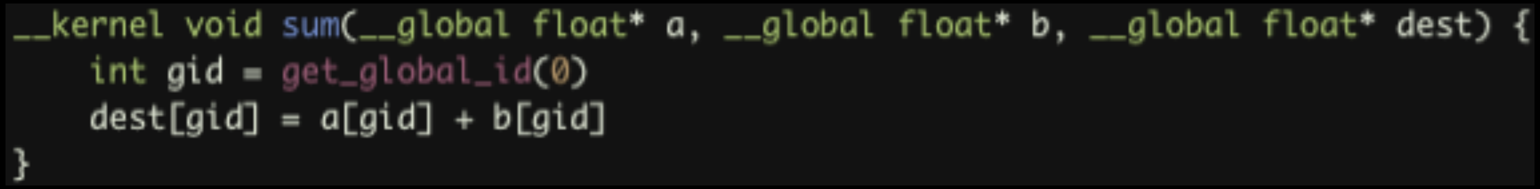
Introduction to OpenCL Programming in Python

DATA PARALLELISM

You want to break your work into thousands of **work items**, each running the same **kernel**.

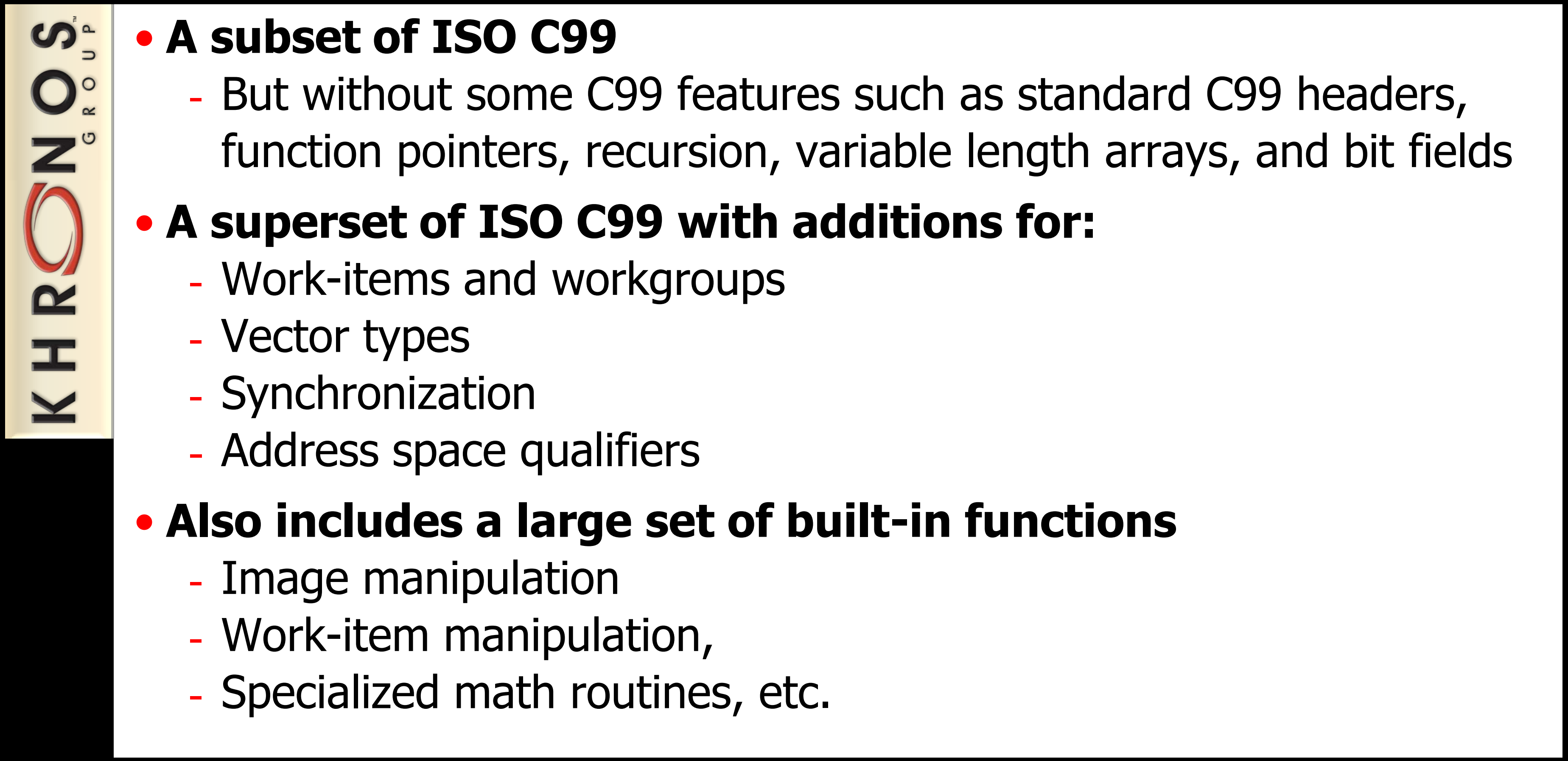
(e.g. do ~ the same thing to each pixel in an image, each neuron in a simulation, each element of a vector)

Example: adding two **buffers** (vectors)



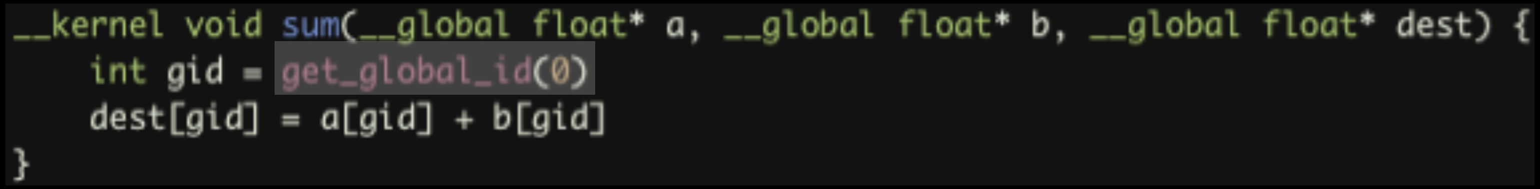
```
__kernel void sum(__global float* a, __global float* b, __global float* dest) {
    int gid = get_global_id(0)
    dest[gid] = a[gid] + b[gid]
}
```

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.

Example: adding two **buffers** (vectors)



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

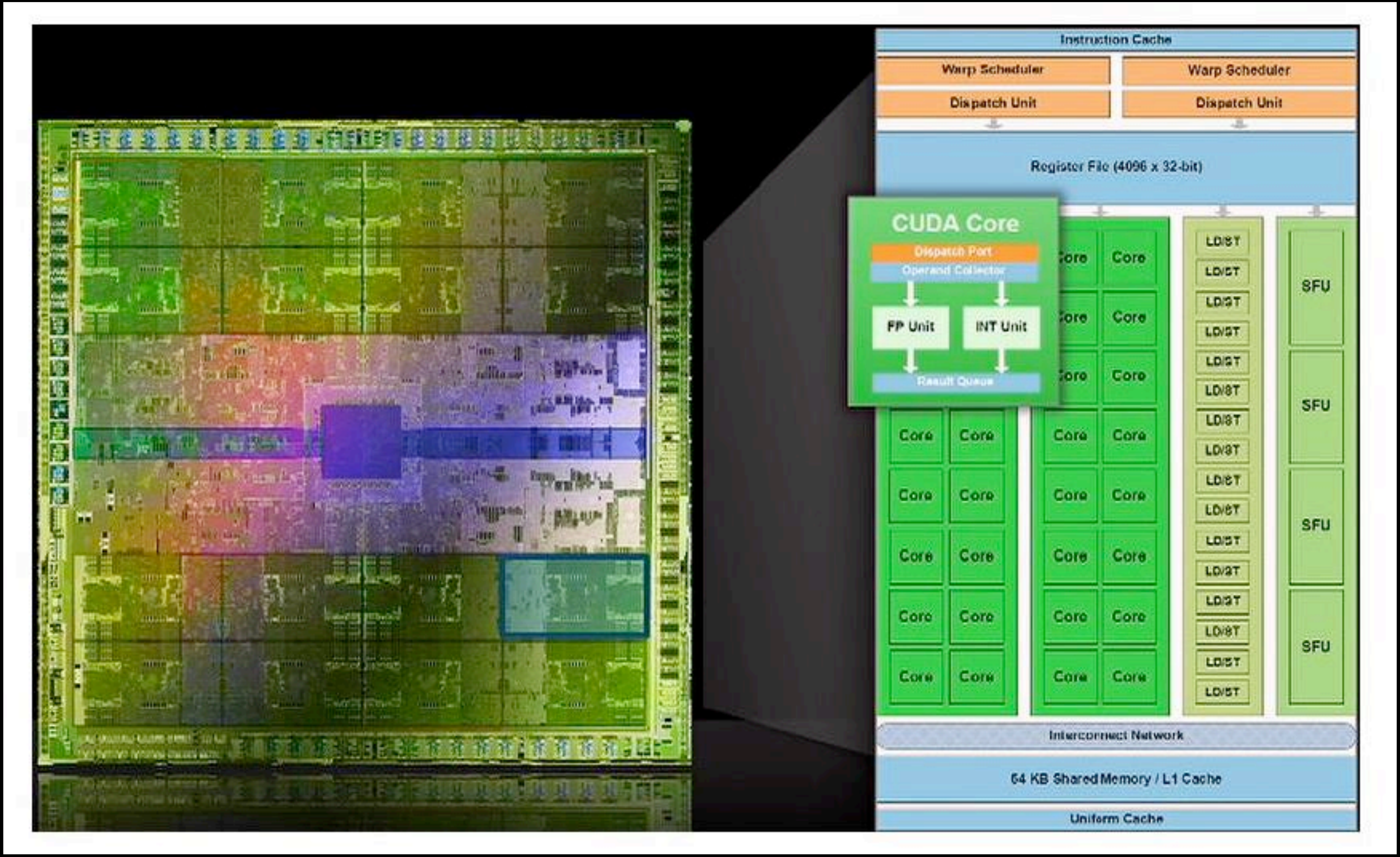
The **global id** allows us to assign each element of our vector (or each pixel, or each neuron) to a different work item.

DATA PARALLELISM

For convenience, the global id can have up to three dimensions. The ‘0’ above reads the first (and only, in this case) dimension.

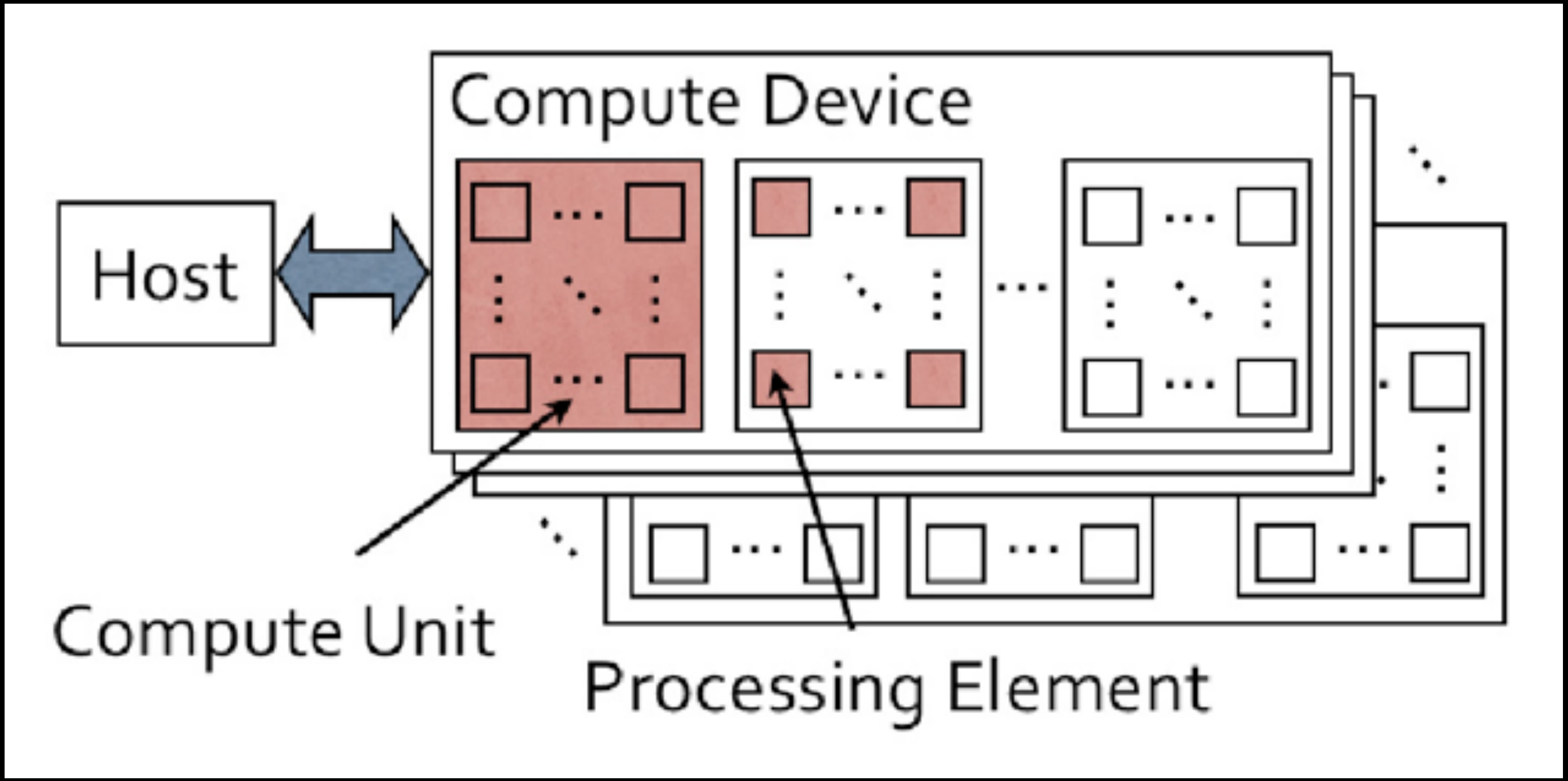
What is a GPU?

a couple dozen parallel **compute units**



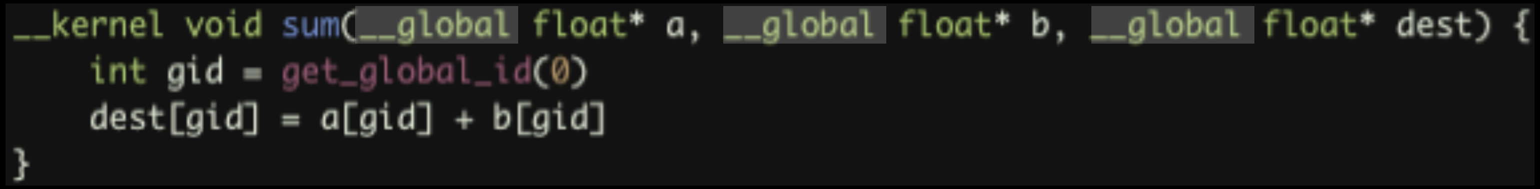
What is a GPU?

Each compute unit consists of several **processing elements**.



All processing elements in a compute unit execute the same instruction on every cycle
(or **wait** if they are in a different branch of the program.)

The Memory Model: Global Memory

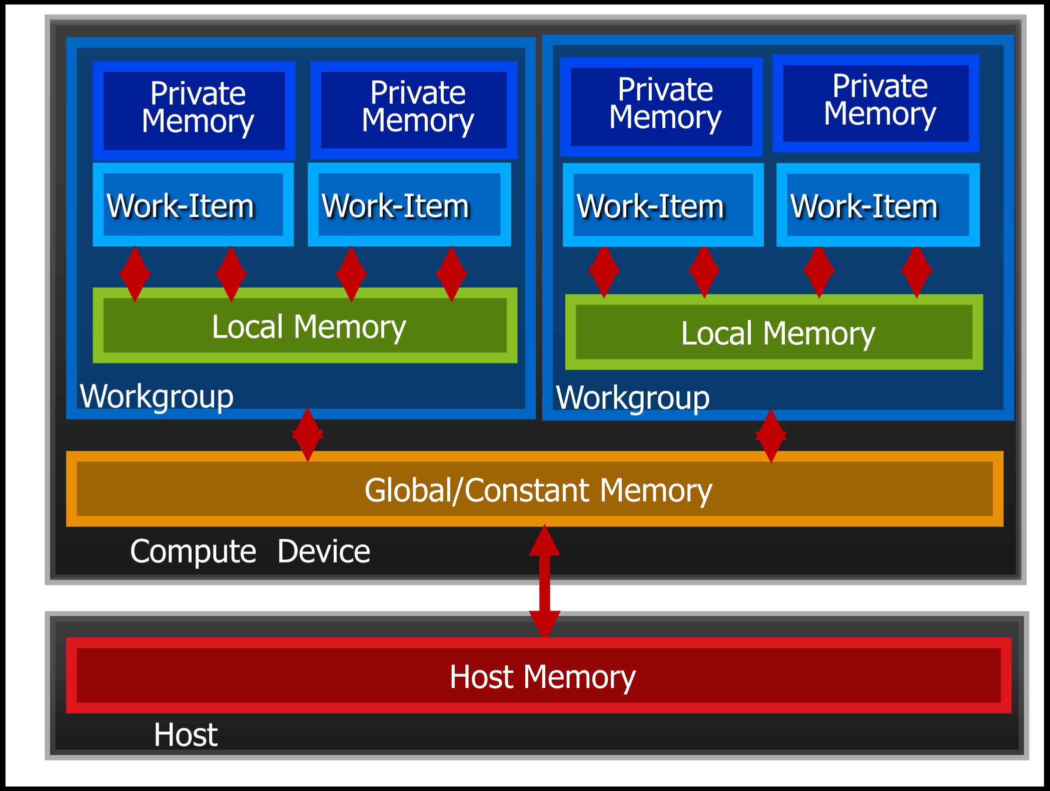


```
__kernel void sum(__global float* a, __global float* b, __global float* dest) {
    int gid = get_global_id(0)
    dest[gid] = a[gid] + b[gid]
}
```

Each **device** has a large amount of **global memory**.

