GPU Programming made Easy

Frédéric Bastien

Laboratoire d'Informatique des Systèmes Adaptatifs

Département d'informatique et de recherche opérationelle

James Bergstra, Olivier Breuleux, Frederic Bastien,

Arnaud Bergeron, Yoshua Bengio, Thierry Bertin-Mahieux, Josh Bleecher Snyder, Olivier Delalleau, Guillaume Desjardins, Douglas Eck, Dumitru Erhan, Xavier Glorot, Ian Goodfellow, Philippe Hamel, Pascal Lamblin, Simon Lemieux, Michael Mandel, Razvan Pascanu, François Savard, Joseph Turian, David Warde-Farley

Presented on June 13th 2011

HPCS 2011, Montréal

Laboratoire d'Informatique des Systèmes Adaptatifs
http://www.iro.umontreal.ca/~lisa

Université de Montréal

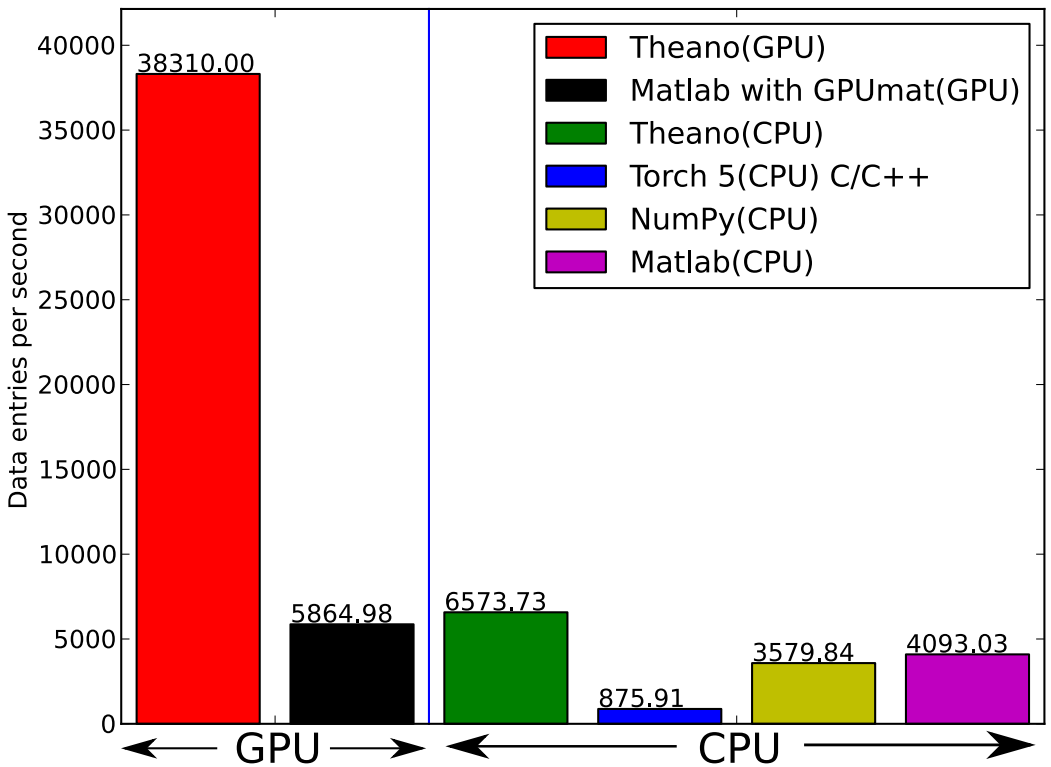
Theano Goal

- Tries to be the **holy grail** in computing: *easy to code* and *fast to execute* !
- Only on mathematical expressions
- So you won't have:
  - Function call inside a theano function
  - Structure, enum
  - Dynamic type (Theano is Fully typed)
  - ...

- And doesn't do coffee!

Université de Montréal

Faster on CPU and GPU



Université de Montréal

Project Status

Why you can rely on Theano:

- Theano has been developed and used since January 2008 (3.5 yrs old)
- Core technology for a funded Silicon-Valley startup
- Driven over 40 research papers in the last few years
- Good user documentation
- Active mailing list with participants from outside our lab
- Many contributors (some from outside our lab)
- Used to teach IFT6266 for two years
- Used by everyone in our lab (~30 people)
- Deep Learning Tutorials
- Unofficial RPMs for Mandriva
- Downloads (June 8 2011, since last January):
  - Pypi 780
  - MLOSS: 483
  - Assembla ("bleeding edge" repository): unknown

Overview 1

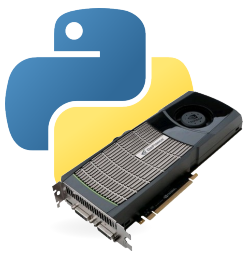
- **Exercises as we go**
- Introduction
  - Why Scripting for GPUs?
  - Theano vs. PyCUDA vs. PyOpenCL vs. CUDA
  - Python in 1 slide
  - NumPy in 1 slide
- Theano
  - Introduction
  - Simple example
  - Real example
  - Theano Flags
  - GPU
  - Symbolic Variables
  - Differentiation Details
  - Benchmarks
- break?

Overview 2

- Advanced Theano
  - Compilation Pipeline
  - Inplace Optimization
  - Profiling
  - Drawing/Printing Theano Graph
  - Debugging
  - Scan (For-Loop generalization)
  - Known Limitations

Overview 3

- PyCUDA
  - Introduction
  - Example
- CUDA Overview
- Extending Theano
  - Theano Graph
  - Op Contract
  - Op Example
  - Theano + PyCUDA
- GpuNdArray
- Conclusion



Overview 4

- Only high level overview of CUDA
- Won't talk about how to optimize GPU code

Why GPU

- Faster, cheaper, more efficient power usage
- How much faster? I have seen numbers from 100x slower to 1000x faster.
  - It depends on the algorithms
  - How the benchmark is done
    - Quality of implementation
    - How much time was spent optimizing CPU vs GPU code
  - In Theory:
    - Intel Core i7 980 XE (107Gf/s float64) 6 cores
    - NVIDIA C2050 (515 Gf/s float64, 1Tf/s float32) 480 cores
    - NVIDIA GTX580 (1.5Tf/s float32) 512 cores
- Theano goes up to 100x faster on th GPU because we don't use multiple core on CPU
  - Theano can be linked with multi-core capable BLAS (GEMM and GEMV)
- If you see 1000x, it probably means the benchmark is not fair

Why Scripting for GPUs

They **Complement each other**

- GPUs are everything that scripting/high level languages are not
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
- CPU: largely restricted to control
  - Optimized for sequential code and *low latency* (rather than high throughput)
  - Tasks (1000/sec)
  - Scripting fast enough

Theano vs PyCUDA vs PyOpenCL vs CUDA

- Theano
  - Mathematical expression compiler
  - Generates costum C and CUDA code
  - Uses Python code when performance is not critical
- CUDA
  - C extension by NVIDA that allow to code and use GPU
- PyCUDA (Python + CUDA)
  - Python interface to CUDA
  - Memory management of GPU objects
  - Compilation of code for the low-level driver
- PyOpenCL (Python + OpenCL)
  - PyCUDA for OpenCL

What is your background ?

Do you have experience with :

- Python
- NumPy / SciPy / Matlab
- Maple / Mathematica / SymPy
- GPU programming / CUDA / OpenCL
- Cython / Weave / Numexpr
- C / Java / Fortran

Python in 1 Slide

- Interpreted language
- General-purpose high-level programming language
- OO and scripting language
- Emphasizes code readability
- Large and comprehensive standard library
- Indentation for block delimiters
- Dynamic type and memory management
- Dictionary `d={'var1':'value1', 'var2':42, ...}`
- List comprehension: `[i+3 for i in range(10)]`

NumPy in 1 Slide

- Base scientific computing package in Python on the CPU
- A powerful N-dimensional array object
  - ndarray.{ndim, shape, size, dtype, itemsize, stride}
- Sophisticated "broadcasting" functions
  - `numpy.random.rand(4,5) * numpy.random.rand(1,5)` $\Rightarrow$ mat(4,5)
  - `numpy.random.rand(4,5) * numpy.random.rand(4,1)` $\Rightarrow$ mat(4,5)
  - `numpy.random.rand(4,5) * numpy.random.rand(5)` $\Rightarrow$ mat(4,5)
- Tools for integrating C/C++ and Fortran code
- Linear algebra, Fourier transform and pseudorandom number generation

Theano

Pointers

- Website: http://deeplearning.net/software/theano/
- Announcements mailing list: http://groups.google.com/group/theano-announce
- User mailing list: http://groups.google.com/group/theano-users
- Deep Learning Tutorials: http://www.deeplearning.net/tutorial/

