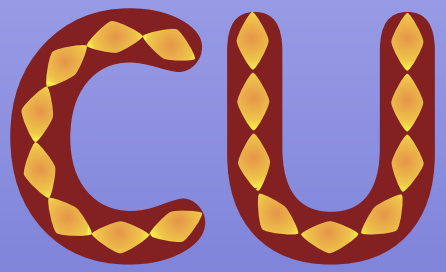
Copperhead

Data Parallel Python

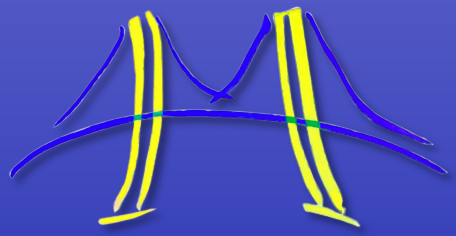


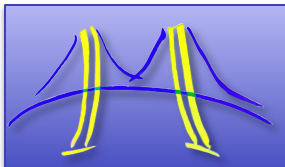
Bryan Catanzaro, Kurt Keutzer

University of California, Berkeley

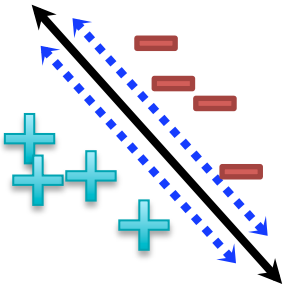
Michael Garland

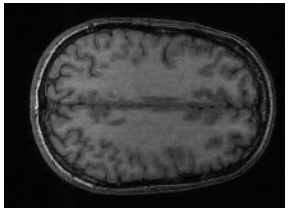
NVIDIA Research

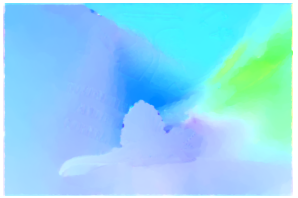




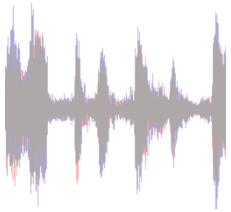
Motivation

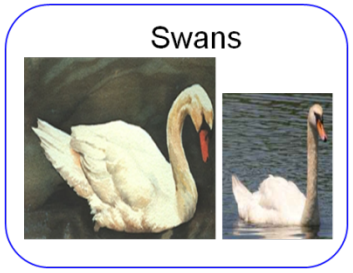




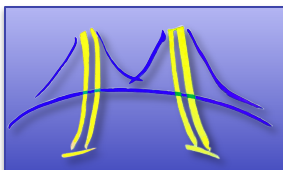


Data Parallelism

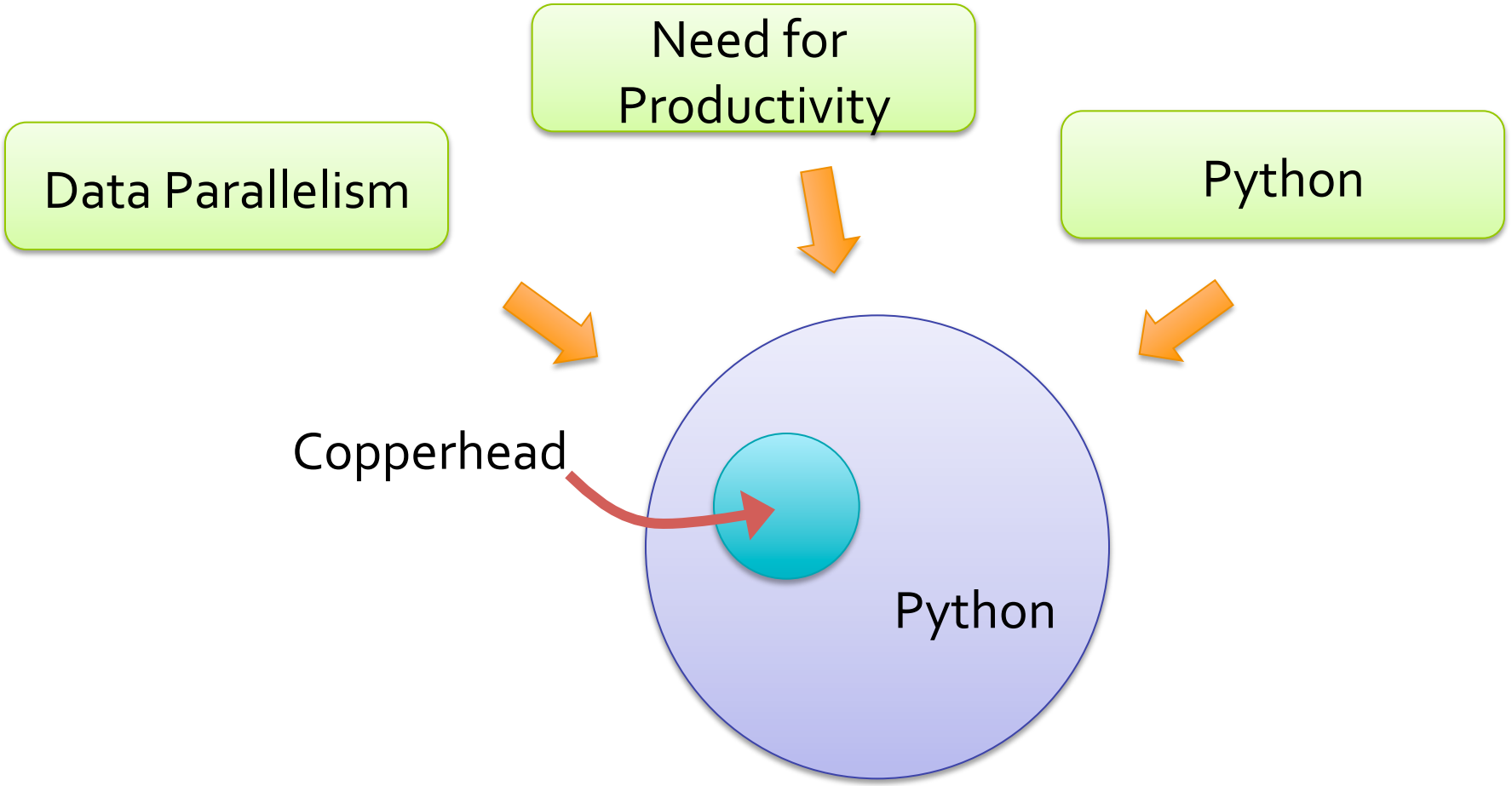


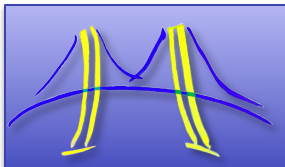


- Data Parallelism is a key parallel pattern
- How can we efficiently and productively use it?



Copperhead


Need for Productivity
Data Parallelism
Python
Copperhead
Python



Hello world of data parallelism

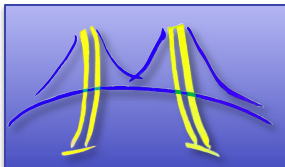
- Consider this intrinsically parallel procedure

```
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



Embedded Subset of Python

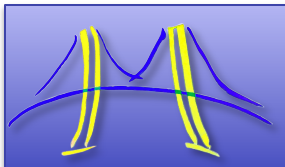
- Using standard Python constructs and syntax

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

- Clearly delineated via @cu function decorator

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

def python_function():
  return 0
```