- Slow (~150 arithmetic ops per read or write on nVidia cards)
- Should **coalesce access**
  - adjacent work items should access adjacent indices

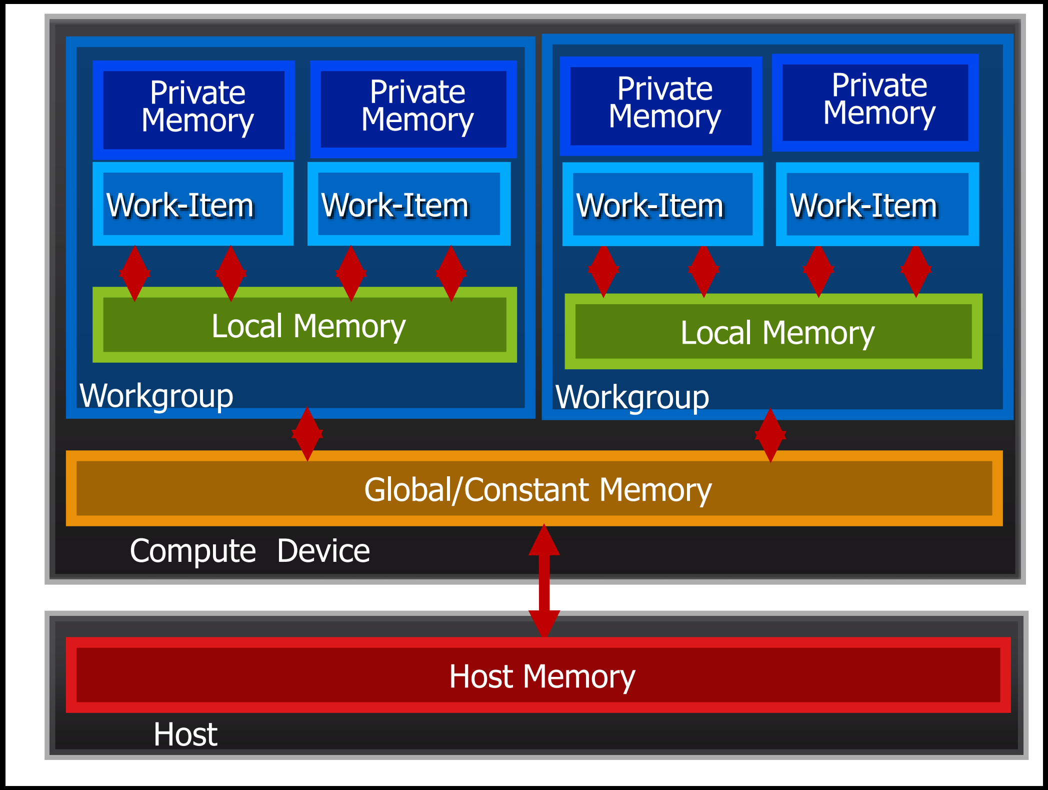
    actually slightly more relaxed: each **half-warp** (16 sequentially indexed threads) should access a contiguous region of memory



The Memory Model: Local Memory

Each **compute unit** has a small amount of **local memory**.

- **Fast** (~same as registers on nVidia)
- **Transient**: cleared between kernel invocations (i.e. global barriers)
- A **shared scratch space** for work items in a work group.

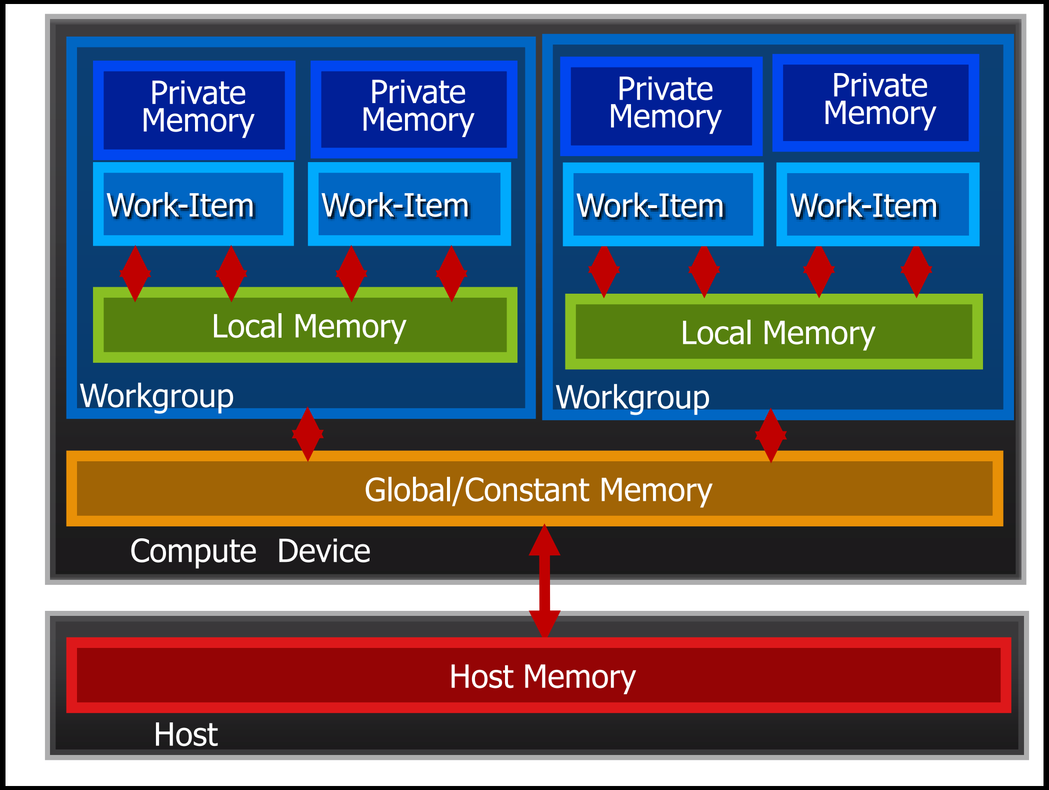


Memory management is **explicit**.

small, fast, unshared, transient (*stack*)

small, fast, shared within work group, transient

big, slow, persistent



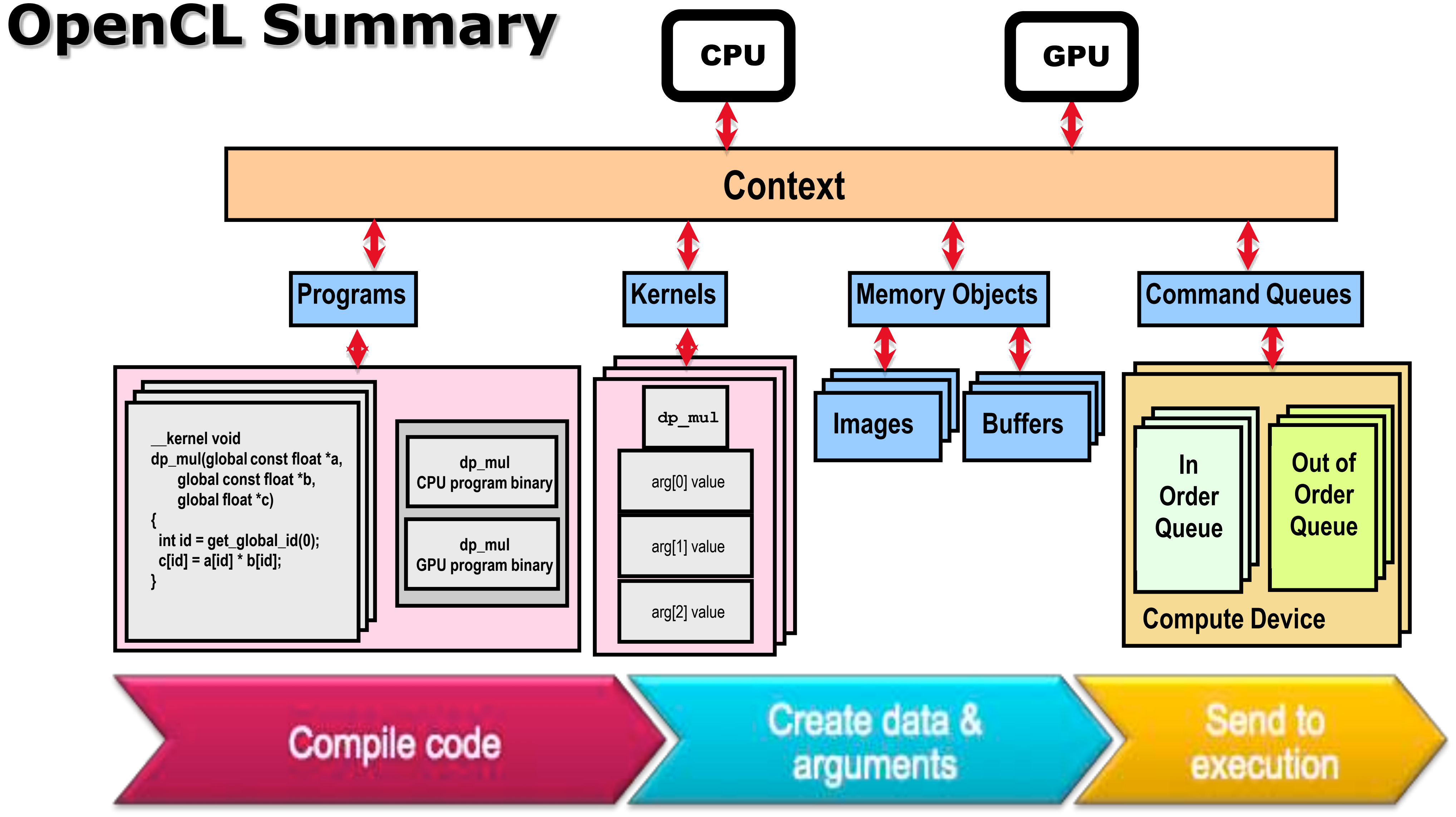
Must move data from
host to **global** to **local** and **back**.

Want speed?

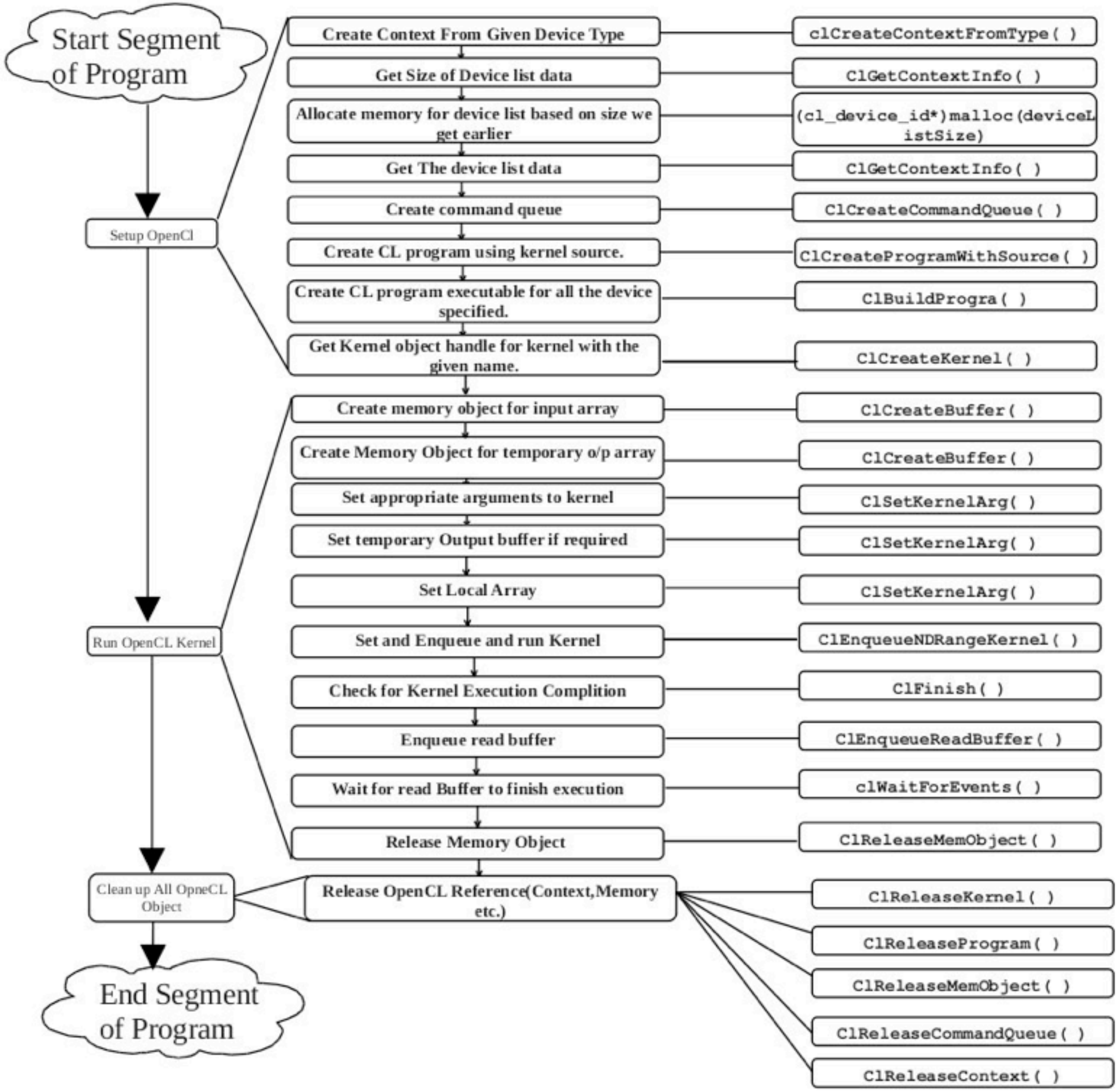
- Data parallel algorithms: should not branch very much
- Computation:I/O ratio should be high
    - If low, need enough threads to hide the latency
- Coalesced memory access patterns
- Good usage of local memory to share common data between work items in a work group
- Minimize host-device transfer - keep it on the device until you absolutely need to bring it back

- >2 TFLOPS (single precision) per ***device***
- 512MB-6GB of ***global memory***
- 16kb of local memory
- Everything is manually managed via ***host*** code

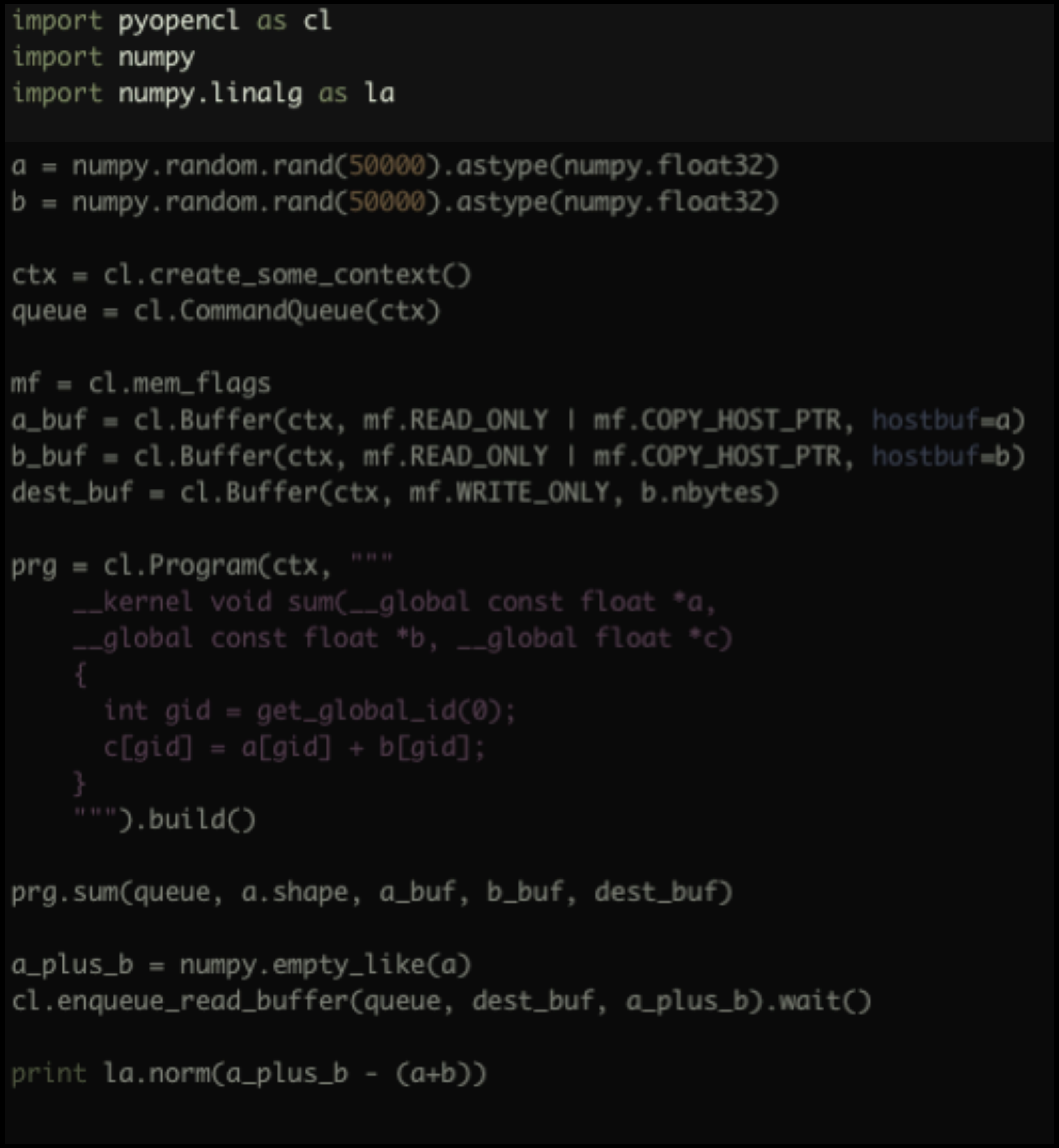
OpenCL Summary



```
__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];
}
```


