GPU Programming Languages

Vilhelm Sjöberg

April 5, 2010
What’s wrong with CUDA?

- Low-level – programs structured by kernels, not data flow.
- Limited metaprogramming features
- It’s just not Haskell! (Or Python, C#, Scala, ...)

Today I Will Talk About


- **GPU Kernels as Data-Parallel Array Computations in Haskell.** Sean Lee, Manuel M. T. Chakravarty, Vinod Grover, and Gabriele Keller. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPAHM 2009)*

Papers get more and more vaporous.
Wouldn’t it be nice if we could access CUDA from Python?

- Integrates with NumPy
- Compile CUDA kernels from Python
- GPUArray class for convenient memory allocation/deallocation, collective operations (map, scan)

3500 lines of Python, 4500 lines of C++.
Wouldn’t it be nice if we could access CUDA from Python?

- Integrates with NumPy
- Compile CUDA kernels from Python
- GPUArray class for convenient memory allocation/deallocation, collective operations (map, scan)

3500 lines of Python, 4500 lines of C++.
import pycuda.autoinit
from pycuda.compiler import SourceModule
mod = SourceModule(''
__global__ void multiply_them(float *dest,
    float *a,
    float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
''')
import pycuda.driver as drv
import numpy

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1))

print dest
Compiling CUDA kernels from Python

When calling \texttt{pycuda.compiler.SourceModule} it

- Writes the kernel source to /tmp.
- Calls \texttt{nvcc} on it to get a “cubin” file.
- Loads the compiled object code \texttt{(cuModuleLoadDataEx)}.
- Remembers the MD5 hash of the source for future use.
from jinja2 import Template

tpl = Template(""
__global__ void add(
    {{ type_name }} * tgt,
    {{ type_name }} * op1,
    {{ type_name }} * op2)
{
    int idx = threadIdx.x +
        {{ block_size }} * {{thread_strides}}
        * blockIdx.x;
    {% for i in range(thread_strides) %}
        {% set offset = i * block_size %}
        tgt[idx + offset ] =
            op1[idx + offset ]
            + op2[idx + offset ];
    {% endfor %}
}"")
from codepy.cgen import FunctionBody, Typedef, FunctionDeclaration, Typedef, POD, Value, Pointer, Module, Block, Initializer, Assign
from codepy.cgen.cuda import CudaGlobal
mod = Module([FunctionBody(CudaGlobal(FunctionDeclaration(Value("void", "add"),
arg_decls=[Pointer(POD(dtype, name))
for name in ["tgt", "op1", "op2"])),
Block([Initializer(POD(numpy.int32, "idx"),
"threadIdx.x + %d*blockIdx.x"
% (thread_block_size*block_size)),
...
Metaprogramming – Motivation

*Nodal Discontinuous Galerkin Methods on Graphics Processors* (Klöckner et al.)
Optimize first memory layout, then loop sizes and use of shared memory.
Convenient array-level operations. E.g.

\[ 5 \times a \]

scales each element.

Also: \textbf{\texttt{sin}}, \textbf{\texttt{cos}}, \textbf{\texttt{log}}, \textbf{\texttt{exp}}, \ldots, and \textbf{\texttt{sum}}, \textbf{\texttt{dot}}, \textbf{\texttt{max}}, \ldots
import pycuda.gpudarray as gpudarray
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand

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

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

c_gpu = gpudarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)
PyCUDA summary

- Complete: exposes all CUDA functionality
- Convenient: no need to worry about setup, memory allocation/freeing, copying
- Low-level: still structured in terms of kernel calls
- ...but provides some support for map/scan.
Observation: organization into CUDA kernels does not match how we would ideally organize into functions.

- Need separate kernels to implement grid-level barriers.
- Collective operations need multiple kernel launches.
- Need to do memory management outside kernels.
- Sometimes put unrelated functionality into a kernel for performance.
Bulk-Synchronous Programming

- Programming model dates back to 1990
- Still Same-Program Multiple-Data
- Computation is organized into *supersteps*, separated by spawn, barrier, sort, scan, reduce, fork,...
spawn and barrier example

spawn n

//integrate
  id = thread.rank
  p = p0[id];
  v = v0[id];
  F = make_float3(0, g, 0)
  v += F*dt
  p += v*dt
  p0[id] = p
  v1[id] = v

barrier

//collide
...

