Department of Computer Science
University of Toronto
http://learning.cs.toronto.edu

6 King's College Rd, Toronto
M5S 3G4, Canada
fax: +1 416 978 1455

November 25, 2009

UTML TR 2009–004

CUDAMat: a CUDA-based matrix class for Python

Volodymyr Mnih
Department of Computer Science, University of Toronto

Abstract

CUDAMat is an open source software package that provides a CUDA-based matrix class for Python. The primary goal of CUDAMat is to make it easy to implement algorithms that are easily expressed in terms of dense matrix operations on a GPU. At present, the feature set of CUDAMat is biased towards providing functionality useful for implementing standard machine learning algorithms, however, it is general enough to be useful in other fields. We have used CUDAMat to implement several common machine learning algorithms on GPUs offering speedups of up to 50x over numpy and MATLAB implementations.

Contents

CUDAMat: a CUDA-based matrix class for Python

Volodymyr Mnih
Department of Computer Science, University of Toronto

1 Introduction

In the past few years GPUs have far surpassed the computational capabilities of CPUs for floating point arithmetic. In the field of machine learning, early adopters of GPUs have reported speedups of well over an order of magnitude for several popular machine learning algorithms [2]. However, adoption of GPUs for machine learning research has been somewhat slow. Given that many machine learning algorithms are easily expressed in terms of dense linear algebra, making them particularly easy to parallelize, this is somewhat surprising.

Based on our communications with other machine learning researchers, the perceived difficulty of programming GPUs appears to be a big hurdle to their widespread use. Tools such as AccelerEyes Jacket [1], which adds nearly transparent GPU support to the popular MATLAB environment, have made it much easier, however, there is currently a lack of free and open source tools for programming GPUs that do not require a low-level understanding of GPU hardware. CUDAMat aims to provide a free, open source alternative to tools such as Jacket, with the main goal being ease of use.

2 Overview of CUDAMat

The CUDAMat library is available as open source software under the New BSD License. The library can be obtained at `http://code.google.com/p/cudamat/` along with installation instructions. The remainder of this section provides a brief overview of the features of CUDAMat.

2.1 Initialization and shutdown

CUDAMat must be initialized before any operations can be performed on the GPU. On machines with a single CUDA-capable device it is enough to call the *init* method. If more than one CUDA-capable device is present, a device should be selected by calling the *cuda_set_device* method with the appropriate device id. It is important to call the *shutdown* method at the end of your program in order to avoid unwanted behavior.

```
import cudamat as cm

cm.cuda_set_device(0)
cm.init()

# Perform computations on GPU.

cm.shutdown()
```

2.2 The *CUDAMatrix* class

The *CUDAMatrix* class represents a matrix of single-precision floats stored on the GPU. Similarly to *ndarray* class from *numpy* a *CUDAMatrix* instance corresponds to a contiguous one-dimensional region of memory. The memory layout of a *CUDAMatrix* is always column-major, because this is the layout required by routines from the CUBLAS library.

2.2.1 Creating matrices on the GPU

There are two ways of creating a matrix on the GPU. The first is by calling the *empty* method with a tuple of length two specifying the shape of the matrix. This will create an empty matrix with the given shape on the GPU. The second way of creating a CUDAMatrix instance involves copying a *numpy ndarray* object to the GPU. This is accomplished by instantiating a new CUDAMatrix object with an *ndarray* object as the argument.

```
import numpy as np

# Create an empty array with 10 rows and 20 columns on the GPU.
empty_array = cm.empty((10, 20))

# Create a numpy array and create a CUDAMatrix instance from it.
cpu_array = np.random.rand(10, 10)
gpu_array = cm.CUDAMatrix(cpu_array)
```

2.2.2 Copying matrices to and from the GPU

There are several ways of copying the contents of a *CUDAMatrix* to CPU memory. One way to do this is by calling the *asarray* method of any matrix residing on the GPU. This will copy its contents to an *ndarray* in CPU memory and return the *ndarray* object. Another way of copying data from the GPU to the CPU is by calling the *copy_to_host* method of a GPU matrix. This method copies the contents of the GPU matrix to an *ndarray* instance that is bound to the *numpy_array* attribute of the GPU matrix.

```
# Create a matrix of random numbers on the GPU.
gpu_array = cm.CUDAMatrix(np.random.rand(10, 10))

# Approach 1: Copy the contents of gpu_array to an ndarray return it.
cpu_array = gpu_array.asarray()

# Approach 2: Copy the contents of gpu_array to an ndarray
gpu_array.copy_to_host()
print gpu_array.numpy_array # print ndarray
```