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

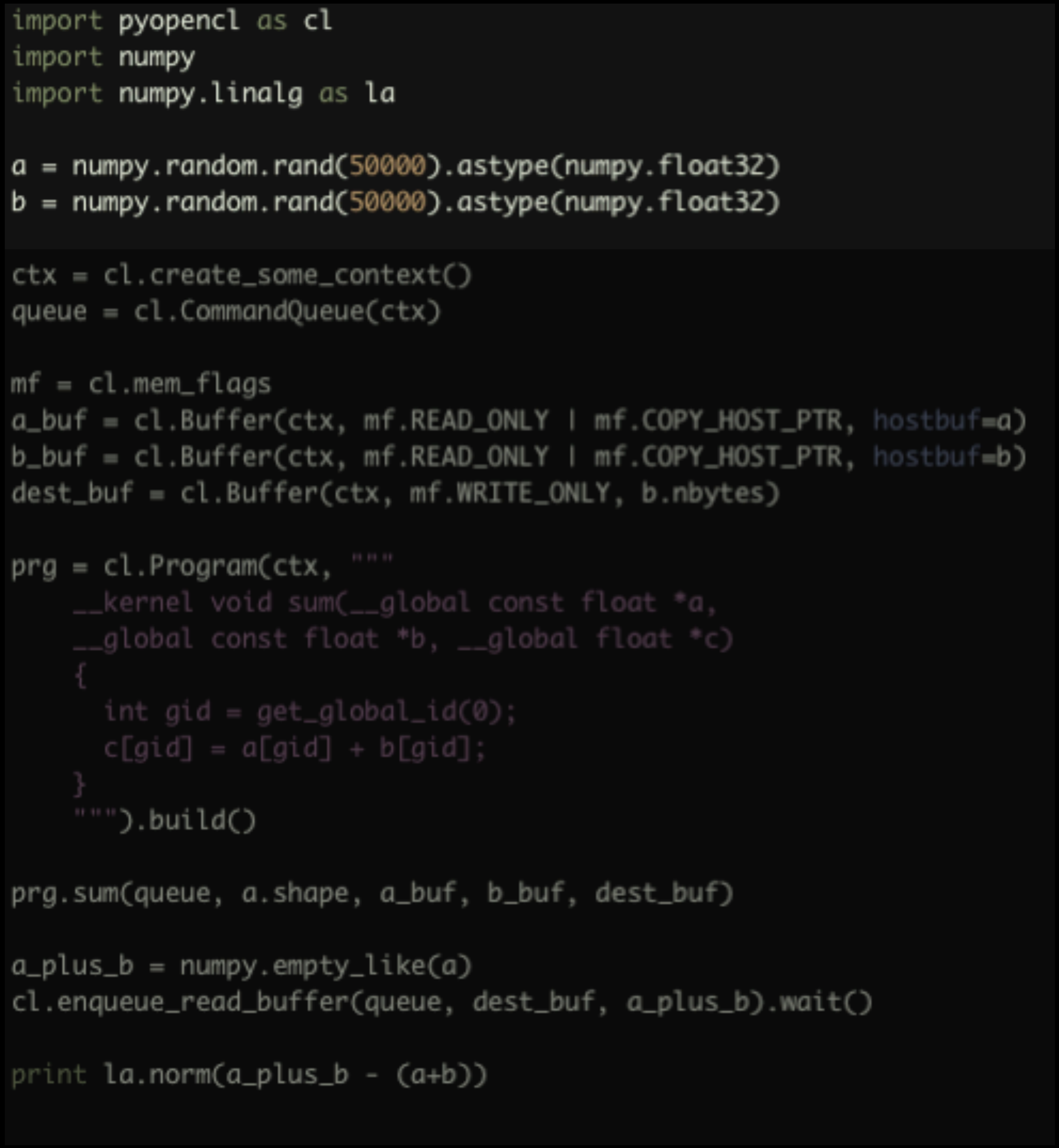
pyopencl



```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))
```

imports

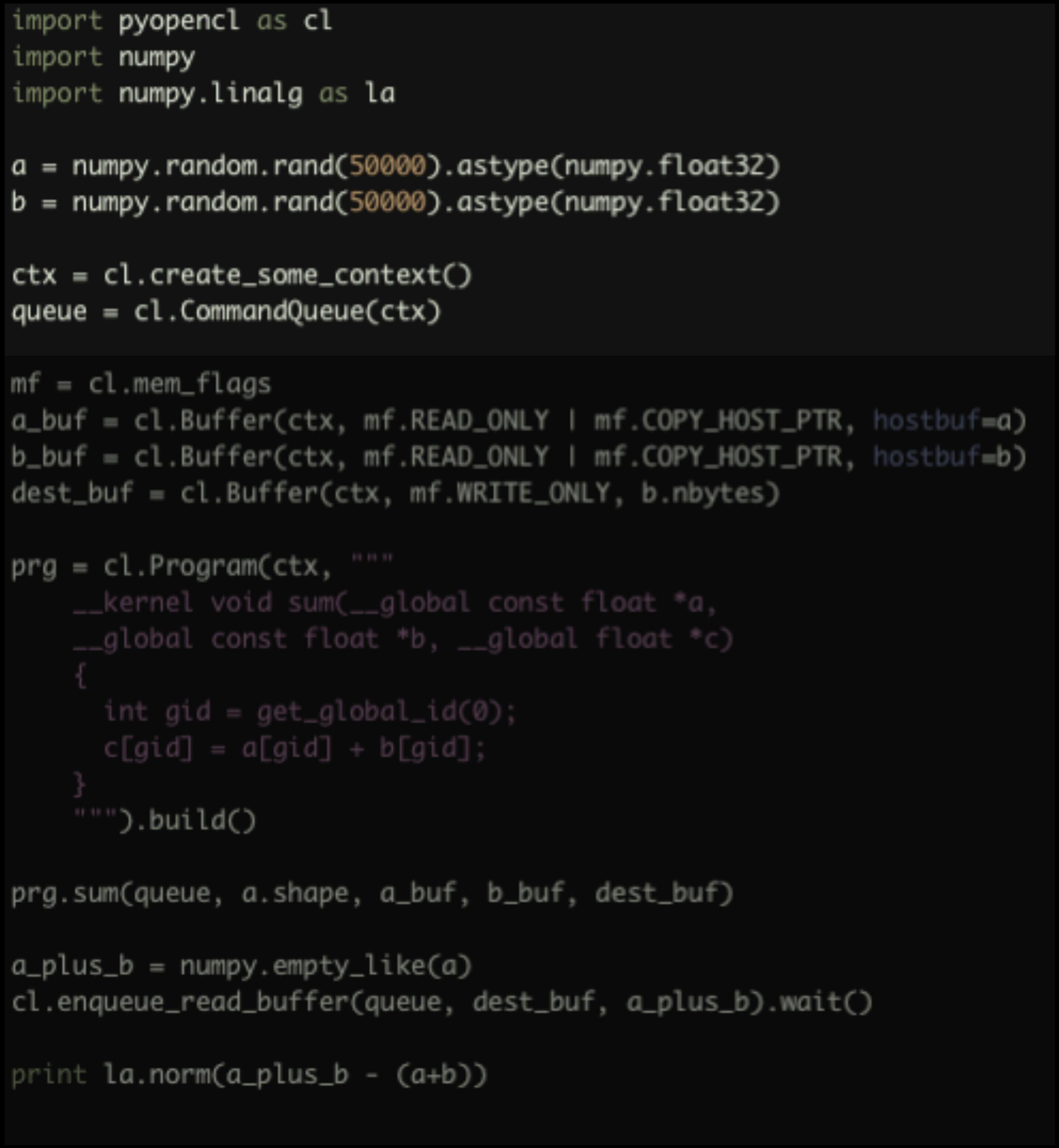
pyopencl



```
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))
```

initialize inputs

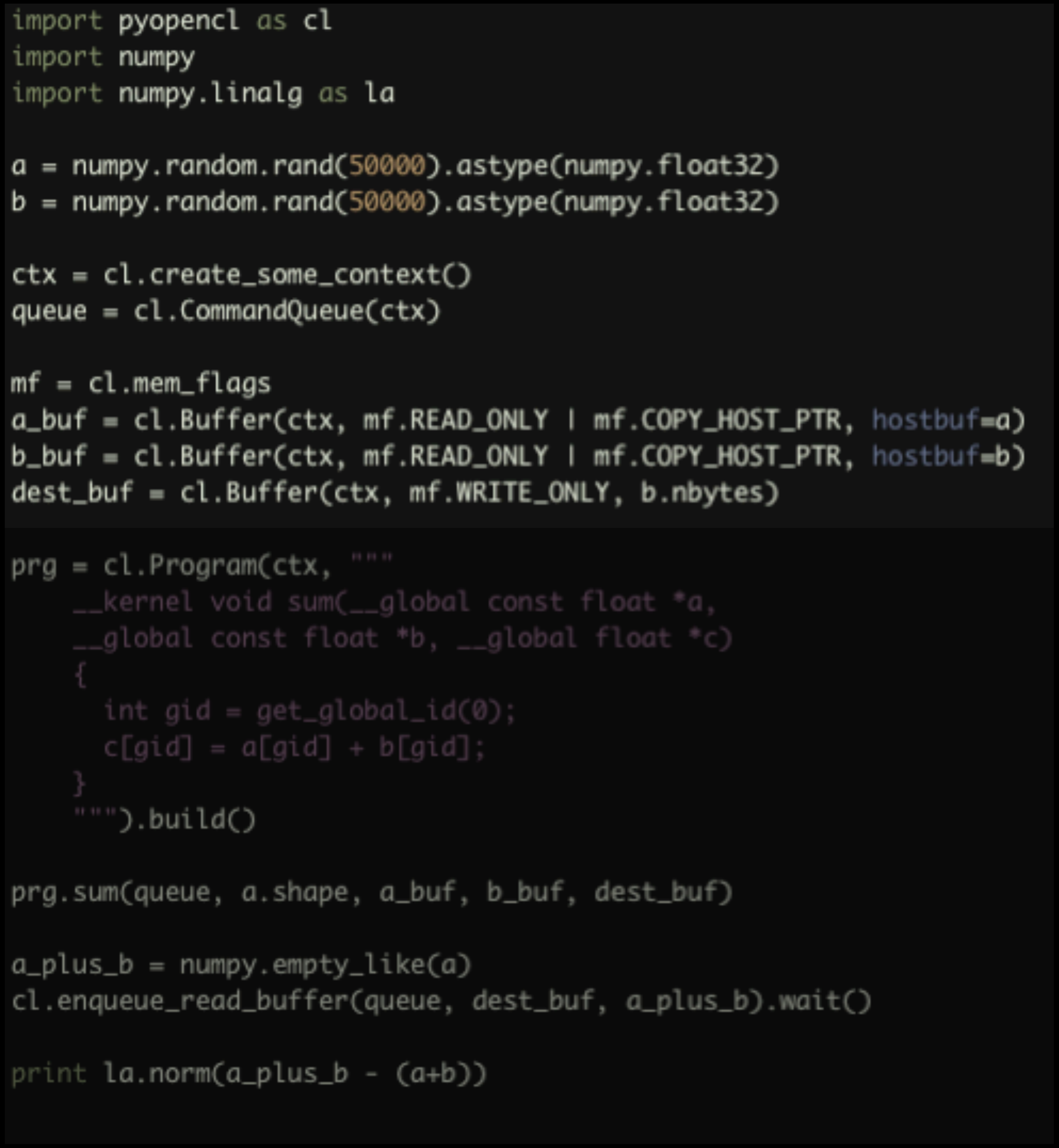
pyopencl



```
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))
```

create an OpenCL context
and a command queue

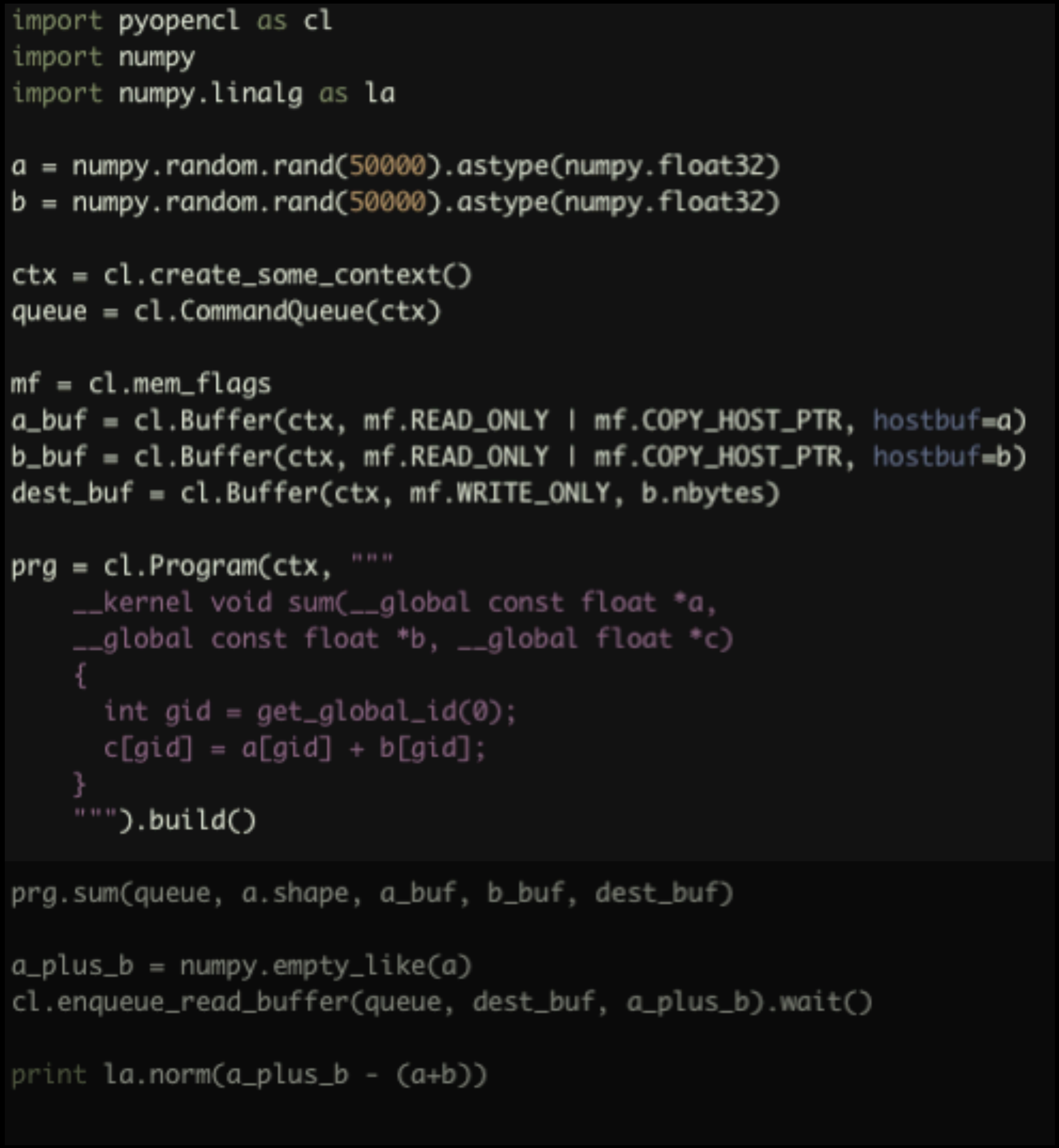
pyopencl



```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))
```

initialize device buffers

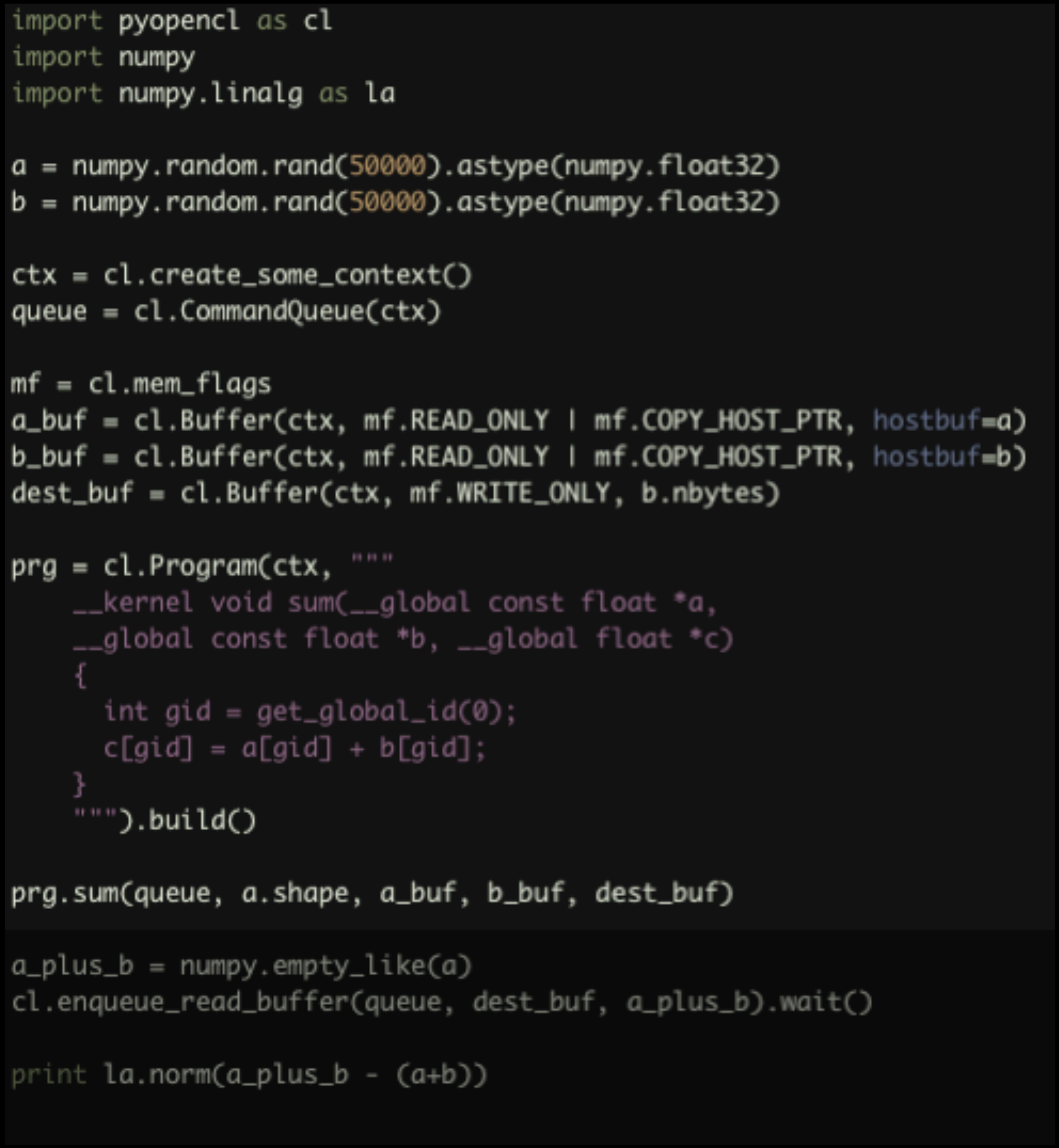
pyopencl



```
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))
```

