

Easy, Effective, Efficient: GPU Programming in Python with PyOpenCL and PyCUDA

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

DTU GPU-Lab Workshop
Lecture 2 · August 17, 2011

Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions



Outline

- 1 Code writes Code
 - The Idea
 - RTCG in Action
 - How can I do it?
 - Case Study: Generic OpenCL Reduction
 - Reasoning about Generated Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions



Outline

1 Code writes Code

■ The Idea

■ RTCG in Action

■ How can I do it?

■ Case Study: Generic OpenCL Reduction

■ Reasoning about Generated Code

2 Interacting with the Rest of the World

3 PyCUDA

4 Loo.py

5 GPU-DG: Challenges and Solutions



The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards



The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

Devices differ by

- Memory Types, Latencies, Bandwidths
- Vector Widths
- Units of Scheduling



The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

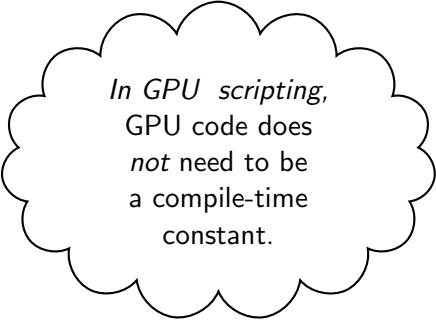
Devices differ by

- Memory Types, Latencies, Bandwidths
- Vector Widths
- Units of Scheduling



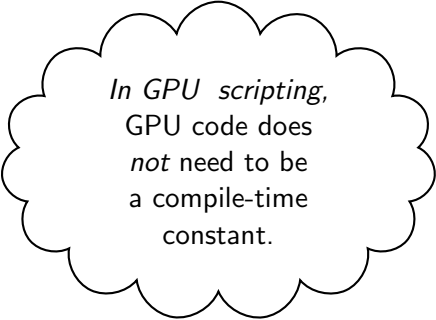
Optimally tuned code will (often)
be different for each device

Metaprogramming



*In GPU scripting,
GPU code does
not need to be
a compile-time
constant.*

Metaprogramming

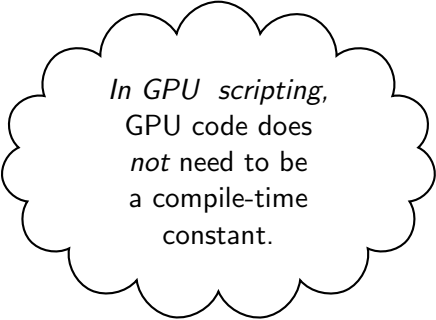


In GPU scripting,
GPU code does
not need to be
a compile-time
constant.

(Key: Code is data—it *wants* to be
reasoned about at run time)

Metaprogramming

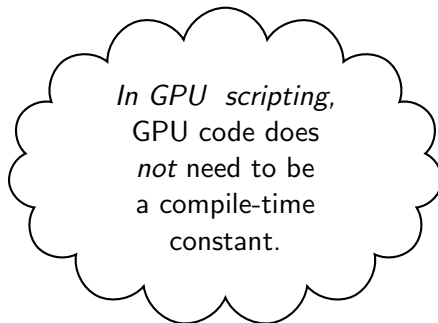
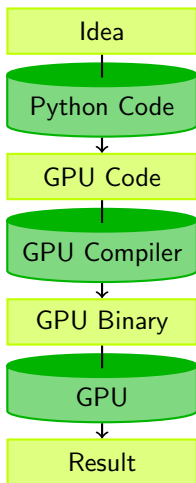
Idea



In GPU scripting,
GPU code does
not need to be
a compile-time
constant.

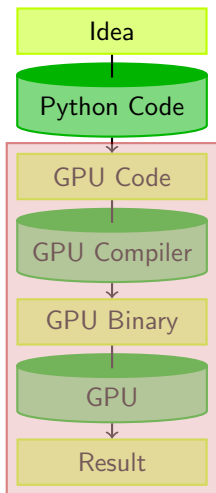
(Key: Code is data—it *wants* to be
reasoned about at run time)

Metaprogramming

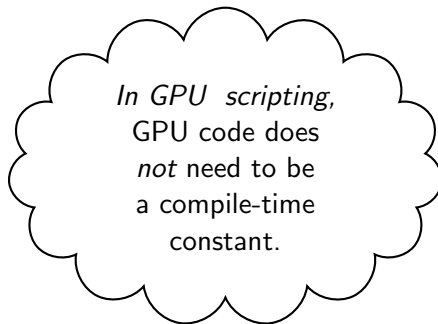


(Key: Code is data—it *wants* to be reasoned about at run time)

Metaprogramming

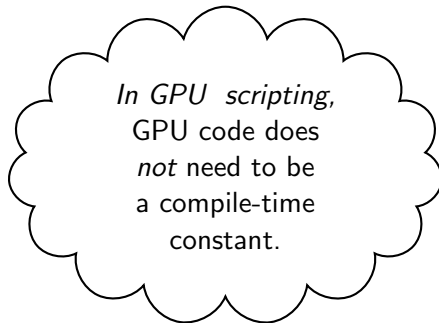
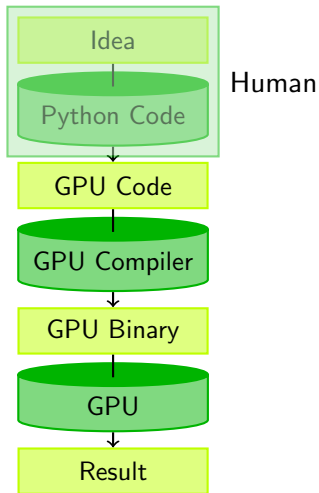


Machine



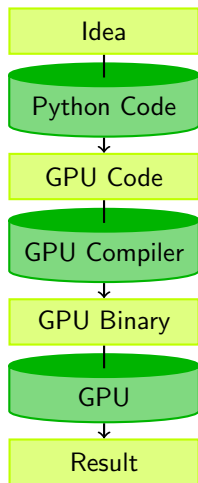
(Key: Code is data—it *wants* to be
reasoned about at run time)

Metaprogramming



(Key: Code is data—it *wants* to be reasoned about at run time)

Metaprogramming

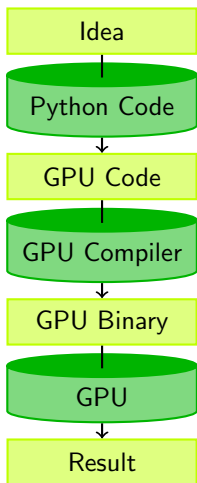


Good for code generation

In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)

Metaprogramming

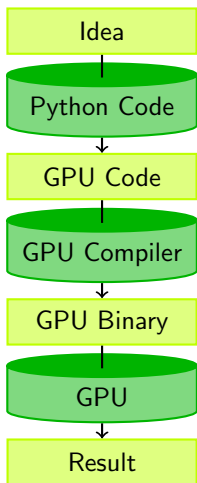


Good for code generation

In **PyCUDA**
GPU code does
not need to be
a compile-time
constant.

(Key: Code is data—it *wants* to be
reasoned about at run time)

Metaprogramming



Good for code generation

In **PyOpenCL**

GPU code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)

Machine-generated Code

Why machine-generate code?

- Automated Tuning
(cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
(→ register pressure)
- Loop Unrolling



PyOpenCL: Support for Metaprogramming

Three (main) ways of generating code:

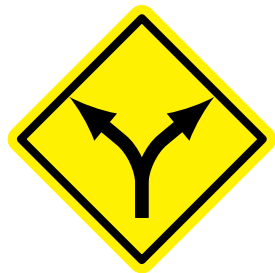
- Simple %-operator substitution
 - Combine with C preprocessor: simple, often sufficient
- Use a templating engine (Mako works very well)
- codepy:
 - Build C syntax trees from Python
 - Generates readable, indented C

Many ways of evaluating code—most important one:

- Exact device timing via events

How are High-Performance Codes constructed?

