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

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

PASI: The Challenge of Massive Parallelism
Lecture 3 · January 7, 2011
Outline

1. Leftovers
2. Code writes Code
3. Case Study: Generic OpenCL Reduction
4. Reasoning about Generated Code
5. Automatic GPU Programming
Outline

1. Leftovers
   - OpenCL implementations

2. Code writes Code

3. Case Study: Generic OpenCL Reduction

4. Reasoning about Generated Code

5. Automatic GPU Programming
Ahem... Show the spec!
Well...

Thank you!
Can’t say this often enough

If you are performing asynchronous transfers, . . .

. . . **beware** of Python’s big yellow garbage truck.
Kernel Attributes

```c
__kernel __attribute__((...))
void foo( __global float4 *p ) { .... }
```

- Implicit ↔ explicit SIMD
  Example:
  ```c
  __kernel __attribute__(( vec_type_hint(float4 )))
  void foo( __global float4 *p ) { .... }
  ```

Auto-vectorize assuming `float4` as the basic computation width.

- Enforcing work group sizes
  ```c
  __attribute__(( reqd_work_group_size(X, Y, Z)))
  ```
Outline

1. Leftovers
   - OpenCL implementations

2. Code writes Code

3. Case Study: Generic OpenCL Reduction

4. Reasoning about Generated Code

5. Automatic GPU Programming
The Nvidia CL implementation

Targets only GPUs

Notes:

- Nearly identical to CUDA
  - No native C-level JIT in CUDA (→ PyCUDA)
- Page-locked memory:
  - Use CL_MEM_ALLOC_HOST_PTR.
  - Careful: double meaning
  - Need page-locked memory for genuinely overlapped transfers.
- No linear memory texturing
- CUDA device emulation mode deprecated → Use AMD CPU CL (faster, too!)
The Apple CL implementation

Targets CPUs and GPUs

General notes:

- Different header name
  OpenCL/cl.h instead of CL/cl.h
  Use -framework OpenCL for C access.

- Beware of imperfect compiler cache implementation
  (ignores include files)

CPU notes:

- One work item per processor

GPU similar to hardware vendor implementation.
  (New: Intel w/ Sandy Bridge)
The AMD CL implementation

Targets CPUs and GPUs (from both AMD and Nvidia)

GPU notes:
  - Wide SIMD groups (64)
  - Native 4/5-wide vectors
    - But: very flop-heavy machine, may ignore vectors for memory-bound workloads
  - $\rightarrow$ Both implicit and explicit SIMD

CPU notes:
  - Many work items per processor (emulated)

General:
  - cl_amd_printf
Outline

1. Leftovers

2. Code writes Code
   - The Idea
   - RTCG in Action
   - How can I do it?

3. Case Study: Generic OpenCL Reduction

4. Reasoning about Generated Code

5. Automatic GPU Programming
Outline

1. Leftovers

2. Code writes Code
   - The Idea
   - RTCG in Action
   - How can I do it?

3. Case Study: Generic OpenCL Reduction

4. Reasoning about Generated Code

5. Automatic GPU Programming
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
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)
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

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)
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

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)
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)
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

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)
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 Leftovers

2 Code writes Code
   ■ The Idea
   ■ RTCG in Action
   ■ How can I do it?

3 Case Study: Generic OpenCL Reduction

4 Reasoning about Generated Code

5 Automatic GPU Programming
pyopencl.array: Simple Linear Algebra

pyopencl.array.Array:

- Meant to look and feel just like numpy.
  - p.a.to_device(ctx, queue, numpy_array)
  - numpy_array = ary.get()
- +, -, *, /, fill, sin, arange, exp, rand, ...
- Mixed types (int32 + float32 = float64)
- print cl_array for debugging.
- Allows access to raw bits
  - Use as kernel arguments, memory maps
PyOpenCL Arrays: General Usage

Remember your first PyOpenCL program?