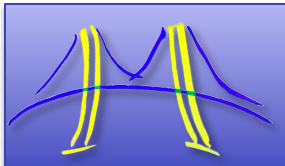


A (Very) Strict Subset

- 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



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)
```

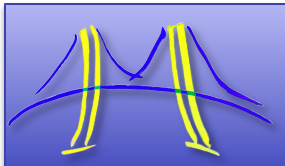
Valid

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

Invalid

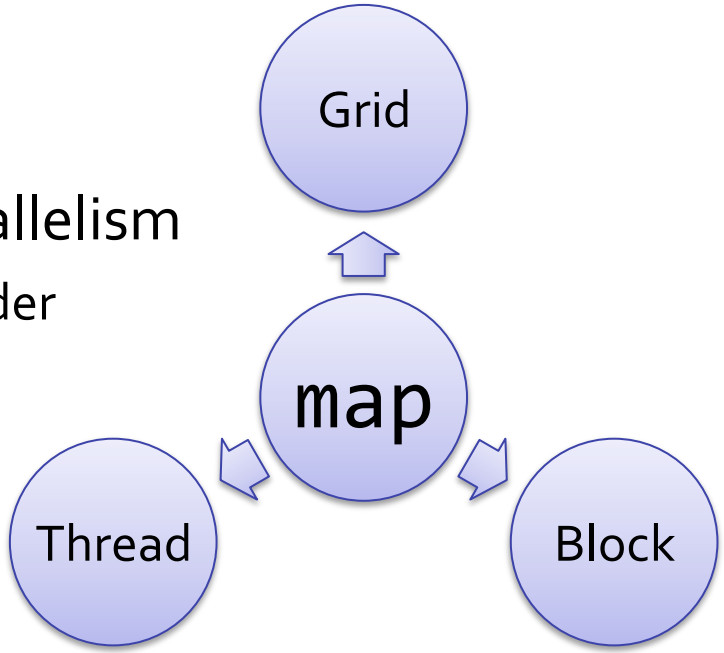
- Side effects are allowed in (sequential) Python code

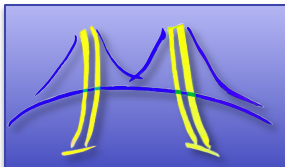
```
a = CuArray([1,2,3,4,5])
a.update([1, 3], [-4, -2])
print a
» [1, -4, 3, -2, 5]
```



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





Synchronization

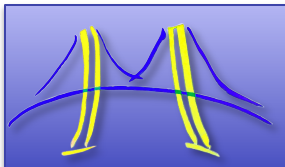
- Synchronization comes from data dependencies

- Higher productivity
  - No data races
  - No explicit barriers

- Compiler's job
  - find data dependencies
  - schedule operations efficiently

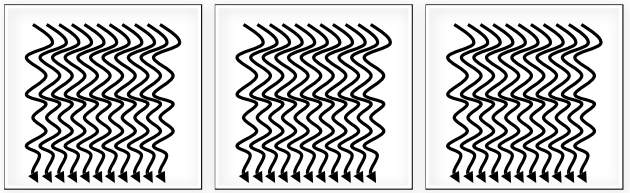
```
b = map(foo, a)
d = scatter(b, i, c)
e = map(bar, d)
```

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

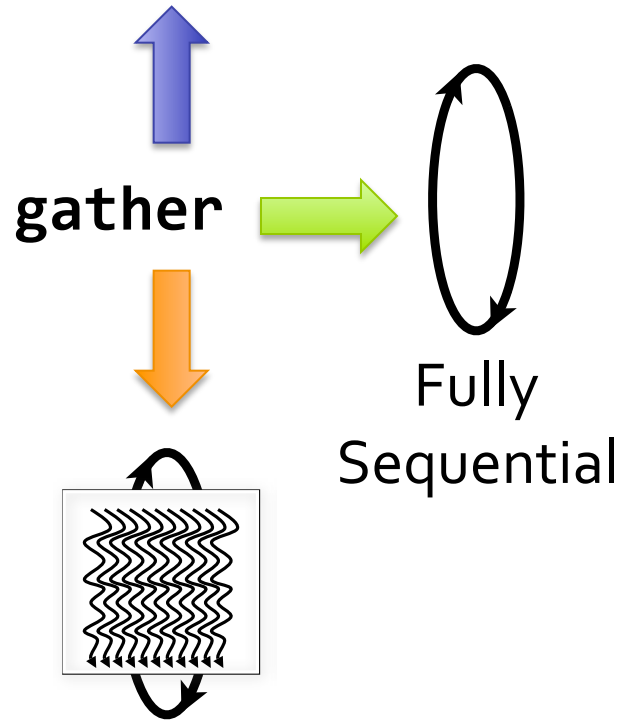