- Installation: https://deeplearning.net/software/theano/install.html

Description

- Mathematical symbolic expression compiler
- Dynamic C/CUDA code generation
- Efficient symbolic differentiation
  - Theano computes derivatives of functions with one or many inputs.
- Speed and stability optimizations
  - Gives the right answer for $\log(1 + x)$ even if x is really tiny.
- Works on Linux, Mac and Windows
- Transparent use of a GPU
  - float32 only for now (working on other data types)
  - Doesn't work on Windows for now
  - On GPU data-intensive calculations are typically between 6.5x and 44x faster. We've seen speedups up to 140x

Description 2

- Extensive unit-testing and self-verification
  - Detects and diagnoses many types of errors
- On CPU, common machine learning algorithms are 1.6x to 7.5x faster than competitive alternatives
  - including specialized implementations in C/C++, NumPy, SciPy, and Matlab
- Expressions mimic NumPy's syntax & semantics
- Statically typed and purely functional
- Some sparse operations (CPU only)
- The project was started by James Bergstra and Olivier Breuleux
- For the past 1-2 years, I have replaced Olivier as lead contributor

Why Theano is better

Executing the code is faster because Theano:

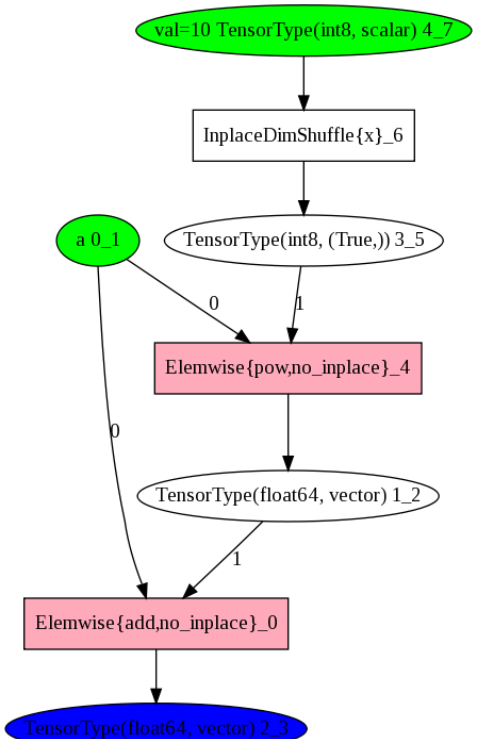
- Rearranges high-level expressions
- Produces customized low-level code
- Uses a variety of backend technologies (GPU,...)

Writing the code is faster because:

- High-level language allows to **concentrate on the algorithm**
- Theano do **automatic optimization**
  - No need to manually optimize for each algorithm you want to test
- Theano do **automatic efficient symbolic differentiation**
  - No need to manually differentiate your functions (tedious & error-prone for complicated expressions!)

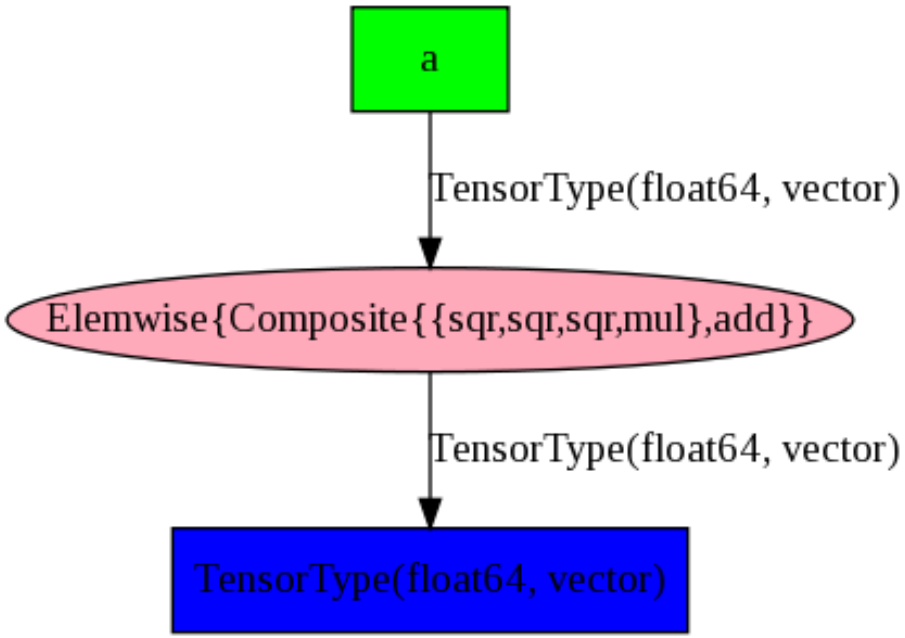
Simple Example

```
import theano
a = theano.tensor.vector("a")  # declare symbolic variable
b = a + a**10                  # build symbolic expression
f = theano.function([a], b)    # compile function
print f([0,1,2])               # prints ‘array([0,2,1026])‘
```



Simple Example: Optimized graph

no pow, fused elemwise op!



Symbolic programming

- Paradigm shift: people need to use it to understand it

Université de Montréal

Exercises 1

```
source /groups/h/hpc2011/bin/GPU.csh
hg clone http://hg.assembla.com/theano Theano
cd Theano/doc/hpcs2011_tutorial
python simple_example.py
```

Modify and execute the example to do this expression: a**2 + b**2 + 2*a*b

A Real Example: Logistic Regression

- GPU-ready
- Symbolic differentiation
- Speed optimizations
- Stability optimizations

A Real Example: Logistic Regression

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats), rng.randint(size=N,low=0, high=2))
training_steps = 10000
```

A Real Example: Logistic Regression

```
# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(100), name="w")
b = theano.shared(0., name="b")
print "Initial model:"
print w.get_value(), b.get_value()
```

A Real Example: Logistic Regression

```
# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(100), name="w")
b = theano.shared(0., name="b")

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))    # Probability that target = 1
prediction = p_1 > 0.5                   # The prediction thresholded
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)# Cross-entropy loss function
cost = xent.mean() + 0.01*(w**2).sum()   # The cost to minimize
gw,gb = T.grad(cost, [w,b])
```

Université de Montréal

A Real Example: Logistic Regression

```
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(100), name="w")
b = theano.shared(0., name="b")
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
prediction = p_1 > 0.5
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
cost = xent.mean() + 0.01*(w**2).sum()
gw,gb = T.grad(cost, [w,b])

# Compile
train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
            updates={w:w-0.1*gw, b:b-0.1*gb})
predict = theano.function(inputs=[x], outputs=prediction)
```

A Real Example: Logistic Regression

```
# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])

print "Final model:"
print w.get_value(), b.get_value()
print "target values for D:", D[1]
print "prediction on D:", predict(D[0])
```

A Real Example: optimization

```
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
prediction = p_1 > 0.5
cost = xent.mean() + 0.01*(w**2).sum()
gw,gb = T.grad(cost, [w,b])

train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
            updates={w:w-0.1*gw, b:b-0.1*gb})  # This is a dictionary
```

Where are those optimization applied?

- Log(1+exp(x))
- 1 / (1 + T.exp(var)) (sigmoid)
- Log(1-sigmoid(var)) (softplus, stabilisation)
- GEMV (matrix-vector multiply from BLAS)
- Loop fusion

Université de Montréal

A Real Example: optimization!

```
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b))
# 1 / (1 + T.exp(var)) -> sigmoid(var)
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1)
# Log(1-sigmoid(var)) -> -sigmoid(var)

prediction = p_1 > 0.5
cost = xent.mean() + 0.01*(w**2).sum()
gw,gb = T.grad(cost, [w,b])

train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
# w-0.1*gw: GEMV with the dot in the grad
            updates={w:w-0.1*gw, b:b-0.1*gb})
```

- Loop fusion in many places

Université de Montréal

Theano Flags

Theano can be configured with flags. They can be defined in two ways

- With an environment variable:
  `THEANO_FLAGS="mode=ProfileMode,ProfileMode.profile_memory=True"`
- With a configuration file that defaults to ~/.theanorc

Exercises 2

