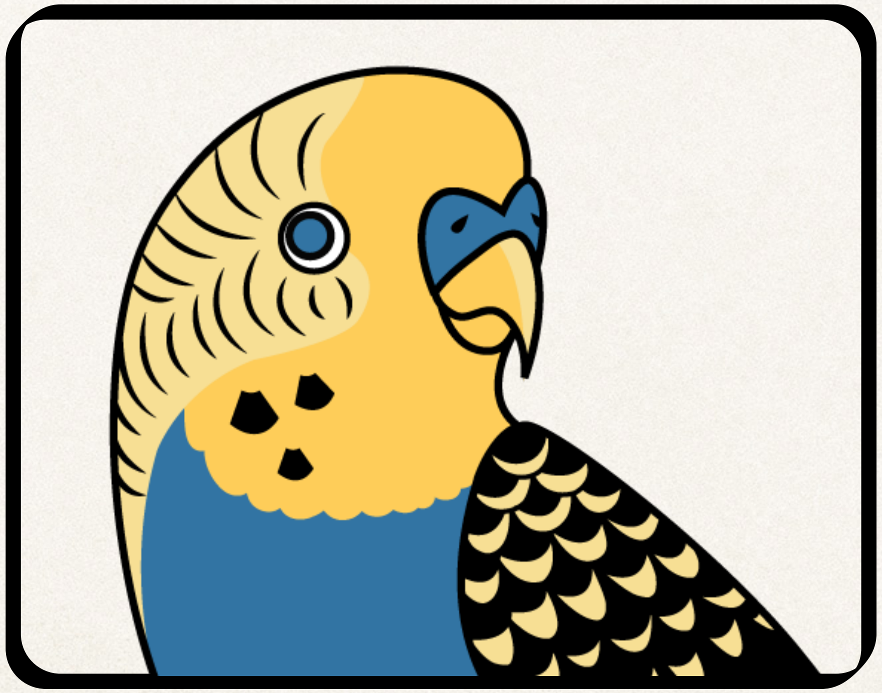
Python on the GPU with Parakeet



Alex Rubinsteyn @ Pydata Nyc (November 10Th, 2013)

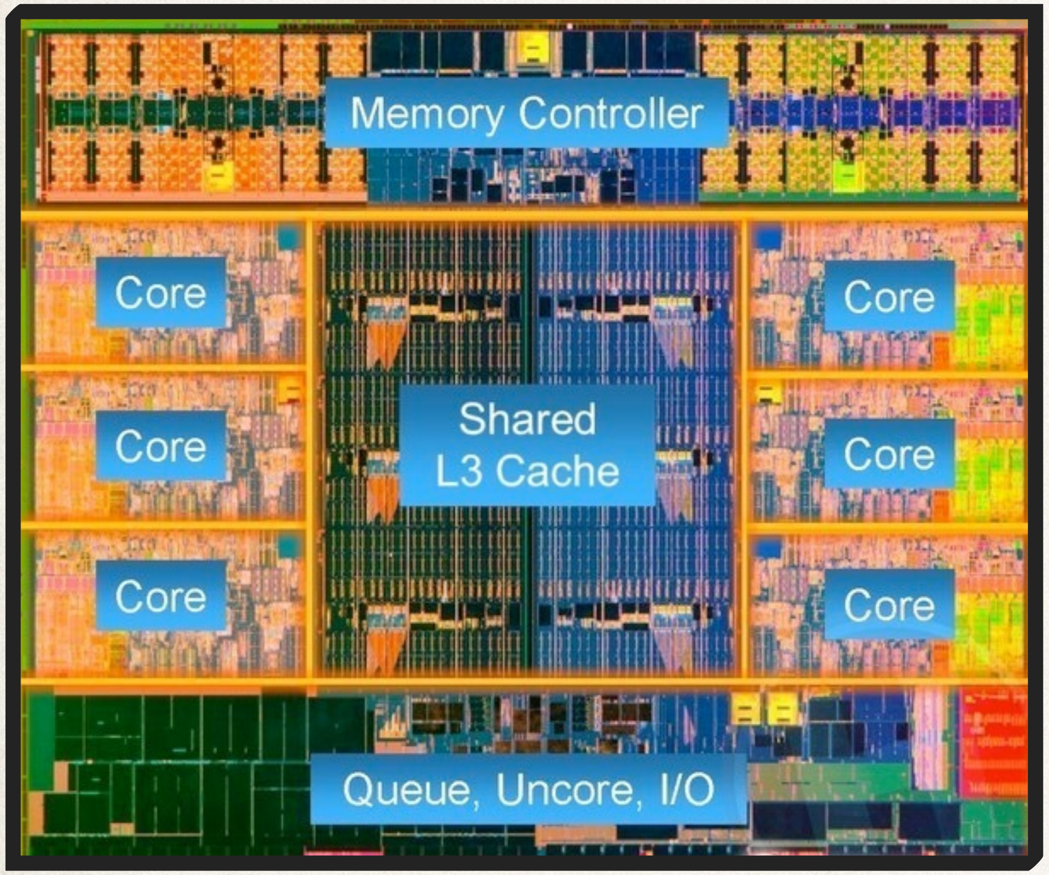
What's a GPU?