- “Traditional” Construction of High-Performance Codes:
 - C/C++/Fortran
 - Libraries
- “Alternative” Construction of High-Performance Codes:
 - Scripting for ‘brains’
 - GPUs for ‘inner loops’
- Play to the strengths of each programming environment.



Outline

1 Code writes Code

- The Idea

- **RTCG in Action**

- How can I do it?

- Case Study: Generic OpenCL Reduction

- Reasoning about Generated Code

2 Interacting with the Rest of the World

3 PyCUDA

4 Loo.py

5 GPU-DG: Challenges and Solutions



pyopencl.elementwise: Elementwise expressions

Avoiding extra store-fetch cycles for elementwise math:

```
n = 10000
a_gpu = cl_array.to_device(
    ctx, queue, numpy.random.randn(n).astype(numpy.float32))
b_gpu = cl_array.to_device(
    ctx, queue, numpy.random.randn(n).astype(numpy.float32))

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

c_gpu = cl_array.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
```

Outline

1 Code writes Code

- The Idea
- RTCG in Action
- **How can I do it?**
- Case Study: Generic OpenCL Reduction
- Reasoning about Generated Code

2 Interacting with the Rest of the World

3 PyCUDA

4 Loo.py

5 GPU-DG: Challenges and Solutions



RTCG via Substitution

```
source = ("""
__kernel void %(name)s(%(arguments)s)
{
    unsigned lid = get_local_id (0);
    unsigned gsize = get_global_size (0);
    unsigned work_item_start = get_local_size (0)*get_group_id (0);

    for (unsigned i = work_item_start + lid; i < n; i += gsize)
    {
        %(operation)s;
    }
}
""" % {
    "arguments": ", ".join(arg.declarator () for arg in arguments),
    "operation": operation,
    "name": name,
    "loop_prep": loop_prep,
})

prg = cl.Program(ctx, source). build ()
```

RTCG via Templates

```

from mako.template import Template

tpl = Template("""
    __kernel void add(
        __global ${ type_name } *tgt,
        __global const ${ type_name } *op1,
        __global const ${ type_name } *op2)
    {
        int idx = get_local_id (0)
            + ${ local_size } * ${ thread_strides }
            * get_group_id (0);

        % for i in range( thread_strides ):
            <% offset = i* local_size %>
            tgt[idx + ${ offset } ] =
                op1[idx + ${ offset } ]
                + op2[idx + ${ offset } ];
        % endfor
    }""")

rendered_tpl = tpl.render(type_name="float",
    local_size = local_size , thread_strides = thread_strides )

```


RTCG via AST Generation

```

from cgen import *
from cgen.opencl import \
    CLKernel, CLGlobal, CLRequiredWorkGroupSize

mod = Module([
    FunctionBody(
        CLKernel(CLRequiredWorkGroupSize((local_size,),
            FunctionDeclaration(Value("void", "twice"),
                arg_decls=[CLGlobal(Pointer(Const(POD(dtype, "tgt")))])),
            Block([
                Initializer(POD(numpy.int32, "idx"),
                    "get_local_id(0) + %d * get_group_id(0)"
                    % (local_size * thread_strides))
            ]+[
                Statement("tgt[idx+%d] *= 2" % (o*local_size))
                for o in range(thread_strides)
            ]))
    ]))

knl = cl.Program(ctx, str(mod)).build().twice

```

Outline

1 Code writes Code

- The Idea
- RTCG in Action
- How can I do it?
- **Case Study: Generic OpenCL Reduction**
- Reasoning about Generated Code

2 Interacting with the Rest of the World

3 PyCUDA

4 Loo.py

5 GPU-DG: Challenges and Solutions



Reduction

$$y = f(\dots f(f(x_1, x_2), x_3), \dots, x_N)$$

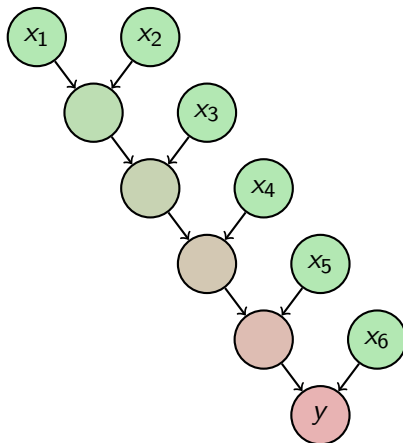
where N is the input size.

Also known as...

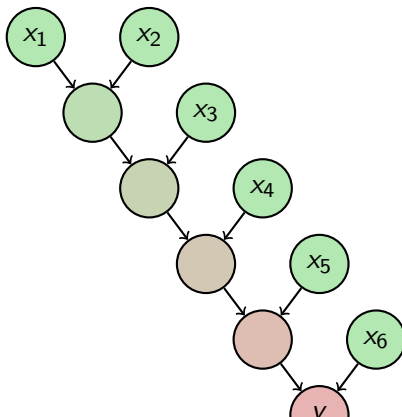
- Lisp/Python function reduce (Scheme: fold)
- C++ STL `std::accumulate`



Reduction: Graph

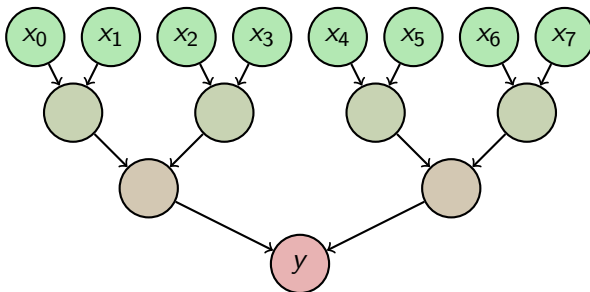


Reduction: Graph



Painful! Not parallelizable.

Reduction: A Better Graph



Mapping Reduction to the GPU

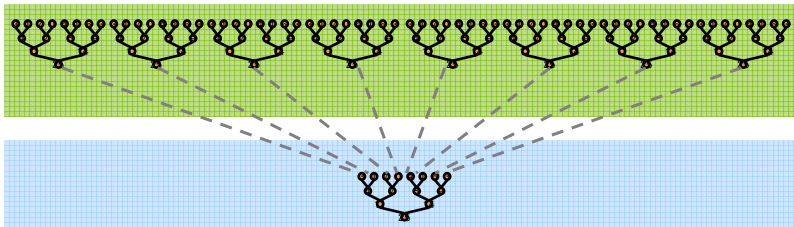
- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
 - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.

With material by M. Harris
(Nvidia Corp.)



Mapping Reduction to the GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
 - Need to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.



In particular: Use multiple grid invocations to achieve inter-group synchronization.

With material by M. Harris
(Nvidia Corp.)