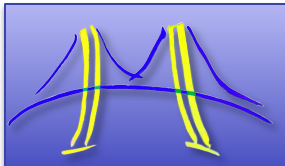
Auto-Sequentializing Compiler

- Data parallel primitives can:
  - Execute in parallel
  - Execute as a sequential loop
  - Or execute at points in between
- Copperhead code leaves this implicit
  - User can influence mapping of nested primitives via compiler options
  - Ultimately, an autotuning problem



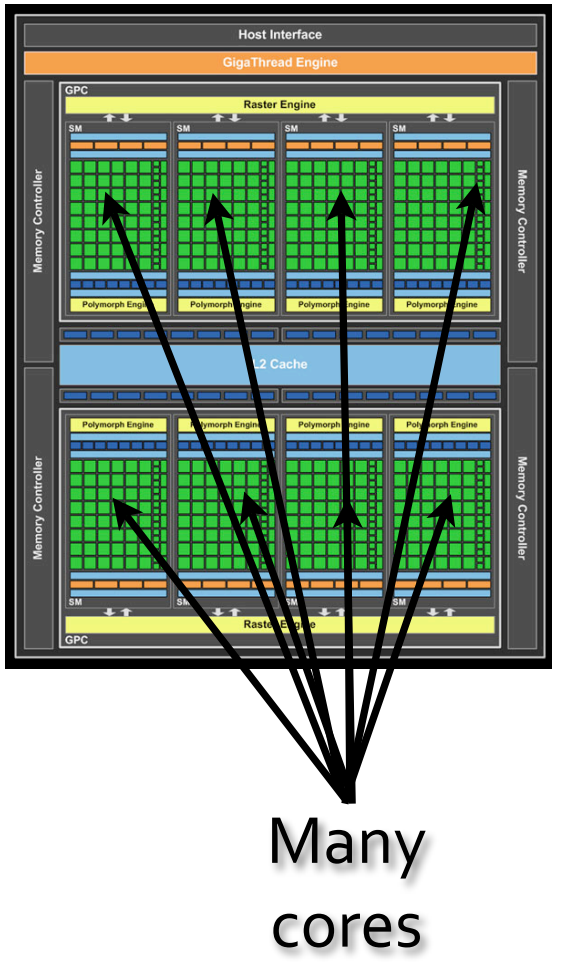
Fully Parallel

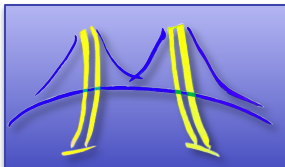




Mapping strategy for modern hardware

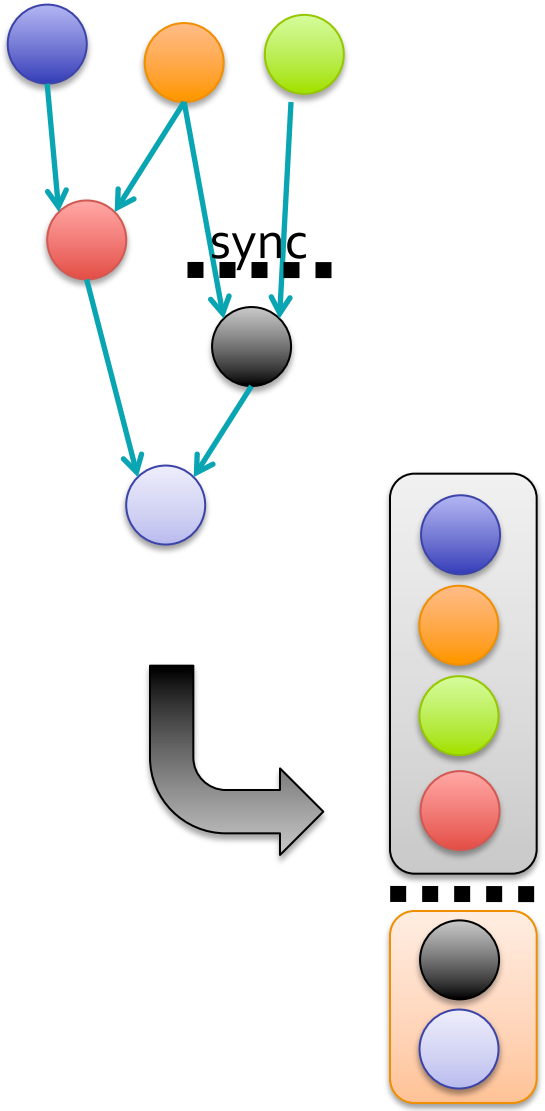
- Modern parallel hardware is MIMD
- Classical data parallel compilers flatten data parallelism for SIMD arrays
  - NESL, Data Parallel Haskell
  - Use segmented operations to flatten nested data parallelism
- In contrast, we map directly to MIMD parallelism
  - No default "flattening transform"
  - Much better performance

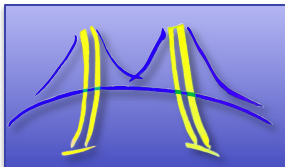




Phase Analysis, Scheduling, Fusion

- ***Phase analysis*** finds synchronization points
