White Paper

Accelerating MATLAB with CUDA™ Using MEX Files

September 2007
WP-03495-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
01	September 11, 2007	KV, NSmith	Initial release

Accelerating MATLAB with CUDA Using MEX Files

This whitepaper illustrates the use of MEX files to perform computations on the GPU using CUDA. MATLAB provides a script, called mex, to compile a MEX file to a shared object or dll that can be loaded and executed inside a MATLAB session. This script is able to parse C, C++ and FORTRAN code. This script can call CUDA code to create greater optimization on GPUs.

MEX files are a powerful tool to call code written in C or FORTRAN. Even though MATLAB is calling optimized libraries internally, there is still room for further optimization. MEX files have been used in the past as a way to invoke multithreaded or vectorized libraries. This document describes a technique to use MEX files to invoke code on the GPU and to handle the data transfer between the host and GPU.

Regular MEX Files

Before demonstrating CUDA code, this section reviews MEX file. All MEX files must include 4 items:

1. #include mex.h (for C and C++ MEX-fles)

2. The gateway routine to every MEX-file is called mexFunction.

   This is the entry point MATLAB uses to access the DLL or .so. In C/C++, it is always:

   mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ]) { . }

   where

   nlhs = number of expected mxArrays (Left Hand Side)
   plhs = array of pointers to expected outputs
   nrhs = number of inputs (Right Hand Side)
   prhs = array of pointers to input data. The input data is read-only

3. mxArray:
   The mxArray is a special structure that contains MATLAB data. It is the C representation of a MATLAB array. All types of MATLAB arrays (scalars, vectors, matrices, strings, cell arrays, etc.) are mxArrays.

4. API functions (like memory allocation and free).

The following code shows a simple MEX file that squares the input array.

Code Sample 1. Listing of MEX file square_me.c

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[])
{
int i, j, m, n;
double *data1, *data2;

/* Error checking */
if (nrhs != nlhs) mexErrMsgTxt
("The number of input and output arguments must be the same.");

for (i = 0; i < nrhs; i++)

/* Find the dimensions of the data */
 m = mxGetM(prhs[i]);
 n = mxGetN(prhs[i]);

/* Create an mxArray for the output data */
 plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);

/* Retrieve the input data */
 data1 = mxGetPr(prhs[i]);

/* Create a pointer to the output data */
 data2 = mxGetPr(plhs[i]);

/* Put data in the output array after squaring them */
 for (j = 0; j < m*n; j++)
 {
  data2[j] = data1[j] * data1[j];
 }
}
}
```

From a MATLAB prompt (or in Linux, from a generic shell), execute the command

```
>> mex square_me.c
```

This command generates a compiled MEX file. (The suffix produced depends on the operating system)

```
square_me.mexw32 (on Windows 32 bit)
square_me.mexglx (on Linux 32 bit)
square_me.mexa64 (on Linux 64 bit)
```

Giving the same name of the original one allows programmers to overload the standard implementation with the accelerated one.

Use the which command to verify that you are calling the compiled version. This command is very useful when the compiled and accelerated MEX file has the name of the native version.

```
>> which square_me
       $pwd/square_me.mexglx

>> a = linspace(1,10,10)
   a=
       1  2  3  4  5  6  7  8  9  10
>> a2 = square_me(a)
   a2=
       1  4  9  16  25  36  49  64  81  100
```

For MATLAB to be able to execute your C functions, you must either put the compiled MEX files containing those functions in a directory on the MATLAB path or run MATLAB in the directory in which they reside. Functions in the current working directory are found before functions on the MATLAB path. Type path to see what directories are currently included in your path. You can add new directories to the path either by using the addpath function, or by selecting File > SetPath to edit the path.

For more information on the memory management in MEX files, we suggest the MATLAB documentation located at:

External Interface > Creating C-Language Mex-Files > Advanced Topics > Memory Management

CUDA-Enabled MEX Files

For most general cases, the MEX file will contain CUDA code (kernel functions, configuration) and in order to be processed by "nvcc", the CUDA compiler, the file needs to have a ".cu" suffix. This suffix .cu could not be parsed from the native MATLAB mex script. To solve this problem, NVIDIA developed new scripts that are able to parse these kind of files.

In particular cases, it is still possible to write standard C code to process data on the GPU, especially when the operations performed on the data could be expressed in terms of calls to CUFFT or CUBLAS functions and use the regular mex infrastructure.

The package downloadable from the NVIDIA web site[1] contains a new script (called nvmex) and a configuration option file for the underlying compiler that simplify the deployment of CUDA based MEX files.

The operation of the mex command is also different between Windows and Linux. In Windows, it invokes a perl script, while in Linux it invokes a shell script. For this reason, there are two different packages on the NVIDIA web site. Please follow the README in the distribution file specific to your OS for the setup.

In order to create a compiled MEX file from a .cu file, the command in MATLAB is:

```
nvmex -f nvmexopts.bat filename.cu -IC:\cuda\include -LC:\cuda\lib
-lcudart
```

The customized version passes the location of the include files (with the -I switch), the location of the CUDA runtime library (with the -L switch), and the name of the libraries needed (cudart, the CUDA runtime library, is mandatory)

[1] Available from http://developer.nvidia.com/object/matlab_cuda.html

The basic CUDA MEX files contain the following parts:

1. Memory allocation on the GPU.

2. Data movement from the host memory to the GPU. It is important to remember that the standard floating point format in MATLAB is double precision, while the current CUDA release and the underlying GPU hardware support only single precision. Before transfer to the card, if the incoming data is in double precision, the data need to be cast to single precision. The example fft_cuda_sp_dp.c shows how to probe if the input data is in single or double precision format.

3. Currently, it is not possible to use "pinned" memory on the host, because the data on the host side needs to be allocated with the special mxMalloc function.

4. Once the data is on the GPU, it is now ready to be processed by CUDA code (custom kernel functions or CUBLAS/CUFFT functions).

5. Data movement from the GPU to host. Once again, data may need to be converted back to double precision

6. Clean-up of the memory allocated on the GPU.

Code Sample 2 shows the CUDA implementation of the square_me code shown in Code Sample 1.

Code Sample 2. Listing of MEX file square_me.c

```
#include "cuda.h"
#include "mex.h"

/* Kernel to square elements of the array on the GPU */
__global__ void square_elements(float* in, float* out, int N)
{
 int idx = blockIdx.x*blockDim.x+threadIdx.x;
 if ( idx < N) out[idx]=in[idx]*in[idx];
}

/* Gateway function */
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
 int i, j, m, n;
 double *data1, *data2;
 float *data1f, *data2f;
 float *data1f_gpu, *data2f_gpu;
 mxClassID category;
 if (nrhs != nlhs)
 mexErrMsgTxt("The number of input and output arguments must be the same."
```

```
for (i = 0; i < nrhs; i++)
   {
    /* Find the dimensions of the data */
    m = mxGetM(prhs[i]);
    n = mxGetN(prhs[i]);

    /* Create an mxArray for the output data */
    plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);

    /* Create an input and output data  array on the GPU*/
    cudaMalloc( (void **) &data1f_gpu,sizeof(float)*m*n);
    cudaMalloc( (void **) &data2f_gpu,sizeof(float)*m*n);

    /* Retrieve the input data */
    data1 = mxGetPr(prhs[i]);

    /* Check if the input array is single or double precision */
    category = mxGetClassID(prhs[i]);

    if( category == mxSINGLE_CLASS)
    {
    /* The input array is single precision, it can be sent directly to the
card */
cudaMemcpy( data1f_gpu, data1, sizeof(float)*m*n,
cudaMemcpyHostToDevice);

    }

    if( category == mxDOUBLE_CLASS)
    {
      /* The input array is in double precision, it needs to be converted t
floats before being sent to the card */
       data1f = (float *) mxMalloc(sizeof(float)*m*n);
       for (j = 0; j < m*n; j++)
        {
          data1f[j] = (float) data1[j];
        }
cudaMemcpy( data1f_gpu, data1f, sizeof(float)*n*m, cudaMemcpyHostToDevice)
    }

       data2f = (float *) mxMalloc(sizeof(float)*m*n);

   /* Compute execution configuration using 128 threads per block */
     dim3 dimBlock(128);
     dim3 dimGrid((m*n)/dimBlock.x);

     if ( (n*m) % 128 !=0 ) dimGrid.x+=1;
```

```
/* Call function on GPU */
 square_elements<<<dimGrid,dimBlock>>>(data1f_gpu, data2f_gpu, n*m);