```
python logreg_example.py
```

Modify and execute the example in the file logreg_example.py to run on CPU with floatX=float32
* You will need to use: theano.config.floatX and ndarray.astype("str")

GPU

- Only 32 bit floats are supported (being worked on)
- Only 1 GPU per process
- Use the Theano flag `device=gpu` to tell to use the GPU device
  - Use `device=gpu0, 1, ...` to specify which GPU if you have more than one
  - Shared variables with float32 dtype are by default moved to the GPU memory space
- Use the Theano flag `floatX=float32`
  - Be sure to use `floatX` (`theano.config.floatX`) in your code
  - Cast inputs before putting them into a shared variable
  - Cast "problem": int32 with float32 $\rightarrow$ float64
    - A new casting mechanism is being developed
    - Insert manual cast in your code or use [u]int8,16
    - Insert manual cast around the mean operator (which involves a division by the length, which is an int64!)

GPU for Exercises

- Intel Core i7 980 XE (107Gf/s float64, 1050$, 6 cores/12 threads)
- NVIDIA C2050 (515 Gf/s float64, 1Tf/s float32, 2400$, 480 cores), compute capability 2.0
- NVIDIA GTX580 (1.5Tf/s float32, 500$, 512 cores), compute capability 2.0

Computers in the class

- Intel Xeon X3450 (?56? flops/s, 383$, 4 cores)
- NVIDIA Quadro FX 580 (71GF/s single, 140$, 32 cores), compute capability 1.1, 'profesionnal card'

Exercises 3

- Modify and execute the code to run with floatX=float32 on GPU
- Time with: `time python file.py`

Creating symbolic variables

- # Dimensions
  - T.scalar, T.vector, T.matrix, T.tensor3, T.tensor4
- Dtype
  - T.[fdczbwil]vector (float32, float64, complex64, complex128, int8, int16, int32, int64)
  - T.vector → floatX dtype
  - floatX: configurable dtype that can be float32 or float64.
- Custom variable
  - All are shortcuts to: T.tensor(dtype, broadcastable=[False]*nd)
  - Other dtype: uint[8,16,32,64], floatX

Creating symbolic variables: Broadcastability

- Remember what I said about broadcasting?
- How to add a row to all rows of a matrix?
- How to add a column to all columns of a matrix?

- Broadcastability must be specified when creating the variable
- The only shorcut with broadcastable dimensions are: **T.row** and **T.col**
- For all others: T.tensor(dtype, broadcastable=**([False or True])*nd**)

Differentiation Details

```
gw,gb = T.grad(cost, [w,b])
```

- T.grad works symbolically: takes and returns a Theano variable
- T.grad can be compared to a macro: it can be applied multiple times
- T.grad takes scalar costs only
- Simple recipe allows to compute efficiently vector $\times$ Jacobian and vector $\times$ Hessian
- We are working on the missing optimizations to be able to compute efficently the full Jacobian and Hessian and Jacobian $\times$ vector

Benchmarks

Example:

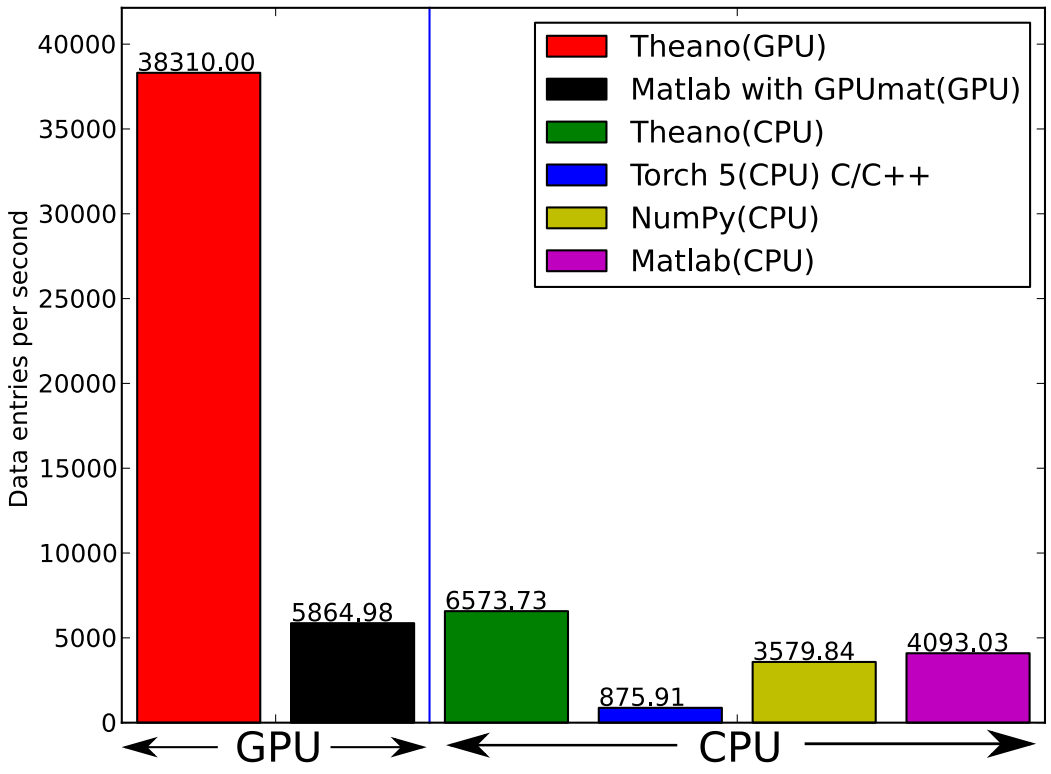
- Multi-layer perceptron
- Convolutional Neural Networks
- Misc Elemwise operations

Competitors: NumPy + SciPy, MATLAB, EBLearn, Torch5, numexpr

- EBLearn, Torch5: specialized libraries written by practitioners specifically for these tasks
- numexpr: similar to Theano, 'virtual machine' for elemwise expressions

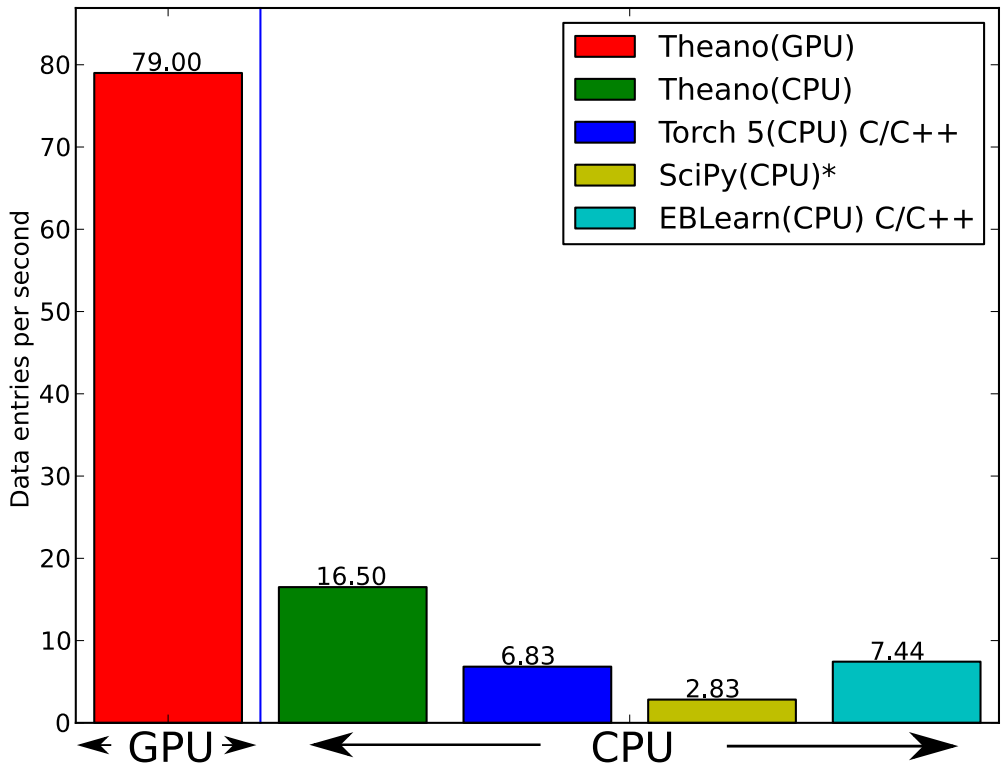
Benchmark MLP

Multi-Layer Perceptron: 60x784 matrix times 784x500 matrix, tanh, times 500x10 matrix, elemwise, then all in reverse for backpropagation



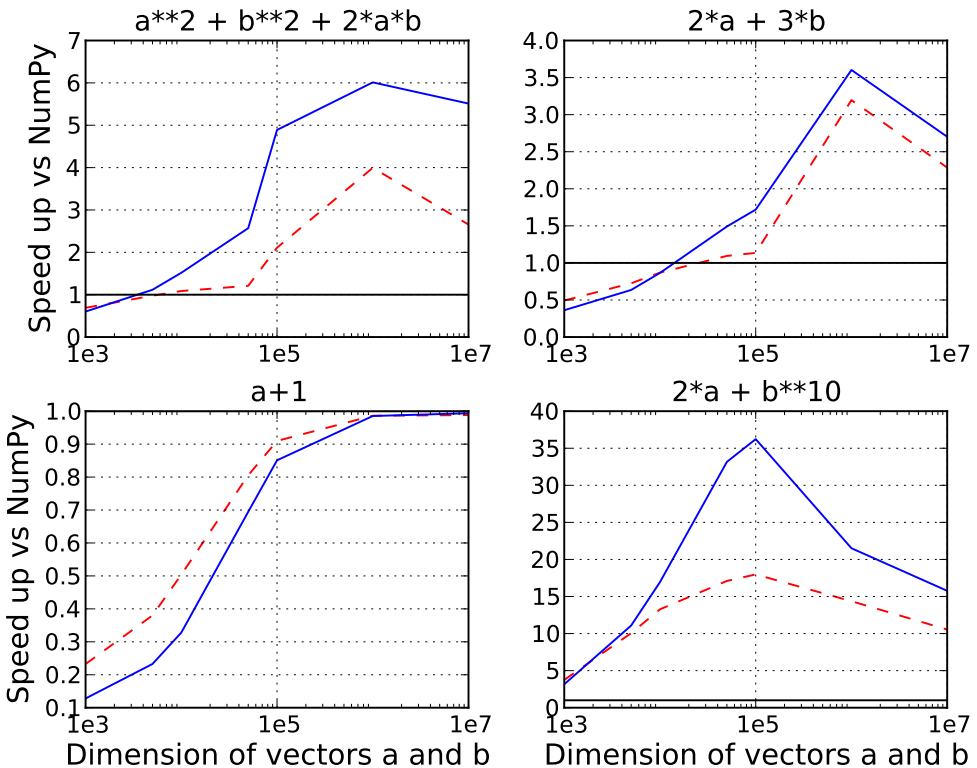
Benchmark Convolutional Network

Convolutional Network: 256x256 images convolved with 6 7x7 filters, downsampled to 6x50x50, tanh, convolution with 16 6x7x7 filter, elementwise tanh, matrix multiply, softmax elementwise, then in reverse

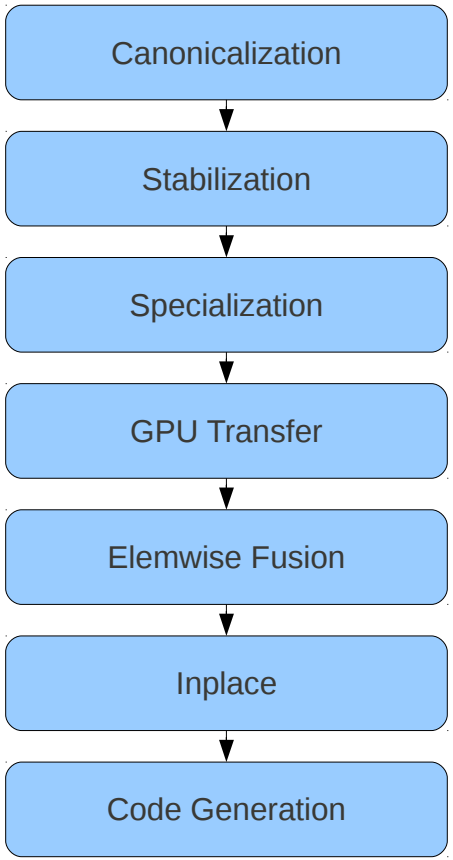


Elemwise Benchmark

- All on CPU
- Solid blue: Theano
- Dashed Red: numexpr (without MKL)



Compilation Pipeline



Inplace Optimization

- 2 type of inplace operations:
  - An op that return a view on its inputs (e.g. reshape, inplace transpose)
  - An op that write the output on the inputs memory space
- This allows some memory optimization
- The Op must tell Theano if they work inplace
- Inplace Op add constraints to the order of execution

Profile Mode

To replace the default mode with this mode, use the Theano flags
`mode=ProfileMode`
To enable the memory profiling use the flags
`ProfileMode.profile_memory=True`