Abstraction is good:

```python
import numpy
import pyopencl as cl
import pyopencl.array as cl_array

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

a_gpu = cl_array.to_device(
    ctx, queue, numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

Why is code generation useful in the implementation of the array type?

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
PyOpenCL Arrays: General Usage

Remember your first PyOpenCL program?

Abstraction is good:

```python
import numpy
import pyopencl as cl
import pyopencl.array as cl_array

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

ga_gpu = cl_array.to_device(
    ctx, queue, numpy.random.randn(4,4).astype(numpy.float32))
da_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

Why is code generation useful in the implementation of the array type?
**pyopencl.elementwise**: Elementwise expressions

Avoiding extra store-fetch cycles for elementwise math:

```python
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
```

Andreas Klöckner  
GPU-Python with PyOpenCL and PyCUDA
**pyopencl.reduction**: Reduction made easy

Example: A dot product calculation

```python
from pyopencl.reduction import ReductionKernel

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

import pyopencl.clrandom as cl_rand
x = cl_rand.rand(ctx, queue, (1000*1000), dtype=numpy.float32)
y = cl_rand.rand(ctx, queue, (1000*1000), dtype=numpy.float32)

x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())
```
Outline

1. Leftovers

2. Code writes Code
   - The Idea
   - RTCG in Action
   - How can I do it?

3. Case Study: Generic OpenCL Reduction

4. Reasoning about Generated Code

5. Automatic GPU Programming

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
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;
        }
    }
""
)

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

```python
from codepy.cgen import *
from codepy.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
```

Andreas Kl"ockner

GPU-Python with PyOpenCL and PyCUDA
Outline

1. Leftovers
2. Code writes Code
3. Case Study: Generic OpenCL Reduction
4. Reasoning about Generated Code
5. Automatic GPU Programming
Reduction

\[ y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N) \]

where \( N \) is the input size.

Also known as...

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

\[ x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_6 \rightarrow y \]

Painful! Not parallelizable.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Reduction: Graph

Painful! Not parallelizable.

Andreas Klöckner  
GPU-Python with PyOpenCL and PyCUDA
Reduction: A Better Graph

x0 \rightarrow x1 \rightarrow x2 \rightarrow x3 \rightarrow x4 \rightarrow x5 \rightarrow x6 \rightarrow x7

\rightarrow y

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
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.)
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.
Kernel V1

```c
__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];
}
```

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Interleaved Addressing

Values (shared memory): 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

- Step 1: Stride 1
  - Thread IDs: 0 2 4 6 8 10 12 14
  - Values: 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2

- Step 2: Stride 2
  - Thread IDs: 0 4 8 12
  - Values: 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2

- Step 3: Stride 4
  - Thread IDs: 0 8
  - Values: 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

- Step 4: Stride 8
  - Thread IDs: 0
  - Values: 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

With material by M. Harris (Nvidia Corp.)
## Interleaved Addressing

<table>
<thead>
<tr>
<th>Values (shared memory)</th>
<th>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong> Thread IDs</td>
<td>0 2 4 6 8 10 12 14</td>
</tr>
<tr>
<td><strong>Step 2</strong> Thread IDs</td>
<td>0 4 8 12</td>
</tr>
<tr>
<td><strong>Step 3</strong> Thread IDs</td>
<td>0 8</td>
</tr>
<tr>
<td><strong>Step 4</strong> Thread IDs</td>
<td>0</td>
</tr>
<tr>
<td><strong>Values</strong></td>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
</tr>
</tbody>
</table>

**Issue:** Slow modulo, Divergence

With material by M. Harris (Nvidia Corp.)
__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

Values (shared memory):

<table>
<thead>
<tr>
<th>Values (shared memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

Step 1
Stride 8

Thread IDs:

- Thread IDs: 0 1 2 3 4 5 6 7
- Values: 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 2
Stride 4

Thread IDs:

- Thread IDs: 0 1 2 3
- Values: 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3
Stride 2

Thread IDs:

- Thread IDs: 0 1
- Values: 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4
Stride 1

Thread IDs:

- Thread IDs: 0
- Values: 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

With material by M. Harris (Nvidia Corp.)

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Better! But still not “efficient”.

Only half of all work items after first round, then a quarter, ...
Thinking about Parallel Complexity

Distinguish:

- **Time on** $T$ processors: $T_P$
- **Step Complexity/Span** $T_\infty$: 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** $T_1/T_\infty$: 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** $T_\infty$: 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** $T_1 / T_\infty$: average amount of work along span
  - $P > T_1 / T_\infty$ doesn’t make sense.

Algorithm-specific!

- How parallel is our current version?
- Can we improve it?
Kernel V3 Part 1

```c
__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);
}
```
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) odata[get_group_id(0)] = ldata [0];
Performance Comparison

With material by M. Harris (Nvidia Corp.)

