Copperhead: Data Parallel Python

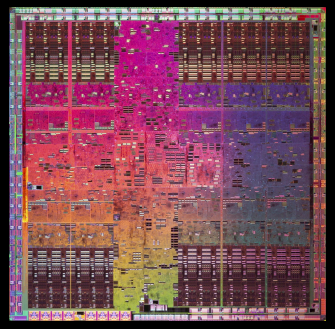
Bryan Catanzaro, NVIDIA Research

Motivation

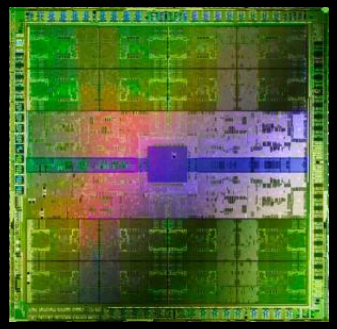
- How can Python programmers use parallelism?

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
5323 bcatanza  20   0  20.2g   24m  9108 R   794   0.4   0:41.76 python
```

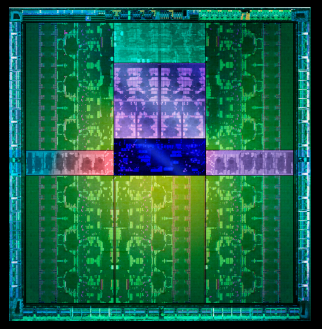
- Parallel architectures evolve quickly
  - Performance portability requires a higher-level, more declarative style



2008

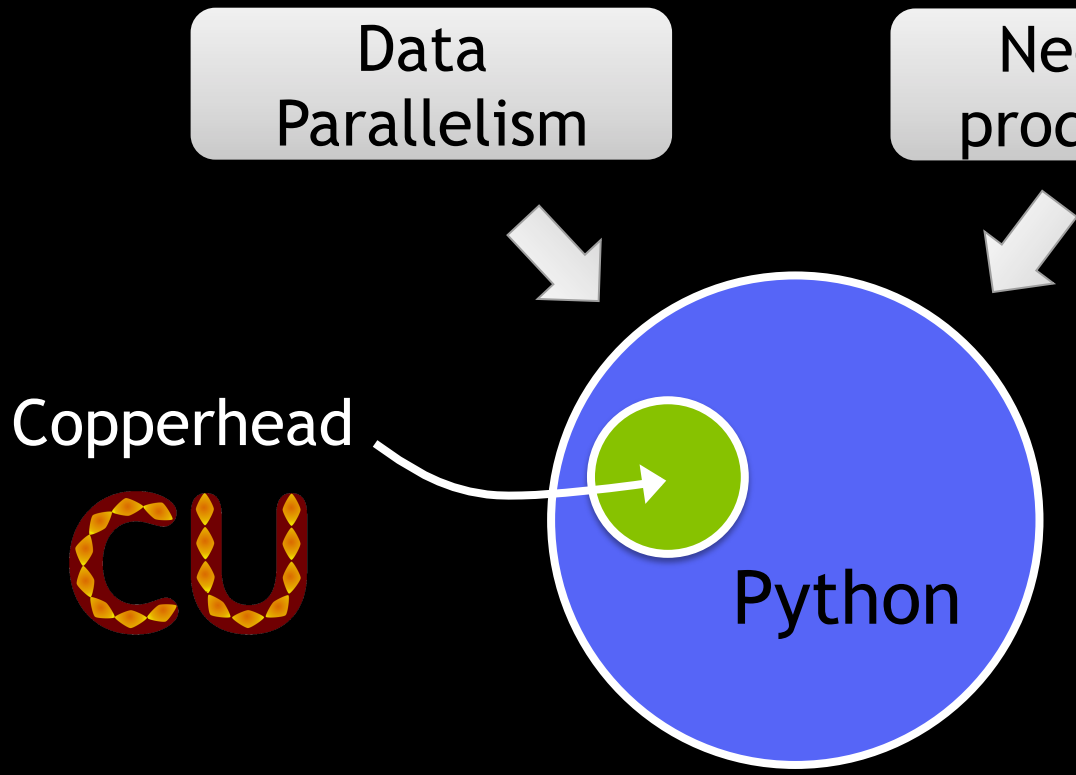


2010



2012

Copperhead



- Goal: Efficiency and Productivity
- Note: Copperhead is a research project, not a product.

Copperhead code is just Python code.
No C-isms, no annotations.

Hello world of data parallelism

- Consider this intrinsically parallel procedure

```
def axpy(a, x, y):
  return map(lambda xi,yi: a*xi + yi, x, y)
```

… or for the lambda averse …

```
def axpy(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 substrates (CUDA, OpenCL, OpenMP, etc.)

