Matlab and CUDA

Brian Dushaw
Applied Physics Laboratory, University of Washington
Seattle, WA USA
email: dushaw AT apl.washington.edu

February 12, 2010

Contents

Introduction

The latest generation of high-end video cards offer considerable computing power using their 100–200 on-card processors, 0.3–1.0+ GB of RAM, and fast inter-processor communications. One promising application of this Graphics Processing Unit (GPU) computing capability is through Matlab and Matlab mex functions. With a properly developed mex function, the user-friendly Matlab interface can be used to perform behind-the-scenes parallel computations on the GPU. The GPU becomes a co-processor for the personal computer. My particular matrix algebra intensive calculation on an 8800GT card gave a factor of ×14 speed up in single precision calculations against a single processor of a 2.3 GHz AMD Phenom quadcore CPU (a factor of ×3.5 against all four cores).

"Tesla" variety devices are available as compute-only devices that eliminate all graphics capability and include additional memory. While such devices may be slightly more reliable than the desktop graphics cards and allow larger computational problems, they do not usually offer faster computation, since they both use the same processor design. Nvidia plans to release the next generation CUDA architecture, “Fermi”, in spring 2010. Fermi devices will feature up to 512 CUDA cores, ECC memory, and offer double precision capabilities 8× faster than existing devices. Fermi based Tesla S2050 and S2070 will also offer display output.

While the "Message Passing Interface" (MPI) implements a means of parallel computing on a cluster of PC's, Nvidia's "Compute Unified Device Architecture" (CUDA) implements a means of computing on the large numbers of processors of a GPU. The former is employed for complicated computations, but is often limited by inter-computer communications, while the latter is employed for massive numbers of simple computations using the fast communication between many processors. Both approaches have their strengths and weaknesses; one does not replace the other. The matrix-intensive computations that occur in Matlab appear to be ideally suited to GPU computation.

Computation on a GPU is basically a three step process: (1) Copy data to the GPU memory, (2) Execute code (the "kernel") to process that data, and (3) Copy the results back from the GPU memory. In general, code should be designed to minimize steps (1)

and (3), which frequently limit the overall speed of the calculations. CUDA calculations usually start to beat ordinary CPU calculations for large-sized problems, e.g., matrices of sizes 1000 or so. It should be noted that this sort of computation is still in its infantcy; it appears to me that things are still a little raw, but developing rapidly.

This is a simple zero-order tutorial I've compiled from my experience in getting started with using the CUDA system of computing on Nvidia's newer graphics cards together with Matlab. Many issues are beyond the purposely-limited scope of this document. The learning curve was rather steep for me, so perhaps these notes will be helpful to get the novice reader started a little easier. For me, at least, the issue of getting CUDA mex functions to work became intertwined with the issue of getting generic mex functions to work. Hence, this document partly addresses generic mex file properties/problems. I am generally unfamiliar with coding in (void*)C, hence that was/is another bumpy road for me. CUDA is centered on a specialized C compiler (nvcc) that translates (mostly ordinary) C code to something that will run on the GPU.

The descriptions here are based on my experience in computing in the Linux environment (Suse Linux, to be precise), but perhaps they will be useful for other environments as well. It is assumed the reader is familiar with Matlab and mex files.

For further information, the software packages, and additional documentation, Google-search CUDA and rummage through the Nvida CUDA website. The Nvidia CUDA forums can be helpful, although there is a mix of C/CUDA Olympians and desperately lost novices (e.g., yours truly) there.

Installation and configuration: CUDA

To install CUDA and use it effectively you need to install three "packages" named something like:

1. NVIDIA-Linux-x86_64-190.53-pkg2.run
2. cudatoolkit_2.3_linux_64_suse11.1.run
3. cudasdk_2.3_linux.run

These files are from the 2.3 release of CUDA for linux.

The first of these is the "ordinary" Nvidia graphics card driver with CUDA support. All newer Nvidia drivers are meant to include CUDA support by default, for those graphics cards that support CUDA. The second of these is the Toolkit, as you see, which includes the compiler (nvcc) and libraries necessary to compile and run CUDA applications. The third of these is the Software Development Kit (SDK) that includes a

number of examples of how to code CUDA routines and a number of testing routines.

