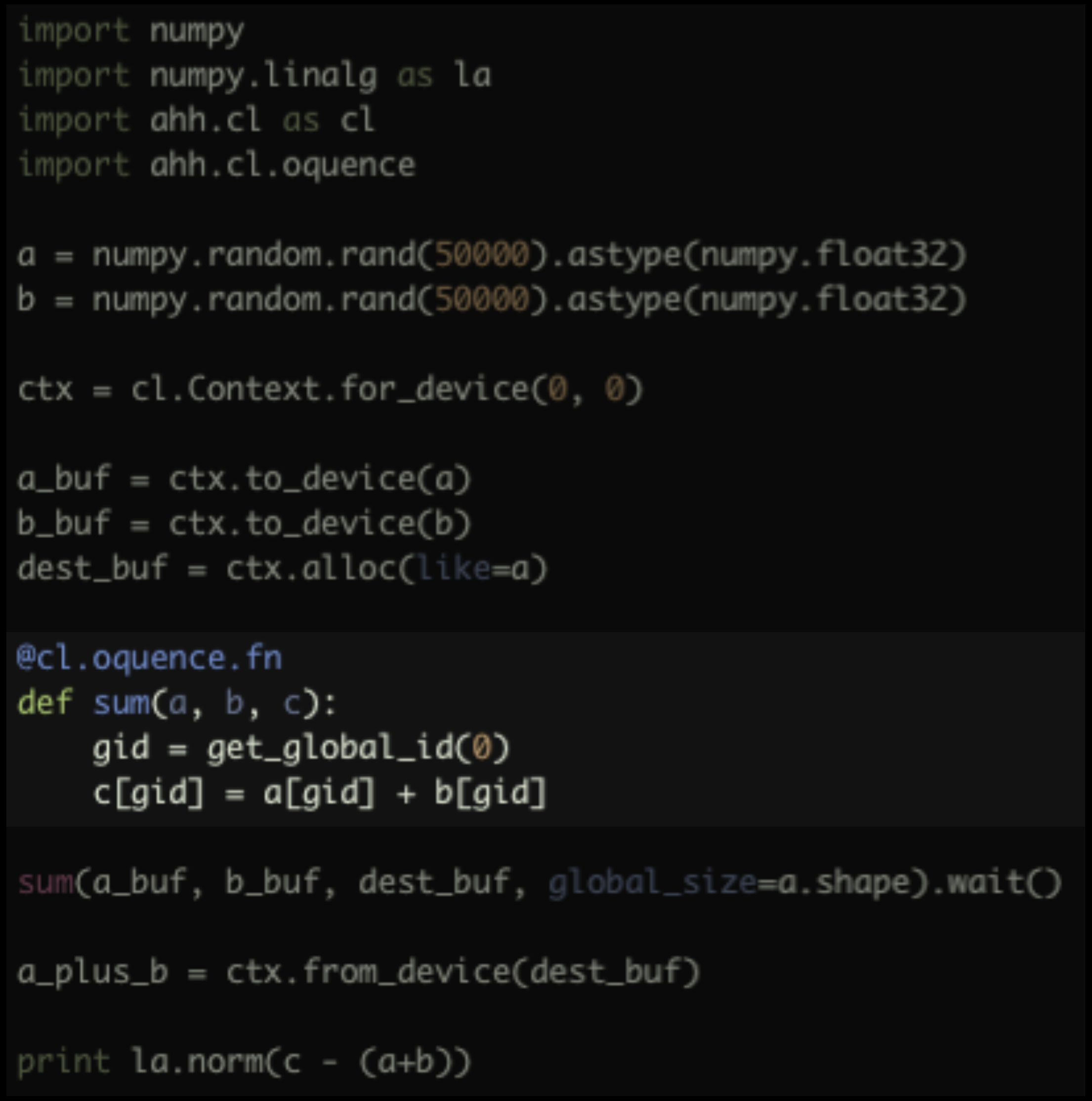
is a
minimal, Pythonic
syntax
for OpenCL
(+extensions)

`cl.oquence`

`sum` is a generic function

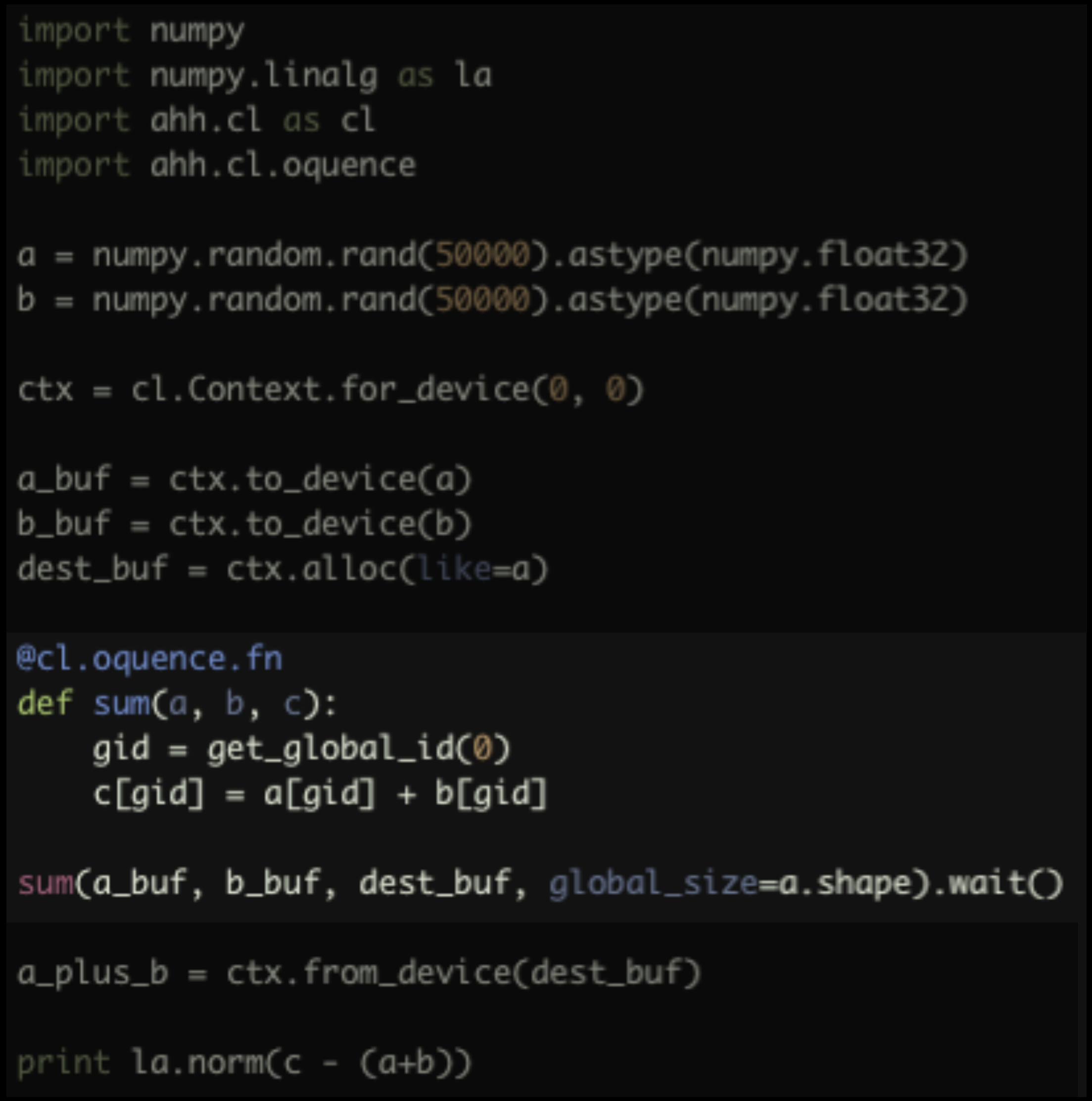
(an instance of `GenericFn`, in fact)