Embedded Subset of Python

- Using standard Python constructs and syntax

```
@cu
def example(x, y):
  a = map(f, x, y)
  return reduce(g, a, 0)
```

- Clearly delineated via @cu function decorator
  - Entry point model

```
@cu
def copperhead_function():
  return 0

def python_function():
  return 0
```

A (Very) Strict Subset

- This is not Cython for GPUs
  - Batteries not included
  - We're aiming for high performance
- Restricted syntax and data structures
  - Homogeneous arrays
  - No classes, metaclasses
- Strong typing
  - Needed for performance

Runtime Static Typing

- Standard Hindley-Milner style type inference, no annotations necessary

```
» def plus1(x): return x+1
plus1 :: Int64 -> Int64
```

- Supporting parametric polymorphism

```
» 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

```
» def ill_typed(p):
      return 1 if p else True
```

Side effects

- Side effects are forbidden in Copperhead code

```
@cu
def axpy(a, x, y):
  def triad(xi, yi):
    return a * xi + yi
  return map(triad, x, y)
```

Valid

```
@cu
def axpy(a, x, y):
  for i in indices(y):
    y[i] = a * x[i] + y[i]
  return y
```

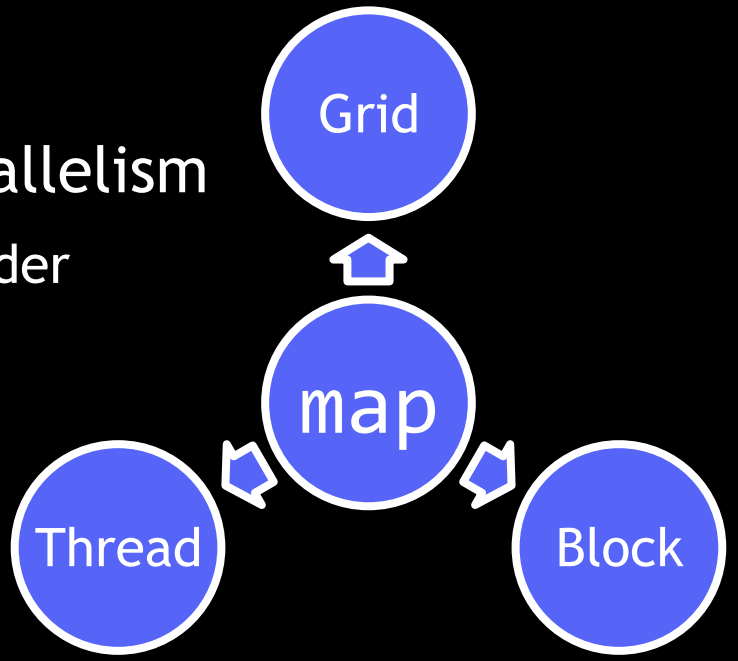
Invalid

- Side effects are allowed in (sequential) Python code

```
a = cuarray([1,2,3,4,5])
a.update([1, 3], [-4, -2])
print a
»  [1, -4, 3, -2, 5]
```

Parallel Semantics

- Python sequentially orders computation
  - Inside to outside, left to right of expressions
  - Top to bottom of statements
  - Left to right `for x in iterable:`
- Copperhead relaxes ordering for parallelism
  - Expressions may be evaluated out of order
  - Data dependencies observed
- Primitives like `map` may be executed in any order
  - Including Python's order



Support for Heterogeneity: Places

- Programmer specifies execution place

```
with places.gpu0:
  gpu_result = axpy(...)
with places.openmp:
  cpu_result = axpy(...)
```

- Currently support:
  - CUDA
  - OpenMP
  - TBB
  - Sequential C++
- At the present time, place selection happens in the Python interpreter
  - We do not yet support mixing and matching places inside Copperhead programs

Primitives

- Programs created through composition of familiar data parallel primitives such as

`map(f, x1, x2, ...)`
Applies function `f` elementwise to all input arrays `x1...xn`

`reduce(f, x, p)`
Reduces a sequence to a scalar
`f`: associative and commutative function
`x`: sequence of data to be reduced
`p`: prefix element

`filter(f, x)`
Compacts a sequence
`f`: predicate function
`x`: sequence of data to be compacted

(Python builtins)

`scan(f, x)`
Performs an inclusive prefix-sum scan
`f`: associative and commutative function
`x`: sequence of data to be scanned

`gather(x, i)`
Gathers from an input array
`x`: source array
`i`: index array

`scatter(x, i, d)`
Scatters into a copy of `d`
`x`: data to be scattered
`i`: indices where data is to be scattered
`d`: destination

Primitives (continued)

`sort(f, x)`
Sorts a sequence
`f`: strict weak ordering comparator
`x`: sequence of data to be sorted

`rotate(x, n)`
Rotates a sequence
`x`: sequence of data to be rotated
`n`: number of elements to rotate (+/-)

`shift(x, n, i)`
Shifts a sequence, fills extra elements with `i`
`x`: sequence of data to be rotated
`n`: number of elements to rotate (+/-)
`i`: element to shift in

`zip(x0, x1, ...)`
Combines several sequences into a sequence of tuples

`unzip(x)`
Distributes a sequence of tuples into a tuple of sequences

`indices(x)`
Produces a sequence which enumerates the elements of `x`

`replicate(x, n)`
Produces a sequence which replicates element `x` by `n` times

Code Generation

- Generates C++ code based on Thrust
- Kernel Fusion