Kernel V1

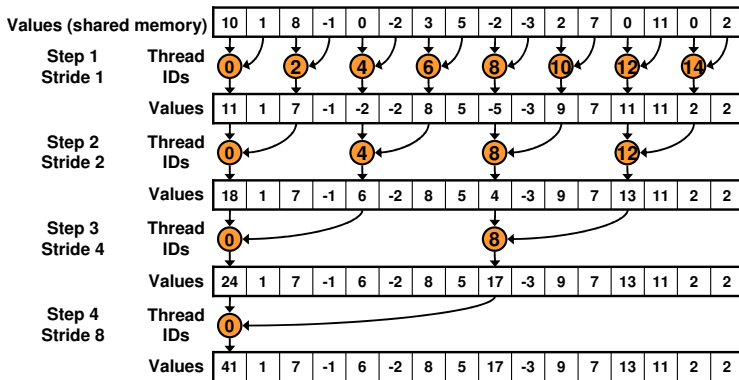
```
__kernel void reduce0( __global T *g_idata, __global T *g_odata,
    unsigned int n, __local T* ldata)
{
    unsigned int lid = get_local_id (0);
    unsigned int i = get_global_id (0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier (CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=1; s < get_local_size (0); s *= 2)
    {
        if ((lid % (2*s)) == 0)
            ldata[lid] += ldata[lid + s];
        barrier (CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
}
```

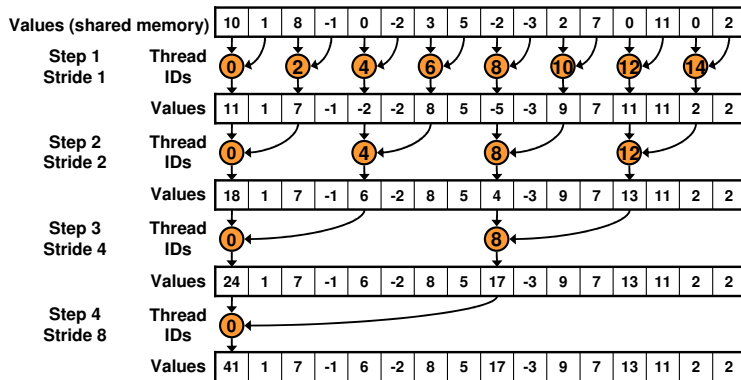
Interleaved Addressing



With material by M. Harris
(Nvidia Corp.)



Interleaved Addressing



Issue: Slow modulo, Divergence

With material by M. Harris
(Nvidia Corp.)



Kernel V2

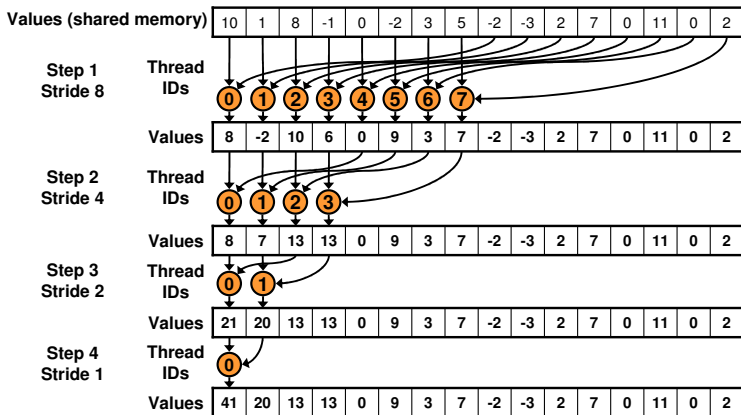
```
__kernel void reduce2( __global T *g_idata, __global T *g_odata,
    unsigned int n, __local T* ldata)
{
    unsigned int lid = get_local_id (0);
    unsigned int i = get_global_id (0);

    ldata[lid] = (i < n) ? g_idata[i] : 0;
    barrier (CLK_LOCAL_MEM_FENCE);

    for(unsigned int s= get_local_size (0)/2; s>0; s>>=1)
    {
        if (lid < s)
            ldata[lid] += ldata[lid + s];
        barrier (CLK_LOCAL_MEM_FENCE);
    }

    if (lid == 0) g_odata[ get_local_size (0)] = ldata [0];
}
```

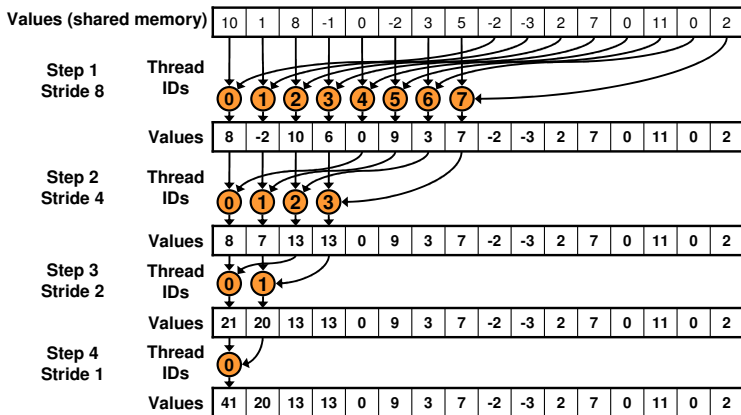
Sequential Addressing



With material by M. Harris
(Nvidia Corp.)



Sequential Addressing



Better! But still not “efficient”.

Only half of all work items after first round
then a quarter, ...

With material by M. Harris
(Nvidia Corp.)



Thinking about Parallel Complexity

Distinguish:

- Time on T processors: T_P
- **Step Complexity/Span** T_∞ : Minimum number of steps taken if an infinite number of processors are available
- Work per step S_t
- **Work Complexity/Work** $T_1 = \sum_{t=1}^{T_\infty} S_t$: Total number of operations performed
- **Parallelism** T_1/T_∞ : average amount of work along span
 - $P > T_1/T_\infty$ doesn't make sense.

Algorithm-specific!

Thinking about Parallel Complexity

Distinguish:

- Time on T processors: T_P
- **Step Complexity/Span** T_∞ : Minimum number of steps taken if an infinite number of processors are available
- Work per step S_t
- **Work Complexity/Work** $T_1 = \sum_{t=1}^{T_\infty} S_t$: Total number of operations performed
- **Parallelism** T_1/T_∞ : average amount of work along span
 - $P > T_1/T_\infty$ d

Algorithm-specific!

- How parallel is our current version?
- Can we improve it?

Kernel V3 Part 1

```
__kernel void reduce6( __global T *g_idata, __global T *g_odata,
    unsigned int n, volatile __local T* ldata)
{
    unsigned int lid = get_local_id (0);
    unsigned int i = get_group_id(0)*(
        get_local_size (0)*2) + get_local_id (0);
    unsigned int gridSize = GROUP_SIZE*2*get_num_groups(0);
    ldata [ lid ] = 0;

    while (i < n)
    {
        ldata [ lid ] += g_idata[i];
        if (i + GROUP_SIZE < n)
            ldata [ lid ] += g_idata[i+GROUP_SIZE];
        i += gridSize;
    }
    barrier (CLK_LOCAL_MEM_FENCE);
}
```

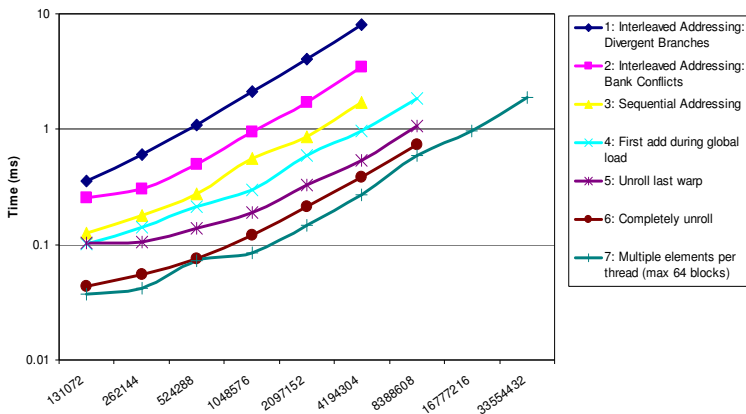
Kernel V3 Part 2

```
if (GROUP_SIZE >= 512)
{
    if (lid < 256) { ldata[lid] += ldata[lid + 256]; }
    barrier (CLK_LOCAL_MEM_FENCE);
}
// ...
if (GROUP_SIZE >= 128)
{ /* ... */ }

if (lid < 32)
{
    if (GROUP_SIZE >= 64) { ldata[lid] += ldata[lid + 32]; }
    if (GROUP_SIZE >= 32) { ldata[lid] += ldata[lid + 16]; }
    // ...
    if (GROUP_SIZE >= 2) { ldata[lid] += ldata[lid + 1]; }
}

if (lid == 0) g_odata[get_group_id(0)] = ldata[0];
}
```

Performance Comparison



With material by M. Harris
(Nvidia Corp.)