- Performed by abstract interpretation using a simple completion space
  - Same analysis for sequential and parallel operations
- Operations are then ***scheduled*** to respect data dependences
  - Greedy scheduler reorders computation to be as early as possible
- Operations between synchronization points are ***fused***
- Critical for performance (2-10x gain)

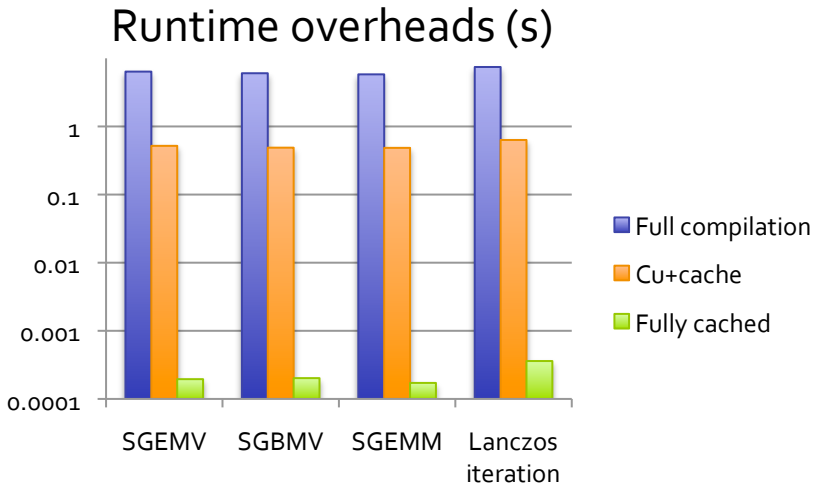


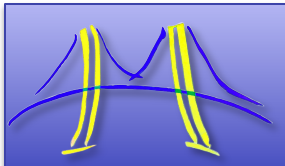


Runtime Model

- Runtime Static Compilation
  - Fits productivity programmer mindset
  - Avoid JIT overspecialization due to compilation overhead

- Currently we have a CUDA backend
- Every Copperhead entry point becomes a C++ function compiled in a shared object
  - May launch multiple kernels
- Runtime overheads carefully engineered
  - Fully cached, on order of 200µs/call
  - Full compilation, ~10 s/call





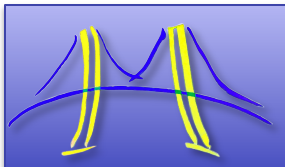
On-chip Memory Usage

- How can we use information from the Copperhead source code to influence on-chip memory placement?
- Consider the following code:

```
@cu
def outer(x, y):
  def inner(xi):
    return sum(map(op_mul, xi, y))
  return map(inner, x)
```

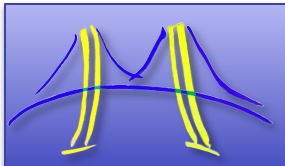
y is closed over

- Since **y** is closed over in a function used in a **map**, we know it is intensively used
- A good candidate for on-chip memory placement



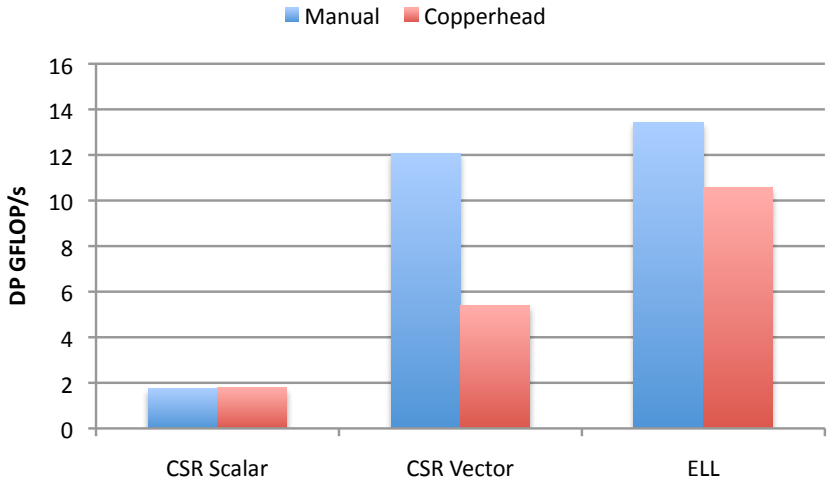
Efficient coexistence with other libraries

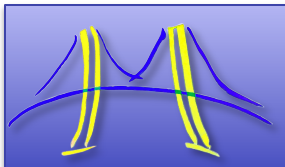
- Copperhead must efficiently coexist with preexisting code
- Copperhead programs can call preexisting libraries
- Libraries must be:
  - Wrapped in C++ to operate on Copperhead data structures in a side-effect free manner
  - Wrapped in Python with Copperhead type and shape information
- As proof of concept, we now can call portions of BLAS from Copperhead programs
- Library calls are compiled into the C++ code for the function
  - No additional Python/Copperhead overhead is introduced



Sparse Matrix Vector Multiply

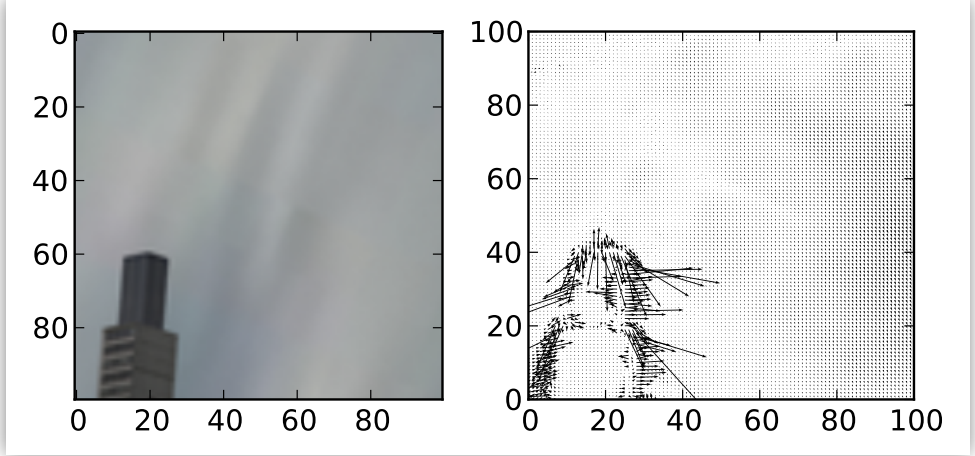
- Three SpMV kernels: CSR scalar, CSR vector, ELL