```
a = map(foo, b)
c = reduce(a)
```

a will never be physically generated in memory

- Array of Structure to Structure of Arrays

```
a = [1,2,3]
b = [4,5,6]
c = zip(a, b)
```

c is a Sequence of Tuples
Uses Structure of Arrays memory layout internally

- Virtual sequences
  - Use no memory

```
a = replicate((1, 1, 0), 10)
b = range(100)
```

Think of these as generators

Types

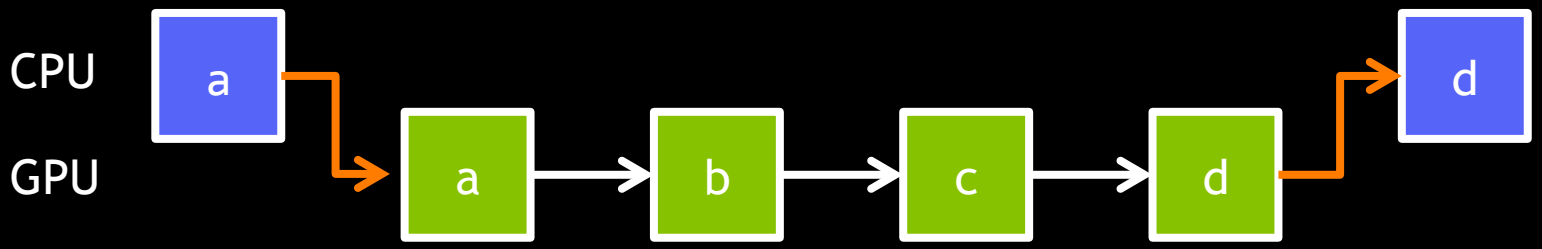
- Scalar types provided by numpy

```
float32     float64     int32       int64       bool
```

- Tuples are heterogeneously typed
  - Can be nested
  - No subscripting
- 1-D, flat sequences
  - May contain scalar types or tuples
  - No nesting at this time
  - Currently, you must index N-D arrays manually

Runtime Data Management

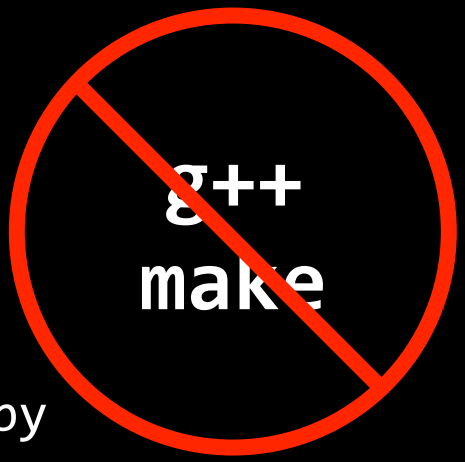
- The Copperhead runtime manages all data
- Data lazily transferred to and from memory spaces



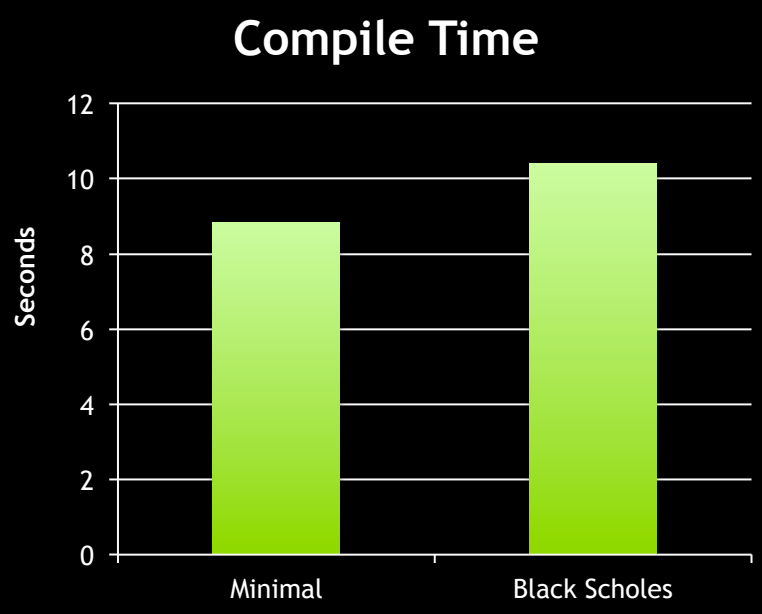