Driver

To install the Nvidia driver one must first stop X-Windows. As root, I execute *"init 3"* which stops X-Windows and puts the computer at runlevel 3. Then install the driver: *"sh NVIDIA-Linux-x86_64-177.13-pkg2.run"*, and answer the questions. The installer needs to compile a kernel module for your system, hence you need to have the packages installed to allow it to do that. The installer will also install a number of libraries and other software. I believe that that the *"nvidia.ko"* kernel module is all the system needs to support CUDA (other than the software libraries of the Toolkit, of course). When you are done, *"init 5"* will start up runlevel 5 again, including (if all went well) X-Windows.

Toolkit

Install the toolkit similarly: *"sh NVIDIA_CUDA_Toolkit_2.0beta2_Suse10.3_x86_64.run"* which will install the cuda software in /usr/local/cuda by default.

Next, edit your *.bashrc* file to aim at the CUDA binary and library directories.

export PATH=(blah blah):/usr/local/cuda/bin

export LD_LIBRARY_PATH=(blah blah):/usr/local/cuda/lib

And either start a new shell, or execute *source .bashrc* to get those variables set.

You'll find documentation in the /usr/local/cuda/doc directory: A programming guide, a CUDA reference manual, and a CUDA BLAS reference manual. Its probably best/essential to take a careful look at the Release Notes as well.

Software Development Kit

Install the software development kit similarly: *"sh NVIDIA_CUDA_sdk_2.0beta2_linux.run"* which will install the example projects into your home space by default to the directory $HOME/NVIDIA_CUDA_SDK. If you then go to that directory and execute "make" and the CUDA Toolkit is installed properly, all the example projects should compile. You can then test out those examples by running them in the NVIDIA_CUDA_SDK/bin/linux/release executables directory.

Visual Profiler

The toolkit now includes the Cuda Visual Profiler which allows you to run a code and measure the GPU time taken up by each called CUDA routine, i.e., identify which elements of your code are taking up the most time. This can show you the bottlenecks in your code, hence give you a chance to develop alternative, faster computation strategies. The Visual Profiler is installed in */usr/local/cuda/cudaprof*. Add the */usr/local /cuda/cudaprof/bin* directory to your LD_LIBRARY_PATH:

export LD_LIBRARY_PATH=(blah blah):/usr/local/cuda/lib:/usr/local /cuda/cudaprof/bin

The profiler binary needs to access libraries in that directory.

AccelerEyes and GP-YOU

Before diving into the material below, the reader should be aware that there are now projects devoted to implementing CUDA on matlab to save the programmer the trouble of debugging home-built CUDA routines. Two such projects are AccelerEyes (http://www.accelereyes.com/) and GP-You (http://gp-you.org). AccelerEyes is a commercial package. GP-You presently offers a freeware library named GPUmat. Both projects implement matlab script wrappers for easily calling the CUDA BLAS or LAPACK routines. This author does not know much about these packages, but they likely offer a more user-friendly access to GPU computing on matlab. I don't believe either project aims to develop their own BLAS or LAPACK routines, rather, they are designing wrappers that make it easy for matlab users to call such routines from within matlab. Both projects are in active development.

Installation and configuration: The mex environment

To compile a mex file for CUDA, you will need four files or scripts available:

1. A *Makefile*.
2. A *nvopts.sh* configuration file.
3. A mex source file with the *.cu extension.
4. The *nvmex* executable.

You can find the Makefile, the nvopts.sh, and the nvmex files in the Matlab package available for download from the Nvidia CUDA site (e.g., http://developer.download.nvidia.com/compute/cuda/1_1/Matlab_Cuda_1.1.tgz). However, there is nothing fancy about these things. nvmex and nvopts.sh appear to be basically copies of Matlab's mex and mexopts.sh scripts, other than: (1) in the nvopts.sh

file the "gcc" compiler is replaced by the "nvcc" compiler with a few of the options changed, and (2) the nvmex script is modified to recognize *.cu extensions, in addition to *.c (c source file). (The nvcc compiler is the very essence of what CUDA is about - converting your code to run on the GPU. nvcc looks for source code with the *.cu extension.)

Examples of mex source code can be found in this document (and elsewhere) copy them to a file with the *.cu extension. Then edit the Makefile to (a) aim at the locations of your CUDA and Matlab installations and (b) have the proper source code filename (The example below is set if your code is named *yourcode.cu*)

Key lines in the *"nvopts.sh"* file:

```
CC='nvcc'
CFLAGS='-O3 -Xcompiler "-fPIC -D_GNU_SOURCE -pthread -fexceptions -m64 -march=native"'
CLIBS="$RPATH $MLIBS -lm -lstdc++"
COPTIMFLAGS='-Xcompiler "-O3 -DNDEBUG -march=native"'
.
.
.
LD="gcc"
```

With these things set and with the Makefile below, executing *"make"* should produce your, e.g., *"yourcode.mexa64"* binary mex file.