generate and compile your kernel

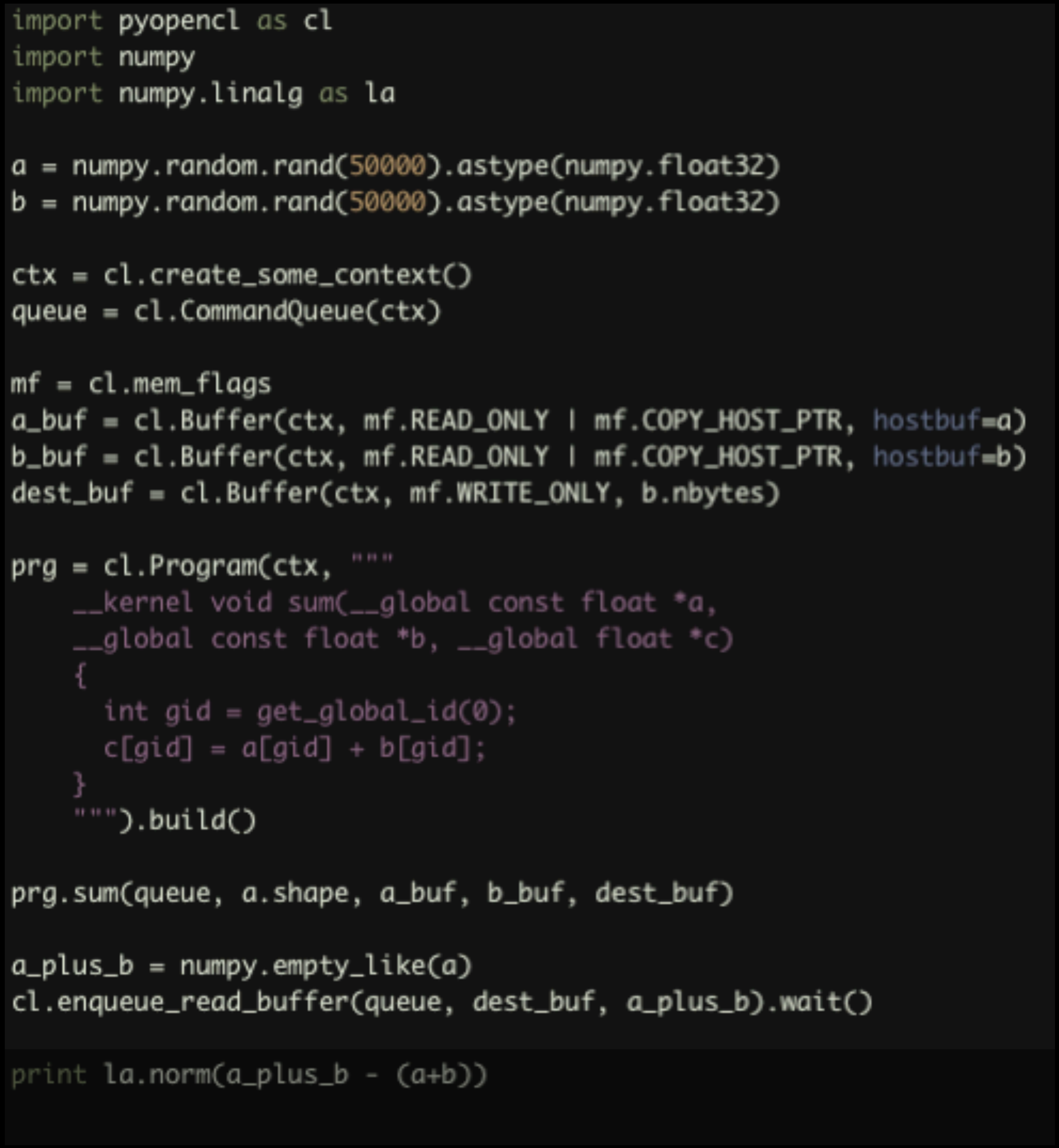
pyopencl



```
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))
```

enqueue and run your kernel, one thread per element

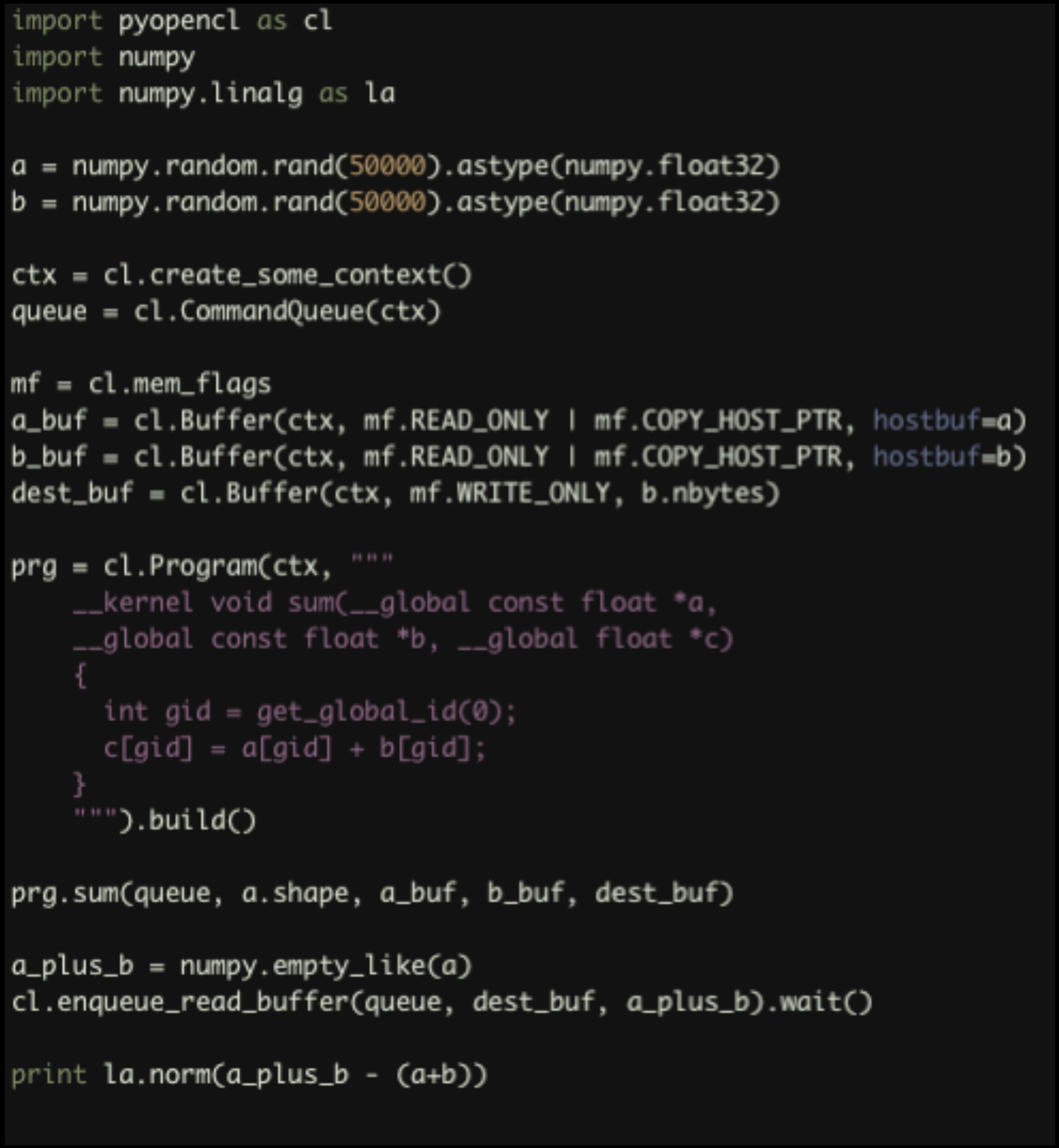
pyopencl



```
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))
```

read your results

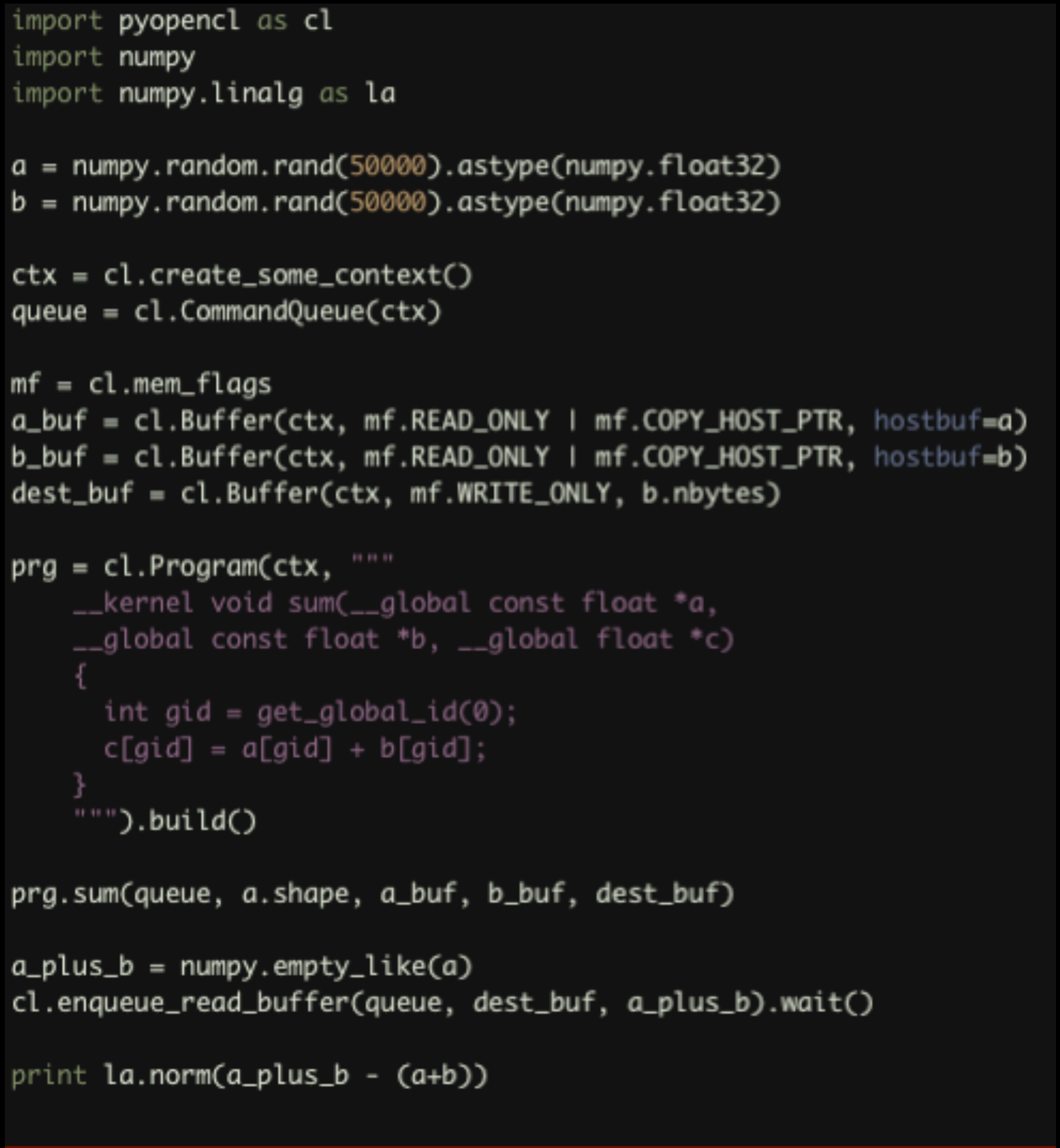
pyopencl



```
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))
```

host-side analysis, plotting, etc.

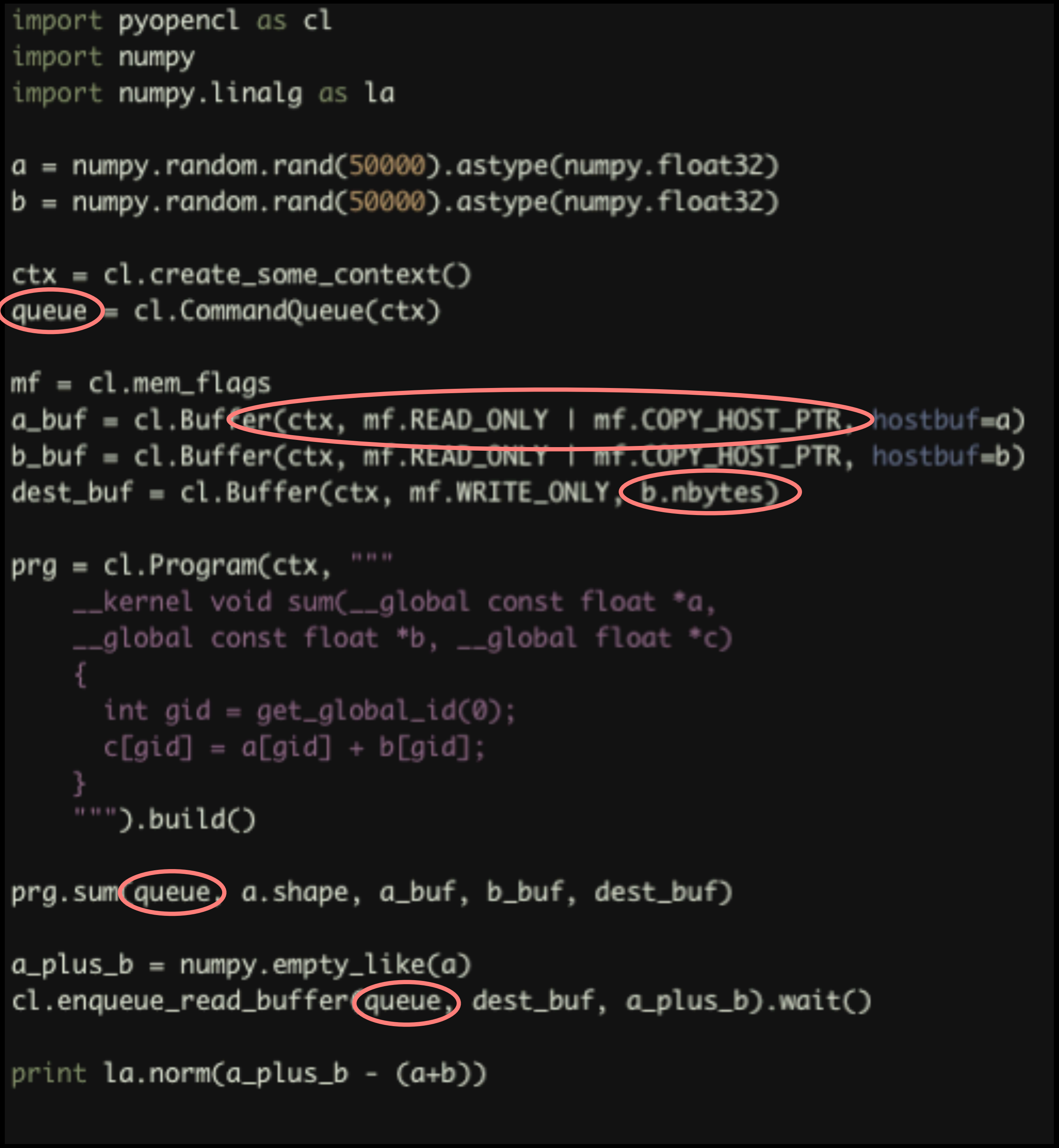
pyopencl



```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))
```

no messy cleanups, its taken care of by the aggressive Python garbage collector by default

pyopencl



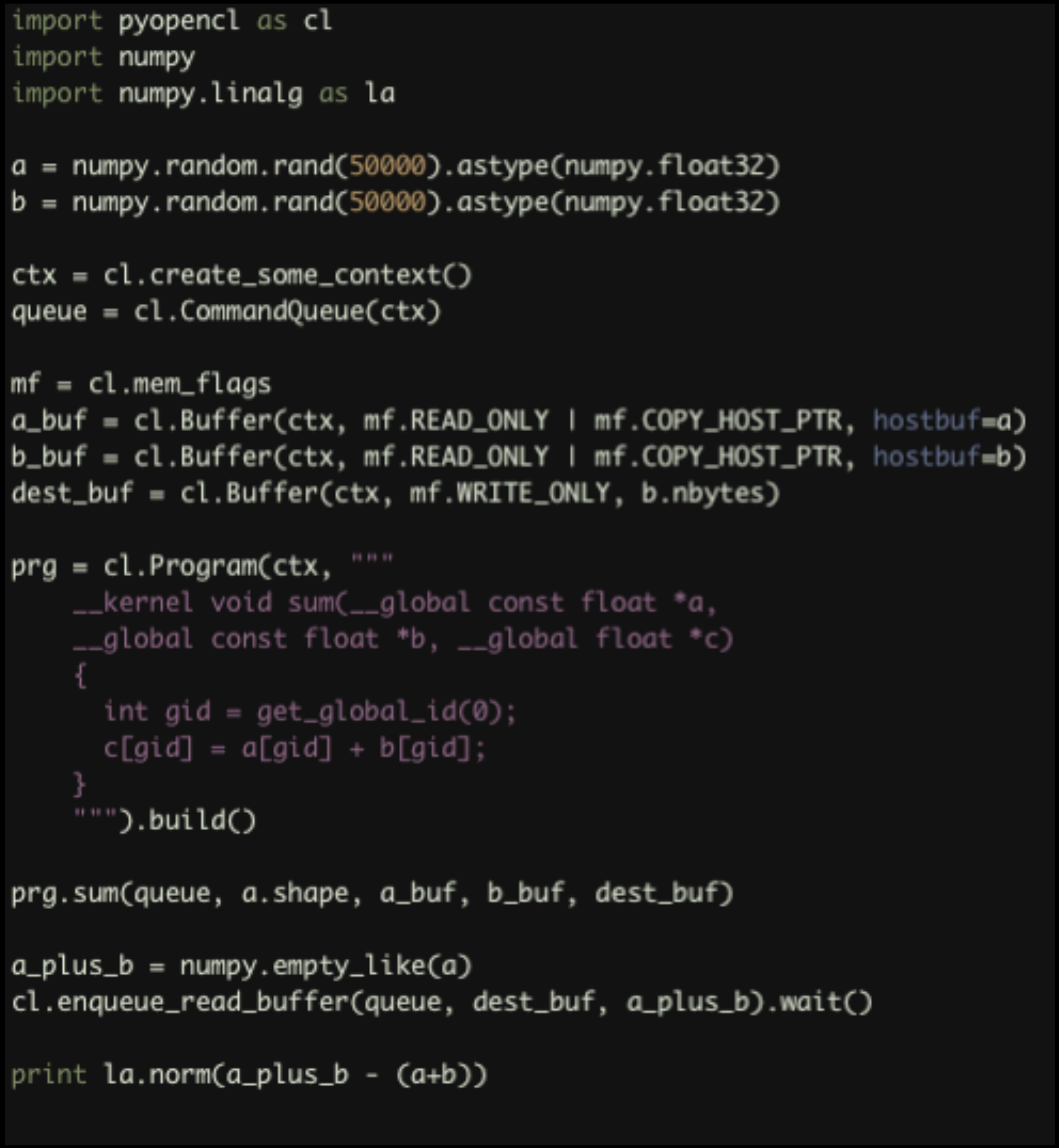
```python
import pyopencl as cl
import numpy
import numpy.linalg as la

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))
```

stateless: *queues everywhere but you almost never have more than one per context*