Generic CL Reduction: Preparation

```
#define GROUP_SIZE ${group_size}
#define READ_AND_MAP(i) (${map_expr})
#define REDUCE(a, b) (${reduce_expr})

% if double_support:
    #pragma OPENCL EXTENSION cl_khr_fp64: enable
% endif

typedef ${out_type} out_type;

${preamble}
```

CL Reduction: Sequential Part

```
__kernel void ${name}({
    __global out_type *out, ${arguments},
    unsigned int seq_count, unsigned int n)
{
    __local out_type ldata[GROUP_SIZE];
    unsigned int lid = get_local_id(0);
    unsigned int i = get_group_id(0)*GROUP_SIZE*seq_count + lid;

    out_type acc = ${neutral};
    for (unsigned s = 0; s < seq_count; ++s)
    {
        if (i >= n) break;
        acc = REDUCE(acc, READ_AND_MAP(i));
        i += GROUP_SIZE;
    }
}
```

CL Reduction: Explicitly Synchronized Part

```
ldata[lid] = acc;

<% cur_size = group_size %>

% while cur_size > no_sync_size:
    barrier (CLK_LOCAL_MEM_FENCE);

    <%
    new_size = cur_size // 2
    assert new_size * 2 == cur_size
    %>

    if (lid < ${new_size})
    {
        ldata[lid] = REDUCE(
            ldata[lid],
            ldata[lid + ${new_size}]);
    }

    <% cur_size = new_size %>

% endwhile
```

CL Reduction: Implicitly Synchronized Part

```

% if cur_size > 1:
    barrier (CLK_LOCAL_MEM_FENCE);

    if ( lid < ${no_sync_size})
    {
        __local volatile out_type *lvdata = ldata;
        % while cur_size > 1:
            <%
                new_size = cur_size // 2
                assert new_size * 2 == cur_size
            %>
            lvdata [ lid ] = REDUCE(
                lvdata [ lid ],
                lvdata [ lid + ${new_size}]);
            <% cur_size = new_size %>
        % endwhile
    }
% endif

if ( lid == 0) out[get_group_id(0)] = ldata [0];
}

```

Outline

1 Code writes Code

- The Idea
- RTCG in Action
- How can I do it?
- Case Study: Generic OpenCL Reduction
- Reasoning about Generated Code

2 Interacting with the Rest of the World

3 PyCUDA

4 Loo.py

5 GPU-DG: Challenges and Solutions



Judging Code Quality

Possible information sources for judging code quality/desirability:

- Heuristics (e.g. Occupancy, Flops/Byte, ... ?)
- OpenCL Event profiling
 - Makes comp. synchronous on Nvidia!
- Wall time (!)
- Compiler build log
- Vendor Profiler



Search Strategies

Possible search strategies:

- Exhaustive
- Exhaustive + Heuristics
- Grouped Orthogonal Search
- Genetic Algorithms
- (your invention here)

Compiler cache makes repeated searches fast.



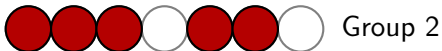
Grouped Orthogonal Search



GAOS: Adrian Tate, Cray, Inc.

Grouped Orthogonal Search

Define groups



GAOS: Adrian Tate, Cray, Inc.

Grouped Orthogonal Search

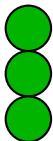
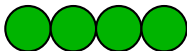
Choose group



GAOS: Adrian Tate, Cray, Inc.

Grouped Orthogonal Search

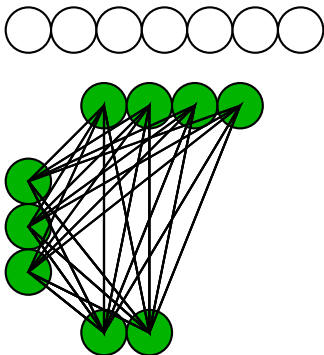
Map admissible options



GAOS: Adrian Tate, Cray, Inc.

Grouped Orthogonal Search

Group-wide exhaustive search



GAOS: Adrian Tate, Cray, Inc.

Grouped Orthogonal Search

Start over with best result → pick new group. . .



GAOS: Adrian Tate, Cray, Inc.

Using the Nvidia profiler in-process

```
1 # enable profiler
2 import os
3 os.environ["COMPUTE_PROFILE"] = "1"
4 with open("/tmp/myprg-prof-config", "w") as prof_config:
5     prof_config.write("\n".join(events))
6 os.environ["COMPUTE_PROFILE_CONFIG"] = "/tmp/myprg-prof-config"
7
8 # obtain timing data
9 prof_f = open("openccl_profile_0.log", "r")
10 gain_count = 0
11
12 while gain_count < 2:
13     # run kernel here
14     prof_output = prof_f.readlines()
15     if prof_output:
16         print "gained %d lines" % len(prof_output)
17         gain_count += 1
18         if gain_count == 2:
19             print "".join(l for l in prof_output[1:-1]
20                             if kernel_name in l)
```

Using the Nvidia profiler in-process

```

1  # enable profiler
2  import os
3  os.environ["COMPUTE_PROFILE"] = "1"
4  with open("/tmp/myprg-prof-config", "w") as prof_config:
5      prof_config.write("\n".join(events))
6  os.environ["COMPUTE_PROFILE_CONFIG"] = "/tmp/myprg-prof-config"
7
8  # obtain timing data
9  prof_f = open("opencl_profile_0.log", "r")
10 gain_count = 0
11
12 while gain_count < 2:
13     # run
14     prof_out = prof_f.read()
15     if prof_out:
16         pri = prof_out.split("\n")
17         gain_count += 1
18     if gain_count == 2:
19         break
20 
```

Sample output:

method=[matvec]	gputime=[7218.048]	cputime=[12.000]	occupancy=[1.000]
method=[matvec]	gputime=[7267.456]	cputime=[14.000]	occupancy=[1.000]
method=[matvec]	gputime=[7264.640]	cputime=[12.000]	occupancy=[1.000]
method=[matvec]	gputime=[7270.048]	cputime=[15.000]	occupancy=[1.000]
method=[matvec]	gputime=[7262.976]	cputime=[12.000]	occupancy=[1.000]
method=[matvec]	gputime=[7237.152]	cputime=[23.000]	occupancy=[1.000]

Nvidia GPU Profiler: Events

`gld_request` : Number of executed global load instructions per warp in a SM

`gst_request` : Number of executed global store instructions per warp in a SM

`divergent_branch` : Number of unique branches that diverge
`instructions` : Instructions executed

`warp_serialized` : Number of SIMD groups that serialize on address conflicts to local memory

And many more: see (root of CUDA
toolkit)/(doc/Compute_Profiler_VERSION.txt
(Careful: CUDA terminology)



Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
 - F2Py: Interacting with Fortran
 - MPI and Python
 - Python and C++
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions

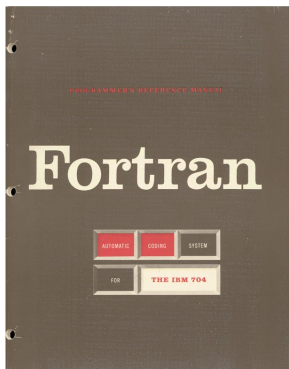


Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
 - F2Py: Interacting with Fortran
 - MPI and Python
 - Python and C++
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions



F2Py



f2py...