The Makefile:
(if you copy this, don't forget that Makefiles are particular about the *TAB* indents!!!)

```
# Define installation location for CUDA and compilation flags compatible
# with the CUDA include files.
CUDAHOME    = /usr/local/cuda
INCLUDEDIR  = -I$(CUDAHOME)/include
INCLUDELIB  = -L$(CUDAHOME)/lib -lcufft -lcudart -lcublas -Wl,-rpath,$(CUDAHOME)/lib
CFLAGS      = -fPIC -D_GNU_SOURCE -pthread -fexceptions
COPTIMFLAGS = -O3 -funroll-loops

# Define installation location for MATLAB.
export MATLAB = /usr/local/matlab
MEX           = $(MATLAB)/bin/mex
MEXEXT        = .$(shell $(MATLAB)/bin/mexext)

# nvmex is a modified mex script that knows how to handle CUDA .cu files.
NVMEX = ./nvmex

# List the mex files to be built.  The .mex extension will be replaced with the
# appropriate extension for this installation of MATLAB, e.g. .mexglx or
# .mexa64.
MEXFILES = yourcode.mex

all: $(MEXFILES:.mex=$(MEXEXT))

clean:
        rm -f $(MEXFILES:.mex=$(MEXEXT))

.SUFFIXES: .cu .cu_o .mexglx .mexa64 .mexmaci

.c.mexglx:
        $(MEX) CFLAGS='$(CFLAGS)' COPTIMFLAGS='$(COPTIMFLAGS)' $< \
        $(INCLUDEDIR) $(INCLUDELIB)

.cu.mexglx:
      $(NVMEX) -f nvopts.sh $< $(INCLUDEDIR) $(INCLUDELIB)

.c.mexa64:
        $(MEX) CFLAGS='$(CFLAGS)' COPTIMFLAGS='$(COPTIMFLAGS)' $< \
        $(INCLUDEDIR) $(INCLUDELIB)

.cu.mexa64:
        $(NVMEX) -f nvopts.sh $< $(INCLUDEDIR) $(INCLUDELIB)

.c.mexmaci:
        $(MEX) CFLAGS='$(CFLAGS)' COPTIMFLAGS='$(COPTIMFLAGS)' $< \
        $(INCLUDEDIR) $(INCLUDELIB)

.cu.mexmaci:
        $(NVMEX) -f nvopts.sh $< $(INCLUDEDIR) $(INCLUDELIB)
```

A first program: SGEMM - matrix multiplication using cudaBLAS

Noting that CUDA included a set of (still incomplete) BLAS libraries, I thought I would start out by developing a mex routine that would multiply two matrices together using CUDA. What could be simpler? After a few false starts, the code below was developed, with a bit of help from the Nvidia mail forum. This mex file is a generic Matlab interface to the CUDA Level 3 BLAS routine for calculating $C = \alpha*A*B + \beta*C$, where A, B and C

are ordinary matrices and α and β are scalars. See the CUDA BLAS reference manual for a complete descriptions of the various functions called in the source code below. There are a few elements worth explicitly pointing out.

- Most GPU hardware is only capable of single precision calculations (video cards capable of double precision have recently arrived), so the routine expects single precision variables as input. This expectation was a design choice motivated by (a) the desire to make that an explicit requirement from the user so he knows that it must be single precision, and (b) the single() function of Matlab is much faster than the initial conversion routine I attempted within the mex file.

- The CUDA BLAS system is initiated with the call *retStatus = cublasInit();* This is specific to the BLAS. The *retStatus* is optional and gives an error status. When first starting out, testing for such errors is essential.

- The CUDA parts of the code consist of the three calls that (a) allocate a matrix on the GPU, (b) zero it out, and then (c) set the values in the GPU matrix to those of the mex matrix:

```
cublasAlloc (...);
cudaMemset(...);
cublasSetMatrix(...)
```

- Be aware that cudaMemset sets the zero value byte-by-byte. This routine cannot be used to set matrix values to an arbitrary number.

- Once the matrices A, B, C are set (with matrix C being unnecessary if β=0) and the values of α, β determined, the next two calls (a) call the CUDA BLAS SGEMM and (b) copy the final results back to the mex matrix "cc":

```
cublasSgemm (...)
cublasGetMatrix (...)
```

- The line *Mc=M+32-M%32;* is used to find the dimension larger than M that is a multiple of 32. The BLAS routines run considerably faster if the arrays are padded with zeros so that the dimensions are a multiple of 32. After the arrays are allocated, they are zeroed out with *cudaMemset(...);* This is necessary because the lines *cublasSetMatrix (...);* only fill the, e.g., M×K section of the Mc×Kc array. It is important to zero the padded elements of the allocated array.