```
Time since import 33.456s
Theano compile time: 1.023s (3.1% since import)
    Optimization time: 0.789s
    Linker time: 0.221s
Theano fct call 30.878s (92.3% since import)
   Theano Op time 29.411s 87.9%(since import) 95.3%(of fct call)
   Theano function overhead in ProfileMode 1.466s 4.4%(since import)
                                                  4.7%(of fct call)
10001 Theano fct call, 0.003s per call
Rest of the time since import 1.555s 4.6%
```

Profile Mode: Function Summary

Theano outputs:

```
Theano fct summary:
<% total fct time> <total time> <time per call> <nb call> <fct name>
   100.0% 30.877s 3.09e-03s 10000 train
    0.0% 0.000s 4.06e-04s 1 predict
```

Profile Mode: Single Op-Wise Summary

Theano outputs:

```
Single Op-wise summary:
<% of local_time spent on this kind of Op> <cumulative %>
    <self seconds> <cumulative seconds> <time per call> <nb_call>
    <nb_op> <nb_apply> <Op name>
   87.3%    87.3%  25.672s  25.672s  2.57e-03s    10000  1   1 <Gemv>
    9.7%    97.0%  2.843s  28.515s  2.84e-04s    10001  1   2 <Dot>
    2.4%    99.3%  0.691s  29.206s  7.68e-06s * 90001 10 10 <Elemwise>
    0.4%    99.7%  0.127s  29.334s  1.27e-05s    10000  1   1 <Alloc>
    0.2%    99.9%  0.053s  29.386s  1.75e-06s * 30001  2   4 <DimShuffle>
    0.0%   100.0%  0.014s  29.400s  1.40e-06s * 10000  1   1 <Sum>
    0.0%   100.0%  0.011s  29.411s  1.10e-06s * 10000  1   1 <Shape_i>
(*) Op is running a c implementation
```

Université de Montréal

Profile Mode: Op-Wise Summary

Theano outputs:

```
Op-wise summary:
<% of local_time spent on this kind of Op> <cumulative %>
    <self seconds> <cumulative seconds> <time per call>
    <nb_call> <nb apply> <Op name>
   87.3%   87.3%  25.672s  25.672s  2.57e-03s    10000  1 Gemv{inplace}
    9.7%   97.0%  2.843s  28.515s  2.84e-04s    10001  2 dot
    1.3%   98.2%  0.378s  28.893s  3.78e-05s * 10000  1 Elemwise{Compos
        scalar_softplus,{mul,scalar_softplus,{neg,mul,sub}}}}
    0.4%   98.7%  0.127s  29.021s  1.27e-05s    10000  1 Alloc
    0.3%   99.0%  0.092s  29.112s  9.16e-06s * 10000  1 Elemwise{Compos
        exp,{mul,{true_div,neg,{add,mul}}}}}[(0, 0)]
    0.1%   99.3%  0.033s  29.265s  1.66e-06s * 20001  3 InplaceDimShuff
   ... (remaining 11 Apply account for 0.7%(0.00s) of the runtime)
(*) Op is running a c implementation
```

Profile Mode: Apply-Wise Summary

Theano outputs:

```
Apply-wise summary:
<% of local_time spent at this position> <cumulative %%>
    <apply time> <cumulative seconds> <time per call>
    <nb_call> <Apply position> <Apply Op name>
   87.3%   87.3%  25.672s  25.672s 2.57e-03s  10000  15 Gemv{inplace}(
        w, TensorConstant{-0.01}, InplaceDimShuffle{1,0}.0, Elemwise{Co
    9.7%   97.0%  2.843s  28.515s 2.84e-04s  10000   1 dot(x, w)
    1.3%   98.2%  0.378s  28.893s 3.78e-05s  10000   9 Elemwise{Composi
    0.4%   98.7%  0.127s  29.020s 1.27e-05s  10000  10 Alloc(Elemwise{i
    0.3%   99.0%  0.092s  29.112s 9.16e-06s  10000  13 Elemwise{Composi
    0.3%   99.3%  0.080s  29.192s 7.99e-06s  10000  11 Elemwise{ScalarS
   ... (remaining 14 Apply instances account for
       0.7%(0.00s) of the runtime)
```

Université de Montréal

Profile Mode: Memory Profile

Theano outputs:

```
Profile of Theano functions memory:
(This check only the output of each apply node. It don't check the
    temporary memory used by the op in the apply node.)
Theano fct: train
    Max without gc, inplace and view (KB) 2481
    Max FAST_RUN_NO_GC (KB) 16
    Max FAST_RUN (KB) 16
    Memory saved by view (KB) 2450
    Memory saved by inplace (KB) 15
    Memory saved by GC (KB) 0
    <Sum apply outputs (bytes)> <Apply outputs memory size(bytes)>
        <created/inplace/view> <Apply node>
    <created/inplace/view> is taked from the op declaration, not ...
         2508800B  [2508800] v InplaceDimShuffle{1,0}(x)
            6272B  [6272] i Gemv{inplace}(w, ...)
            3200B  [3200] c Elemwise{Composite{...}}(y, ...)
```

Profile Mode: Tips

Theano outputs:

```
Here are tips to potentially make your code run faster
(if you think of new ones, suggest them on the mailing list).
Test them first, as they are not guaranteed to always provide a speedup
  - Try the Theano flag floatX=float32
```

Exercises 4

- In the last exercises, do you see a speed up with the GPU?
- Where does it come from? (Use ProfileMode)
- Is there something we can do to speed up the GPU version?

Text Printing of Your Theano Graph: Pretty Printing

theano.printing.pprint(variable)

```
>>> theano.printing.pprint(prediction)
gt((TensorConstant{1} / (TensorConstant{1} + exp(((-(x \\dot w)) - b)))
TensorConstant{0.5})
```

Text Printing of Your Theano Graph: Debug Print

theano.printing.debugprint(fct, variable, list of variables)

```
>>> theano.printing.debugprint(prediction)
Elemwise{gt,no_inplace} [@181772236] ''
 |Elemwise{true_div,no_inplace} [@181746668] ''
 | |InplaceDimShuffle{x} [@181746412] ''
 | | |TensorConstant{1} [@181745836]
 | |Elemwise{add,no_inplace} [@181745644] ''
 | | |InplaceDimShuffle{x} [@181745420] ''
 | | | |TensorConstant{1} [@181744844]
 | | |Elemwise{exp,no_inplace} [@181744652] ''
 | | | |Elemwise{sub,no_inplace} [@181744012] ''
 | | | | |Elemwise{neg,no_inplace} [@181730764] ''
 | | | | | |dot [@181729676] ''
 | | | | | | |x [@181563948]
 | | | | | | |w [@181729964]
 | | | | |InplaceDimShuffle{x} [@181743788] ''
 | | | | | |b [@181730156]
 |InplaceDimShuffle{x} [@181771788] ''
 | |TensorConstant{0.5} [@181771148]
```

Université de Montréal

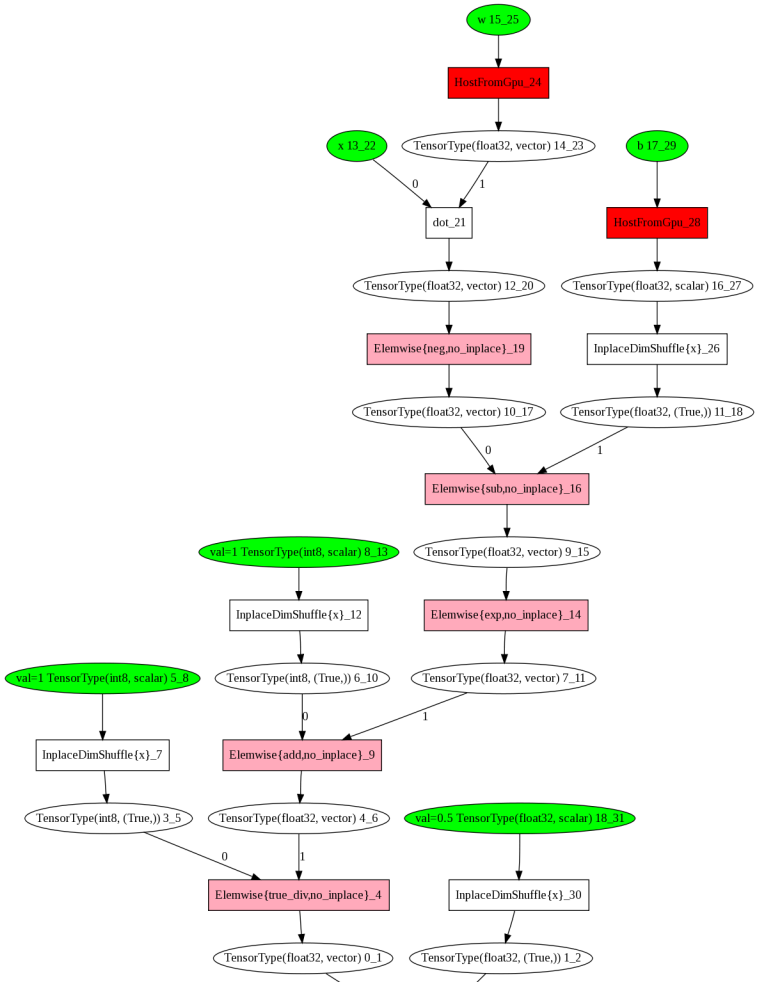
Text Printing of Your Theano Graph: Debug Print

theano.printing.debugprint(fct, variable, list of variables)

```
>>> theano.printing.debugprint(predict)
Elemwise{Composite{neg,{sub,{{scalar_sigmoid,GT},neg}}}} [@183160204] ''   2
 |dot [@183018796] ''   1
 | |x [@183000780]
 | |w [@183000812]
 |InplaceDimShuffle{x} [@183133580] ''   0
 | |b [@183000876]
 |TensorConstant{[ 0.5]} [@183084108]
```

Picture Printing of Graphs

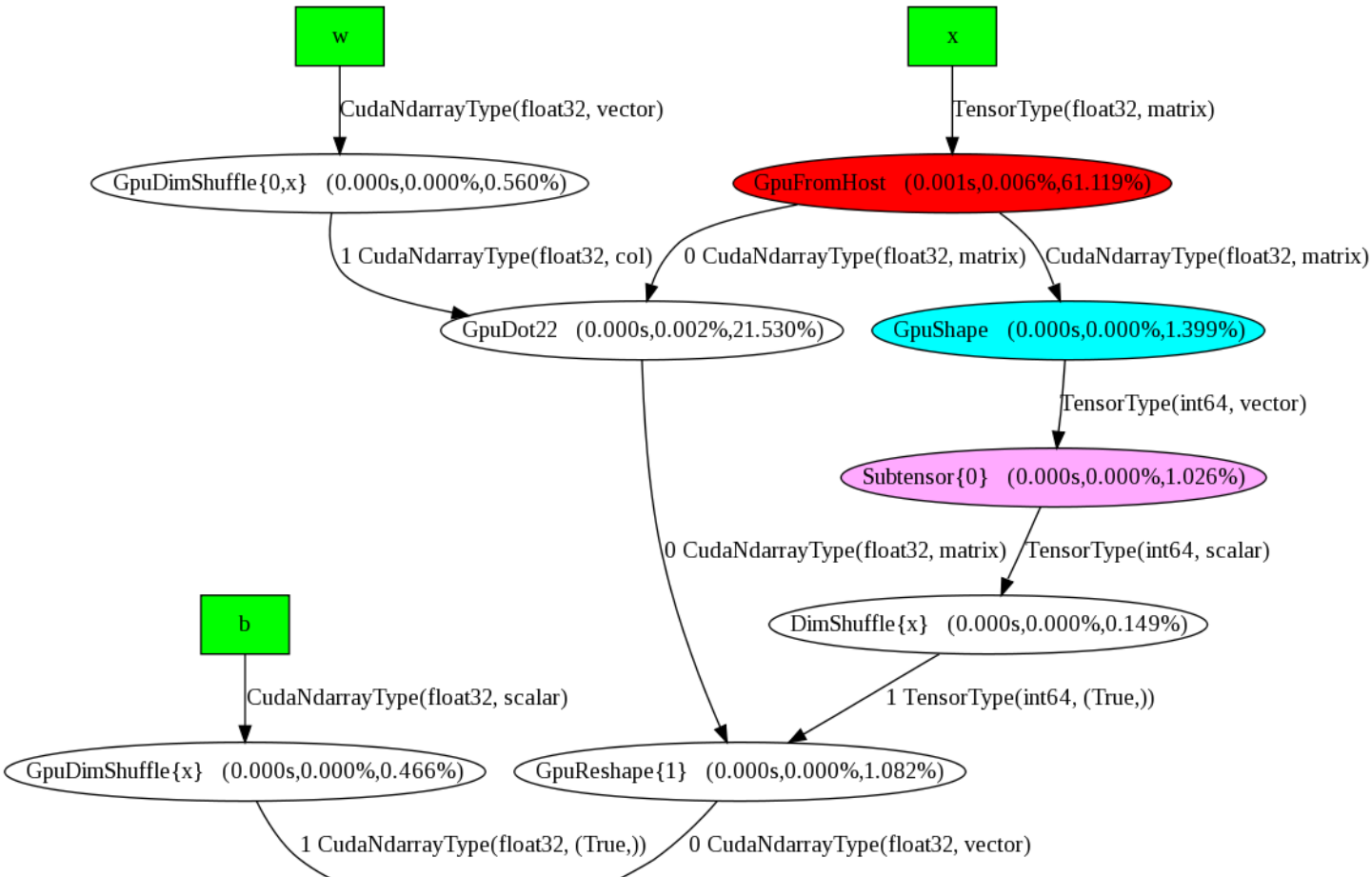
```
>>> theano.printing.pydotprint_variables(prediction)
```



Université de Montréal

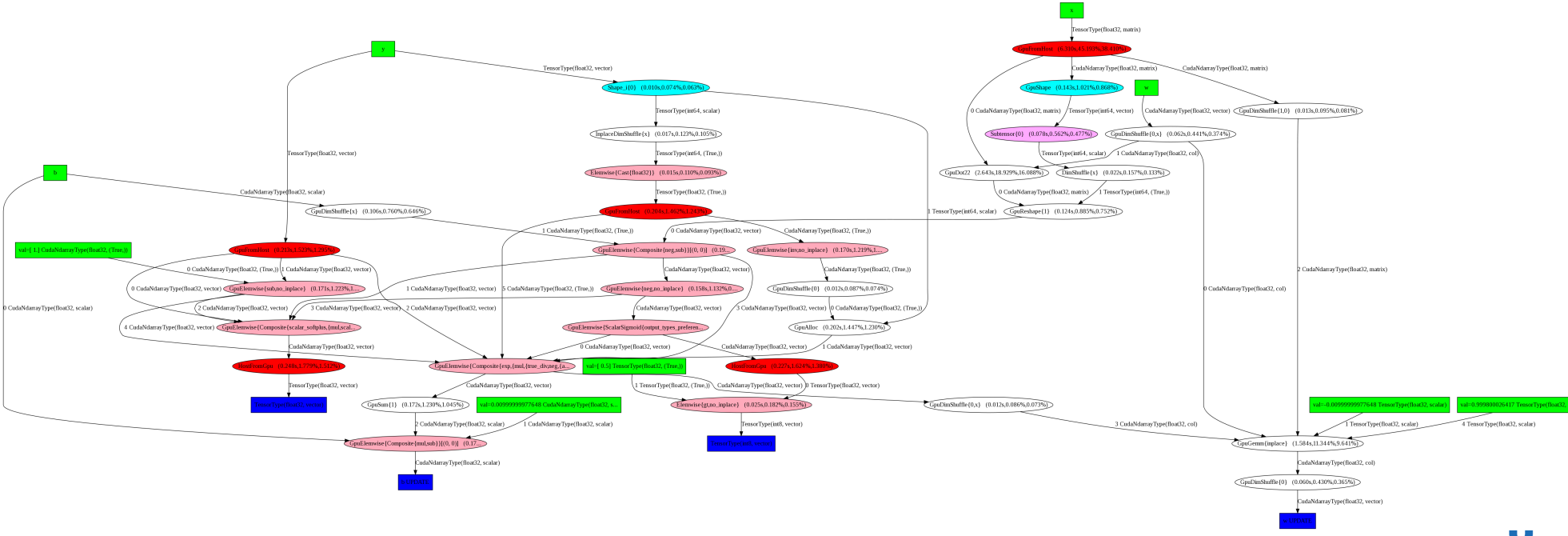
Picture Printing of Graphs