In some cases one may want to modify a matrix on the CPU and copy the results back to the GPU. This can be accomplished by performing *in place* modifications to the *numpy_array* attribute and calling the *copy_to_device* method. Note that in some cases a *CUDAMatrix* instance may not have a *numpy_array* attribute (if, for example, it was created using the *empty* method) so one may need to call the *copy_to_host* method to create it.

```
# Create a matrix of random numbers on the GPU.
gpu_array = cm.CUDAMatrix(np.random.rand(10, 10))
```

```
# Copy contents of gpu_array to CPU, modify the matrix, and copy the modified
# version back to the GPU.
gpu_array.copy_to_host()
gpu_array.numpy_array += 1.
gpu_array.copy_to_device()
```

2.2.3 Accessing and modifying matrix dimensions

All *CUDAMatrix* instances have a *shape* property, which is a tuple of length two specifying the shape of the matrix that the instance should be interpreted to have. Sometimes, it may be convenient to change the shape of a matrix and this can be accomplished by calling the *reshape* method with a new shape. Note that the *reshape* method only changes the shape that the matrix is interpreted to have and does not move any data.

```
# Create an empty array with 10 rows and 20 columns on the GPU.
empty_array = cm.empty((10, 20))

# Reshape empty_array.
empty_array.reshape((5, 40))
```

2.3 Matrix operations

2.3.1 Assignment

The *assign* method of the *CUDAMatrix* class takes a single argument that can be a scalar or a *CUDAMatrix* instance. If the argument is a scalar it is assigned to every element of the GPU matrix. If the argument is a *CUDAMatrix* instance of the same size its contents are assigned to the contents of the GPU matrix.

```
# Create a matrix of zeros on the GPU.
zeros = cm.empty((10, 20))
zeros.assign(0)

empty_array = cm.empty((10, 20))
random_array = cm.CUDAMatrix(np.random.rand(10, 20))

# Assign the contents of random_array to the contents of empty_array.
empty_array.assign(random_array)
```

2.3.2 Basic algebraic operations

The *CUDAMatrix* class supports elementwise addition, subtraction, multiplication, and division between two matrices or a matrix and a scalar. The *add*, *subtract*, *mult*, *divide* methods of the *CUDAMatrix* class each take an argument named *val* and an optional argument named *target*. If *val* is a scalar a matrix/scalar operation is performed, if *val* is a *CUDAMatrix* instance an elementwise matrix/matrix operation is performed. If *target* is provided it is used to store the result, otherwise it is stored in the matrix whose method was called.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
```

```
B = cm.CUDAMatrix(np.random.randn(10, 20))
T = cm.CUDAMatrix(np.random.randn(10, 20))

# Add A and B and store the result in T.
A.add(B, target = T)

# Multiply A by 42.
A.mult(42)
```

2.3.3 Mathematical functions

The *cudamat* module class has a number of routines for applying various mathematical functions to each element of a *CUDAMatrix* instance. All of these functions take an optional *target* argument that is used to store the result if provided. If no target is provided a new GPU matrix is allocated to store the result.

Function	Description
exp(mat, target)	Elementwise exp.
log(mat, target)	Elementwise natural logarithm.
sqrt(mat, target)	Elementwise square root.
pow(mat, p, target)	Take every element of mat to the power of p.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
T = cm.CUDAMatrix(np.random.randn(10, 20))

# Apply the exp function to each element of A and store the result in T.
cm.exp(A, target = T)

# Apply the log function to each element of A and assign the result to B.
B = cm.log(A)
```

2.3.4 Slicing operations

Instances of *CUDAMatrix* have limited slicing support. Since matrices are stored in column-major order CUDAMat allows one to obtain a view of a slice of columns without copying any data. Obtaining a slice of some rows of a matrix is also possible, but currently requires copying data. Slicing is done with the *get_col_slice* and *get_row_slice* methods, which take the indices of the first (inclusive) and last (non-inclusive) column/row in the slice as well as an optional target matrix. Assigning to a column/row slice can be done using the *set_col_slice* and *set_row_slice* methods.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
col_slice = cm.empty((10, 5))