it expresses *what* the sum algorithm is, but without types, it is still unspecified how to do it



```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

cl.oquence

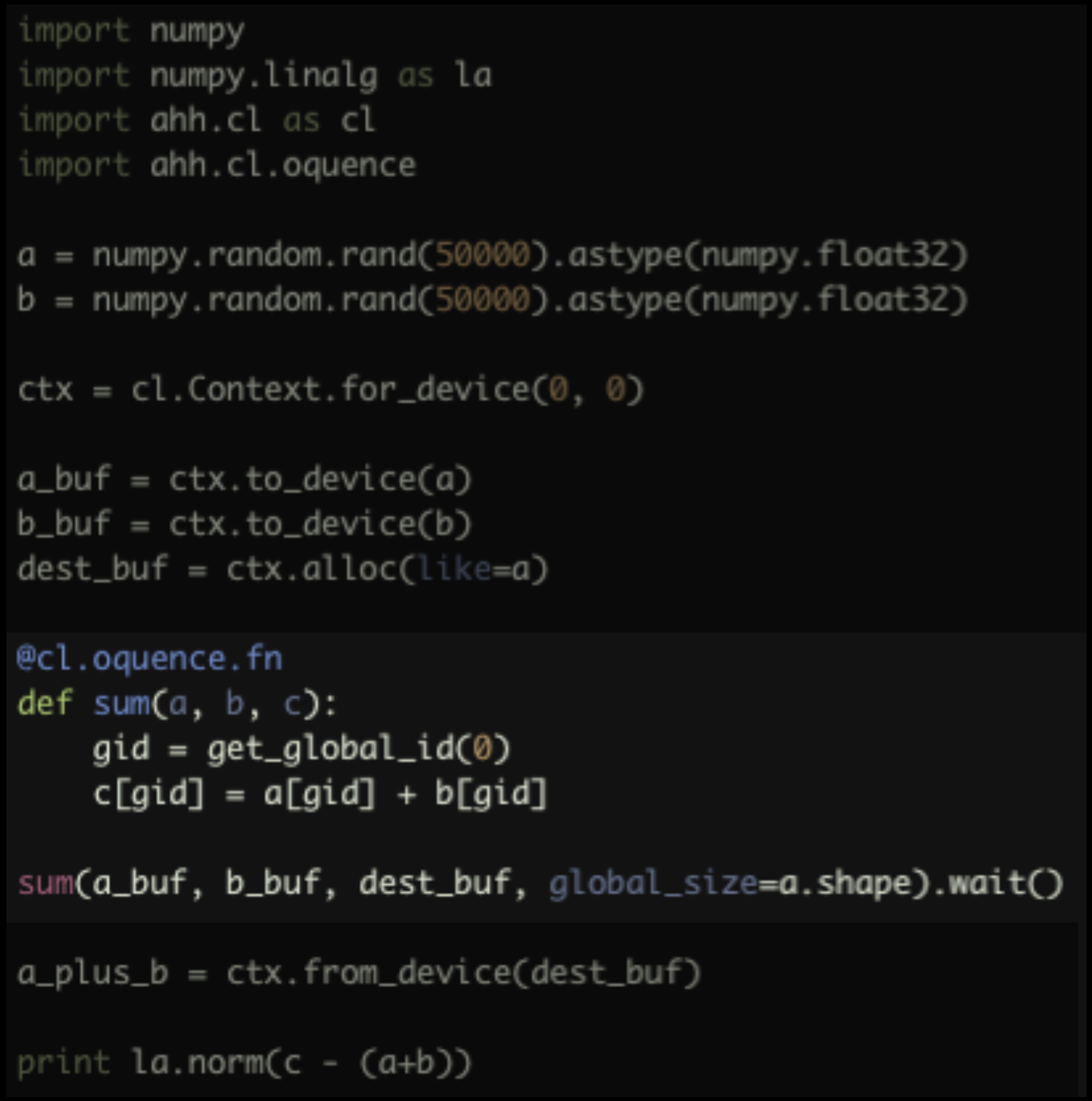


```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

when you call `sum`, the types of the arguments become known (because `ahh.cl` saved them!)