- very easily exposes Fortran routines
 - just add `intent(in/out)`
- supports F77 and much of F90 (not data types)
- Allows call-back into Python
- Much easier than C
 - No issues with pointer ownership
- PyOpenCL+F2Py: Good way of bringing Fortran codes into the “GPU age”

Simple f2py example

```
subroutine fib(a,n)
  integer, intent(in) :: n
  real*8, intent(out) :: a(n)

  do i=1,n
    if (i.eq.1) then
      a(i) = 0.0d0
    elseif (i.eq.2) then
      a(i) = 1.0d0
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end
```

Code Credit: Pearu Peterson

Simple f2py example

```

subroutine fib(a,n)
  integer, intent(in) :: n
  real*8, intent(out) :: a(n)

  do i=1,n
    if (i.eq.1)
      a(i) = 0
    elseif (i.eq.2)
      a(i) = 1
    else
      a(i) = a(i-1) + a(i-2)
    endif
  enddo
end

```

```
$ f2py -m f2pyex -c f2pyex.f90
```

```
$ python
```

```
>>>import f2pyex
```

```
>>>print f2pyex.fib.__doc__
```

```
fib - Function signature:
```

```
    a = fib(n)
```

```
Required arguments:
```

```
    n : input int
```

```
Return objects:
```

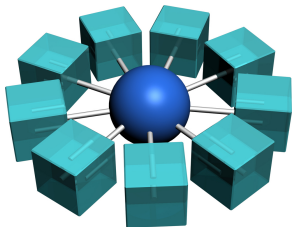
```
    a : rank-1 array('d') with bounds (n)
```


Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
 - F2Py: Interacting with Fortran
 - MPI and Python
 - Python and C++
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions



mpi4py: Using MPI from Python



mpi4py...

- wraps MPI 2 into Python
- aimed at numpy arrays and general data
- interfaces with Fortran/F2Py, SWIG, Cython

<https://code.google.com/p/mpi4py/>

Simple MPI with Python objects

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Can communicate arbitrary Python objects
- Versatile, potentially slow
 - CPU needs to pack data into buffer

Code Credit: Lisandro Dalcin (CIMEC, Argentina)

MPI with numpy objects

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

- Can communicate “POD” numpy arrays
- Fast: No Copying
- Note difference: Send vs. send

Code Credit: Lisandro Dalcin (CIMEC, Argentina)

MPI with numpy objects

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

- Can communicate “POD” numpy arrays
- Fast: No Copying
- Note difference: Send vs

If non-blocking, same issue with buffer lifetime as PyOpenCL.

Code Credit: Lisandro Dalcin (CIME)

MPI collectives

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

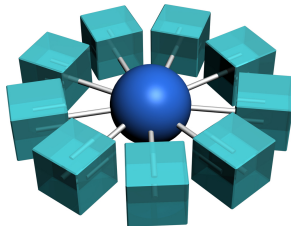
Also available:

- One-sided, non-blocking, synchronous, ...
- Dynamic process creation
- MPI I/O

Code Credit: Lisandro Dalcin (CIMEC, Argentina)

mpi4py: Final Observations

- Common theme with PyOpenCL:
numpy arrays work well as a vehicle for bulk data
 - Side Corollary: Avoid touching bulk data in Python
- Nvidia GPUDirect:
 - Use host-mapped GPU arrays for MPI communication
 - Avoid GPU → ~~Host~~ → IB copies
 - ```
b = cl.Buffer(...,
 cl.mem_flags.ALLOC_HOST_PTR)
cl.enqueue_map_buffer(
 queue, b, ...)
```



# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
  - F2Py: Interacting with Fortran
  - MPI and Python
  - Python and C++
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions





# Boost.Python: Connecting C/C++ and Python



Boost.Python (BPL)...

- uses template “magic” to introspect C++
- provides enough tools to write ‘idiomatic’ wrappers
- wraps the Python/C API to make it ‘idiot-proof’
- deals well with complicated pointer ownership
- is used by PyOpenCL, PyCUDA, ...

<http://boost.org/>

# Simple BPL example

```
#include <boost/python.hpp>

static char const *greet()
{
 return "hello world";
}

BOOST_PYTHON_MODULE(module)
{
 boost::python::def("greet", &greet);
}
```

- Wrapper generation happens in pure C++
  - Good if wrapping complicated types
  - Wrapper generator exists
- Compile with 'Distutils' (see PyOpenCL build system for example)

# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA**
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions



# Whetting your appetite

```
1 import pycuda.driver as cuda
2 import pycuda.autotinit, pycuda.compiler
3 import numpy
4
5 a = numpy.random.randn(4,4).astype(numpy.float32)
6 a_gpu = cuda.mem_alloc(a.nbytes)
7 cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCUDA distribution.]

# Whetting your appetite

```
1 mod = pycuda.compiler.SourceModule("""
2 __global__ void twice(float *a)
3 {
4 int idx = threadIdx.x + threadIdx.y*4;
5 a[idx] *= 2;
6 }
7 """)
8
9 func = mod.get_function("twice")
10 func(a_gpu, block=(4,4,1))
11
12 a_doubled = numpy.empty_like(a)
13 cuda.memcpy_dtoh(a_doubled, a_gpu)
14 print a_doubled
15 print a
```

# Whetting your appetite

```

1 mod = pycuda.compiler.SourceModule("""
2 __global__ void twice(float *a)
3 {
4 int idx = threadIdx.x + threadIdx.y*4;
5 a[idx] *= 2;
6 }
7 """)
8
9 func = mod.get_function("twice")
10 func(a_gpu, block=(4,4,1))
11
12 a_doubled = numpy.empty_like(a)
13 cuda.memcpy_dtoh(a_doubled, a_gpu)
14 print a_doubled
15 print a

```

Compute kernel

# Whetting your appetite, Part II

Did somebody say “Abstraction is good”?



# Whetting your appetite, Part II