# Obtain a view into A (no copying is performed).
view = A.get_col_slice(5, 10)

# Obtain the same slice as a copy.
A.get_col_slice(5, 10, col_slice)
```

```
# Get a row slice (copying is performed).
row_slice = A.get_row_slice(0, 5)

# Assign to a row slice.
A.set_row_slice(5, 10, row_slice)
```

2.3.5 Matrix transpose

GPU matrices can be transposed using the *transpose* method. This method takes an optional target parameter and returns a transposed copy of the matrix whose method was called.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
T1 = cm.CUDAMatrix(np.random.randn(20, 10))

# Put transpose of A into T1.
A.transpose(target = T1)

# Get transpose of A (new matrix is created).
T2 = A.transpose()
```

2.3.6 Matrix multiplication

The *cudamat* module provides the *dot* function for performing matrix multiplication. One often needs to transpose a matrix just to use it in a matrix multiplication and the *dot* function makes this easy. If A and B are instances of *CUDAMatrix* then one can pass $A.T$ and/or $B.T$ to *dot* to use the transposed matrices in a matrix multiplication. Since *dot* uses the CUBLAS library it is able to use the transpose of one or both matrices without explicitly computing and storing it. The *dot* method takes an optional target parameter. If a target is not provided, a new matrix will be created on the GPU to store the result.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
B = cm.CUDAMatrix(np.random.randn(20, 30))
C = cm.CUDAMatrix(np.random.randn(30, 20))
target = cm.empty((10, 30))

# Multiply A and B putting the result in target.
cm.dot(A, B, target)

# Multiply A and the transpose of C.
result = cm.dot(A, C.T)
```

2.3.7 Summing along an axis

Instances of *CUDAMatrix* have a *sum* method that sum a matrix along a given axis. As in *numpy* summing along axis 0 means summing along the columns, while summing along axis 1 means summing along the rows. The *sum* method takes an optional target parameter. If a target is not provided, a new matrix will be created on the GPU to store the result.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
col_sums = cm.CUDAMatrix(np.random.randn(1, 20))
row_sums = cm.CUDAMatrix(np.random.randn(10, 1))

# Sum along rows and columns.
A.sum(axis = 0, target = col_sums)
A.sum(axis = 1, target = row_sums)
```

2.4 Matrix-vector operations

CUDAMat supports a number of basic matrix-vector operations. It is possible to add a row or column vector to a matrix using the *add_col_vec* and *add_row_vec* methods. These methods add a column or a row to every column or row of a matrix respectively. Similarly, it is possible to multiply every column or row of a matrix by a given column or row vector using the *mult_by_col* and *mult_by_row* methods. All of these methods take an optional target parameter.

```
# Create some matrices on the GPU.
A = cm.CUDAMatrix(np.random.randn(10, 20))
row = cm.CUDAMatrix(np.random.randn(1, 20))
col = cm.CUDAMatrix(np.random.randn(10, 1))

# Add row to every row of A.
A.add_row_vec(row)

# Multiply every column of A by col.
A.mult_by_col(col)
```

2.5 Random number generation

CUDAMat currently provides support for sampling from uniform and Gaussian distributions. Before random numbers can be generated one must seed the random number generator (at present multiply-with-carry is used) by calling the *init_random* static method of the *CUDAMatrix* class. The *init_random* method takes an optional *seed* parameter, which is 0 by default. Once initialized, a matrix can be filled with samples from a $Uniform(0, 1)$ distribution by calling its *fill_with_rand*. Similarly, a matrix can be filled with samples from a standard normal distribution by calling its *fill_with_randn* method.

```
# Create some matrices on the GPU.
uniform_samples = cm.CUDAMatrix(np.random.randn(10, 20))
normal_samples = cm.CUDAMatrix(np.random.randn(10, 20))

# Initialize and seed the random number generator.
cm.CUDAMatrix.init_random(seed = 42)

# Generate random numbers.
uniform_samples.fill_with_rand()
normal_samples.fill_with_randn()
```

References

[1] AccelerEyes. Jacket, May 2009. `http://www.accelereyes.com`.

[2] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880. ACM, 2009.