`cl.oquence`

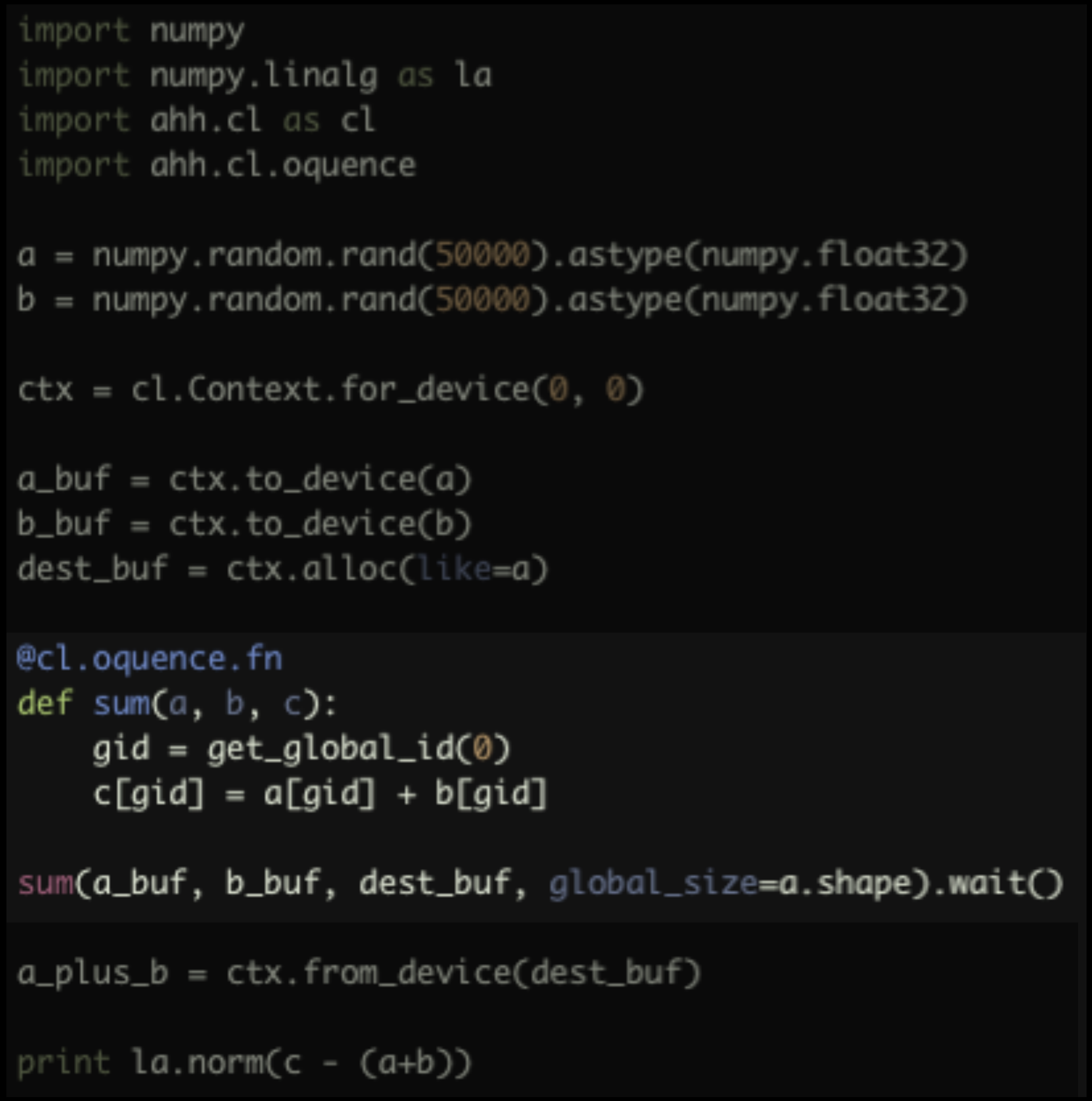
call-time type inference: the types of all the intermediate expressions, and thus the types of the local variables and return type of the function, can be exactly inferred once there are no free variables



```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

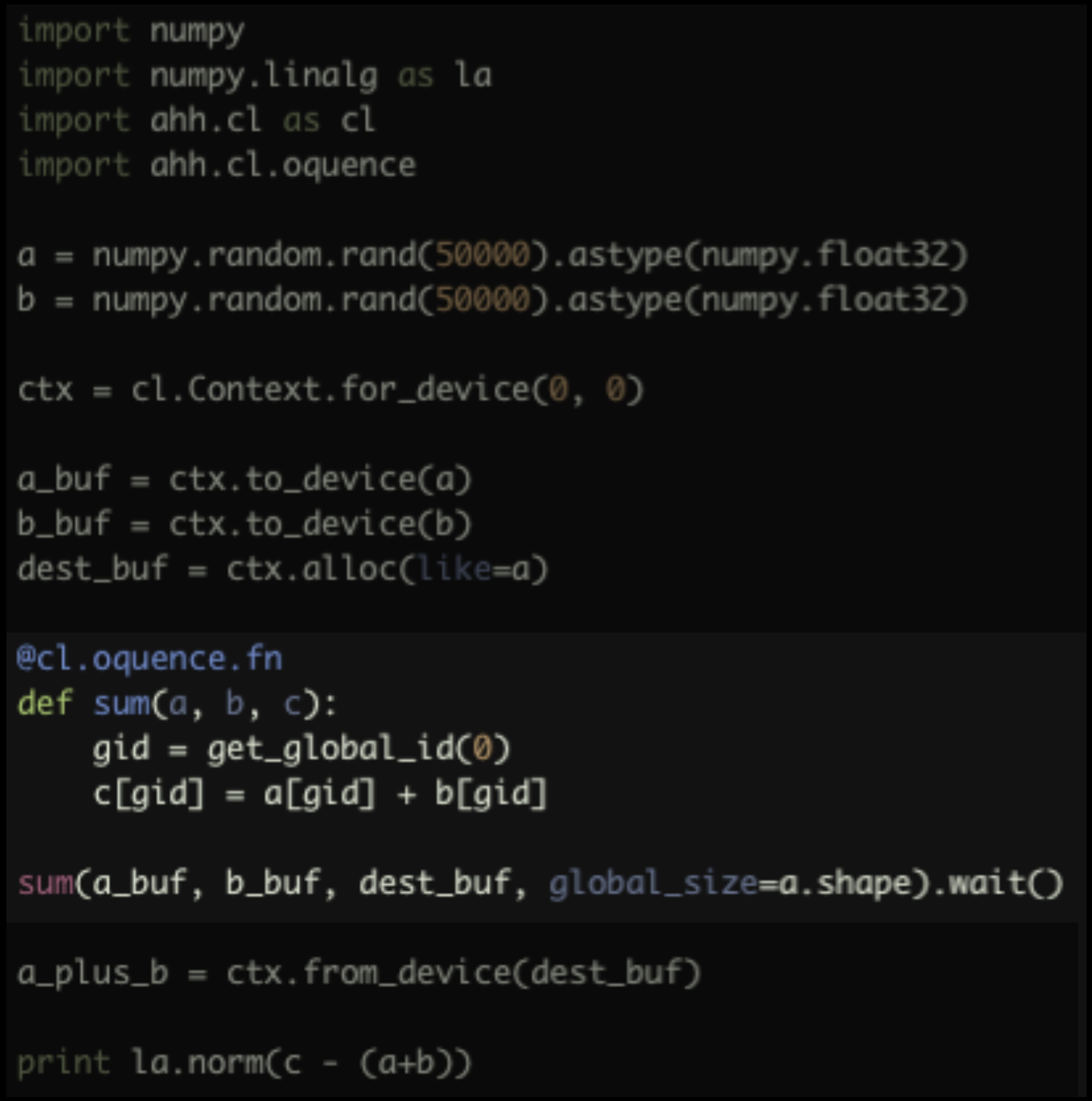
`cl.oquence`

a: `__global float*`
b: `__global float*`
c: `__global float*`
gid: `size_t`
a[gid]: `float`
b[gid]: `float`
...



