

# gpustats: GPU Library for Statistical Computing

Andrew Cron, Wes McKinney

Duke University

July 13, 2011

Why GPUs?

GPU Programming

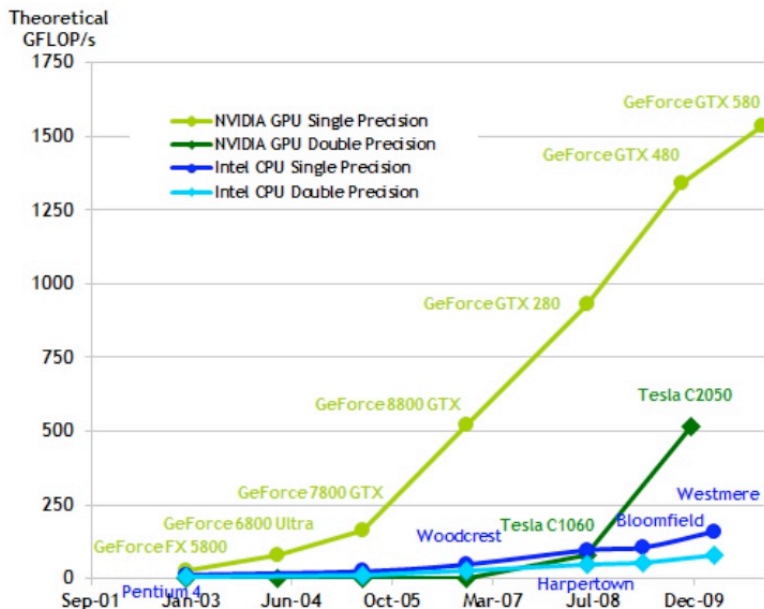
GPU Computing in Likelihood Based Inference

Benchmarks

Immediate Impacts in Python: PyMC

Future Work

# Why GPUs?



# GPU Programming

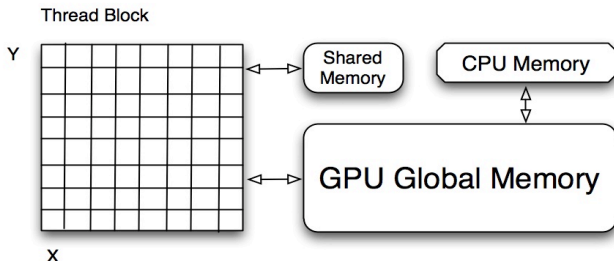
- ▶ C/C++ General Purpose GPU Computing
  - ▶ Nvidia: CUDA
  - ▶ Kronos: OpenCL
- ▶ Python GPU Computing
  - ▶ PyCUDA
  - ▶ PyOpenCL

gpustats currently depends on PyCUDA (since CUDA had “first mover advantage”), but most ideas apply to PyOpenCL.

# SIMD Development Challenges

GPUs operate under the Single Instruction Multiple Data framework:

- ▶ Each thread performs the same instruction on different data.
- ▶ Threads are organized into blocks with some shared cache memory.
- ▶ Threads operate in groups that run instruction in lockstep. If the threads diverge, they are executed serially.



# GPU Computing in Likelihood Based Inference

Many statistical models assume data,  $x_i$ , arise independently from some distribution  $p(x_i|\Theta)$ . Where  $\Theta$  are the parameters of interest.

Monte Carlo algorithms (and others) often require evaluating

$$\log L(\Theta|X) = \sum_{i=1}^n \log p(x_i|\Theta)$$

for many values of  $\Theta$ . If  $n$  is large, this is very expensive, but embarrassingly parallel.

# Common/Repetitive Issues in GPU Computing

For optimal performance, GPU developers must carefully design their thread instruction designs to take advantage of the memory architecture:

- ▶ *Coalescing* transactions between global (slow) and shared memory (fast).
- ▶ Avoiding shared memory *bank conflicts*.

Furthermore, there is an abundance of boiler plate code for memory copies, thread grid design optimization, and kernel launches that do not change across various pdf implementation.

## Metaprogramming to eliminate boilerplate code

The difference between implementing the normal logged pdf and the gamma logged pdf is literally just a couple lines the kernel:

```
float log_normal_pdf(float* x, float* params) {  
    float std = params[1];  
    float xstd = (*x - params[0]) / std;  
    return - (xstd * xstd) / 2 - 0.5 * LOG_2_PI  
        - log(std);  
}
```

Hopefully, this is *all* we'd have to write. Since PyCUDA compiles code on the fly, we can use string templates to dynamically generate CUDA code.

## Metaprogramming to eliminate boilerplate code

**Step 1:** Write “unnamed” inline CUDA device function implementing the PDF logic:

```
_log_pdf_normal = """
__device__ float %(name)s(float* x, float* params) {
    // mean stored in params[0]
    float std = params[1];

    // standardize
    float xstd = (*x - params[0]) / std;
    return - (xstd * xstd) / 2 - 0.5f * LOG_2_PI
           - log(std);
}
"""
```

# Metaprogramming to eliminate boilerplate code

**Step 2:** Pass that inline function to the code generation class:

```
log_pdf_normal = DensityKernel('log_pdf_normal',  
                               _log_pdf_normal)  
pdf_normal = Exp('pdf_normal', log_pdf_normal)
```

**Note**, the Exp class uses some metaprogramming trickery to apply an elementwise transformation to an existing kernel:

```
class Exp(Flop):  
    op = 'expf'
```

# Metaprogramming to eliminate boilerplate code

**Step 3:** Write Python interface function (could always do more metaprogramming here)

```
def normpdf_multi(x, means, std, logged=True):
    if logged:
        cu_func = mod.get_function('log_pdf_normal')
    else:
        cu_func = mod.get_function('pdf_normal')
    packed_params = np.c_[means, std]
    return _univariate_pdf_call(cu_func, x,
                                packed_params)
```

# Benchmarks

- ▶ Comparisons were made on a NVIDIA GTS 250 (128 CUDA cores) and an Intel Core i7 930.
- ▶ Comparisons are made with and without reallocated GPUArrays.
- ▶ Several sizes of density values are considered with both single and multiple parameter values.

## Benchmarks

	1e3	1e4	1e5	1e6
Single	0.22	1.27	7.95	23.05
Single (GPUArray)	0.24	1.29	9.36	38.72
Multi	1.46	7.04	26.19	43.73
Multi (GPUArray)	1.79	8.35	30.79	49.26

Table: Univariate Normal PDF Speedup

	1e3	1e4	1e5	1e6
Single	0.70	4.17	12.55	14.09
Single (GPUArray)	0.85	6.03	32.59	64.12
Multi	3.13	18.41	60.18	63.89
Multi (GPUArray)	3.14	19.80	74.39	82.00

Table: Multivariate Normal PDF Speedup

# Immediate Impacts in Python: PyMC

- ▶ The PyMC package allows users to specify complex Bayesian models and does MCMC integration for inference. ( Similar to WinBUGS/OpenBUGS/JAGS. )
- ▶ Each iteration of the sample involves an evaluation of the likelihood. If the data is large, this becomes infeasible.
- ▶ PyMC allows users to use custom likelihoods! Using our GPU bases likelihood could make PyMC more practical for much larger data sets.

## Future Work

- ▶ Use PyOpenCL for ATI cards and CPUs.
- ▶ Seamlessly use multiple GPUs.
- ▶ Reimplement the probability distributions in `scipy.stats` (densities, cumulative distribution functions, and samplers). This will allow seamless integration with most statistics packages.

Thanks!

Questions?