- The allocated memory is freed at the end with *cublasFree()* and cublasShutdown at the end. See "Memory issues" later in this document.

A Matlab script to test this routine:

```
A=[1.5 2 3;4 4 4];
B=[44 55 66;11 11 11]';

% Let's use some large arrays...
A=randn(4000,2000);
B=randn(2000,4000);

[m n]=size(A);
[mm nn]=size(B);
C=randn(m,nn);
alpha=-1;
beta=0;

disp('Matlab:')
tic
C1d=alpha*A*B + beta*C;
toc

% In single precision, Matlab is twice as fast! (go figure...)
tic
A1=single(A);
B1=single(B);
C1=single(C);
C1s=alpha*A1*B1 + beta*C1;
toc

% The call here is testing out the transposes of the code.
disp('CUDA:')
tic
C2=sgemm_cu(0,1,single(alpha),single(beta),single(A),single(B'),single(C));
toc

% Compare the CUDA results with the Matlab results
min(min(C2-C1s))/min(min(C1s))
max(max(C2-C1s))/max(max(C1s))
```

The source code for sgemm_cu.cu:
(edit the Makefile above to apply to it, type *"make"* to compile):

```
#include "mex.h"
#include "cublas.h"

/*  sgemm_cu.cu - Gateway function for subroutine sgemm
 C = sgemm_cu(transa,transb,single(alpha),single(beta),single(A),single(B),single(C))
 transa,transb = 0/1 for no transpose/transpose of A,B
 Input arrays must be single precision.
*/

void mexFunction( int nlhs, mxArray *plhs[],
   int nrhs, const mxArray *prhs[])
{
    cublasStatus status;
    int M,K,L,N,MM,NN,KK;
    int Mc,Kc,Lc,Nc,MMc,NNc,KKc;
    int dims0[2];
    int ta,tb;
    float alpha,beta;
    float *a,*b,*c,*cc;
    float *ga,*gb,*gc;
    char transa,transb;
    cublasStatus retStatus;

    if (nrhs != 7) {
        mexErrMsgTxt("sgemm requires 7 input arguments");
    } else if (nlhs != 1) {
        mexErrMsgTxt("sgemm requires 1 output argument");
    }

    if ( !mxIsSingle(prhs[4]) ||
         !mxIsSingle(prhs[5]) ||
         !mxIsSingle(prhs[6]))   {
         mexErrMsgTxt("Input arrays must be single precision.");
    }

    ta = (int) mxGetScalar(prhs[0]);
    tb = (int) mxGetScalar(prhs[1]);
    alpha = (float) mxGetScalar(prhs[2]);
    beta = (float) mxGetScalar(prhs[3]);

    M = mxGetM(prhs[4]);   /* gets number of rows of A */
    K = mxGetN(prhs[4]);   /* gets number of columns of A */
    L = mxGetM(prhs[5]);   /* gets number of rows of B */
    N = mxGetN(prhs[5]);   /* gets number of columns of B */

    if (ta == 0) {
        transa='n';
        MM=M;
        KK=K;
    } else {
        transa='t';
        MM=K;
        KK=M;
    }

    if (tb == 0) {
        transb='n';
        NN=N;
    } else {
        transb='t';
        NN=L;
    }
```

```
 /*    printf("transa=%c\n",transa);
    printf("transb=%c\n",transb);
    printf("alpha=%f\n",alpha);
    printf("beta=%f\n",beta);     */

/* Left hand side matrix set up */
    dims0[0]=MM;
    dims0[1]=NN;
    plhs[0] = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
    cc = (float*) mxGetData(plhs[0]);

/* Three single-precision arrays */
    a = (float*) mxGetData(prhs[4]);
    b = (float*) mxGetData(prhs[5]);
    c = (float*) mxGetData(prhs[6]);

/* STARTUP   CUBLAS */
    retStatus = cublasInit();
   // test for error
   retStatus = cublasGetError ();
   if (retStatus != CUBLAS_STATUS_SUCCESS) {
      printf("CUBLAS: an error occurred in cublasInit\n");
    }

   Mc=M+32-M%32;
   Kc=K+32-K%32;
/* ALLOCATE SPACE ON THE GPU AND COPY a INTO IT */
    cublasAlloc (Mc*Kc, sizeof(float), (void**)&ga);
    // test for error
    retStatus = cublasGetError ();
    if (retStatus != CUBLAS_STATUS_SUCCESS) {
        printf("CUBLAS: an error occurred in cublasAlloc\n");
    }
    cudaMemset(ga,0,Mc*Kc*4);                /* zeros the array ga byte-by-byte  */
    retStatus = cublasSetMatrix (M, K, sizeof(float),
             a, M, (void*)ga, Mc);

    Lc=L+32-L%32;
    Nc=N+32-N%32;
/* SAME FOR B, C */
    cublasAlloc (Lc*Nc, sizeof(float), (void**)&gb);
    cudaMemset(gb,0,Lc*Nc*4);
    retStatus = cublasSetMatrix (L, N, sizeof(float),
             b, L, (void*)gb, Lc);

    MMc=MM+32-MM%32;
    NNc=NN+32-NN%32;
    KKc=KK+32-KK%32;
    cublasAlloc (MMc*NNc, sizeof(float), (void**)&gc);
    if (beta != 0.0 ) {
       cudaMemset(gc,0,MMc*NNc*4);
       retStatus = cublasSetMatrix (MM, NN, sizeof(float),
                c, MM, (void*)gc, MMc);
    }
/*  PADDED ARRAYS */
/*     printf("Op(A) has No. rows = %i\n",MMc);
       printf("Op(B) has No. cols = %i\n",NNc);
       printf("Op(A) has No. cols = %i\n",KKc);
       printf("A has leading dimension = %i\n",Mc);
       printf("B has leading dimension = %i\n",Lc);
       printf("C has leading dimension = %i\n",MMc); */
```

```
/* READY TO CALL SGEMM */
  (void) cublasSgemm (transa, transb, MMc, NNc, KKc, alpha,
                              ga, Mc, gb, Lc, beta, gc, MMc);
  status = cublasGetError();
  if (status != CUBLAS_STATUS_SUCCESS) {
      fprintf (stderr, "!!!! kernel execution error.\n");
  }

/* NOW COPY THE RESULTING gc ON THE GPU TO THE LOCAL c */
   retStatus = cublasGetMatrix (MM, NN, sizeof(float), gc, MMc, cc, MM);
   if (retStatus != CUBLAS_STATUS_SUCCESS) {
        printf("CUBLAS: an error occurred in cublasGetMatrix\n");
   }

/* FREE UP GPU MEMORY AND SHUTDOWN (OPTIONAL?) */
    cublasFree (ga);
    cublasFree (gb);
    cublasFree (gc);
    cublasShutdown();

}
```