```
1 import numpy
2 import pycuda.autoinit
3 import pycuda.gpuarray as gpuarray
4
5 a_gpu = gpuarray.to_gpu(
6 numpy.random.randn(4,4).astype(numpy.float32))
7 a_doubled = (2*a_gpu).get()
8 print a_doubled
9 print a_gpu
```



# gpuarray: Simple Linear Algebra

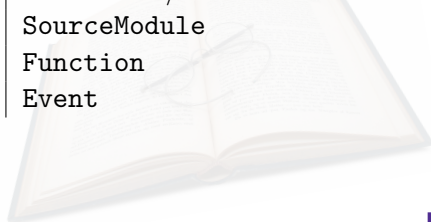
pycuda.gpuarray:

- Meant to look and feel just like numpy.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- `+`, `-`, `*`, `/`, `fill`, `sin`, `exp`, `rand`,  
basic indexing, `norm`, inner product, ...
- Mixed types (`int32 + float32 = float64`)
- `print gpuarray` for debugging.
- Allows access to raw bits
  - Use as kernel arguments, textures, etc.

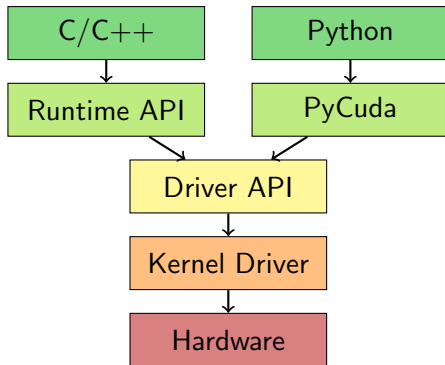


# PyOpenCL $\leftrightarrow$ PyCUDA: A (rough) dictionary

| PyOpenCL                   | PyCUDA                       |
|----------------------------|------------------------------|
| Context                    | Context                      |
| CommandQueue               | Stream                       |
| Buffer                     | mem_alloc / DeviceAllocation |
| Program                    | SourceModule                 |
| Kernel                     | Function                     |
| Event (eg. enqueue_marker) | Event                        |



# PyCUDA in the CUDA ecosystem



CUDA has two Programming Interfaces:

- “Runtime” high-level (separate install)
- “Driver” low-level (`libcuda.so`, comes with GPU driver)

# PyCUDA: Vital Information

- <http://mathematician.de/software/pycuda>
- Complete documentation
- X Consortium License  
(no warranty, free for all use)
- Convenient abstractions  
Array, Fast Vector Math, Reductions
- Requires: numpy, Python 2.4+  
(Win/OS X/Linux)



# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py**
- 5 GPU-DG: Challenges and Solutions



# Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

Obvious idea: Let the computer do it.

- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

# Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

Obvious idea: Let the computer do it.

- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

**BAD  
IDEA**

# Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

Obvious idea: Let the computer do it.

- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Brute enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

BAD  
IDEA  
WORSE  
IDEA



# Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

Obvious idea: Let

Well, what then?

- One way: Smart enumeration
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Brute enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

**WORSE  
IDEA**

# Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

Obvious idea: Let

- One way: Small
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Build en
  - Enumerate loop slic
  - Enumerate prefetch
  - Choose by running re

Well, what then?

My experience:

*Users know what they're doing,  
but they're lazy and in a hurry.*

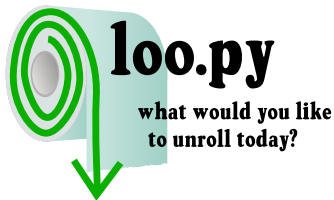
So give them tools and get out of  
the way.

WORTH  
IDEA

# Enter: Loo.py

*Idea:* Expose loop transformations that compilers can do *to the user*.

- Somewhat similar to directive-based model
- But not really: Loop kernel is a Python object
  - Can query it for its properties
  - Can control transformation according to target device
  - Can easily integrate into other software
- Use polyhedral model to represent loop bounds
  - And knowledge about divisibility, parameters, etc.



# Loo.py Example

```
knl = lp.LoopKernel(ctx.devices[0],
 "{[i,j,k]: 0<=i,j,k<1600}",
 [
 (c[i, j], a[i, k]*b[k, j])
],
 [
 lp.ArrayArg("a", dtype, shape=(1600, 1600), order="C"),
 lp.ArrayArg("b", dtype, shape=(1600, 1600), order="C"),
 lp.ArrayArg("c", dtype, shape=(1600, 1600), order="C"),
],
 name="matmul")

knl = lp.split_dimension(knl, "i", 16, outer_tag="g.0", inner_tag="l.1")
knl = lp.split_dimension(knl, "j", 16, outer_tag="g.1", inner_tag="l.0")
knl = lp.split_dimension(knl, "k", 16)
knl = lp.add_prefetch(knl, 'a', ["k_inner", "i_inner"])
knl = lp.add_prefetch(knl, 'b', ["j_inner", "k_inner",])

```

Lo

```

#define i_outer (0 + (int) get_group_id(0)) /* [0, 10) */
#define j_outer (0 + (int) get_group_id(1)) /* [0, 10) */
#define j_inner (0 + (int) get_local_id(0)) /* [0, 16) */
#define i_inner (0 + (int) get_local_id(1)) /* [0, 16) */

__kernel void __attribute__((reqd_work_group_size(16, 16, 1)))
matmul(__global float const *restrict a, __global float const *restrict b, __global float *restrict c)
{
 __local float prefetch_a_0 [16][16];
 __local float prefetch_b_1 [16][16];
 float tmp_c_0 = 0;
 for (int k_outer = 0; 9 - k_outer >= 0; ++k_outer)
 {
 barrier(CLK_LOCAL_MEM_FENCE);
 {
 int const loopy_prefetch_dim_idx_0 = (int) get_local_id(0);
 int const loopy_prefetch_dim_idx_1 = (int) get_local_id(1);
 prefetch_b_1[loopy_prefetch_dim_idx_0][loopy_prefetch_dim_idx_1] =
 b[loopy_prefetch_dim_idx_0 + j_outer*16 + 160*(loopy_prefetch_dim_idx_1 + k_outer*16)];
 }
 {
 int const loopy_prefetch_dim_idx_0 = (int) get_local_id(0);
 int const loopy_prefetch_dim_idx_1 = (int) get_local_id(1);
 prefetch_a_0[loopy_prefetch_dim_idx_0][loopy_prefetch_dim_idx_1] =
 a[loopy_prefetch_dim_idx_0 + k_outer*16 + 160*(loopy_prefetch_dim_idx_1 + i_outer*16)];
 }
 barrier(CLK_LOCAL_MEM_FENCE);

 for (int k_inner = 0; 15 - k_inner >= 0; ++k_inner)
 tmp_c_0 += prefetch_b_1[j_inner - 0][k_inner - 0]*prefetch_a_0[k_inner - 0][i_inner - 0];
 }
 c[j_outer*16 + j_inner + 160*(i_outer*16 + i_inner)] = tmp_c_0;
}

```

# Loo.py Example

```
knl = lp.LoopKernel(ctx.devices[0],
 "{[i,j,k]: 0<=i,j,k<1600}",
 [
 (c[i, j], a[i, k]*b[k, j])
],
 [
 lp.ArrayArg("a", dtype, shape=(1600, 1600), order="C"),
 lp.ArrayArg("b", dtype, shape=(1600, 1600), order="C"),
 lp.ArrayArg("c", dtype, shape=(1600, 1600), order="C"),
],
 name="matmul")

knl = lp.split_dimension(knl, "i", 16, outer_tag="g.0", inner_tag="l.1")
knl = lp.split_dimension(knl, "j", 16, outer_tag="g.1", inner_tag="l.0")
knl = lp.split_dimension(knl, "k", 16)
knl = lp.add_prefetch(knl, 'a', ["k_inner", "i_inner"])
knl = lp.add_prefetch(knl, 'b', ["j_inner", "k_inner",])

```

# Loo.py Example II

```

knl = lp.LoopKernel(ctx.devices[0],
 "[n] -> {[i,j,k]: 0<=i,j,k<n}",
 [
 (c[i, j], a[i, k]*b[k, j])
],
 [
 lp.ImageArg("a", dtype, 2),
 lp.ImageArg("b", dtype, 2),
 lp.ArrayArg("c", dtype, shape=(n_sym, n_sym), order=order),
 lp.ScalarArg("n", np.int32, approximately=1000),
],
 name="matmul")

ilp = 4; j_inner_split = 16
knl = lp.split_dimension(knl, "i", 2, outer_tag="g.0", inner_tag="l.1")
knl = lp.split_dimension(knl, "j", ilp * j_inner_split, outer_tag="g.1")
knl = lp.split_dimension(knl, "j_inner", j_inner_split,
 outer_tag="ilp", inner_tag="l.0")
knl = lp.split_dimension(knl, "k", 16)
knl = lp.add_prefetch(knl, 'a', ["i_inner", "k_inner"])
knl = lp.add_prefetch(knl, 'b', [("j_inner_outer", "j_inner_inner"), "k_inner"])

```

# Loo.py Example II