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Generic CL Reduction: Preparation

```c
#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

```c
__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;
    }
```

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
CL Reduction: Explicitly Synchronized Part

```python
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
```

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
CL Reduction: Implicitly Synchronized Part

```c
% 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. Leftovers
2. Code writes Code
3. Case Study: Generic OpenCL Reduction
4. Reasoning about Generated Code
5. Automatic GPU Programming
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

Group 1

Group 2

Group 3

GAOS: Adrian Tate, Cray, Inc.
Grouped Orthogonal Search

Choose group

GAOS: Adrian Tate, Cray, Inc.
Grouped Orthogonal Search

Map admissible options

Group 1
Group 2
Group 3

GAOS: Adrian Tate, Cray, Inc.
Grouped Orthogonal Search

Group-wide exhaustive search

GAOS: Adrian Tate, Cray, Inc.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Grouped Orthogonal Search

Start over with best result → pick new group...
Using the Nvidia profiler in-process

```python
# enable profiler
import os
os.environ['COMPUTE_PROFILE'] = '1'
with open('/tmp/myprg-prof-config', 'w') as prof_config:
    prof_config.write('
'.join(events))
os.environ['COMPUTE_PROFILE_CONFIG'] = '/tmp/myprg-prof-config'

# obtain timing data
prof_f = open('opencl-profile_0.log', 'r')
gain_count = 0

while gain_count < 2:
    # run kernel here
    prof_output = prof_f.readlines()
    if prof_output:
        print 'gained %d lines' % len(prof_output)
        gain_count += 1
    if gain_count == 2:
        print ''.join(l for l in prof_output[1:-1]
                      if kernel_name in l)
```

Sample output:
```
```

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Using the Nvidia profiler in-process

```python
# enable profiler
import os
os.environ["COMPUTE_PROFILE"] = "1"

with open("/tmp/myprg−prof−config", "w") as prof_config:
    prof_config.write("\n".join(events))

os.environ["COMPUTE_PROFILE_CONFIG"] = "/tmp/myprg−prof−config"

# obtain timing data
prof_f = open("opencl_profile_0.log", "r")
gain_count = 0

while gain_count < 2:
    # run
    prof_output = prof_f.readlines()
    if prof_output:
        print("gained %d lines" % len(prof_output))
        gain_count += 1
    if gain_count == 2:
        print("".join(l for l in prof_output[1:] if kernel name in l))

Sample output:

```
```
```
Nvidia GPU Profiler: Events

\textbf{gld}\_request: Number of executed global load instructions per warp in a SM

\textbf{gst}\_request: Number of executed global store instructions per warp in a SM

\textbf{divergent}\_branch: Number of unique branches that diverge

\textbf{instructions}: Instructions executed

\textbf{warp}\_serialized: Number of SIMD groups that serialize on address conflicts to local memory

And many more: see \texttt{(root of CUDA toolkit)/(doc/Compute_Profiler_VERSION.txt)}

(Careful: CUDA terminology)
Outline

1. Leftovers
2. Code writes Code
3. Case Study: Generic OpenCL Reduction
4. Reasoning about Generated Code
5. Automatic GPU Programming
Automating GPU Programming

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

- Obvious idea: Let the computer do it.
- One way: Smart compilers
Automating GPU Programming

CPU 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
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
Empirical GPU loop optimization:

\[
\begin{align*}
a, b, c, i, j, k &= [\text{var}(s) \text{ for } s \text{ in } "abcijk"] \\
n &= 500 \\
k &= \text{make_loop_kernel}([ \\
    \text{LoopDimension}("i", n), \\
    \text{LoopDimension}("j", n), \\
    \text{LoopDimension}("k", n), \\
    ], \[
    (c[i+n*j], a[i+n*k]*b[k+n*j]) \\
    ])
\end{align*}
\]

gen_kwargs = {
    "min_threads": 128,
    "min_blocks": 32,
}

→ Ideal case: Finds 160 GF/s kernel without human intervention.
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using “loopy” model
    (i.e. indices decompose into “output” and “reduction”)
  - Enough for DG, LA, FD, ...
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using “loopy” model (i.e. indices decompose into “output” and “reduction”)
  - Enough for DG, LA, FD, . . .

- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels
Questions?
Image Credits

- Garbage Truck: sxc.hu/mzacha
- Nvidia logo: Nvidia Corporation
- Apple logo: Apple Corporation
- AMD logo: AMD Corporation
- Apples and Oranges: Mike Johnson - TheBusyBrain.com
- Machine: flickr.com/13521837@N00
- Adding Machine: flickr.com/thomashawk
- Clock: sxc.hu/cema
- Magnifying glass: sxc.hu/topfer