Copperhead: Data-Parallel Python

Bryan Catanzaro, UC Berkeley  
Michael Garland, NVIDIA Research  

(catanzar@cs.berkeley.edu) 
(mgarland@nvidia.com) 

Armando Fox, Yunsup Lee, Mark Murphy, 
Kurt Keutzer, UC Berkeley
Warning: Evolving Project

- Copperhead is still in embryo
- We can compile and execute a few small programs
- Lots more work to be done in both language definition and code generation
- Currently, Copperhead only targets CUDA GPUs
  - Multicore x86, Larrabee, OpenCL in future

- Feedback is encouraged
  - Please interrupt me with questions!
Why parallelism?

- Brick wall
- ILP Wall
- Memory Wall
- Power Wall

---

Intel Clock Speed

Data from Mark Horowitz & Wikipedia
What kind of parallelism?

- Single socket parallelism
  - Shared memory
- Multiple threads per core
- SIMD parallelism per thread
- On chip memory subsystem
  - Hierarchical caches or local stores
Motivation

- We are looking for ways to raise the abstraction level of parallel programming
- Observation: weak performance guarantees acceptable for (serial) productive languages
  - Maybe my code isn’t as fast as C, but it’s fast enough, and in a few years it will be faster running on a newer processor
- We want similar guarantees for parallel programming
- Not trying for absolute performance, but scalability
- Python has a lot of momentum in scientific computing
Copperhead: Data Parallel Python

- Consider this intrinsically parallel procedure
  ```python
def saxpy(a, x, y):
    return map(lambda xi, yi: a*xi + yi, x, y)
    ... or for the lambda averse ...

def saxpy(a, x, y):
    return [a*xi + yi for xi, yi in zip(x, y)]
```

- This procedure is both
  - completely valid Python code
  - compilable to data parallel languages like CUDA or OpenCL
» from copperhead import *

» @cu
   def saxpy(a, x, y):
       return [a*xi + yi for x, y in zip(x, y)]

» x = [1.0, 1.0, 1.0, 1.0]  # can use NumPy or
» y = [0.0, 1.0, 2.0, 3.0]  # CuArrays, too

» parallelResult = saxpy(2.0, x, y)

» serialResult = saxpy(2.0, x, y, cuEntry=False)
A subset of Python

- Copperhead is a subset of Python, designed for data parallelism
- Why Python?
  - Extant, well accepted high level scripting language
  - Productive programming
    - Interactive algorithm prototyping
  - Flexible enough syntax to allow executing Copperhead programs in Python natively
- Other sources of inspiration:
  - Nesl, Data Parallel Haskell, APL, MATLAB
Runtime: Selective, Embedded, Just in Time Specialization

- Copperhead procedures live in normal Python modules
  - delineated by @cu function decorator

- Runtime intercepts & specializes function when called
  - infers type, shape, etc. using runtime data
  - generates C code
  - moves data as needed
  - runs in parallel
PyCUDA

- The Copperhead runtime relies on the PyCUDA project for CUDA bindings
  - Memory allocation, data movement
  - Compilation
  - Execution
- PyCUDA automatically garbage collects allocated memory on the GPU
- Provides binary cache of .cubin, so nvcc can be avoided if you’re reexecuting the program
  - Determination based on md5 sum of CUDA program
Data types

- The runtime operates on CuArrays, which extend Numpy.ndarray
  - Allows Copperhead programs to interact with existing libraries for computation and visualization
- Runtime also accepts Numpy arrays, Python sequences
- CuArrays are associated with a “place”
  - Places have local storage (in Python interpreter)
  - and remote storage (out of interpreter)
    - GPU, Multicore x86, ...
- Data moved lazily between places
Runtime Static Typing

- Standard Hindley-Milner style type inference
  ```python
  def plus1(x): return x+1
  plus1 :: int -> int
  ```

- Supporting parametric polymorphism
  ```python
  def saxpy(a, x, y):
      return map(lambda xi,yi: a*xi+yi, x, y)
  saxpy :: (a, [a], [a]) -> [a]
  ```

- And rejecting ill-typed procedures
  ```python
  def ill_typed(p):
      return 1 if p else True
  ```
Side-effects are forbidden

- An acceptable Copperhead procedure:
  ```python
def saxpy(a, x, y):
    return map(lambda xi, yi: a*xi + yi, x, y)
```

- Valid Python but forbidden in Copperhead:
  ```python
def saxpy(a, x, y):
    for i in indices(y):
      y[i] = a*x[i] + y[i]
    return y
```
  Parallelization requires knowledge of indices(y)
Data-driven parallelism

- Parallelism arises from map

- Or primitive procedures built from it
  - reduce
  - scan
  - sort
  - ...

Data-driven synchronization

- Joining previously independent sequences
  » \texttt{join(map(sort, split(A))}

- Data access patterns that can’t be statically localized
  » \texttt{B = gather(map(f, A), indices)}
  » \texttt{total = sum(B)}
Functions

- Support for higher-order functions
  - Currently limited to higher-order functions which can be statically flattened to avoid function pointers

```python
@cu
def reduce(f, A):
    tiles = split_by_size(A, tilesize)
    partials = map(lambda tile: reduceP(f, tile), tiles)
    return reduceP(f, partials)
```
Assume we had a reduction primitive reduceP
  - Details of reduceP might be complicated
  - Assume it does a parallel reduction on a small enough array that access to that array is PRAM-style (hence reduceP)

```python
@cu
def reduce(f, A):
    tiles = splitP(A)
    partials = [reduceP(f, tile) for tile in tiles]
    return reduceP(f, partials)
```
Auto-sequentialization

- Compiler chooses sequential vs. parallel

  \[\text{total} = \text{reduce}(\text{map}(\text{reduce}, \text{split}(A, 128)))\]

- Deeply nested primitives are sequentialized

- Note that there is no "correct" answer here
  - depends on input size
  - depends on architecture
  - Autotuning useful
Kernel fission & fusion

- Compiler infers & schedules “phase” boundaries
  - Points where synchronization is required
  - Only possible because of lack of side effects

B = sum(map(f,A))
D = sum(map(g,C))
More on runtime

Python code with @cu decorator

Grabs AST through `inspect` & `ast` transforms to Cu AST

Desugars code. Remains valid
Copperhead

Inference, adds state. Remains valid
Python

Generates parallel C & Python code for coordinating C.

Python function
Copperhead is an open source project (Apache 2.0)
However, it’s not yet generally available
  ▪ Lots of changes going on
  ▪ Can compile simple programs:
    ▪ Compositions of maps and reduces
  ▪ Speed looks good
    ▪ Comparable with handwritten CUDA on small problems
    ▪ But compiler still too immature to make strong performance claims