 /* Copy result back to host */
 cudaMemcpy( data2f, data2f_gpu, sizeof(float)*n*m,  cudaMemcpyDeviceToHos

 /* Create a pointer to the output data */
 data2 = mxGetPr(plhs[i]);

 /* Convert from single to double before returning */
  for (j = 0; j < m*n; j++)
   {
           data2[j] = (double) data2f[j];
   }

 /* Clean-up memory on device and host */
  mxFree(data1f);
  mxFree(data2f);
  cudaFree(data1f_gpu);
  cudaFree(data2f_gpu);

 }
}
```

Sample Application – Vortex Dynamics

The examples used in this whitepaper are from a class on atmospheric research at the University of Washington[2] and are included in the package with the nvmex script[3]. The MATLAB scripts solve the Euler equation in vorticity-stream function using a pseudo-spectral method. Pseudo-spectral methods are very well conditioned and some operations can be safely performed in single precision without affecting the overall quality of the solution.

The MATLAB code can be easily modified to solve problems with different initial conditions or forcing, for example, to study the evolution of an elliptic vortex or 2D isotropic turbulence. The code is heavily FFT based. All the results were obtained with MATLAB R2006B, both under Windows and Linux.

NVIDIA performed the work in two steps. The first was to write MEX files for the FFT2 and IFFT2 functions in MATLAB calling the CUFFT library. The second was to port the function computing the non-linear term of the Euler equation to CUDA. The MATLAB code is running in double-precision and the data is transformed to single-precision before it is transferred to the GPU. The computation on the GPU is performed in single precision and the result is transformed back to double precision before it is returned to MATLAB.

NVIDIA uses 2 different systems for this benchmark. The first one (system A) has an Opteron 250 processor (2.4 GHz). The second one (system B) has an Opteron 2210 (dual-core 1.8 GHz). Both systems are using a Quadro FX5600. Aside from the clock speed, the systems have a different bandwidth through the PCI Express bus. Using the SDK example, "BandwithTest", we can measure the chipset performances: system A is able to sustain 1,135 MB/s in write (transfer to the GPU) and 1,003 MB/s in read (transfer from the GPU); system B is able to sustain 1,483 MB/s in write, and 1,223 MB/s in read.

Table 1 shows the timing details for 400 Runge-Kutta steps on a 1024×1024 mesh for 2D isotropic turbulence simulation on Windows.

Table 1. Timing Details