```
All pydotprint* requires graphviz and pydot
>>> theano.printing.pydotprint(predict)
```



Picture Printing of Graphs

```
>>> theano.printing.pydotprint(train) # This is a small train example!
```



How to Debug

- Run with the flag `mode=DebugMode`
  - 100-1000x slower
  - Test all optimization steps from the original graph to the final graph
  - Checks many things that Op should/shouldn't do
  - Executes both the Python and C code versions
- Run with the Theano flag `compute_test_value = ''off'', ''ignore'', ''warn'', ''raise''`
  - Run the code as we create the graph
  - Allows you to find the bug earlier (ex: shape mismatch)
  - Makes it easier to identify where the problem is in *your* code
  - Use the value of constants and shared variables directly
  - For pure symbolic variables uses `x.tag.test_value = numpy.random.rand(5,10)`
- Run with the flag `mode=FAST_COMPILE`
  - Few optimizations
  - Run Python code (better error messages and can be debugged interactively in the Python debugger)

Université de Montréal

Scan

- General form of **recurrence**, which can be used for looping.
- **Reduction** and **map**(loop over the leading dimensions) are special cases of Scan
- You *scan* a function along some input sequence, producing an output at each time-step
- The function can see the **previous K time-steps** of your function
- "sum()" could be computed by scanning the $z + x_i$ function over a list, given an initial state of "z=0".
- Often a for-loop can be expressed as a "scan()" operation, and "scan" is the closest that Theano comes to looping.
- The advantage of using "scan" over for loops
  - The number of iterations to be part of the symbolic graph
  - Minimizes GPU transfers if GPU is involved
  - Compute gradients through sequential steps
  - Slightly faster then using a for loop in Python with a compiled Theano function
  - Can lower the overall memory usage by detecting the actual amount of memory needed

Scan Example: Computing pow(A,k)

```
k = T.iscalar("k"); A = T.vector("A")

def inner_fct(prior_result, A): return prior_result * A
# Symbolic description of the result
result, updates = theano.scan(fn=inner_fct,
                              outputs_info=T.ones_like(A),
                              non_sequences=A, n_steps=k)

# Scan has provided us with A**1 through A**k.  Keep only the last
# value. Scan notices this and does not waste memory saving them.
final_result = result[-1]

power = theano.function(inputs=[A,k], outputs=final_result,
                        updates=updates)

print power(range(10),2)
#[  0.   1.   4.   9.  16.  25.  36.  49.  64.  81.]
```

Université de Montréal

Scan Example: Calculating a Polynomial

```
coefficients = theano.tensor.vector("coefficients")
x = T.scalar("x"); max_coefficients_supported = 10000

# Generate the components of the polynomial
full_range=theano.tensor.arange(max_coefficients_supported)
components, updates = theano.scan(fn=lambda coeff, power, free_var:
                                      coeff * (free_var ** power),
                                  outputs_info=None,
                                  sequences=[coefficients, full_range],
                                  non_sequences=x)
polynomial = components.sum()
calculate_polynomial = theano.function(inputs=[coefficients, x],
                                       outputs=polynomial)

test_coeff = numpy.asarray([1, 0, 2], dtype=numpy.float32)
print calculate_polynomial(test_coeff, 3)
# 19.0
```

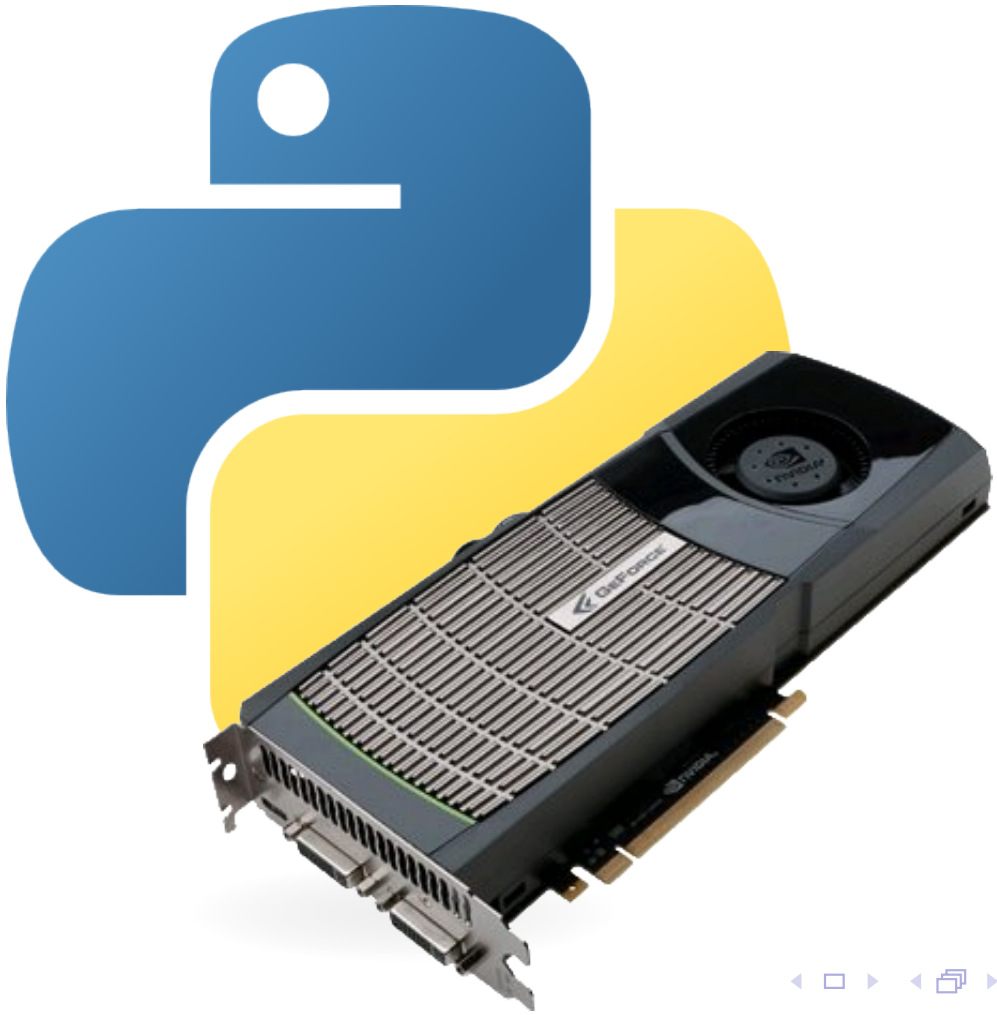
Exercises 5

- Run the example in the file scan_pow.py and scan_poly.py
- Modify and execute the polynomial example to have the reduction done by scan

Known Limitations

- Compilation phase distinct from execution phase
- Compilation time can be significant
  - Amortize it with functions over big input or reuse functions
- Execution overhead
  - Needs a certain number of operations to be useful
  - We have started working on this in a branch
- Compilation time superlinear in the size of the graph.
  - A few hundreds nodes is fine
  - Disabling a few optimizations can speed up compilation
  - Usually too many nodes indicates a problem with the graph
- Lazy evaluation in a branch (We will try to merge this summer)

PyCUDA



Intro

Authors: Andreas Klöckner

- PyCUDA can access Nvidia's CUDA parallel computation API from Python
- Object cleanup tied to lifetime of objects (RAII, Resource Acquisition Is Initialization).
  - Makes it much easier to write correct, leak- and crash-free code
  - PyCUDA knows about dependencies (e.g.. it won't detach from a context before all memory allocated in it is also freed)
- Convenience
  - Abstractions to compile CUDA code from Python: `pycuda.driver.SourceModule`
  - A GPU memory buffer: `pycuda.gpuarray.GPUArray`
- Completeness
  - Binding to all of CUDA's driver API
- Automatic Error Checking
  - All CUDA errors are automatically translated into Python exceptions
- Speed
  - PyCUDA's base layer is written in C++
- Helpful documentation

Example

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")
```