- Originally for drawing pixels on your screen...
- Lots of simple processors ("massively parallel")
- Can be dramatically faster than a CPU for numerically intensive programs (possibly orders of magnitude)
- Very annoying to program
  - *warp divergence, occupancy, coalescing, bank conflicts,* &c!

CPU vs. GPU highlights

	Cost	Specs	Memory Bandwidth	Float Math
Intel Core i7 3960X	$1000	6 cores @ 3.3ghz (8 float SIMD)	51 GB/s	140 billion FLOPS
NVIDIA GTX 780	$700	2880 “cores” @ 928mhz	336 GB/s	5 trillion FLOPS

Why is a GPU faster?



Intel Core i7

NVIDIA GTX 780

- CPU spends transistors on cache, memory, branch prediction, &c
- GPU is ruthlessly minimalist, mostly math & logic units

How do you program a GPU?

- Two competing languages: CUDA & OpenCL
  - Extensions to C/C++
  - Distinguish boundary between "host" (CPU) code and sections which run on the graphics card with attributes like `__host__` and `__device__`.
  - Program uses special *"which thread is this?"* variables to figure out what data elements to read and where to write results.

Example CUDA Program

Matrix Transpose (Naive Version)

GPU ("device") code

```
int TILE_DIM = 32;
int BLOCK_ROWS = 8;
int NUM_REPS = 100;

// entry-point to GPU program
__global__
void transposeNaive(float *out, float *in)
{
  // blockIdx and threadIdx are CUDA structs
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    out[x*width + (y+j)] = in[(y+j)*width + x];
}
```

Performs actual transpose

CPU ("host") code

```
void transpose(float* src, float* dst,int nx, int ny)
{
  int n = nx*ny*sizeof(float);
  dim3 dimGrid(nx/TILE_DIM, ny/TILE_DIM, 1);
  dim3 dimBlock(TILE_DIM, BLOCK_ROWS, 1);
  float *d_src, *d_dst;
  checkCuda(cudaMalloc(&d_src, n));
  checkCuda(cudaMalloc(&d_dst, n));
  checkCuda(cudaMemcpy(d_src, src, n,
cudaMemcpyHostToDevice));
  // special CUDA syntax for launching a kernel
  transposeNaive<<<dimGrid, dimBlock>>>(d_dst, d_src);
  checkCuda(cudaMemcpy(src, d_src, n,
cudaMemcpyDeviceToHost));
  checkCuda(cudaFree(d_src));
  checkCuda(cudaFree(d_dst));
}
```

Moves data to / from GPU,
sets up & launches computation

GPU Programming in Python

- **PyCUDA / PyOpenCL**
  - Low-level GPU programs as literal strings in Python
  - Library compiles kernels & moves data
  - GPUArray container implements small subset of NumPy's array interface
- **scikits.cuda**
  - Wraps precompiled NVIDIA libraries (BLAS, FFT,

Anything higher-level?

- **Theano**
  - Expression trees compile into GPU kernels (loved by neural network folks, only supports float32)
- **Copperhead**
  - Purely functional data parallel DSL in Python
  - Reinterprets list comprehensions as parallel maps
  - Compiles to Thrust (C++ CUDA library)

Parakeet

A runtime compiler for numerical Python

```
@jit
def now_faster(x):
  return np.mean(x < 0)
```

When you call a `@jit` wrapped function, Parakeet compiles it to native code.

Only a numerical/array-oriented subset of Python is supported.

Array operations run in parallel using OpenMP or CUDA.

What subset of Python works?

- Array + Scalar Expression

```
a * x[2:-3:k] + y[2:, 4:]  / b
```

- Tuples

```
a,b,c = (1,2) + (3,)
```

- Data Parallel Operators

```
parakeet.map(f_three_inputs, x, y, z)
parakeet.reduce(add, x, axis=1)
parakeet.imap(from_index, x.shape)
array([f(xi) for xi in x])
```

- (some) NumPy functions

```
np.arange(n) + np.linspace(a,b)
```

- Functions (with keywords)
- Loops & Conditionals

```
def f(pred, y = 1, *zs):
  while pred(y):
    y += sum(zs)
  return y
```

What *doesn't* compile? (Most things)

- Types other than arrays, slices, scalars: dicts, sets, lists, &c
- User-defined objects
- Assertions and exceptions
- Modifying/mutating anything other than array data
- Most library functions

Parakeet **doesn't** compete with PyPy, it's a small DSL

How does Parakeet work?