	Runtime Opteron 250	Speed-up	Runtime Opteron 2210	Speed-up
Standard MATLAB	8,098 s		9,525 s	
Overload FFT2 and IFFT2	4,496 s	1.8 x	4,937 s	1.9x
Overload Non-linear term	735 s	11 x	789 s	12x
Overload Non-linear term, FFT2 and IFFT2	577 s	14 x	605 s	15.7x

---

[2] Available from http://www.amath.washington.edu/courses/571-winter-2006/matlab

[3] Available from http://developer.nvidia.com/object/matlab_cuda.html

The CUDA code is not yet taking advantage of the symmetries of real transforms (the original code was written when CUFFT was only supporting complex to complex transforms). The speed-up will increase even further by using real-to-complex and complex-to-real transforms where only half the data needs to be transferred and computed.

Figure 1 and Figure 2 compare the final results of the original MATLAB implementation with the one accelerated with CUDA on system A. The stream function and vorticity fields are the same, but the elapsed time drops from 168 seconds to 14.9 seconds (11 times faster).

These results were performed on Linux.

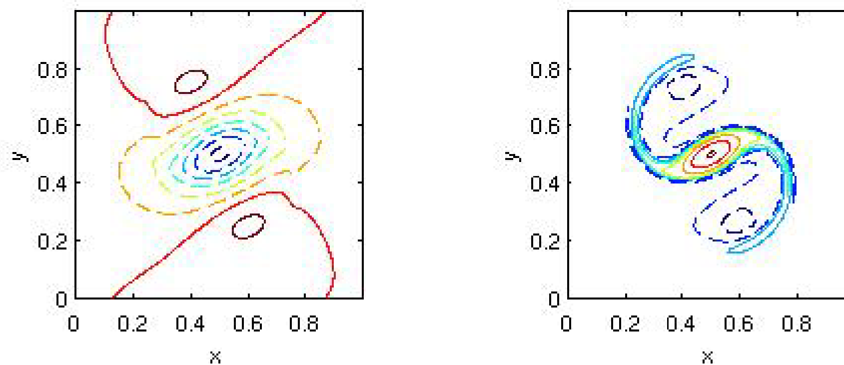

Figure 1. Final Results Using MATLAB

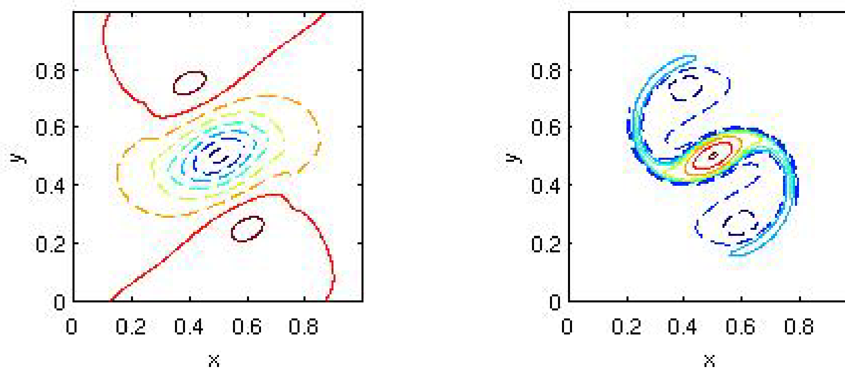

Figure 2. Final Results Using MATLAB with CUDA

Using MATLAB on Linux, the results for the computation of the advection of an elliptic vortex on a 256×256 mesh, stream function (left) and vorticity (right) in Figure 1 required 168 seconds. By contrast, the results using MATLAB with CUDA in Figure 2 required only 14.9 seconds.

For a better comparison of the quality of the results, we ran a 2D isotropic turbulence simulation compared the vorticity power spectra of the different runs. The first used the original MATLAB code (Figure 3) and the second used MATLAB accelerated with CUDA code (Figure 4). Even for this quantity, that is very sensitive to fine scales, the results are in close agreement.

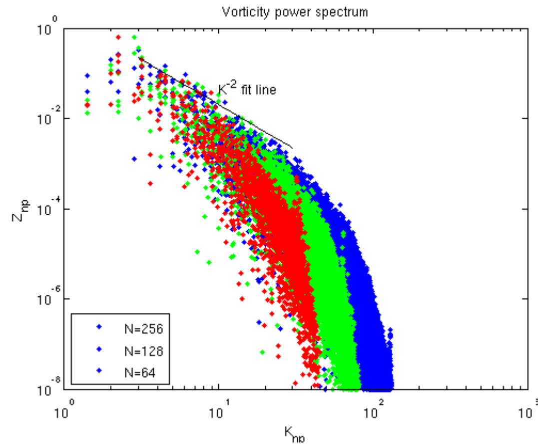

Figure 3. Final Results Using MATLAB

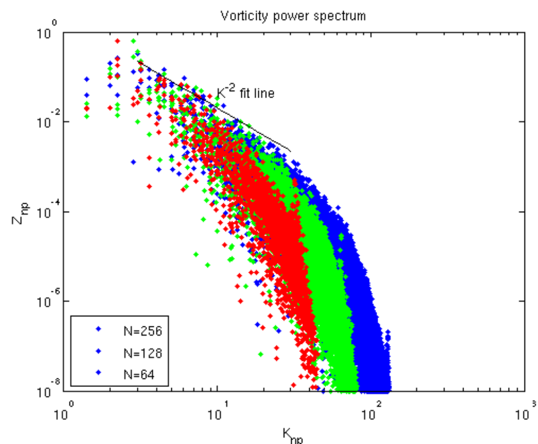

Figure 4. Final Results Using MATLAB with CUDA

Conclusion

The combination of MATLAB and CUDA enables high-productivity and high-performance solutions and with GeForce 8 series GPUs now available even in laptops, will be a very effective tool for engineers and scientists.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, and GeForce are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007 NVIDIA Corporation. All rights reserved.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com