Example

```
multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))
```

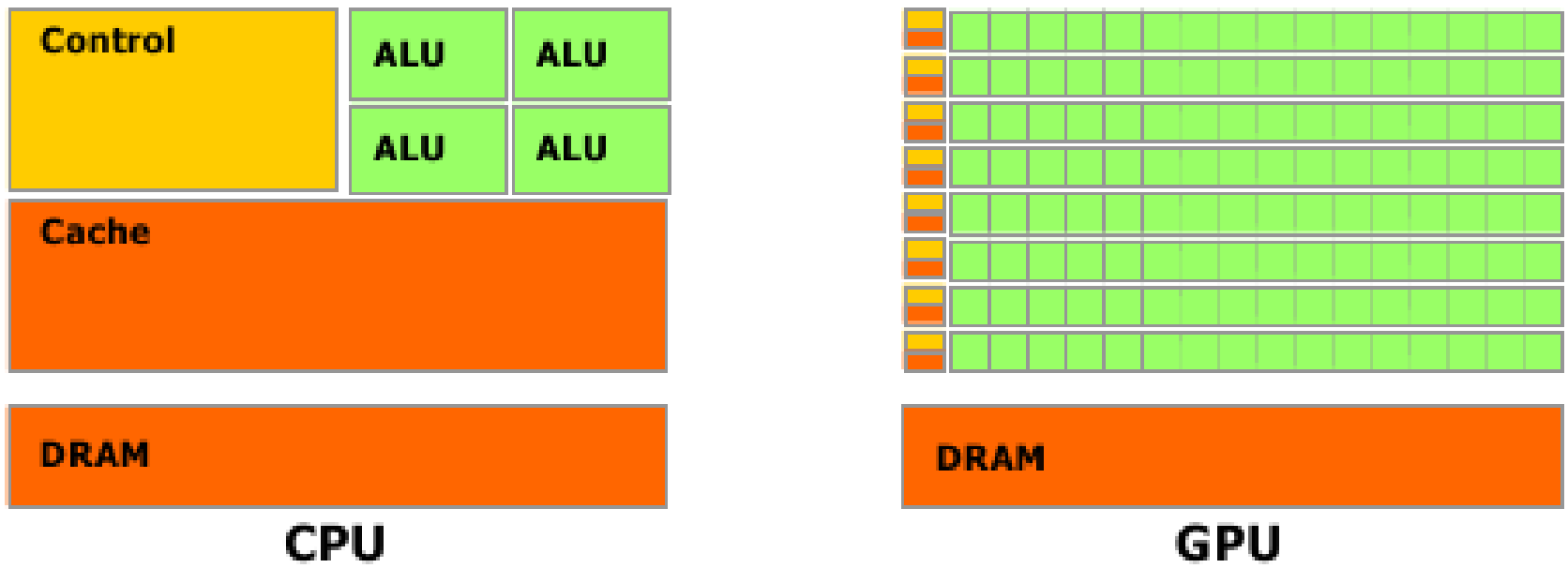
GPU Programming: Gains and Losses

- Gains:
  - Memory Bandwidth (140GB/s vs 12 GB/s)
  - Compute Bandwidth(Peak: 1 TF/s vs 0.1 TF/s in float)
  - Data-parallel programming
- Losses:
  - No performance portability guaranty
  - Data size influence more the implementation code on GPU
  - Cheap branches
  - Fine-grained malloc/free*
  - Recursion*
  - Function pointers*
  - IEEE 754FP compliance*

* Less problematic with new hardware (NVIDIA Fermi)

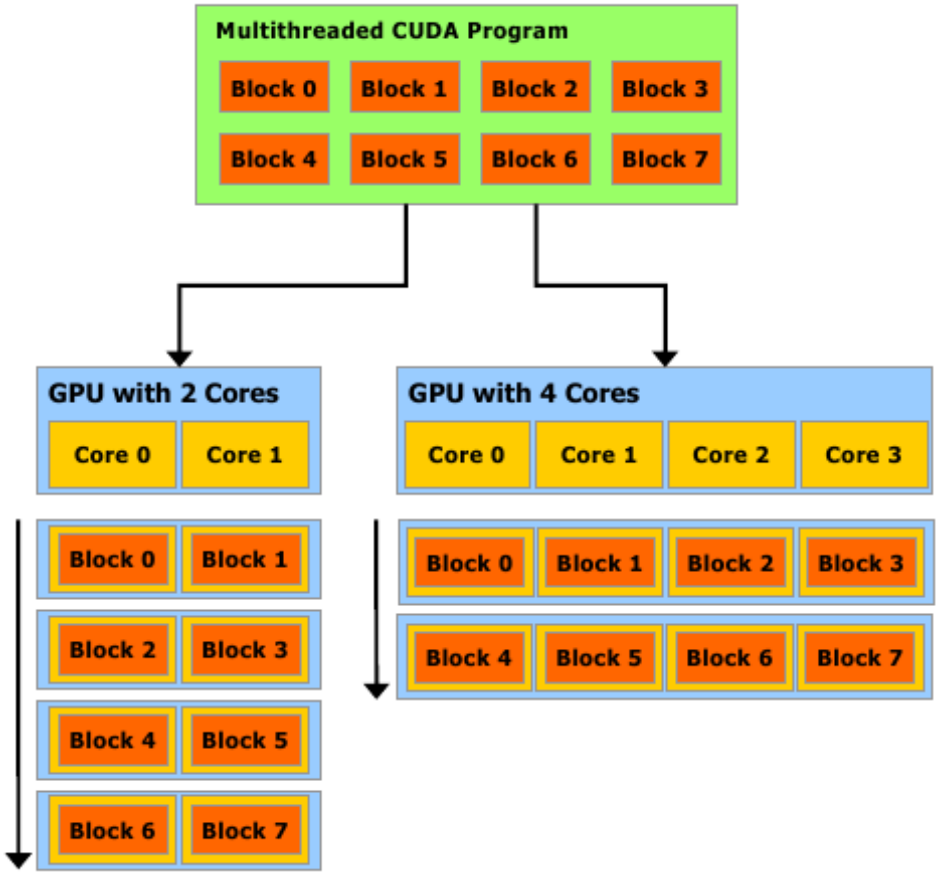
[slide from Andreas Klöckner]