1. *wrap*

```
@jit
def f(x):
  return x + 1
```

Decorator parses function source, translates to untyped IR

2. *specialize*

```
f(673.6)
```

```
f(x : float64): return x +float 1.0
```

```
f(np.arange(5))
```

```
f(x : array1<int>): return map(+int, x, 1)
```

3. *optimize*

Compiler magic: *Simplify, CSE, DCE, LICM, Fusion, LowerArrayOperators, Indexify, …*

4. *codegen*

Generate C (sequential), OpenMP (multi-core), or CUDA (GPU)

Data Parallel Operators

- **map**: Apply a function to the elements of some array(s), or to each slice along a specified array axis.
- **reduce:** Combine the elements of an array with a binary operator.
- **scan:** Cumulative sums, products, &c
- **outer_map**: Apply function to cartesian product of array elements.
- **imap:** Apply function to indices in cartesian product of ranges.
- **ireduce**: Apply a function to index ranges, collect/reduce results with a binary operator.

Data Parallelism Without Trying

You don't need to always use data parallel operators explicitly.

Type Specializer expands array broadcasting into maps

```
matrix + scalar
```
⇝
```
map(lambda x,y: x + y, matrix, scalar)
```

Comprehensions become maps

```
[sqrt(xi) for xi in x]
```
⇝
```
map(sqrt,x)
```

NumPy library functions reimplemented w/ data parallelism explicitly

```
@jit
def prod(x, axis = None):
  return reduce(prims.multiply, x, init = 1, axis = axis)
```

Example: Matrix-Multiply

```
def matmult(X, Y):
  return array([[dot(row,col) for col in Y.T] for row in X])
```

Timings for 1200x1200 float32 arrays w/ 4-core Xeon 2.67ghz & GTX 780:

	Execution Time	Compile Time
Parakeet (single core)	14.65s	0.336s
Parakeet (multi-core)	4.08s	0.280s
Parakeet (GPU)	**0.11s**	**2.16s**
Numba	*fails*	--
Numba w/ explicit loops	14.79s	0.146s
Python + NumPy dot	17.4s	--
Python (*dot = sum(row*col)*)	~12 minutes	--
ATLAS (multi-core BLAS)	0.40s	--
cuBLAS (GPU)	**0.008s**	--

Example: Image Convolution

```
def conv_3x3_trim(image, weights):
  return array([[(image[i-1:i+2, j-1:j+2] * weights).sum()
                 for i in xrange(1, image.shape[0]-2]
                 for j in xrange(1, image.shape[1]-2])
```

Timings for 1200x1200 float64 array:

	Execution	Compile Time
Parakeet (single core)	16.8ms	247ms
Parakeet (4 cores)	11.5ms	220ms
Parakeet (GPU)	**3ms**	**~2.5 seconds**
Numba w/ loops	17ms	317ms
Python	10,975ms	--

Example: Simple Regression

```
def covariance(x,y):
  return ((x-x.mean()) * (y-y.mean())).mean()

@jit
def fit_simple_regression(x,y):
  slope = covariance(x,y) / covariance(x,x)
  offset = y.mean() - slope * x.mean()
  return slope, offset
```

	Execution Time
Parakeet (single core)	202ms
Parakeet (4 cores)	**95ms**
Parakeet (GPU)	*(death by memory transfer!)* 308ms
Numba	357ms
NumPy	362ms

(timings for x,y = 10 billion doubles)

What's next?

- **Fleshing out library functions** Tedious, but has to be done.
- **Improving the GPU backend**
  - *Data movement:* Currently, every parallel operator copies data to & from the GPU. Possible to infer when this isn't necessary.
  - *Data layout:* Currently, all new arrays are row-major. Can choose layouts more intelligently based on access pattern.
  - *Compile Time:* NVIDIA's compiler is slow, should cache compiled modules from hash of generated source file.
  - *Loop Parallelizer:* Simple loops can be turned into parallel

Thanks!

Try out Parakeet: **pip install parakeet**

Website: www.parakeetpython.com

Which NumPy functions work?

Types	bool, uint8, int8, uint16, int16, uint32, int32, uint64, int64, float32, float64
Constructors and Views	empty_like, empty, zeros_like, zeros, ones_like, ones, arange, transpose, ravel, copy
Array Properties	alen, size, rank, dtype
Reductions	min, max, argmin, argmax, all, any, sum, mean
Scans	cumsum, cumprod
Basic Math & Logic	minimum, maximum, abs, add, subtract, multiply, divide, true_divide, mod, remainder, sign, reciprocal, logical_and, logical_or, logical_not
Comparisons	less, less_equal, equal, not_equal, greater, greater_equal
Logs and Exponents	sqrt, square, power, exp, exp2, expm1, log, log10, log2, log1p, logaddexp, logaddexp2
Rounding	trunc, rint, floor, ceil, round
Trig	cos, arccos, cosh, arccosh, sin, arcsin, sinh, cosh, tan, arctan, arctan2, tanh, arctanh

What's missing from NumPy?

- Assertions and exceptions
- Complex numbers & structured dtypes
- Iterators (`flatiter`, `nditer`, `ndindex`, etc..)
- Random numbers (`numpy.random`)
- Linear Algebra (`numpy.linalg`)
- ...and lots more!

Math & Logic ufuncs were the easy part...