```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

cl.oquence



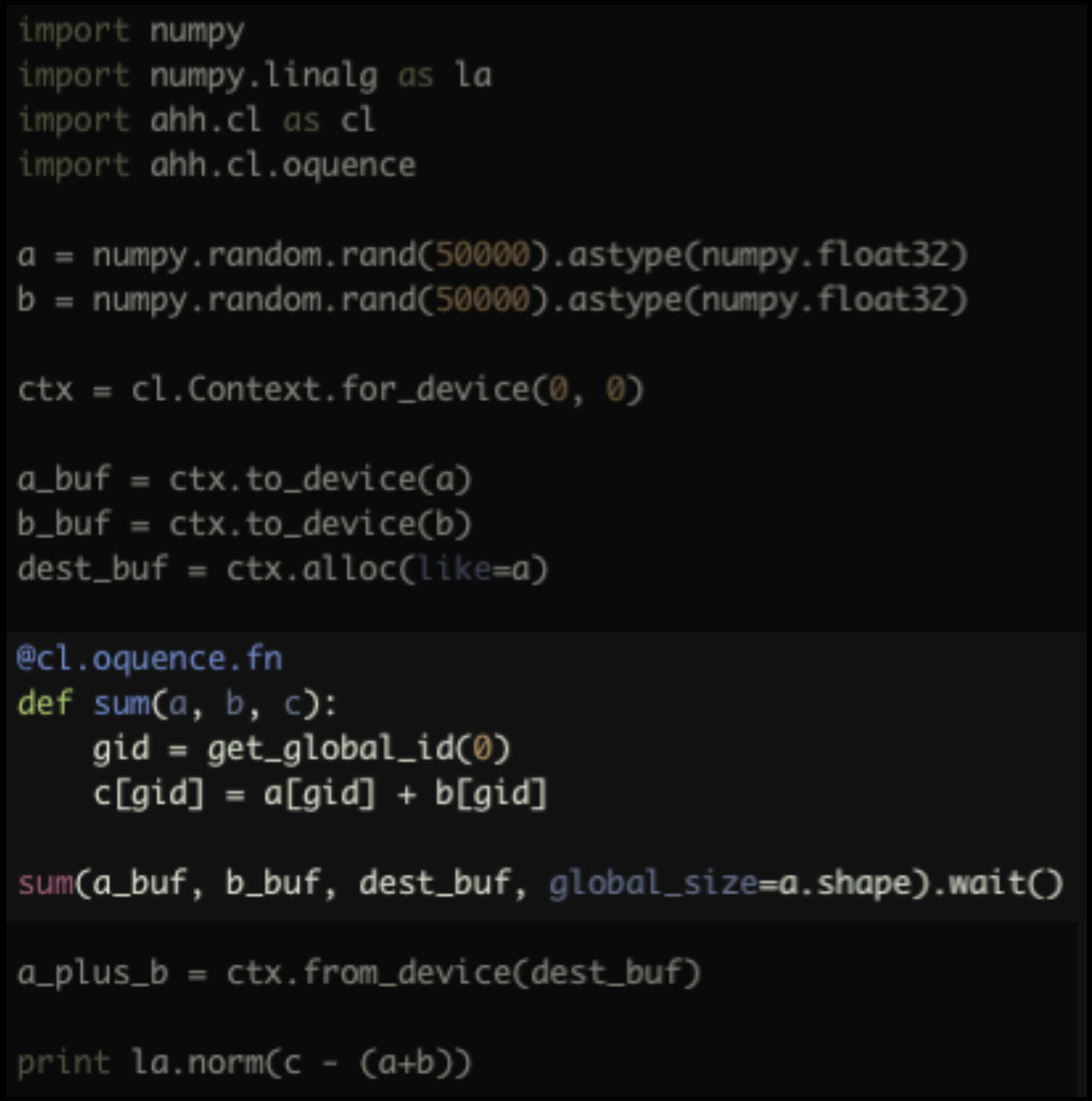
```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

From there, we can generate OpenCL source with the correct types