```

knl = lp.LoopKernel(ctx.devices[0],
 "[n] -> {[i,j,k]: 0<=i,j,k<n}",
 [
 (c[i, j], a[i, k]*b[k, j])
],
 [
 lp.ImageArg("a", dtype, 2),
 lp.ImageArg("b", dtype, 2),
 lp.ArrayArg("c", dtype, shape=(n_sym, n_sym), order=order),
 lp.ScalarArg("n", np.int32, approximately=1000),
],
 name="matmul")

ilp = 4; j_inner_split = 16
knl = lp.split_dimension(knl, "i", 2, outer_tag="g.0", inner_tag="l.1")
knl = lp.split_dimension(knl, "j", ilp * j_inner_split, outer_tag="g.1")
knl = lp.split_dimension(knl, "j_inner", j_inner_split,
 outer_tag="ilp", inner_tag="l.0")
knl = lp.split_dimension(knl, "k", 1)
knl = lp.add_prefetch(knl, 'a', ["i", "j"])
knl = lp.add_prefetch(knl, 'b', [("j", "j_inner")])

```

Resulting code 1000+ lines long



# Loo.py Summary

Can do today:

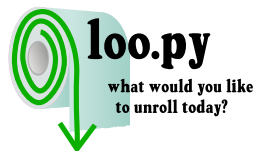
- Split dimensions
- “Tag” dimensions: GPU axes, unroll, ILP
- Lmem prefetch (with dim. merging)
- Loop schedule
- Conditional avoidance
- Warn about lmem conflicts
- Images, constant memory

Status:

- Initial goal: recipes for BLAS, DG
  - Obviously not for every computation
  - Quite general: convolutions, stencils, ...

- Public, but pre-alpha:

<http://github.com/inducer/loopy>



# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions
  - Introduction
  - Challenges
  - Benefits of Metaprogramming
  - GPU-DG: Performance and Generality



# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions
  - Introduction
  - Challenges
  - Benefits of Metaprogramming
  - GPU-DG: Performance and Generality



# Discontinuous Galerkin Method

Let  $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$ .



# Discontinuous Galerkin Method

Let  $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$ .



## Goal

Solve a *conservation law* on  $\Omega$ :

$$u_t + \nabla \cdot F(u) = 0$$

# Discontinuous Galerkin Method

Let  $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$ .



## Goal

Solve a *conservation law* on  $\Omega$ :

$$u_t + \nabla \cdot F(u) = 0$$

## Example

*Maxwell's Equations*: EM field:  $E(x, t)$ ,  $H(x, t)$  on  $\Omega$  governed by

$$\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -\frac{j}{\varepsilon},$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon},$$

$$\partial_t H + \frac{1}{\mu} \nabla \times E = 0,$$

$$\nabla \cdot H = 0.$$

# Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

$$\begin{aligned} 0 &= \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx \\ &= \int_{D_k} u_t \varphi - F(u) \cdot \nabla \varphi \, dx + \int_{\partial D_k} (\hat{n} \cdot F)^* \varphi \, dS_x, \end{aligned}$$

Substitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices  $S$ ,  $M$ ,  $M_A$ :

$$\partial_t u^k = - \sum_{\nu} D^{\partial_{\nu}, k} [F(u^k)] + L^k [\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial D_k}.$$

For straight-sided simplicial elements:

Reduce  $D^{\partial_{\nu}}$  and  $L$  to reference matrices.



# DG on GPUs: Possible Advantages



## DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
  - The majority of DG is local.
- Compute Bandwidth  $\gg$  Memory Bandwidth
  - DG is arithmetically intense.
- GPUs favor dense data.
  - Local parts of the DG operator are dense.
- GPUs don't like scattered access.
  - DG's cell connectivity is sparser than CG's
  - and more regular.



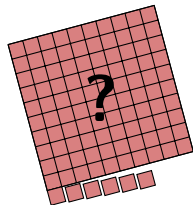
# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions
  - Introduction
  - **Challenges**
  - Benefits of Metaprogramming
  - GPU-DG: Performance and Generality



# Work Partition for Element-Local Operators

**Natural Work Decomposition:**  
One Element per Block

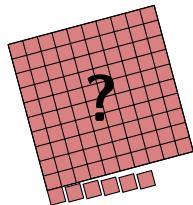


# Work Partition for Element-Local Operators

## Natural Work Decomposition:

One Element per Block


- ➕ Straightforward to implement
- ➕ No granularity penalty
- ➖ Cannot fill wide SIMD: unused compute power for small to medium elements
- ➖ Data alignment: Padding wastes memory
- ➖ Cannot amortize cost of preparation steps (e.g. fetching)



# Loop Slicing for element-local parts of GPU DG

**Per Block:**  $K_L$  element-local mat.mult. + matrix load



**Question:** How should one assign work to threads? 

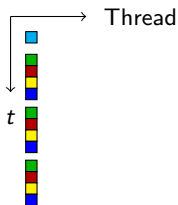
# Loop Slicing for element-local parts of GPU DG

**Per Block:**  $K_L$  element-local mat.mult. + matrix load



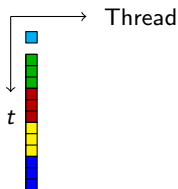
**Question:** How should one assign work to threads? ?

$w_s$ : in sequence



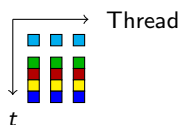
(amortize preparation)

$w_i$ : "inline-parallel"



(exploit register space)

$w_p$ : in parallel



# Outline

- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions
  - Introduction
  - Challenges
  - **Benefits of Metaprogramming**
  - GPU-DG: Performance and Generality



# Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms



# Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity





# Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes

# Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
- Loop Unrolling

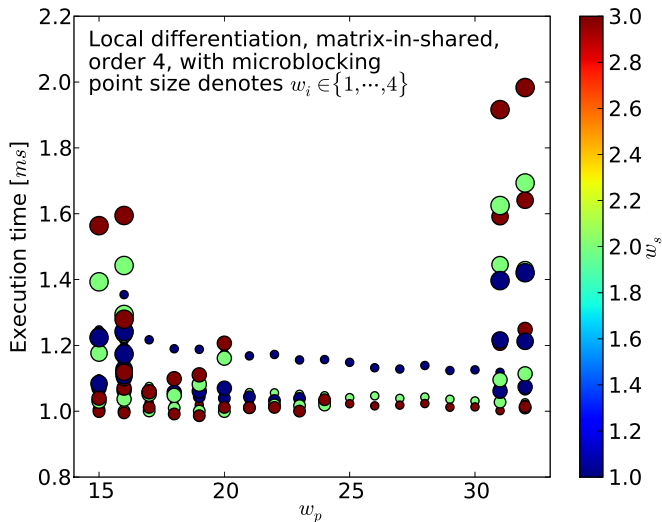


# Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms (\*)
- Automated Tuning:
  - Memory layout
  - Loop slicing (\*)
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
- Loop Unrolling



# Loop Slicing for Differentiation



# Metaprogramming DG: Flux Terms

$$0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \underbrace{\int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x}_{\text{Flux term}}$$

# Metaprogramming DG: Flux Terms

$$0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \underbrace{\int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x}_{\text{Flux term}}$$

Flux terms:

- vary by problem
- expression specified by user
- evaluated pointwise

# Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times (\llbracket H \rrbracket - \alpha \hat{n} \times \llbracket E \rrbracket)]$$

# Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times (\llbracket H \rrbracket - \alpha \hat{n} \times \llbracket E \rrbracket)]$$

**User writes:** Vectorial statement in math. notation

```
flux = 1/2*cross(normal, h.int - h.ext
 - alpha*cross(normal, e.int - e.ext))
```



# Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell's Equations

$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times (\llbracket H \rrbracket - \alpha \hat{n} \times \llbracket E \rrbracket)]$$

**We generate:** Scalar evaluator in C (6×)