//integrate
barrier implementation

- Split into two kernels, before and after barrier.
- At the end of the “before” kernel, save all live local variables to global memory.
- Launch the two kernels after each other.
// collide
...
gash=compute_hash(gridpos(p,0,csz),gsz)
thread.sortby(ghash)
id0=id
id=thread.rank
$shuf[id0]=id
...

sort implementation

- Write out all thread context to global memory
- Sort. (Using a merge-sort variant implemented using shared memory)
• Similarly, after a call to `scan(func, x)`, the variable `x` will contain the cumulative result of scanning using the function `func`.
• Implemented using local scan; global scan; local result adding.
• The local steps are congealed with surrounding supersteps.
thread.fork(3*e+v)
// The old thread.rank is now thread.oldrank

Implementation:
- Use a collective scan to calculate the new rank of each thread’s first child.
- Then fill in oldrank
Implementation

- A real compiler: Source $\rightarrow$ SSA $\rightarrow$ Optimized SSA [liveness etc] $\rightarrow$ PTX
- Optimize memory-usage through a graph-flow analysis
- Binary-only distribution ("research-quality" software)
Examples

- GPU Ray Tracer (using “stackless kd-tree traversal”)
- Particle based fluid simulation
- GPU X3D Parser
- GPU Adaptive Tessellation
- Large example: the *RenderAnts* Reyes renderer (by the same authors).

Small examples same speed or faster than handwritten CUDA.
BSGP: Summary

- Higher-level, abstracts away from kernels
- No attempt to expose full CUDA API or give detailed control over generated code.
- No control over shared memory usage?
- Convenient collective operations from within the threads.
The collective operations, `scan`, `sort`, ... look a lot like functional programming.

Wouldn’t it be nice to program them in a functional language?

GpuGen – A Haskell library for array manipulation on the GPU. (PyCUDA + lots of types everywhere!)
GpuGen Example

saxpy_CPU :: Float -> [Float] -> [Float] -> [Float]
saxpy_CPU alpha xs ys =
  zipWith (\ x y -> alpha * x + y) xs ys

saxpy_GPU :: Exp Float
  -> Array DIM1 Float
  -> Array DIM1 Float
  -> Array DIM1 Float
saxpy_GPU alpha xs ys
  = GPU.run $ do
    xs' <- use xs
    ys' <- use ys
    zipWith_GPU (\ x y -> alpha * x + y) xs' ys'
Types for array indices

-- Typeclass for array indices
class Ix ix where
  dim :: ix -> Int
  size :: ix -> Int
  index :: ix -> ix -> Int --shape, index
...

instance Ix ()
instance Ix (Int)
instance Ix (Int, Int)

-- Shorthand for common shape types
type DIM0 = ()
type DIM1 = (Int)
type DIM2 = (Int, Int)
Types for Arrays, and Scalar datatypes

-- In Java, this would be
--
--   class Array<Dim, E> {
--       ...

    data Array dim e where
    ...  

    class IsScalar a where
        -- no methods.

    instance IsScalar Float
    instance IsScalar Int
    ...

**Exp – type for scalar sublanguage**

data Exp t where

... 

(+) :: IsScalar t => Exp t -> Exp t -> Exp t 

(*) :: IsScalar t => Exp t -> Exp t -> Exp t 

(?) :: IsScalar t => 
    Exp Bool -> (Exp t, Exp t) -> Exp t 

(==) :: IsScalar t => 
    Exp t -> Exp t -> Exp Bool 

For example:

fun foo :: Exp Float -> Exp Float -> Exp Float 

foo x y = x + y
data GPU t where

...  

map :: (Ix dim, Elem a, Elem b) =>
        (Exp a -> Exp b) ->
        GPU (Array dim a) -> GPU (Array dim b)

zipWith :: (Ix dim, Elem a, Elem b, Elem c) =>
           (Exp a -> Exp b -> Exp c) ->
           GPU (Array dim a) -> GPU (Array dim b) ->
           GPU (Array dim c)

scan :: Elem a =>
       (Exp a -> Exp a -> Exp a) -> Exp a
       -> GPU (Vector a)
       -> (GPU (Vector a), GPU (Scalar a))
GpuGen – Implementation

- The type $\text{Exp}$ is the type of syntax trees!
- To reify a function, invent a fresh name and call it on that.
- Once we have the syntax tree, proceed like PyCUDA.
GpuGen – Summary

- Does not attempt to cover the full CUDA API. No detailed control over output.
- High-level language for collective operations.
- About 50% as fast as handwritten CUDA.
- Complete vapourware. (But starting to materialize in a generalized form as Data.Array.Accelerate)
Questions?