```
>>> print sum.get_concrete_fn(cl.cl_float.global_ptr,
        cl.cl_float.global_ptr, cl.cl_float.global_ptr).program_source

__kernel void sum(__global float* a, __global float* b, __global float* c)
{
    size_t gid;
    gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

cl.oquence

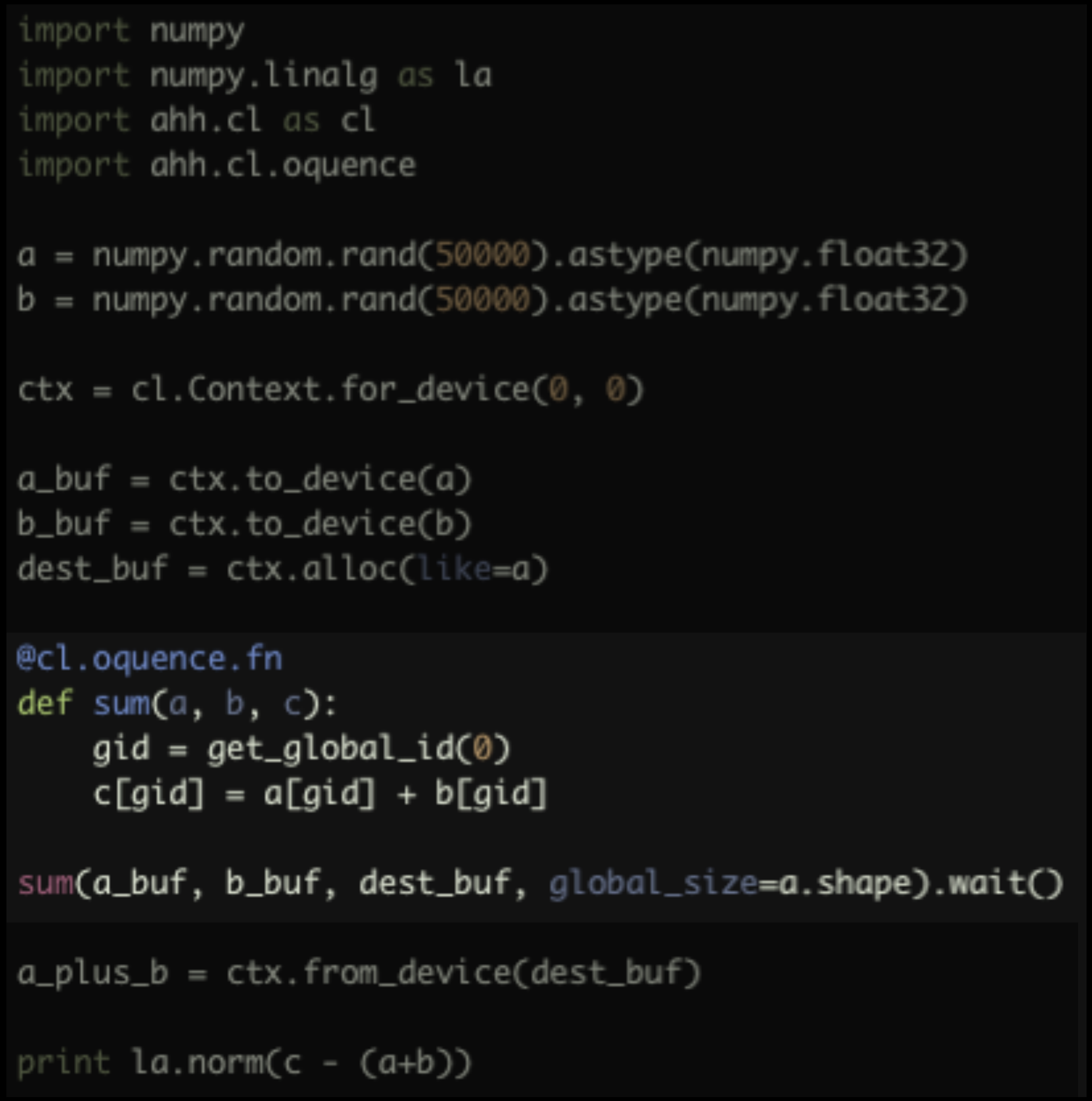
All functions are **automatically** generic, so you can call sum with different types later and it will generate a new (properly renamed) version of sum with those types too.



```python
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

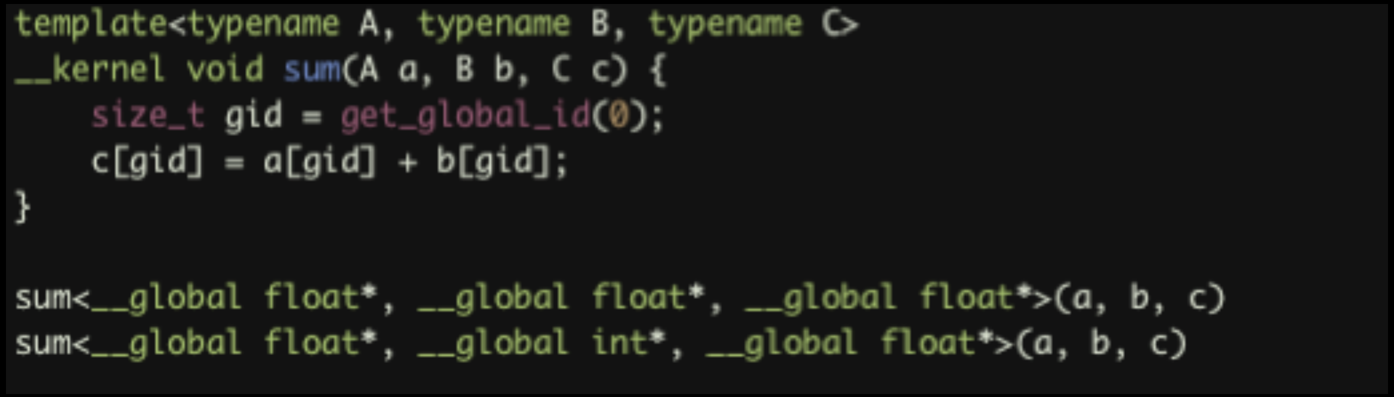
```
>>> print sum.get_concrete_fn(cl.cl_int.global_ptr,
        cl.cl_float.global_ptr, cl.cl_float.global_ptr).program_source