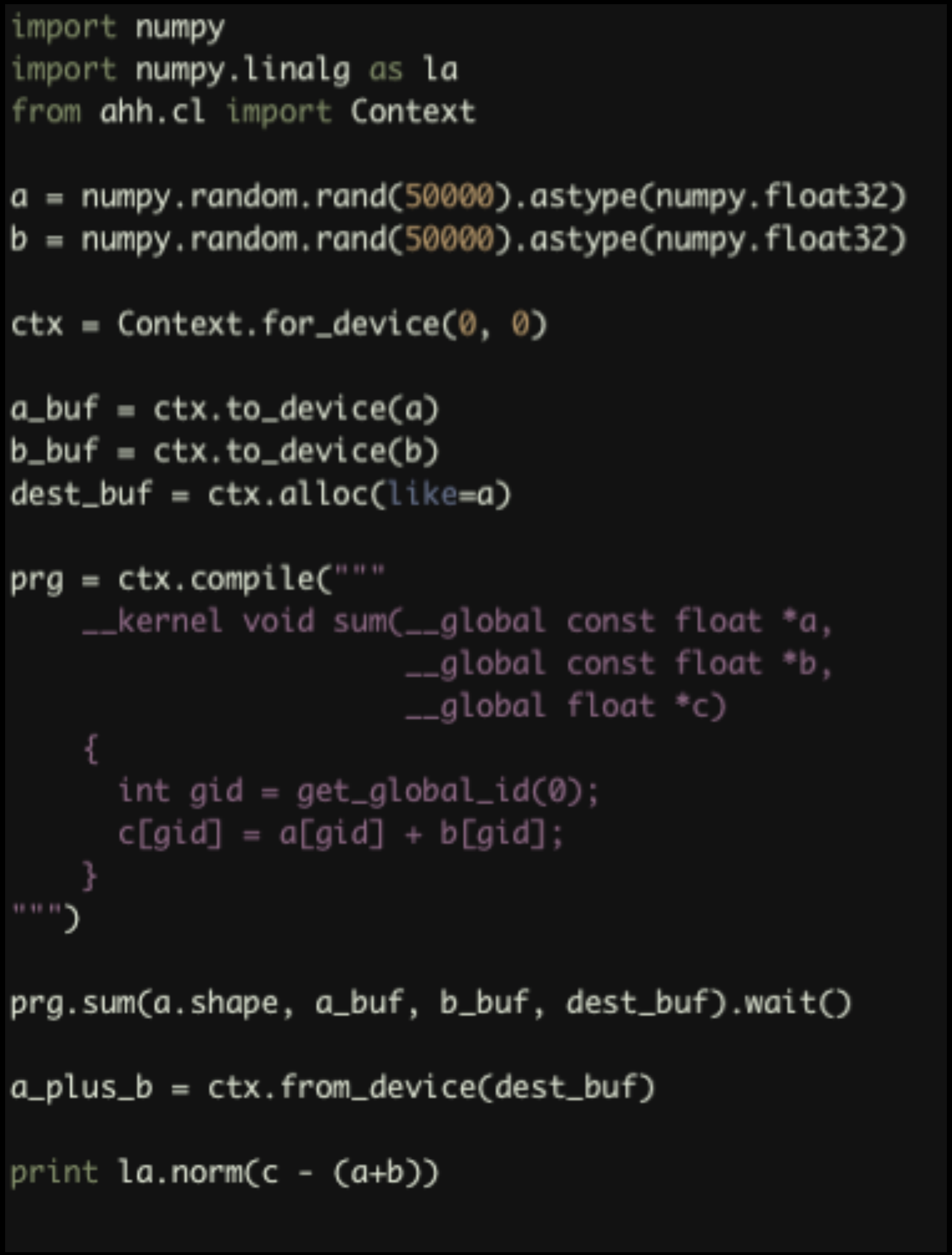
verbose: *flags everywhere*

opaque: *buffers are just a bunch of bytes* atomic hedgehog (ahh)

pyopencl $\subset$ 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))
```

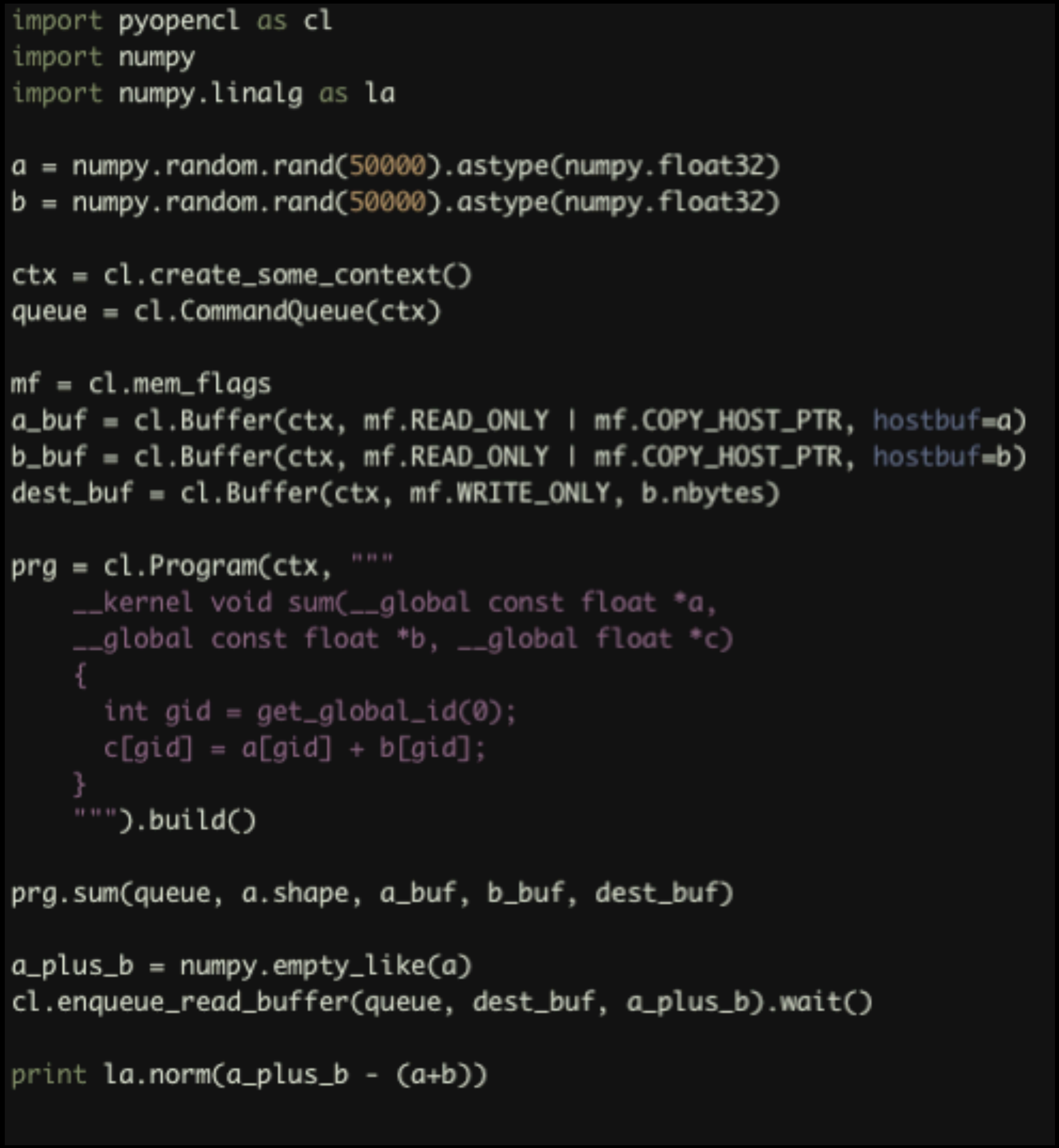


```
import numpy
import numpy.linalg as la
from ahh.cl import Context

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))
```

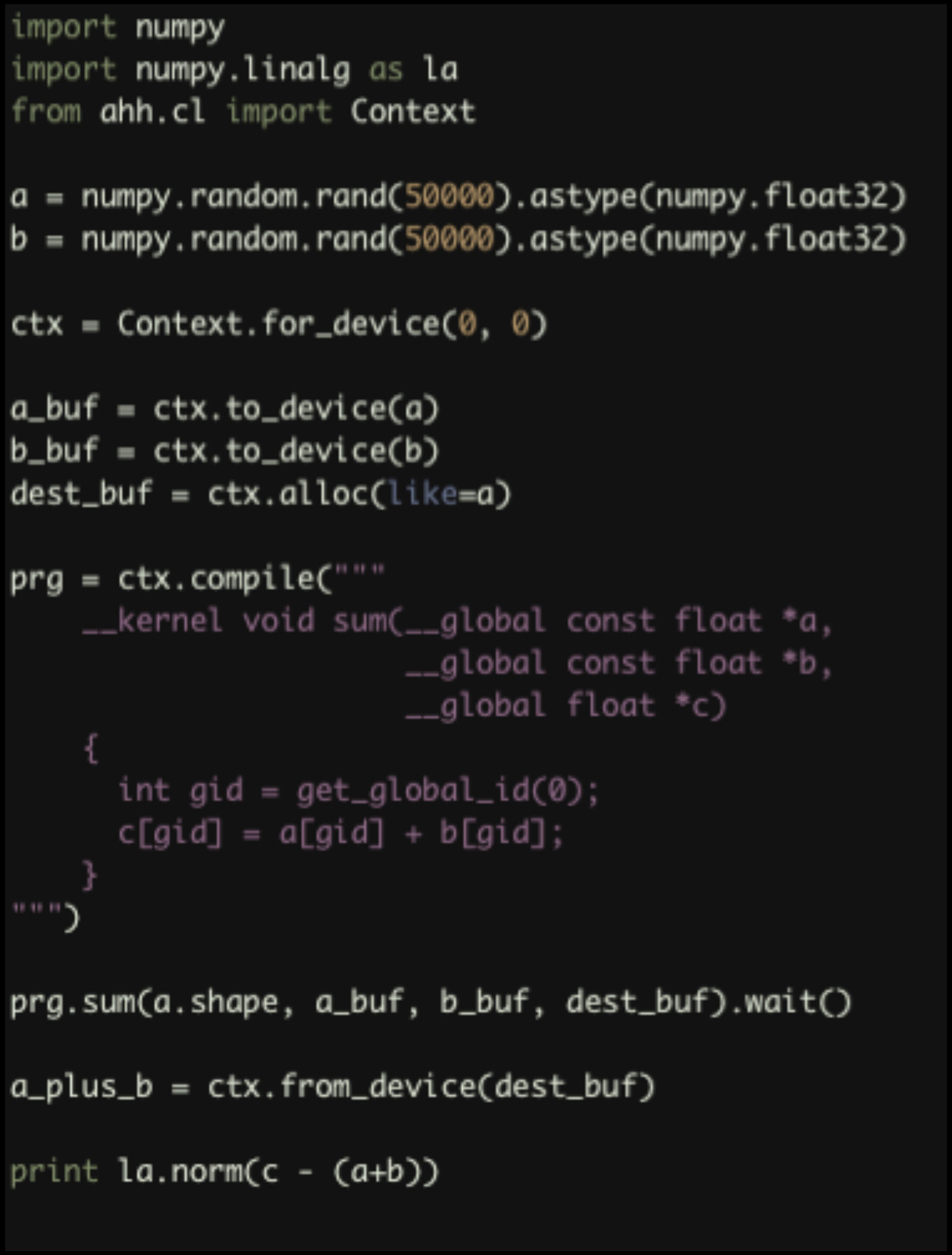
ahh.cl is a superset of pyopencl

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

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))
```

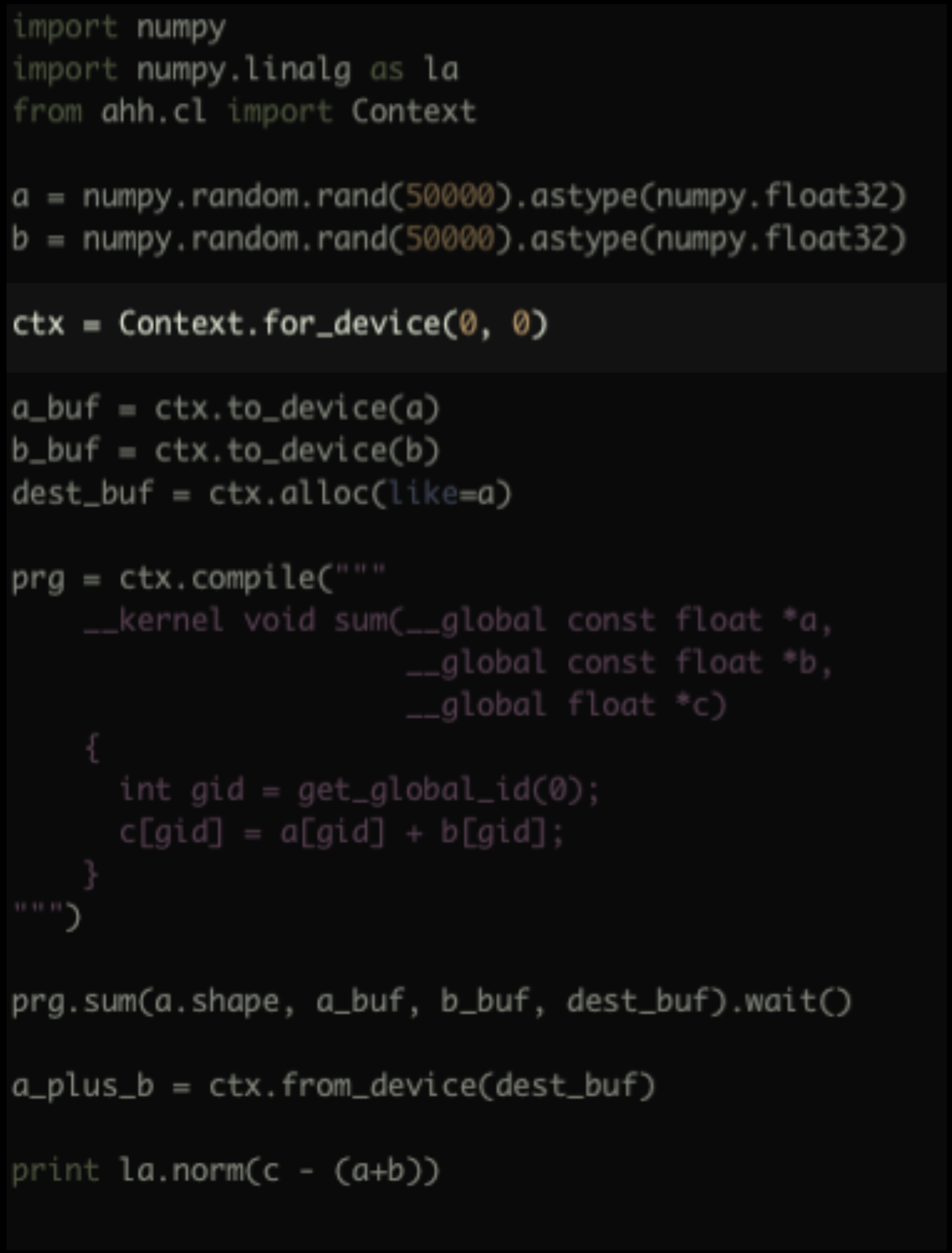


```python
import numpy
import numpy.linalg as la
from ahh.cl import Context

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))
```