- On average, achieve 98%, 45%, and 79% of hand-coded performance (CUSP Library)
- This is average performance over the Matrix suite from Sam Williams *et al.*

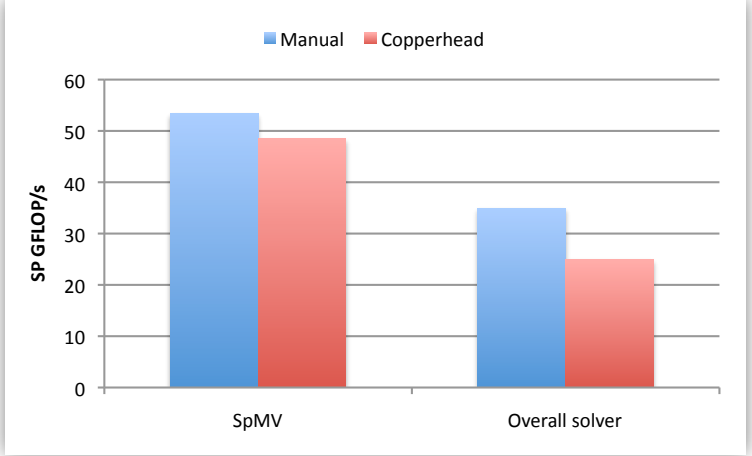


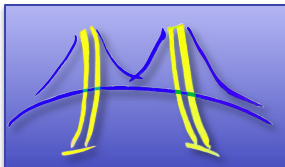


Preconditioned Conjugate Gradient Solver

- Preconditioned Conjugate Gradient solver implemented in Copperhead
- Based on linear step in non-linear optical flow solver
  - Block Jacobi Preconditioner
- Custom SpMV reaches 90% of hand-coded CUDA
- Overall solver reaches 71% of hand-coded CUDA

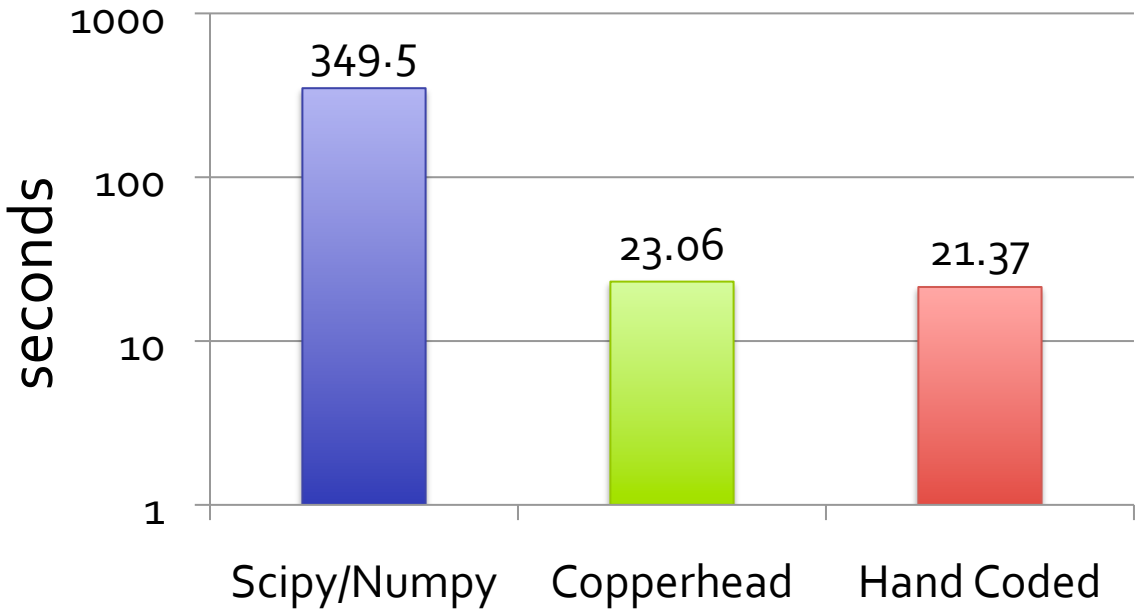






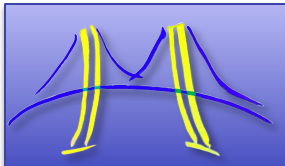
Lanczos Eigensolver

- Case study: calling BLAS from Copperhead
- Lanczos eigensolver from Damascene image contour detector
- Including all SEJITS overheads and Python iteration loop overhead, we come close to a handcoded C++ implementation (CUSP/CUBLAS)



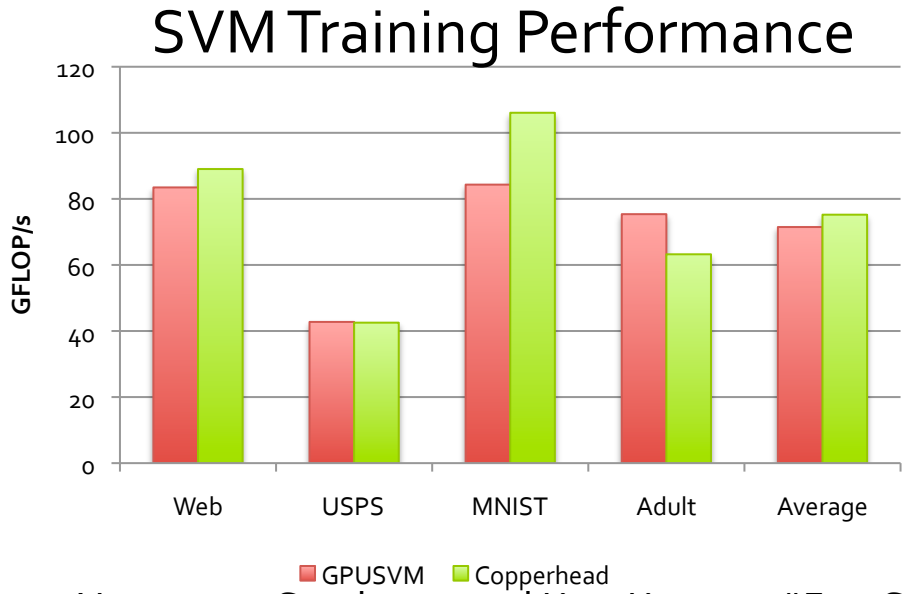
93% of handcoded performance