__kernel void sum___1(__global int* a, __global float* b, __global float* c)
{
    size_t gid;
    gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

`cl.oquence`



```python
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

This is analagous to templates in C++ (and CUDA), but implicit and concise.



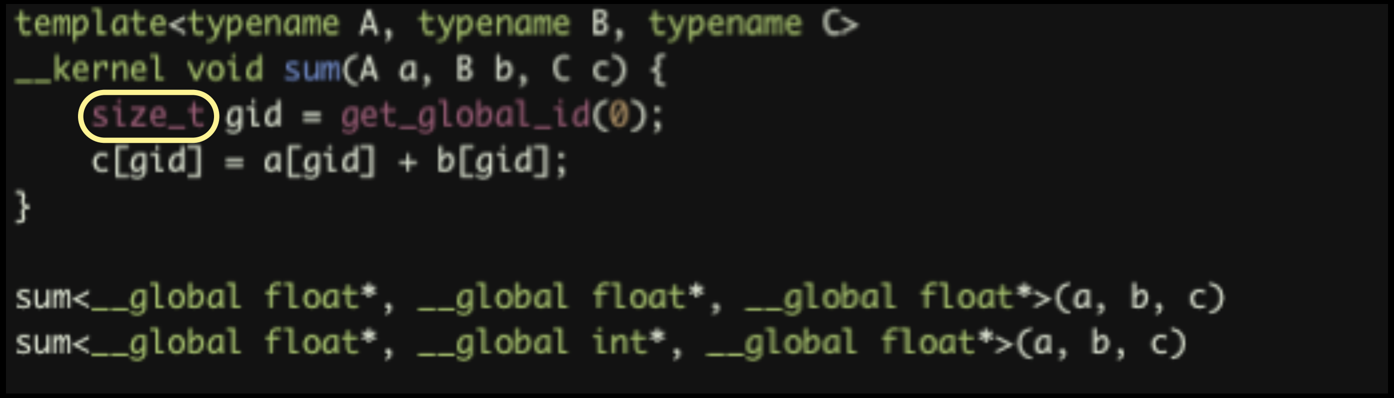
```cpp
template<typename A, typename B, typename C>
__kernel void sum(A a, B b, C c) {
    size_t gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}

sum<__global float*, __global float*, __global float*>(a, b, c)
sum<__global float*, __global int*, __global float*>(a, b, c)
```

(OpenCL doesn't have templates, anyway)

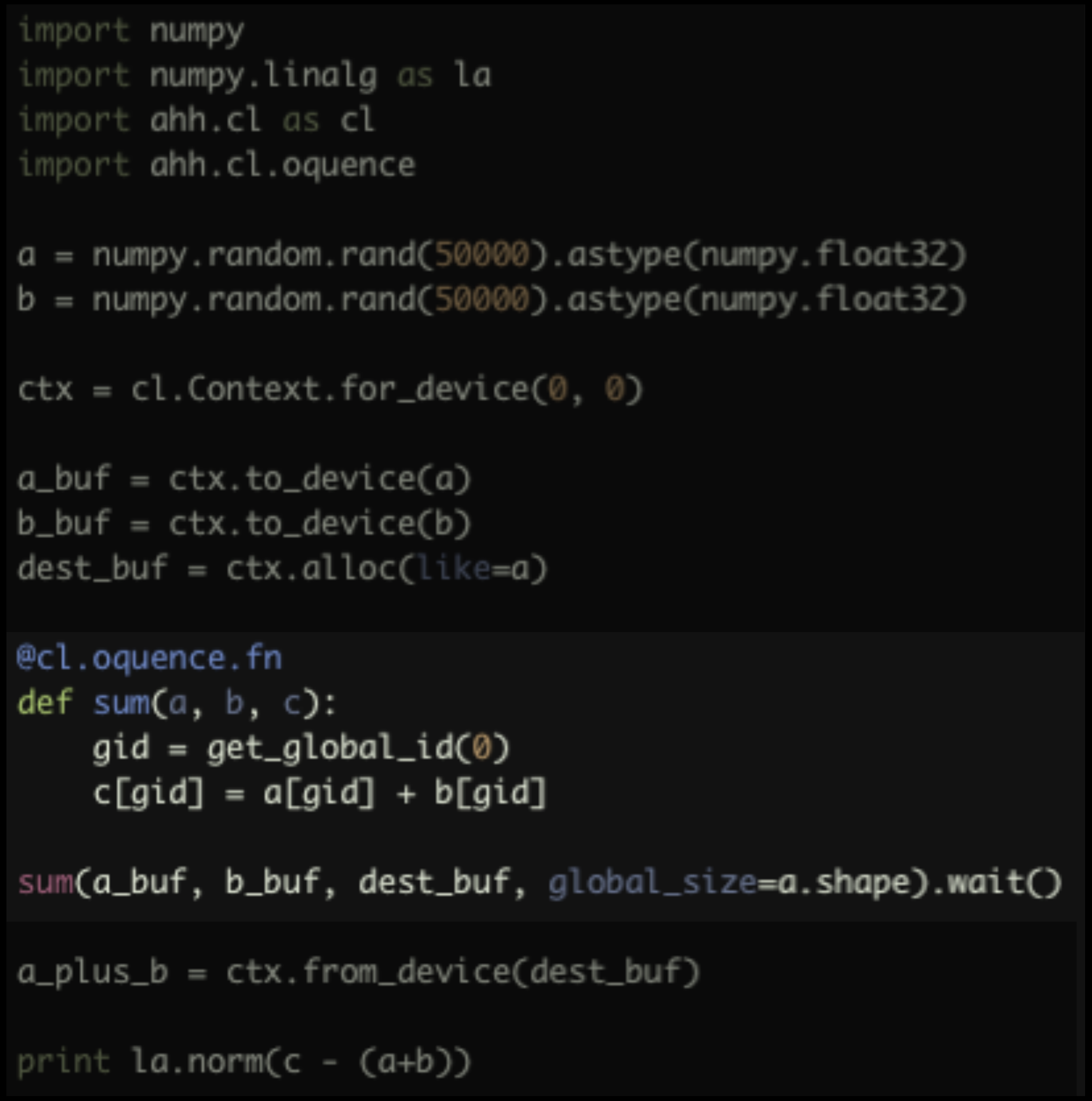
cl.oquence

Implicit typing prevents a class of errors.

Usually people use 'int' here, which could
lead to subtle errors on future platforms.