+ 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



```
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, intelligent memory management

four flexible functions:

`alloc`
allocates empty buffer

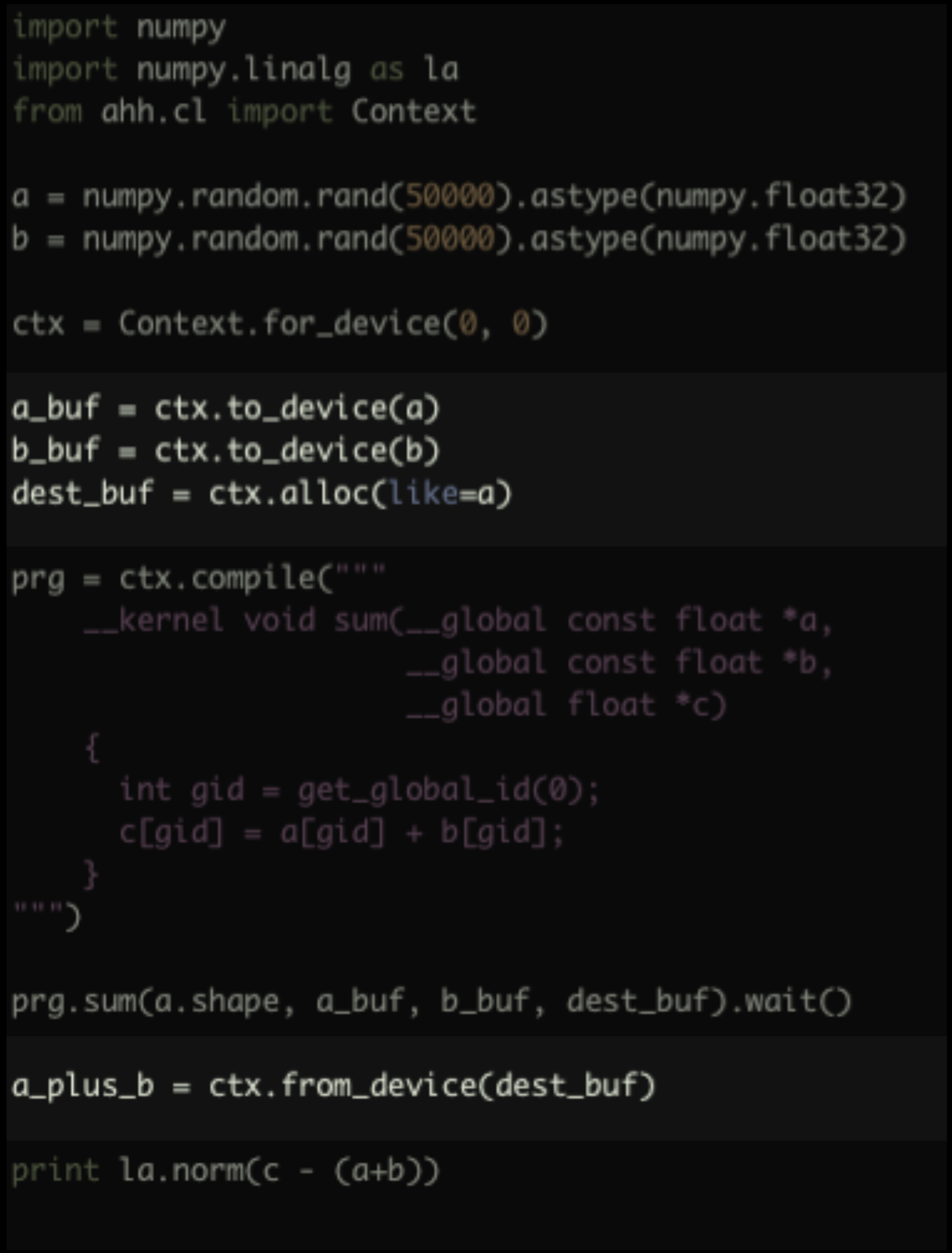
`to_device`
numpy array => new buffer

`from_device`
buffer => new numpy array

`memcpy`
copies between existing buffers or arrays

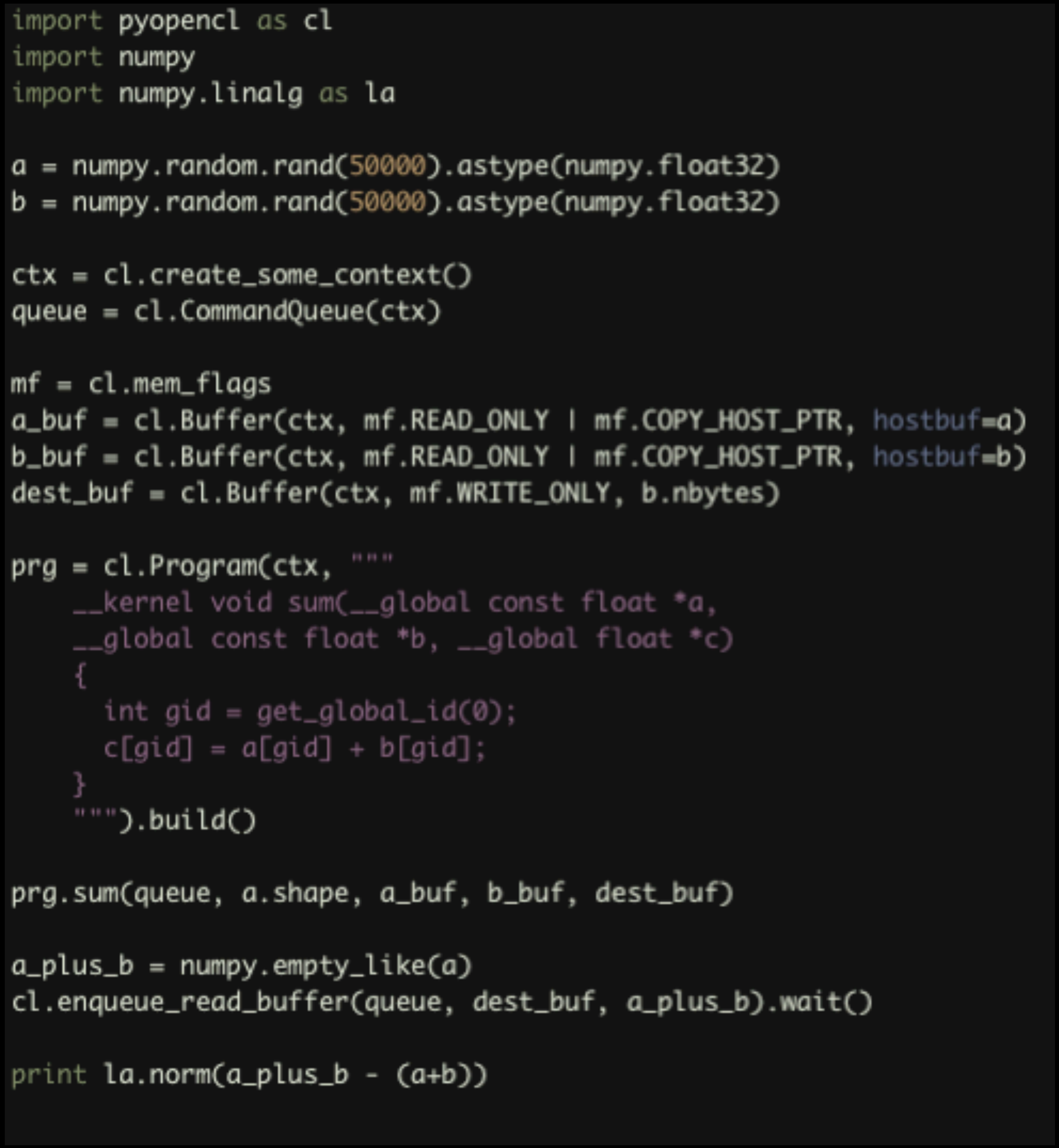
buffers save metadata (type, shape, order), so you never have to repeat yourself (this will be useful later)

ahh.cl



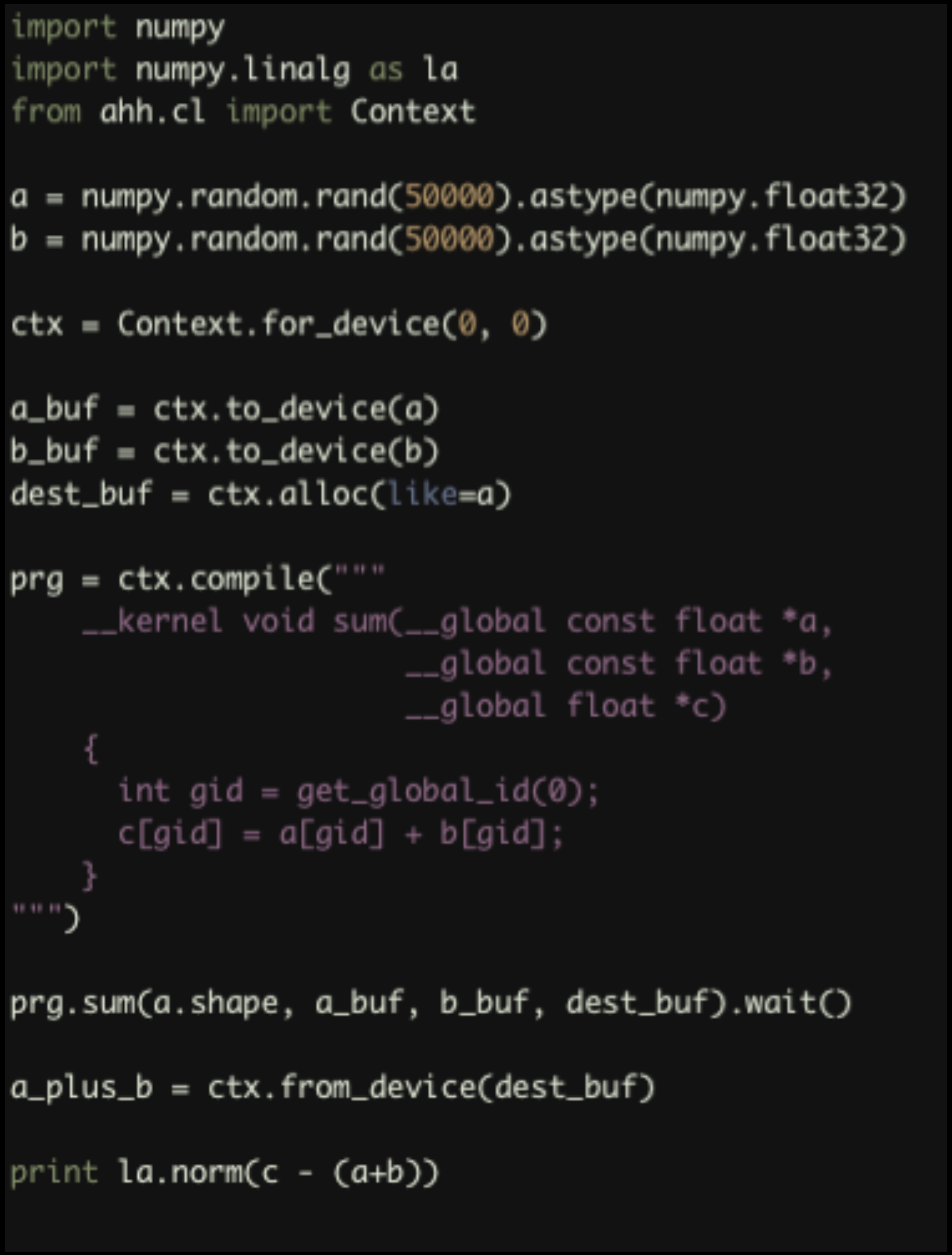
```
import numpy
import numpy.linalg as la
from ahh.cl import Context

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



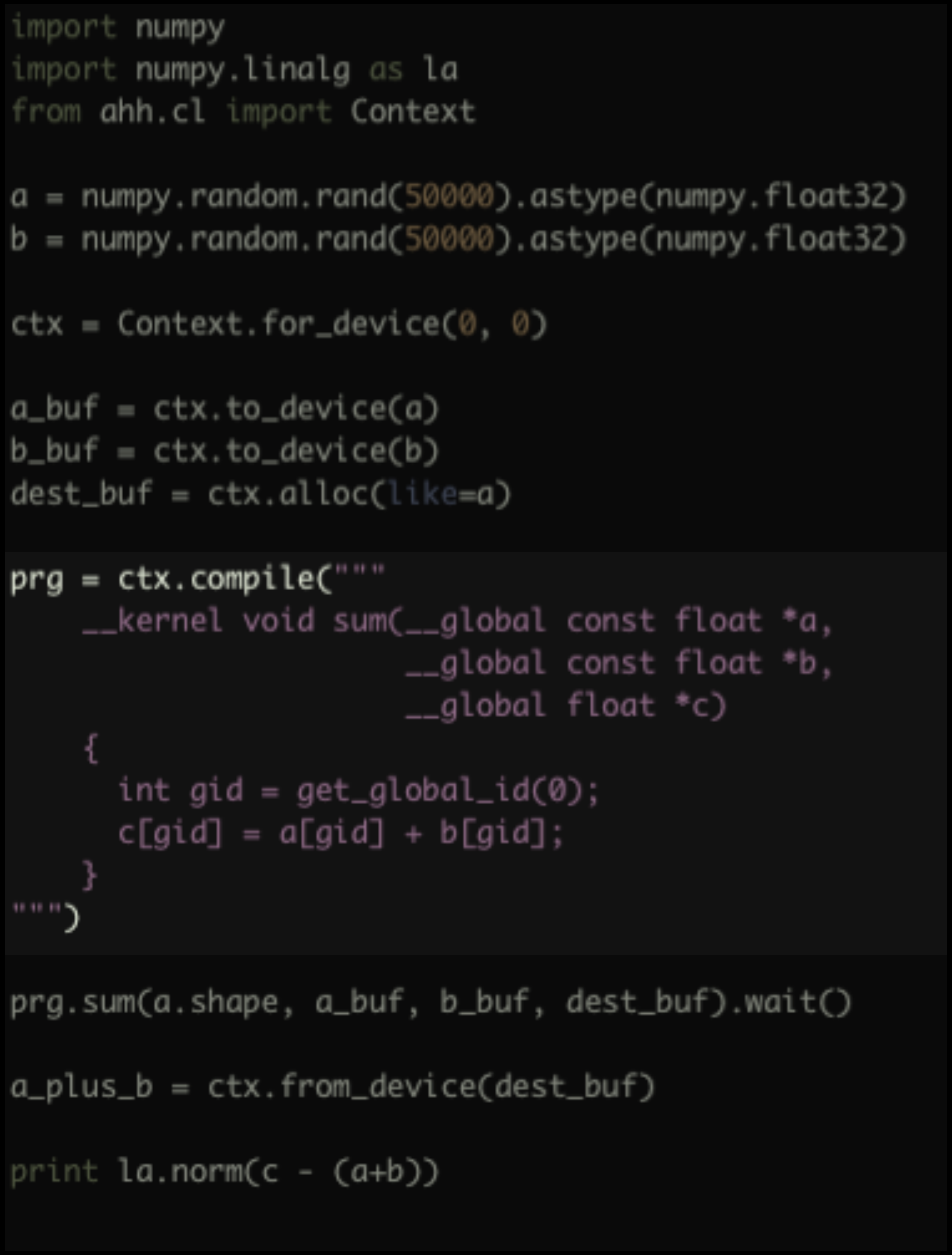
```python
import pyopencl as cl
import numpy
import numpy.linalg as la

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))
```

ahh.cl



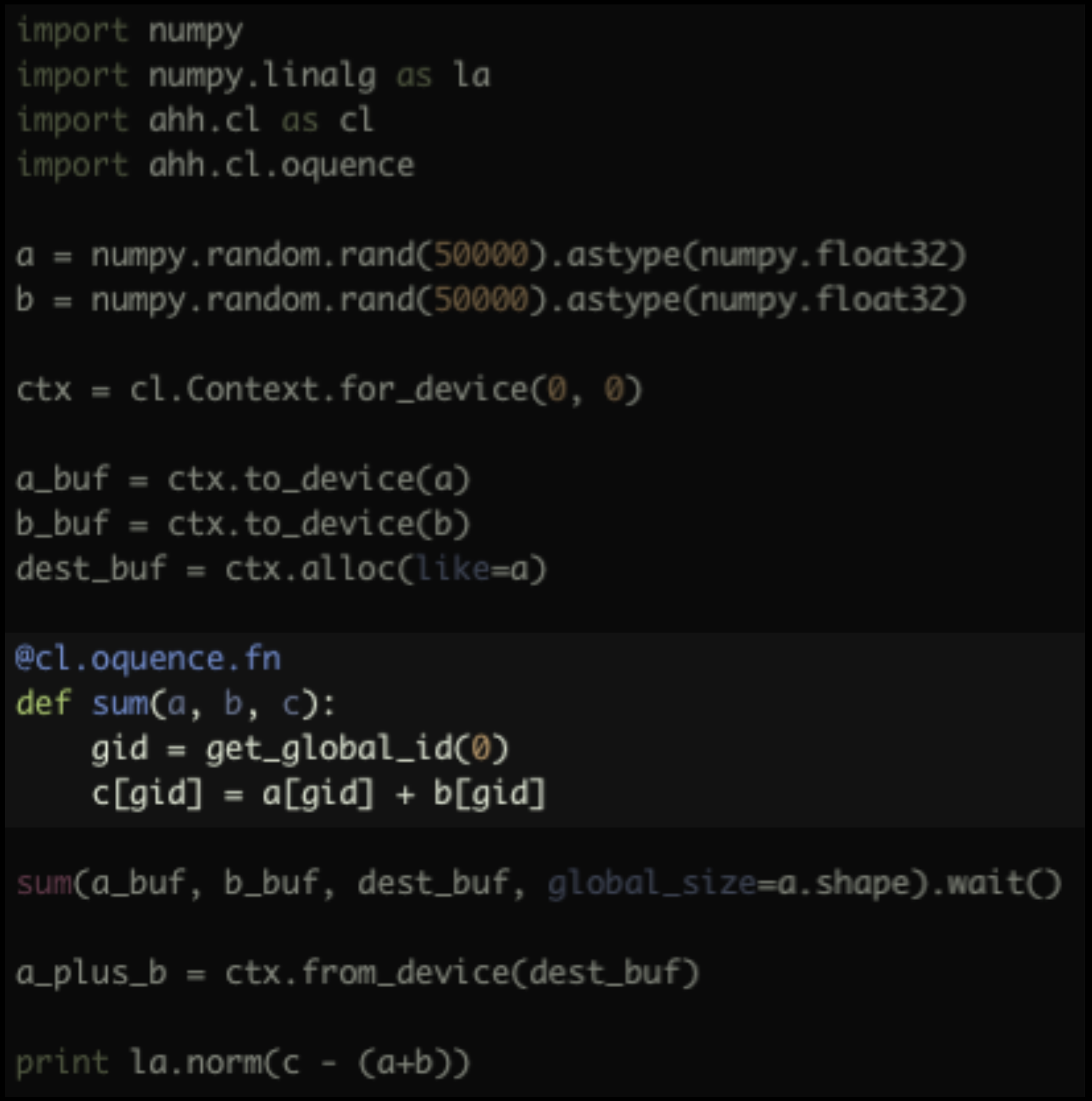
```python
import numpy
import numpy.linalg as la
from ahh.cl import Context

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))
```

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))
```

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))
```