CPU vs GPU Architecture



Source NVIDIA CUDA_C_Programming_Guide.pdf document

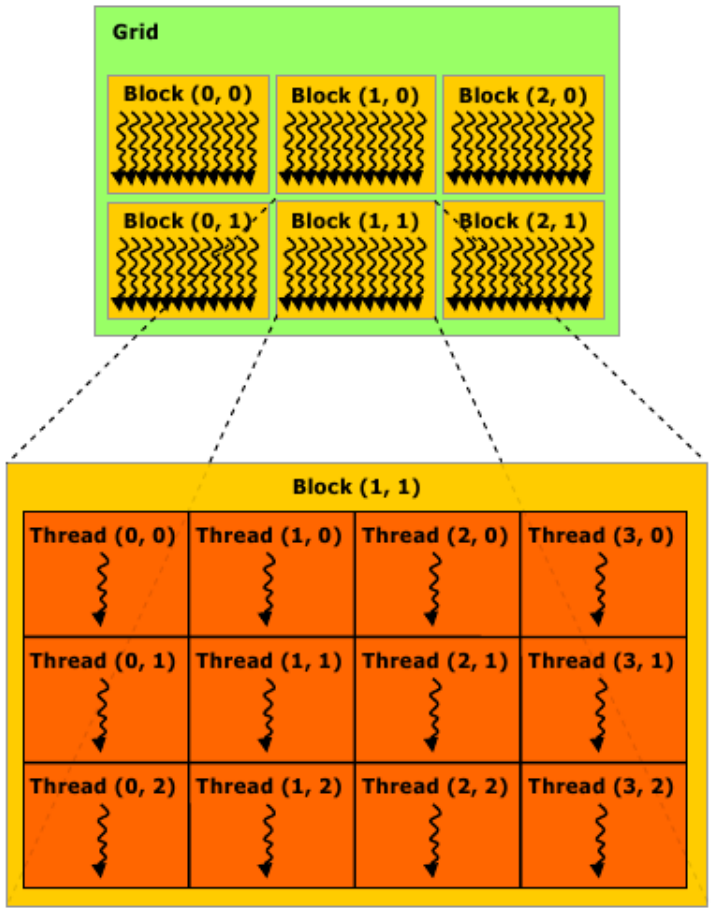
Different GPU Block Repartition



Source NVIDIA CUDA_C_Programming_Guide.pdf document

Université de Montréal

GPU thread structure



Source NVIDIA CUDA_C_Programming_Guide.pdf document

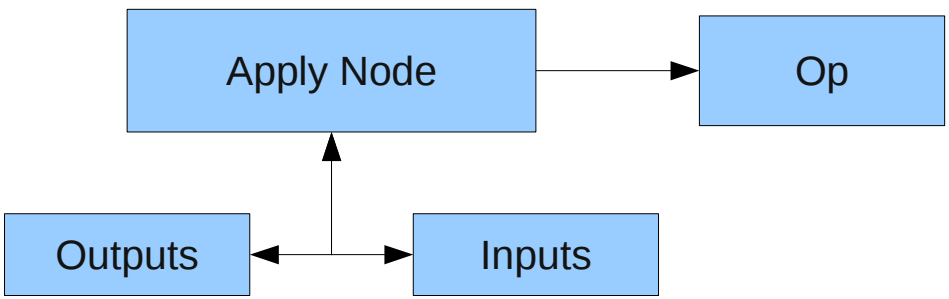
Université de Montréal

Exercises 6

- Run the example in the file pycuda_simple.py
- Modify and execute it to work for a matrix of 20×10

Theano Graph

- Theano works with symbolic graphs
- Those graphs are bi-partite graphs (graph with 2 types of nodes)
- Those 2 nodes types are Apply and Variable nodes
- Inputs and Outputs are lists of Theano variables



Op Contract

```
class MyOp(Op):
    def __eq__(self, other):
    def __hash__(self):
    def __str__(self):
    def make_node(self, *inputs):

# Python implementation:
    def perform(self, node, inputs_storage, outputs_storage):
# C implementation: [see theano web site]
# others implementation (pycuda, ...):
     def make_thunk(self, node, storage_map, _, _2):

# optional:
    def __init__(self, ...):
    def grad(self, inputs, g):
    def infer_shape(node, (i0_shapes, ...))
```

Op Example

```
import theano

class DoubleOp(theano.Op):
    def __eq__(self, other):
        return type(self) == type(other)
    def __hash__(self):
        return hash(type(self))
    def __str__(self):
        return self.__class__.__name__
    def make_node(self, x):
        x = theano.tensor.as_tensor_variable(x)
        return theano.Apply(self, [x], [x.type()])
    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = x * 2
```

Theano Op Example: Test it!

```
x = theano.tensor.matrix()
f = theano.function([x],DoubleOp()(x))

import numpy
inp = numpy.random.rand(5,5)
out = f(inp)
assert numpy.allclose(inp*2, out)
print inp
print out
```

Exercises 7

- Run the code in the file double_op.py.
- Modify and execute to compute: $x * y$
- Modify and execute the example to return 2 outputs: $x + y$ and $x - y$
  - Our current elemwise fusion generate computation with only 1 outputs

Theano+PyCUDA Op Example

```
import numpy, theano
import theano.misc.pycuda_init
from pycuda.compiler import SourceModule
import theano.sandbox.cuda as cuda

class PyCUDADoubleOp(theano.Op):
    def __eq__(self, other):
        return type(self) == type(other)
    def __hash__(self):
        return hash(type(self))
    def __str__(self):
        return self.__class__.__name__
    def make_node(self, inp):
        inp = cuda.basic_ops.gpu_contiguous(
           cuda.basic_ops.as_cuda_ndarray_variable(inp))
        assert inp.dtype == "float32"
        return theano.Apply(self, [inp], [inp.type()])
```

Theano + PyCUDA Op Example: make_thunk

```
def make_thunk(self, node, storage_map, _, _2):
    mod = SourceModule( THE_C_CODE )

    pycuda_fct = mod.get_function("my_fct")
    inputs = [ storage_map[v] for v in node.inputs]
    outputs = [ storage_map[v] for v in node.outputs]
    def thunk():
        z = outputs[0]
        if z[0] is None or z[0].shape!=inputs[0][0].shape:
            z[0] = cuda.CudaNdarray.zeros(inputs[0][0].shape)
        grid = (int(numpy.ceil(inputs[0][0].size / 512.)),1)
        pycuda_fct(inputs[0][0], z[0], numpy.intc(inputs[0][0].size
                   block=(512,1,1), grid=grid)
    return thunk
```

Université de Montréal

Theano + PyCUDA Op Example: GPU Code

```
THE_C_CODE = """
__global__ void my_fct(float * i0, float * o0, int size) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<size){
        o0[i] = i0[i]*2;
    }
}""")
```

Theano + PyCUDA Op Example: Test it!

```
x = theano.tensor.fmatrix()
f = theano.function([x], PyCUDADoubleOp()(x))
xv=numpy.ones((4,5), dtype="float32")

assert numpy.allclose(f(xv), xv*2)
print numpy.asarray(f(xv))
```

Exercises 8

- Run the example in the file pycuda_double_op.py
- Modify and execute the example to multiple two matrix: $x * y$
- Modify and execute the example to return 2 outputs: $x + y$ and $x - y$
  - Our current elemwise fusion generate computation with only 1 outputs
- Modify and execute the example to support stride? (Don't force the input to be c contiguous)

Why a common GPU ndarray?

- Currently there are at least 4 different GPU array data structures in use by Python packages
  - CudaNdarray (Theano), GPUArray (PyCUDA), CUDAMatrix (cudamat), GPUArray (PyOpenCL), ...
  - There are even more if we include other languages
- All of them are a subset of the functionality of `numpy.ndarray` on the GPU
- Lots of duplicated effort
  - GPU code is harder/slower to do **correctly** and **fast** than on the CPU/Python
- Lack of a common array API makes it harder to port/reuse code
- Also harder to find/distribute code
- Divides development work

Design Goals

- Make it VERY similar to `numpy.ndarray`
- Be compatible with both CUDA and OpenCL
- Have the base object accessible from C to allow collaboration with more projects, across high-level languages
  - We want people from C, C++, Ruby, R, ... all use the same base GPU N-dimensional array

Final GpuNdArray Note

- Under development
- Will be the next GPU array container for Theano (this summer!)
- Probably also for PyCUDA, PyOpenCL
- Mailing list: http://lists.tiker.net/listinfo/gpundarray

Conclusion

- I presented a tool that tries to be the holy grail in computing: **easy to code** and **fast to execute**!
- Generates fast, custom CPU code *and* GPU code
- You can easily wrap existing CPU/GPU code with Theano
- It **works** and is **used in the real world** by academic researchers *and* industry

Thanks

- Thanks for attending this tutorial

- Thanks to our agencies that resources for this projects: Calcul Québec, CIFAR, Compute Canada, FQRNT, MITACS, NSERC, SciNet, SHARCNET, Ubisoft and WestGrid.

Questions/Comments?