```
template<typename A, typename B, typename C>
__kernel void sum(A a, B b, C c) {
    size_t gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}

sum<__global float*, __global float*, __global float*>(a, b, c)
sum<__global float*, __global int*, __global float*>(a, b, c)
```



```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context.for_device(0, 0)

a_buf = ctx.to_device(a)
b_buf = ctx.to_device(b)
dest_buf = ctx.alloc(like=a)

@cl.oquence.fn
def sum(a, b, c):
    gid = get_global_id(0)
    c[gid] = a[gid] + b[gid]

sum(a_buf, b_buf, dest_buf, global_size=a.shape).wait()

a_plus_b = ctx.from_device(dest_buf)

print la.norm(c - (a+b))
```

cl.oquence

Type inference follows the
C99 Usual Unary and Arithmetic Conversions,
no surprises.

It understands multiple assignments and return statements if their types are compatible, even if not identical.

```
@cl.oquence.fn
def threshold(x):
    if x > 0:
        return x
    else:
        return 0
```

← this has type int, but x need not

The **full** OpenCL library and type system is supported,
including questionable antics like pointer arithmetic.

`cl.oquence`

Code is generated just-in-time just **once** for each set of argument types, minimizing overhead.

Using Python's own parser (all cl.oquence functions are thus valid Python syntax, so your editor won't choke)

A **large** spiking neuron network kernel took on the order of **10ms** to go through this process. More aggressive caching to disk can also be used.

`cl.oquence`

also supports

extension inference

(for some common extensions)

`cl_khr_fp16`, `cl_khr_fp64`,
`cl_khr_byte_addressable_store` and *all atomics*

and

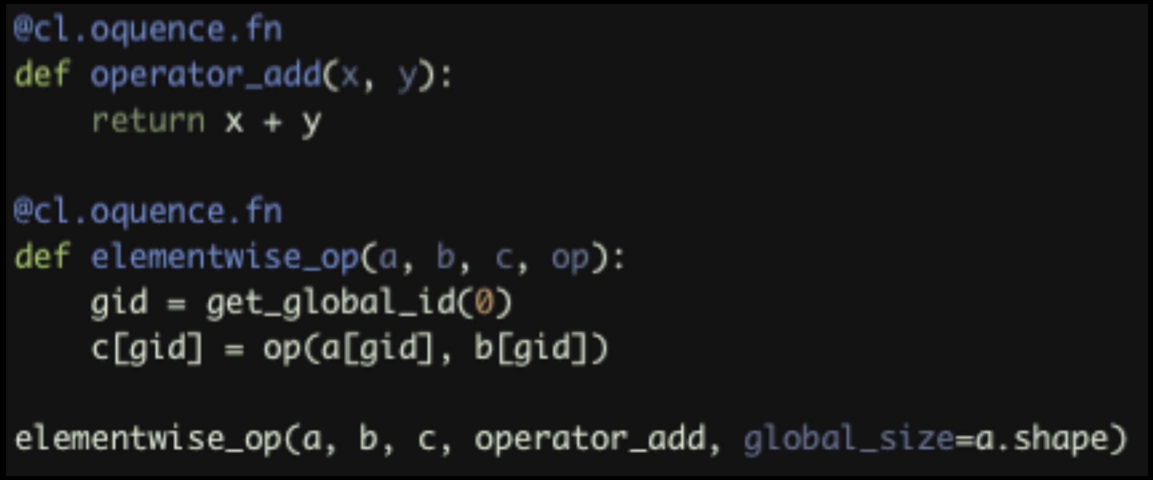
kernel inference

`cl.oquence`

also supports

higher order functions

(at compile-time)



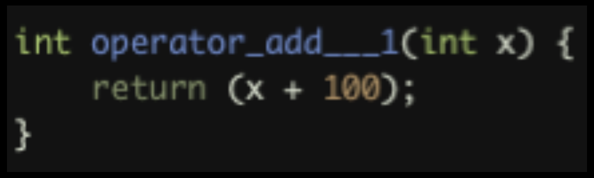
```
@cl.oquence.fn
def operator_add(x, y):
    return x + y

@cl.oquence.fn
def elementwise_op(a, b, c, op):
    gid = get_global_id(0)
    c[gid] = op(a[gid], b[gid])

elementwise_op(a, b, c, operator_add, global_size=a.shape)
```

`cl.oquence`

also supports

dynamic constant binding