```
a = …
b = foo(a)
c = foo(b)
d = foo(c)
print(d)
```

- Memory is garbage collected via Python's garbage collector
- Data interoperates with `numpy`, `matplotlib`, etc.

Runtime code generation

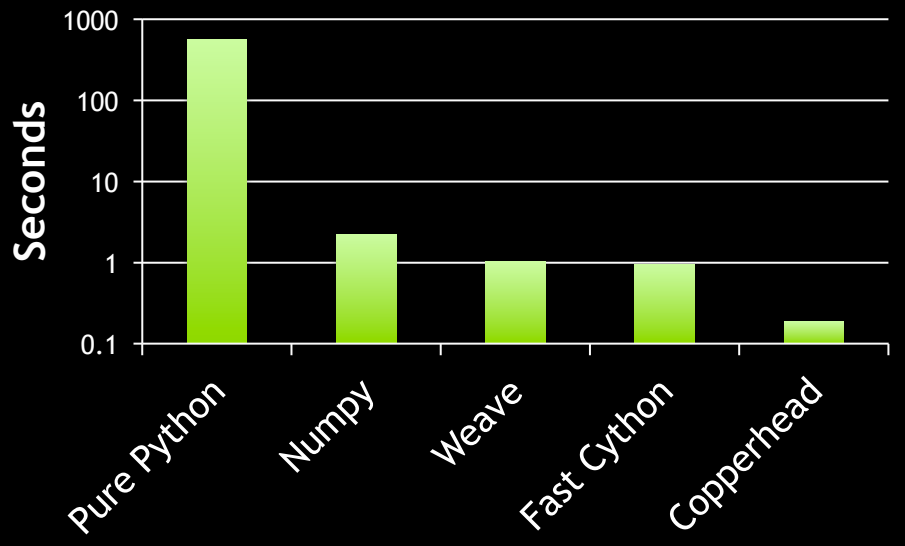


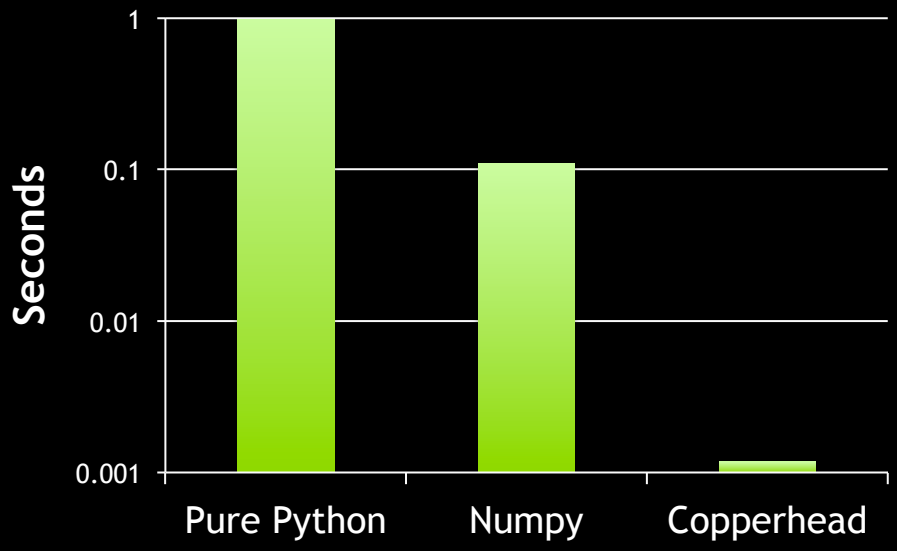
- Runtime code generation
- Copperhead compiler produces C++ code
- C++ code is compiled to a dynamic library using `codepy`
- Compilation artifacts persistently stored in `__pycache__`
- Runtime overhead: ~10-100 µsec (from Python, per fn call)



Some results (GTX480)

- Solving Laplace's equation (from Travis Oliphant's blog)



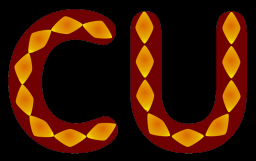


- Sorting array of 1M float32 elements

Ongoing work

- Bugs/Stability/Performance tuning
  - More data parallel primitives
- Nested data parallelism
  - Essential for efficiency & expressivity
- Dynamic heterogeneity
  - Use CPU and GPU together in the same Copperhead program

Conclusion



- Copperhead is a data parallel dialect of Python
- Runtime compiler generates code for CUDA, OpenMP and TBB
- Ongoing open source project, Apache 2.0, available at

http://copperhead.github.com

- Questions?

bcatanzaro@nvidia.com