```
a_flux += (
 (((val_a_field5 - val_b_field5) * fpair ->normal[2]
 - (val_a_field4 - val_b_field4) * fpair ->normal[0])
 + val_a_field0 - val_b_field0) * fpair ->normal[0]
 - (((val_a_field4 - val_b_field4) * fpair ->normal[1]
 - (val_a_field1 - val_b_field1) * fpair ->normal[2])
 + val_a_field3 - val_b_field3) * fpair ->normal[1]
) * value_type (0.5);
```

# Hedge DG Solver



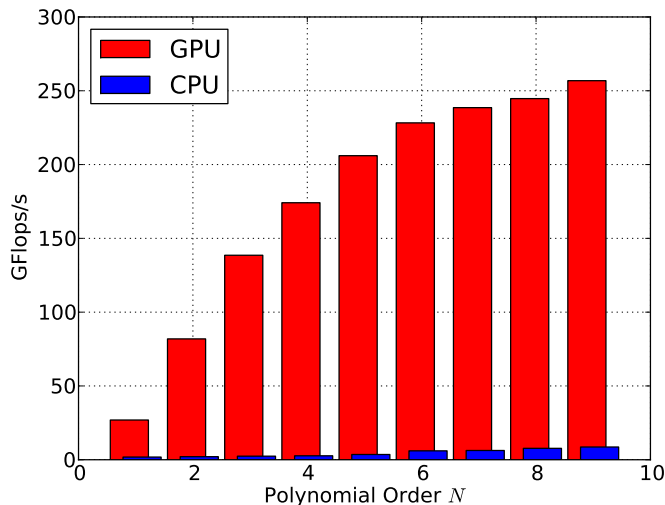
- High-Level Operator Description
  - Maxwell's
  - Euler
  - Poisson
  - Compressible Navier-Stokes, ...
- One Code runs...
  - ...on CPU, CUDA
  - ...on {CPU,CUDA}+MPI
  - ...in 1D, 2D, 3D
  - ...at any order
- Uses CPU, GPU code generation
- Open Source (GPL3)
- Written in Python,

# Outline

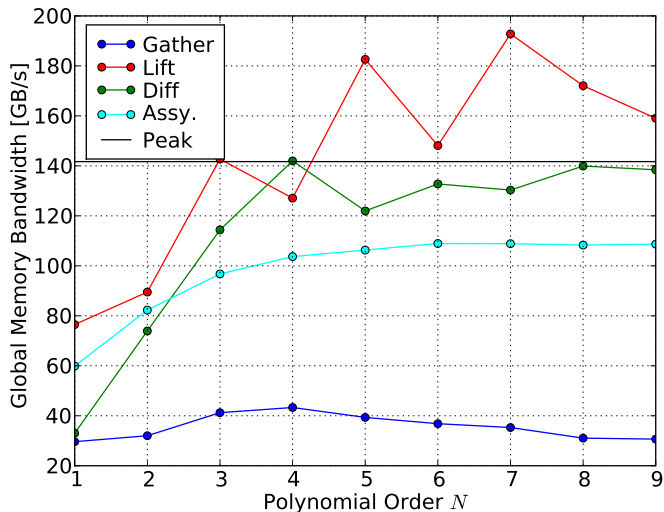
- 1 Code writes Code
- 2 Interacting with the Rest of the World
- 3 PyCUDA
- 4 Loo.py
- 5 GPU-DG: Challenges and Solutions
  - Introduction
  - Challenges
  - Benefits of Metaprogramming
  - GPU-DG: Performance and Generality



# Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400

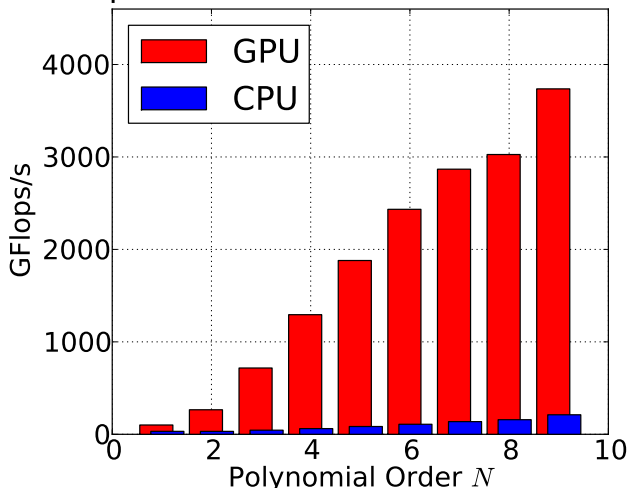


# Memory Bandwidth on a GTX 280



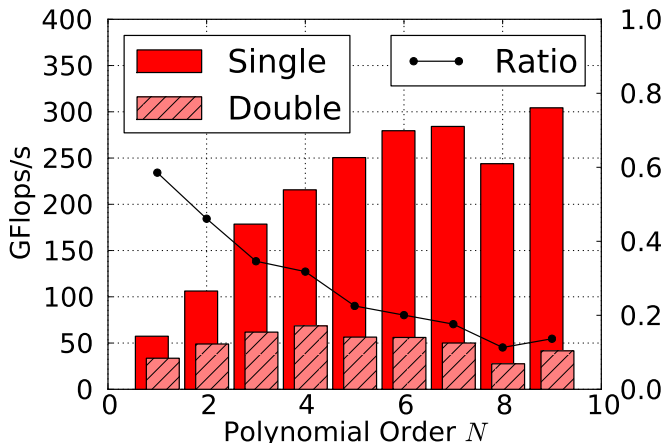
# Multiple GPUs via MPI: 16 GPUs vs. 64 CPUs

## Flop Rates: 16 GPUs vs 64 CPU cores

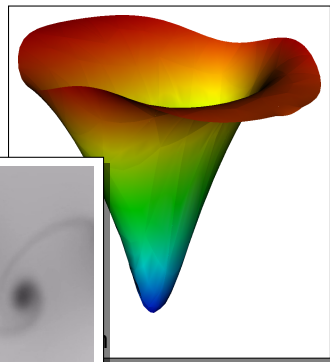
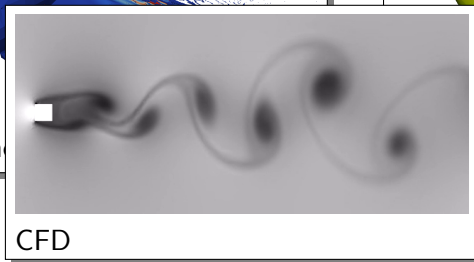
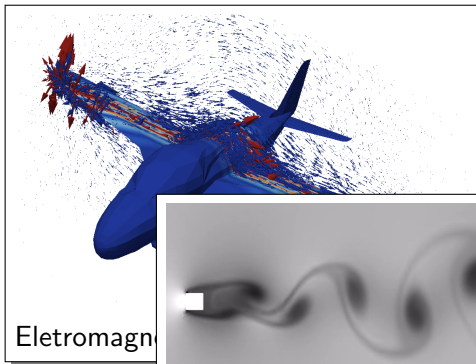


# GPU-DG in Double Precision

## GPU-DG: Double vs. Single Precision



# GPU DG Showcase





# Conclusions

- OpenCL: A way to talk to *all* computing power in the box
  - Many exciting developments in/around/for CL
- OpenCL and Python scripting work surprisingly well together
  - Python/numpy connect to existing software
  - Enable run-time code generation (“RTCG”)
  - Works well for tool-building
- PyOpenCL aims to realize these possibilities
  - Easy to use, safe, fast, complete

# Questions?

?

Thank you for your attention!

<http://www.cims.nyu.edu/~kloeckner/>

► image credits



# Image Credits

- Apples and Oranges: Mike Johnson - TheBusyBrain.com (cc)
- Machine: flickr.com/13521837@N00 (cc)
- Clock: sxc.hu/cema
- Magnifying glass: sxc.hu/topfer
- Fortran book: Wikipedia
- Network switch: sxc.hu/martwork
- Boost logo: The Boost C++ project
- Adding Machine: flickr.com/thomashawk (cc)
- Floppy disk: flickr.com/ethanhein (cc)
- Carrot: OpenClipart.org