```
add_100 = cl.oquence.fn(operator_add, y=100)
```



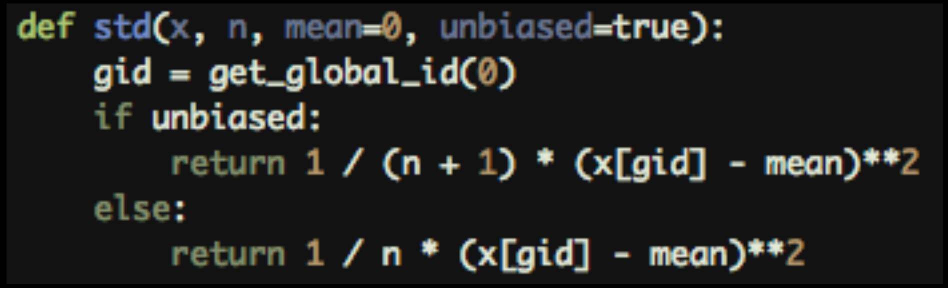
```
int operator_add___1(int x) {
    return (x + 100);
}
```

you can inline buffers as constants too.

future backends may use this to speed up code by relieving register pressure.

`cl.oquence`

also supports
default arguments

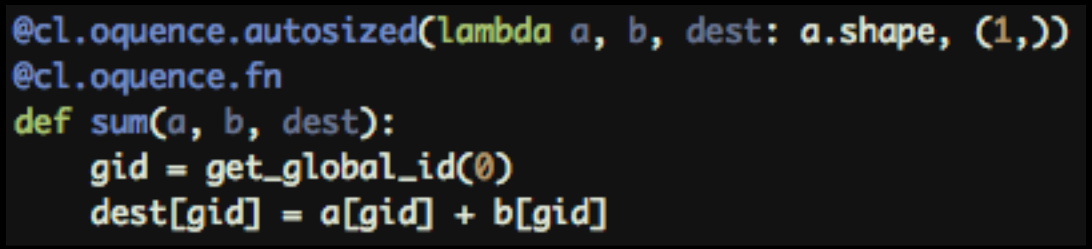


```
def std(x, n, mean=0, unbiased=true):
    gid = get_global_id(0)
    if unbiased:
        return 1 / (n + 1) * (x[gid] - mean)**2
    else:
        return 1 / n * (x[gid] - mean)**2
```

`cl.oquence`

also supports

automatic kernel size calculators



```python
@cl.oquence.autosized(lambda a, b, dest: a.shape, (1,))
@cl.oquence.fn
def sum(a, b, dest):
    gid = get_global_id(0)
    dest[gid] = a[gid] + b[gid]
```

I’ve shown you

an **alternative syntax** for OpenCL
with **automatic type inference**
that *looks like Python*.

It functions as an aggressive just-in-time
(JIT) compiler, embedded within Python.

I've shown you

an **alternative syntax** for OpenCL
with **automatic type inference**
that *looks like Python*.

It functions as an aggressive just-in-time (JIT) compiler, embedded within Python.

IT IS NOT A PYTHON TO OPENCL COMPILER.

No indirection.
No dynamic types.
No Python library functions.

Embracing A Hybrid Approach

By building a system defaulting to Python's dynamicism for **productivity-limited tasks**, but integrating a well-designed statically-typed data-parallel language based on industry standards as a library for **performance-limited tasks**, we achieve a flexible balance.

Related to the concept of **gradual typing**.

Problem 3: Language Extensibility

The extensions I showed you are nice, but
we aren't done researching language features
to support high-performance computing.
&
Domain-specific languages have proven to be useful
tools for practitioners.

Ideally, the language's semantics should be extensible
so that **PL research** can make it out to users who need it.

An extensible type system

cl.oquence has an
extensible type system
based on a merging of the concepts of a **macro**
and a **metaobject protocol**

When the type inference and code generation syntax tree visitors encounter an expression, they pass the appropriate subtrees of the syntax tree to a **type implementation** written in Python, e.g.

```
a[b] ⟶ infer(a).infer_Subscript(Name(‘a’), Name(‘b’), visitor)
       infer(a).generate_Subscript(Name(‘a’), Name(‘b’), visitor)
```

```
def ptr_infer_Subscript(arr, idx, visitor):
    infer = visitor.infer
    idx_type = infer(idx)
    if isinstance(idx_type, IntegerType):
        arr_type = infer(arr)
        return arr_type.target_type
    raise InvalidTypeError(...)
```

```
def ptr_generate_Subscript(arr, idx, visitor):
    generate = visitor.generate
    return (generate(arr), “[“, generate(idx), “]”)
```

`An extensible type system`

cl.oquence has an
extensible type system
based on a merging of the concepts of a **macro**
and a **metaobject protocol**

This is like operator overloading or a metaobject protocol, combined with LISP-style macros. The following operations can be implemented:

Assignment
Subscript Access and Assignment
Attribute Access and Assignment
All Unary and Binary Operators
Functor Syntax

An extensible type system

cl.oquence has an
extensible type system
based on a merging of the concepts of a **macro**
and a **metaobject protocol**

With these, the *entirety of the C99 type system* is implemented as a pluggable library with no runtime performance overhead.

But if you're in academia, pick your favorite type system with stronger guarantees!

An extensible type system

cl.oquence has an
extensible type system
based on a merging of the concepts of a **macro**
and a **metaobject protocol**

Many mechanisms that would have been implemented in entirely new *compilers* can be implemented as *libraries* without any run-time performance overhead.

Higher-order functions
Type cast syntax
Units for numeric types
A dynamic object type (real gradual typing!)
Domain-specific constructs and transformations
High-performance computing primitives
Typestate?
...

Summary I

Scientific applications demand both **productivity** and **performance**. By embedding an established static data parallel language with call-time type inference into an established dynamic host language, we can achieve a balance between these needs **today**.

Summary II

More generally, cl.oquence provides an **extensible** language framework, allowing PL researchers to distribute better type systems as libraries, and domain and platform experts to implement domain-specific constructs and constraints into the language.

The extensible type system in cl.oquence leverages Python as a metalanguage for defining **type implementations**, a system which is intuitive and extremely flexible.

atomic hedgehog

Everything is developed on a **public repository**.
It's **well documented!**
There is an **issue tracker**.

Mailing lists:
ahh-announce
(if you just wanna know about releases)

ahh-discuss
(discussions, suggestions, questions)

ahh-dev
(commit messages, bug reports, etc. are sent here)

http://ahh.bitbucket.org/ (docs)
http://bitbucket.org/ahh/ahh (hg repository)

atomic hedgehog

This isn't a throwaway research language.
It is meant to be used!

http://ahh.bitbucket.org/ (docs)
http://bitbucket.org/ahh/ahh (hg repository)

http://cyrus.omar.name/

DOE CSGF

Carnegie Mellon

CNBC
Center for the Neural Basis of Cognition

Discussions with Michael Rule
(BS, CMU CS; will be at Brown) have been helpful.

He also came up with the name, designed the sweet logo and set up the documentation system!