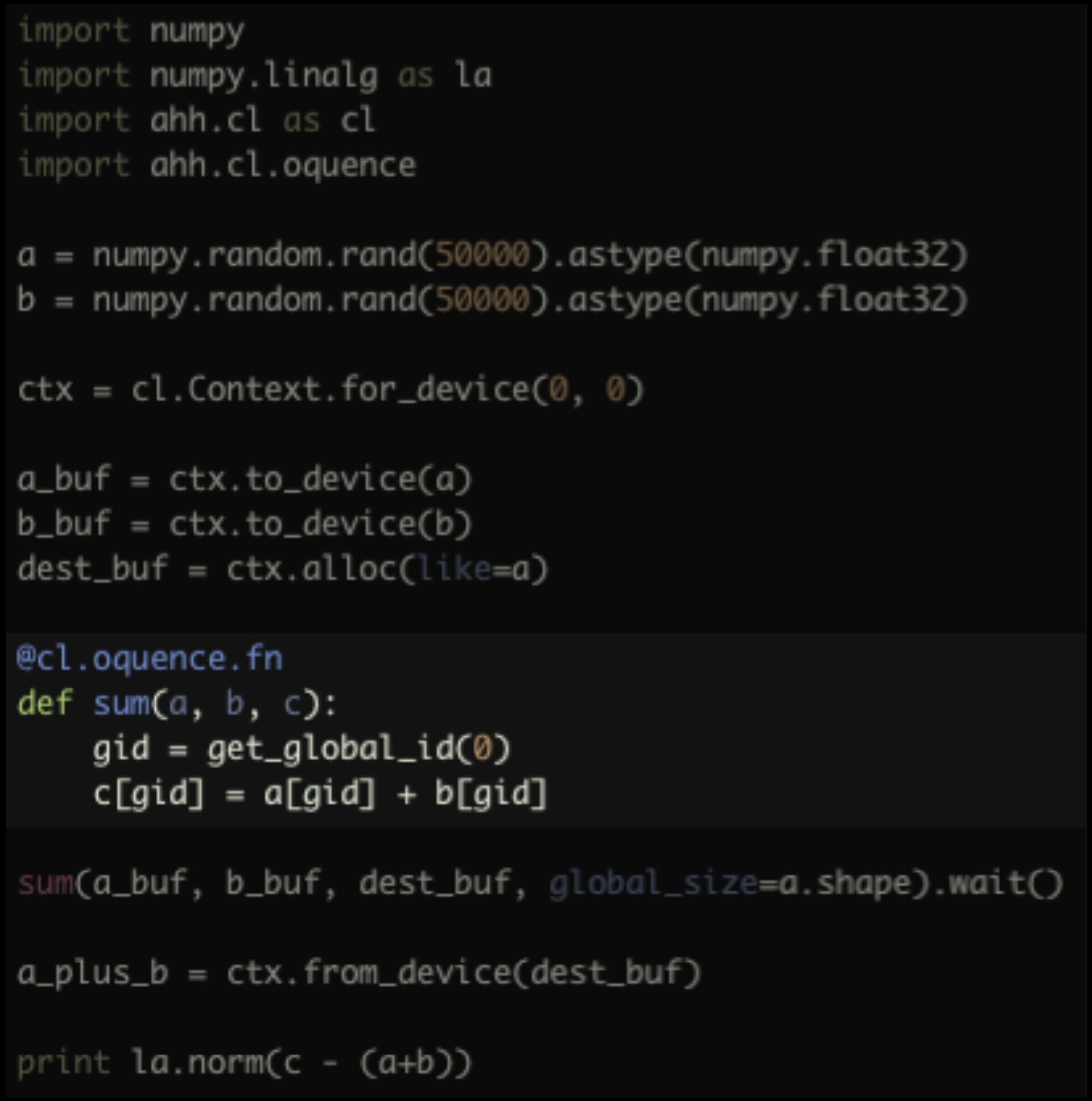
an
alternative
syntax
for OpenCL

ahh.cl.oquence

`sum` is a generic function

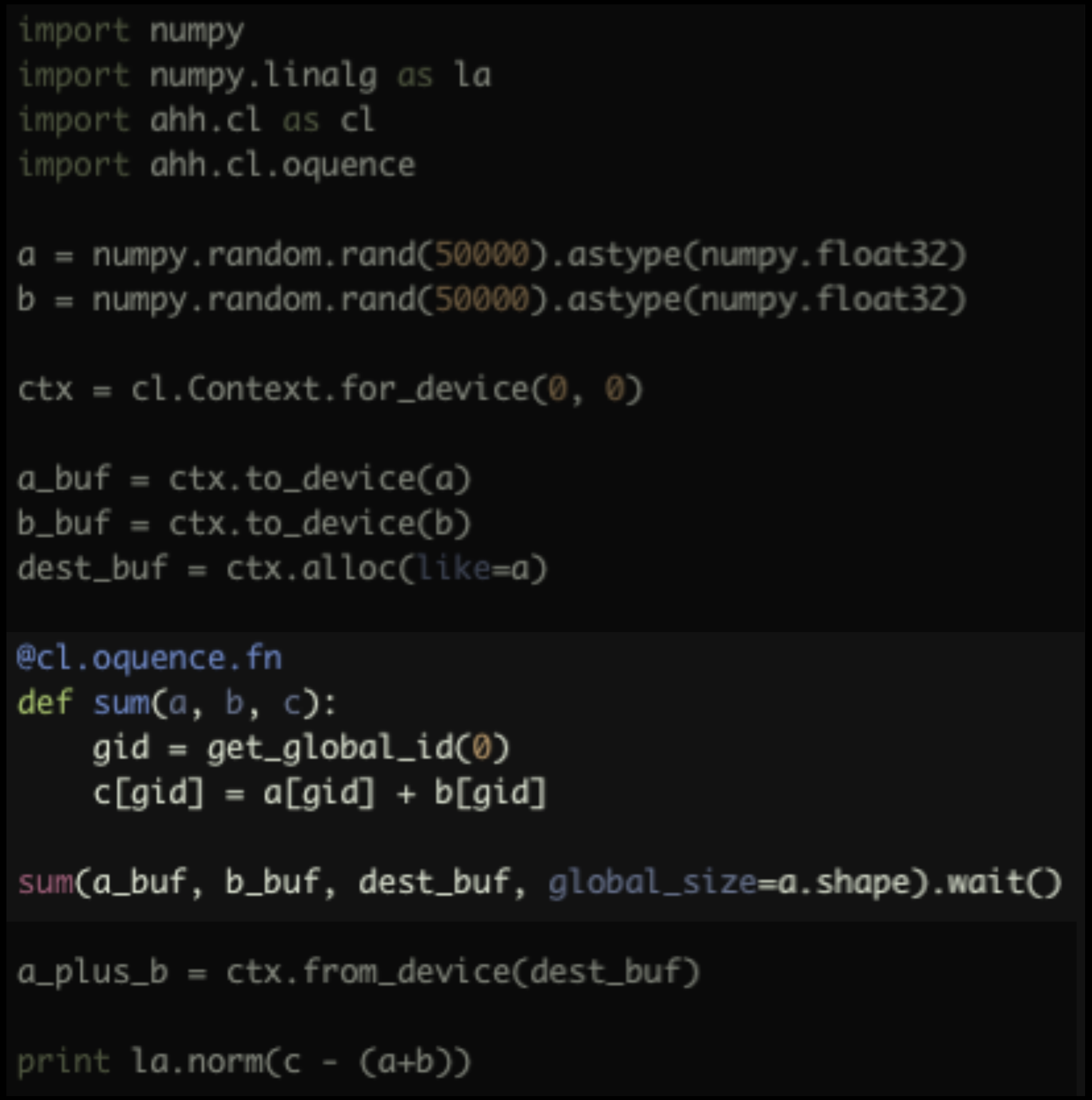
(an instance of `GenericFn`, in fact)

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



```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))
```

ahh.cl.oquence

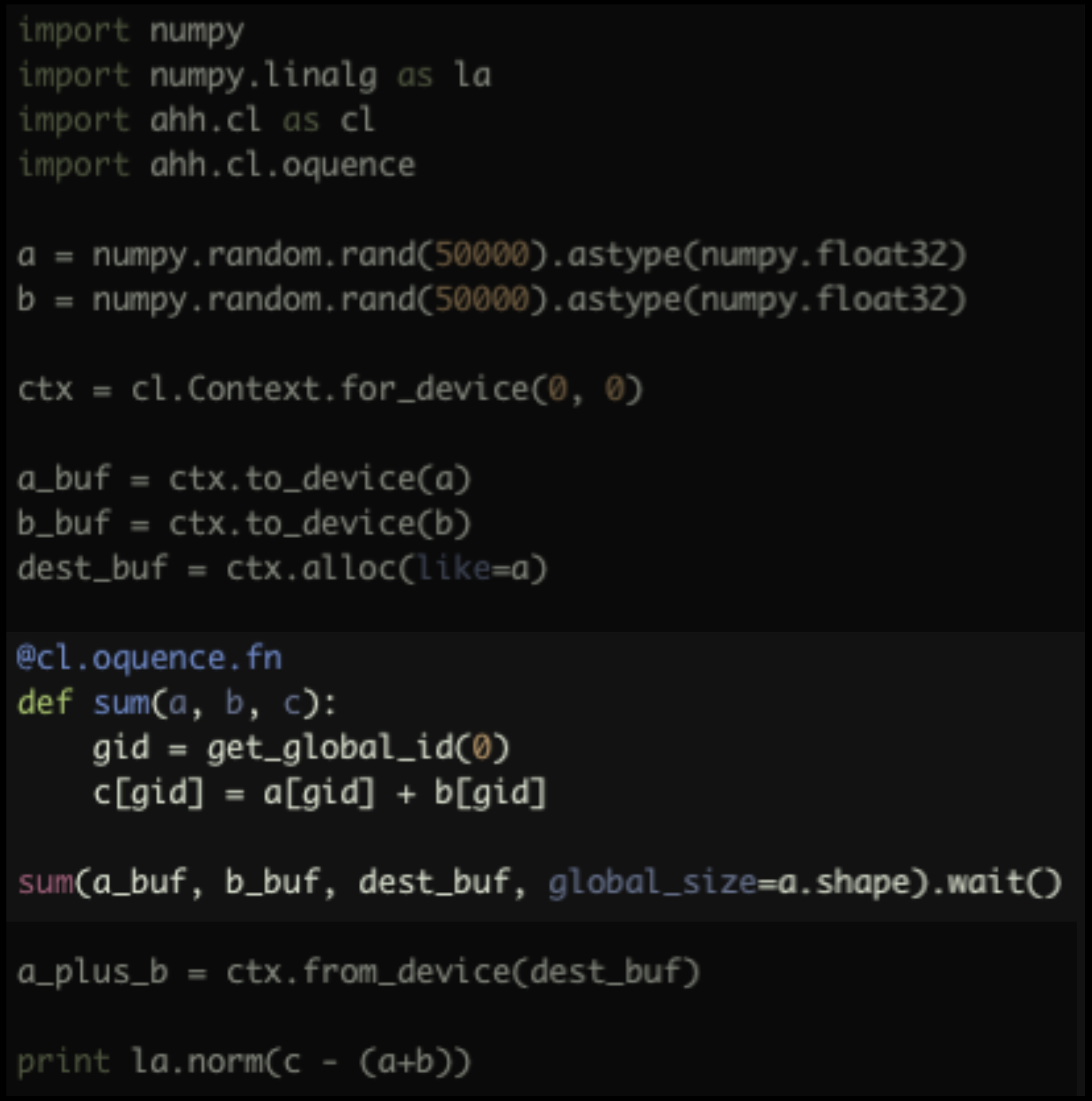


```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

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 are now known (because `ahh.cl` saves them!)

ahh.cl.oquence

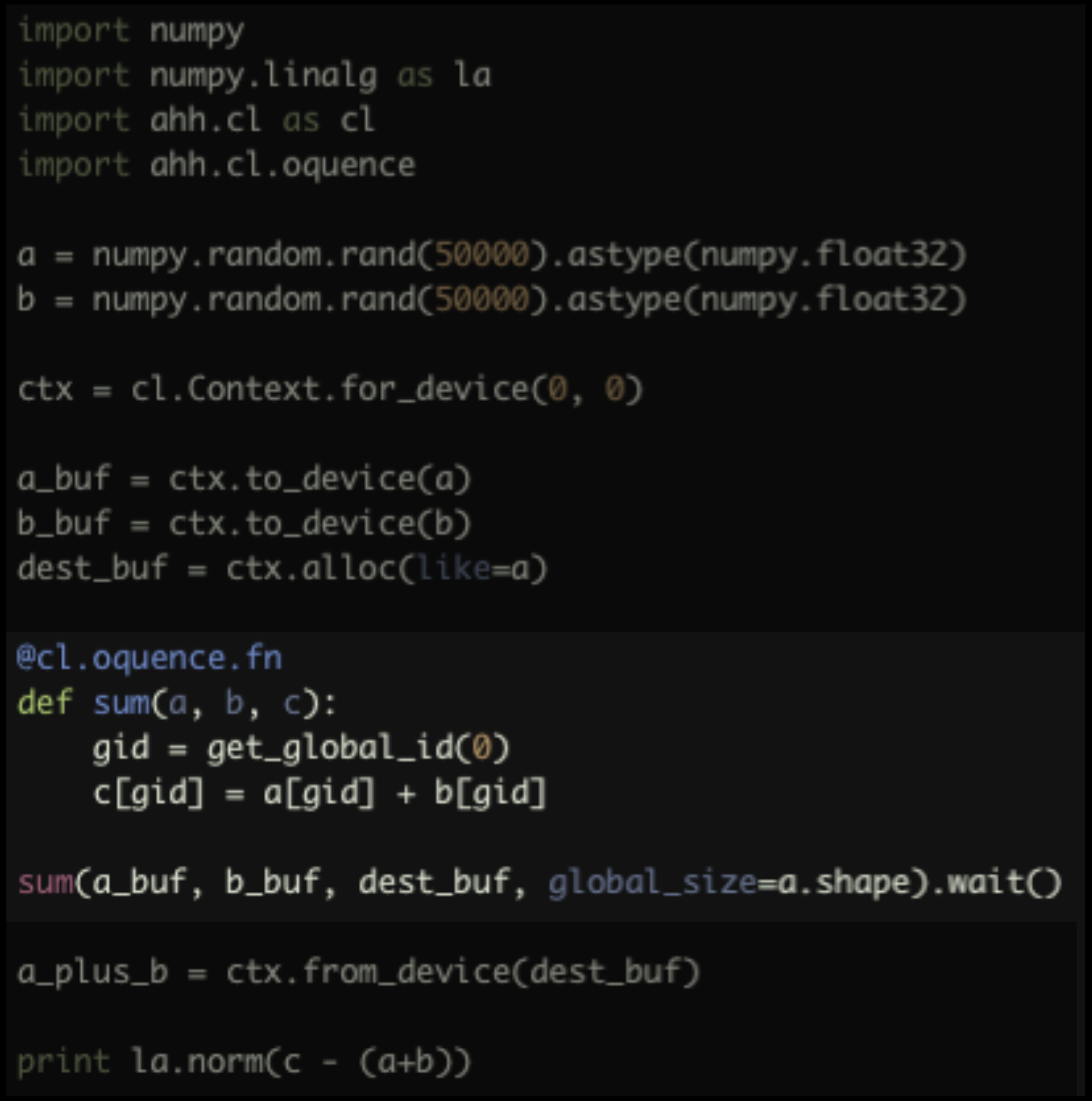
type inference: the types of all the intermediate expressions, and thus the types of the local variables and return type of the function, can now be exactly inferred, since there are no free variables



```
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))
```

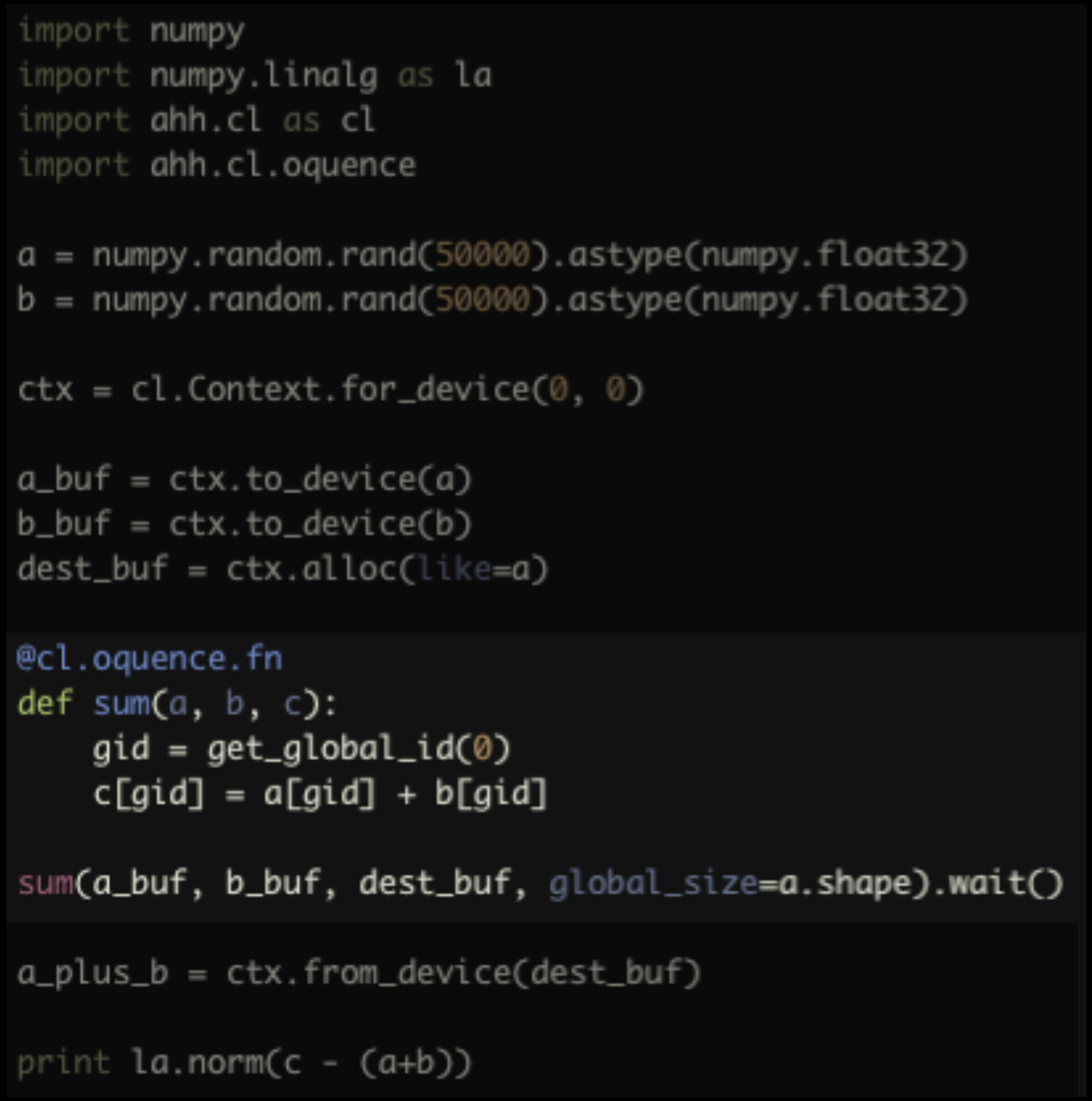
ahh.cl.oquence

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



```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))
```

ahh.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, you can generate an OpenCL function 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];
}
```

ahh.cl.oquence

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

```
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)
```