Bryan Catanzaro, Bor-Yiing Su, Narayanan Sundaram, Yunsup Lee, Mark Murphy, Kurt Keutzer, "*Efficient, High-Quality Image Contour Detection*". ICCV 2009.



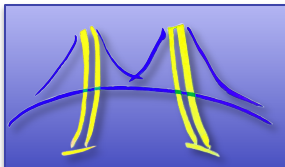
SVM Training

- SVM Training is a Quadratic Programming optimization problem
- Implemented in Copperhead, comparing against our own GPUSVM
- Good performance requires fusion, scheduling, on-chip memories



Competitive with our GPUSVM hand coded implementation

Bryan Catanzaro, Narayanan Sundaram and Kurt Keutzer, *"Fast Support Vector Machine Training and Classification on Graphics Processors"*. ICML 2008.



Conclusion

- Copperhead provides good performance, while increasing productivity
  - On average, 3.75x less lines of code, ability to tap into Python ecosystem
  - 45-105% of handcoded CUDA performance
- More details in our PPoPP 2011 paper:

Bryan Catanzaro, Michael Garland, Kurt Keutzer, "*Copperhead: Compiling an Embedded Data Parallel Language*", PPoPP 2011.

Copperhead is freely available (Apache 2.0 license) at:
http://code.google.com/p/copperhead