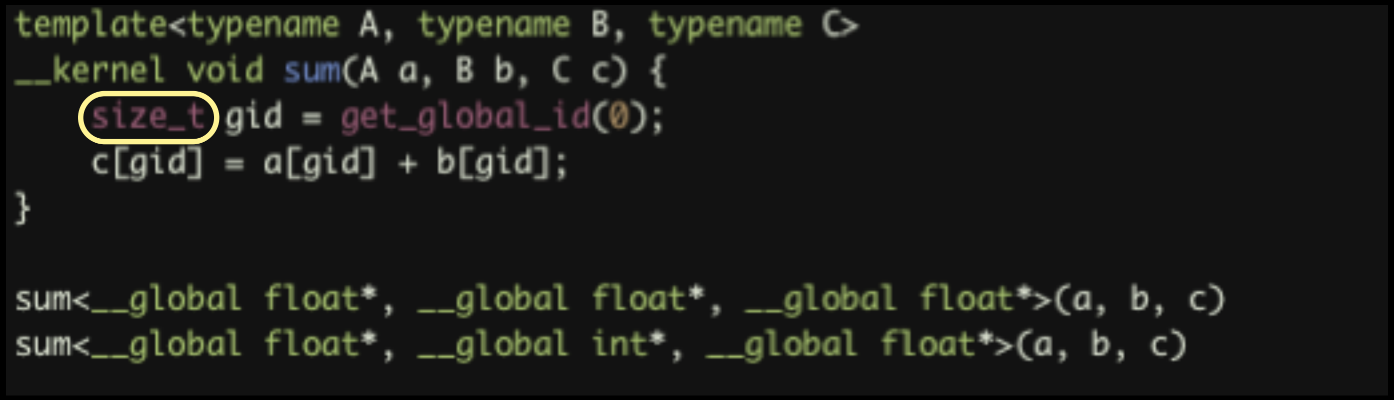
(OpenCL doesn't have templates.)

```
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))
```

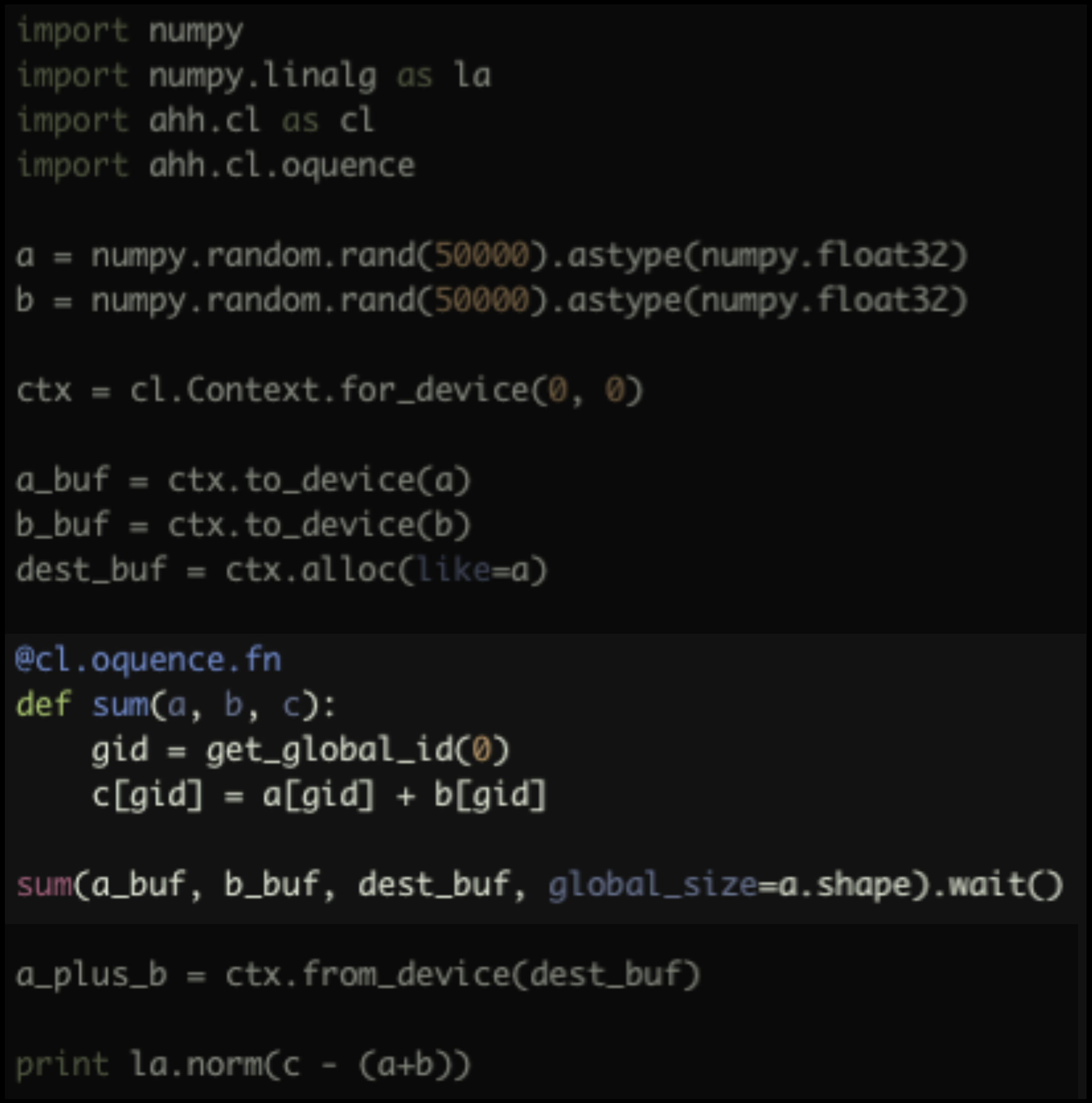
ahh.cl.oquence

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)
```

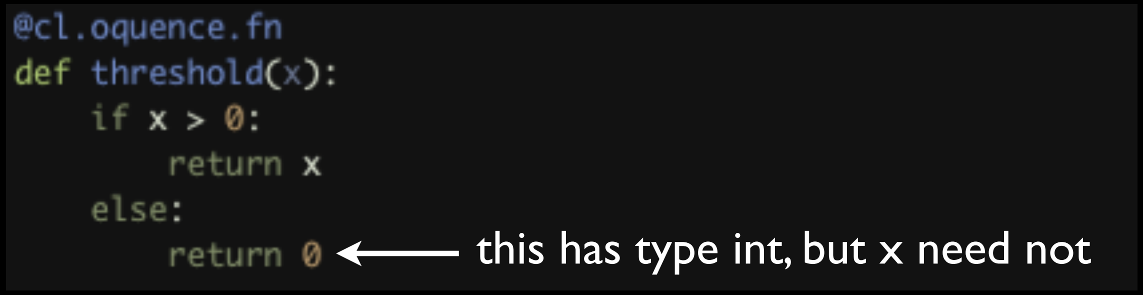


```
import numpy
import numpy.linalg as la
import ahh.cl as cl
import ahh.cl.oquence

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))
```

ahh.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.



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

$\longleftarrow$ this has type int, but x need not

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

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.

`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

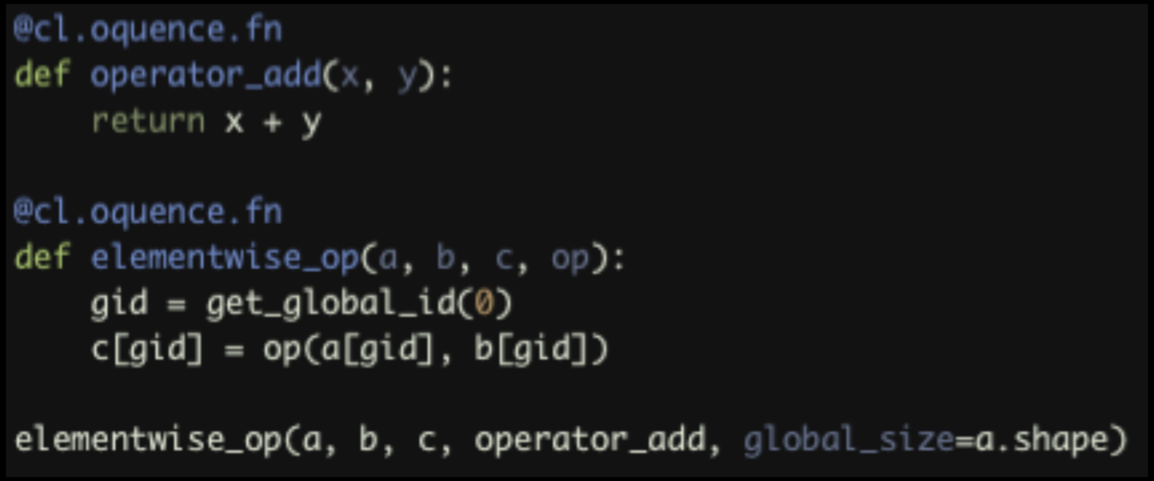
kernel inference

`cl.oquence`

also supports

higher order functions

(at compile-time)

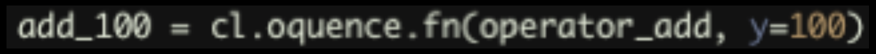


```
@cl.oquence.fn
def operator_add(x, y):
    return x + y

@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)
```

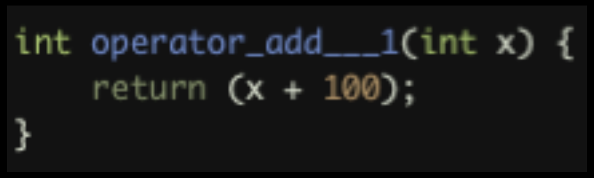
cl.oquence

also supports

dynamic constant binding



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



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

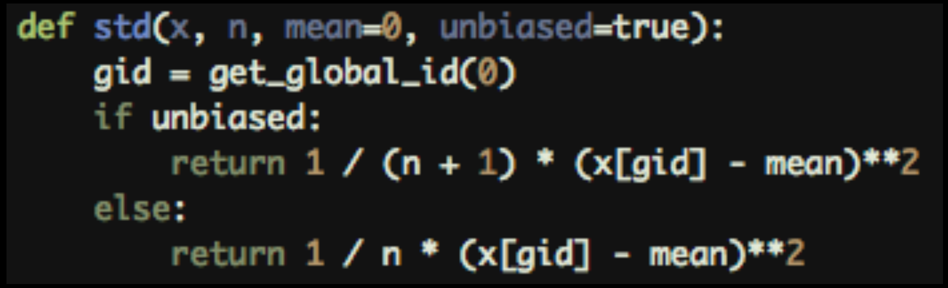
you can inline buffers as constants too.

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

cl.oquence

also supports

default arguments

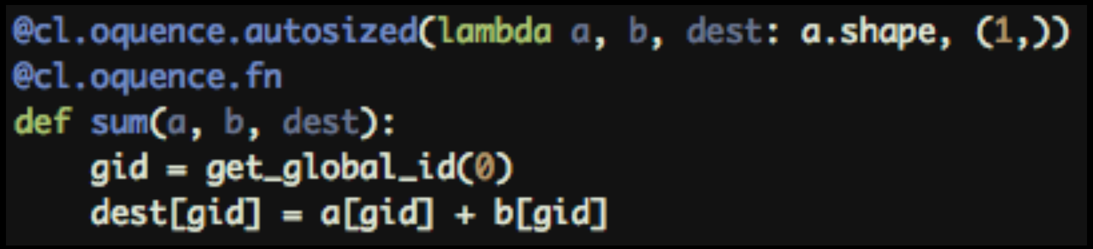


```
def std(x, n, mean=0, unbiased=true):
    gid = get_global_id(0)
    if unbiased:
        return 1 / (n + 1) * (x[gid] - mean)**2
    else:
        return 1 / n * (x[gid] - mean)**2
```

cl.oquence

also supports

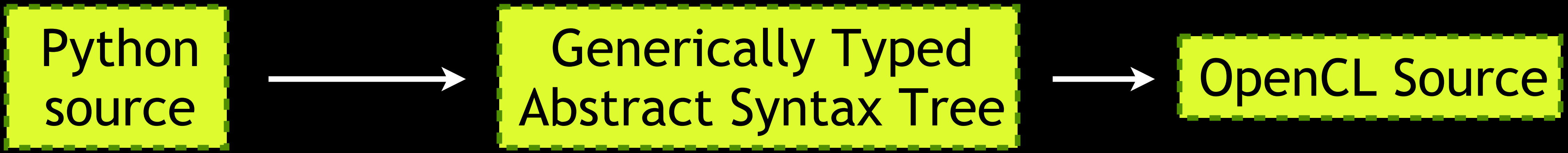
automatic kernel size calculators



```
@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]
```

`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.

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 better balance.

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)

GNU LGPL cc

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!

GNU LGPL cc

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!

For the programming language nerds

A novel type system

cl.oquence has an
extensible
code generation based
structural
static type system.

A novel type system

cl.oquence has an
extensible
code generation based
structural
static type system.

Define **“virtual” types** in Python which cl.oquence will defer to during type inference and compilation when values claiming to have that type are manipulated
(unary operators, binary operators, function calls, attribute access, subscript access, assignment, passed as argument)

A novel type system

cl.oquence has an
extensible
code generation based
structural
static type system.

Define **“virtual” types** in Python which cl.oquence will defer to during type inference and compilation when values claiming to have that type are manipulated
(unary operators, binary operators, function calls, attribute access, subscript access, assignment, passed as argument)

Used to implement as libraries many of the core features described earlier
C99’s type semantics, higher order functions, numeric literal syntax, type cast syntax, etc.
-- a truly extensible core language.

Can use as a macro system
(LISP reborn?)

You could add a **dynamic object** type to the language which does attribute lookup at run-time.
As a library!

`cl.oquence`

makes possible

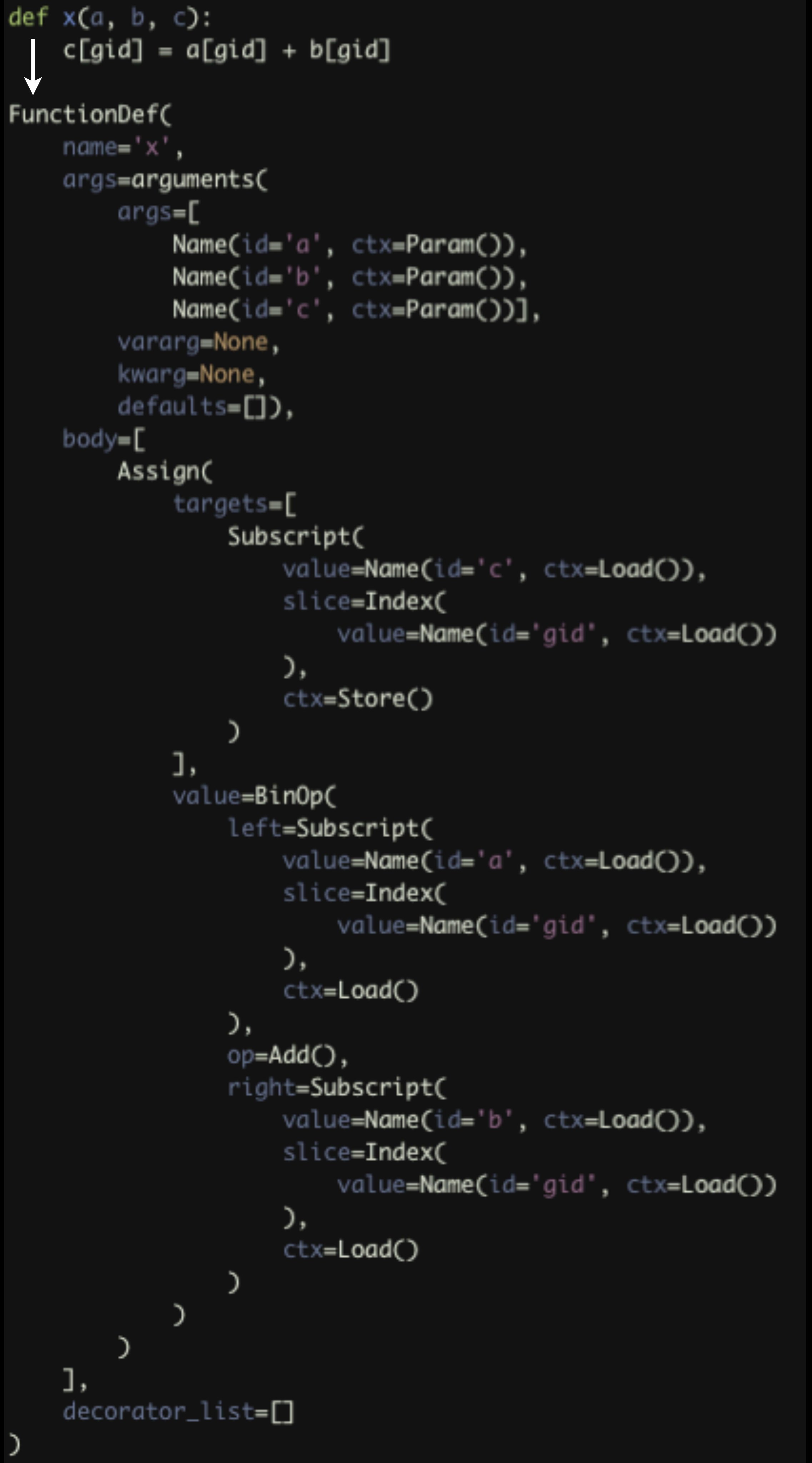
syntax tree transformations

(hairy, but far simpler and more managable than the equivalent C syntax tree, if you can even parse it at all)

I showed you constant inlining.

Possible and relatively simple:
replace variable *x* with expression *y*
replace all but first occurrence of *x* with *y*
...

More sophisticated:
Yang, Xiang, Kong & Zhou, *A GPGPU Compiler for Memory Optimization and Parallelism Management*. PLDI, 2010.
...



```
def x(a, b, c):
    c[gid] = a[gid] + b[gid]

FunctionDef(
    name='x',
    args=arguments(
        args=[
            Name(id='a', ctx=Param()),
            Name(id='b', ctx=Param()),
            Name(id='c', ctx=Param())],
        vararg=None,
        kwarg=None,
        defaults=[]),
    body=[
        Assign(
            targets=[
                Subscript(
                    value=Name(id='c', ctx=Load()),
                    slice=Index(
                        value=Name(id='gid', ctx=Load())
                    ),
                    ctx=Store()
                )
            ],
            value=BinOp(
                left=Subscript(
                    value=Name(id='a', ctx=Load()),
                    slice=Index(
                        value=Name(id='gid', ctx=Load())
                    ),
                    ctx=Load()
                ),
                op=Add(),
                right=Subscript(
                    value=Name(id='b', ctx=Load()),
                    slice=Index(
                        value=Name(id='gid', ctx=Load())
                    ),
                    ctx=Load()
                )
            )
        )
    ],
    decorator_list=[]
)
```

Future Work

- Methods to guarantee buffer overflows don't happen
- **Richer type systems**: dependent types, typestate, ... open to guidance
- Integration with a **live environment** (remember Self and Smalltalk?) focused on scientific productivity
- DSL development for neurobiological circuit and system design
  what programming language is the brain written in?