Array ordering conventions

One potential pitfall is that CUDA uses both conventions for ordering of arrays. The Matlab convention is that the array A = [1 2 3;4 5 6] is ordered: A(1:6)=[1 4 2 5 3 6], that is, by columns ("column major"). This is the same convention used by mex functions and the cudaBLAS routines. The C convention is the opposite — in C, A(1:6) = [1 2 3 4 5 6] ("row major"). The C convention is used by most of the CUDA routines, e.g., cudaMemcpy2D which "copies a matrix (height rows of width bytes each) from the memory area pointed to by src to the memory area pointed to by dst." However, what cudaMemcpy2D means by rows is understood to be columns in Matlab. An example:

Suppose you have a array A sized M×N that is already on the GPU device. A is equivalent to a column vector of height M*N, and suppose A can be reshaped to an array K×L (Matlab convention). You wish to reshape A to be K×L and copy it to a padded array B also on the GPU, sized Kp×Lp. Kp,Lp are padded from K,L so that they are multiples of, e.g., 32. For example, a float* vector [1 2 3 4 5 6 7 8 9]' on the device needs to be copied into a 5X4 float* array on the device like this:

```
1 4 7 0
2 5 8 0
3 6 9 0
0 0 0 0
0 0 0 0
```

The call to do that is:

```
cudaMemcpy2D(B, Kp*sizeof(float), A, K*sizeof(float),
       K*sizeof(float), L, cudaMemcpyDeviceToDevice);
```

This makes L copies of K elements each of A to B, leaving the lower and right padded areas of B unchanged (Kp is 5, K is 3, and L is 3 in the simple example). In the description of cudaMemcpy2D in the CUDA reference manual, the "height" and "width" conventions are the opposite of Matlab's. Note that the value for the "height", L, (which is really the width of the array in Matlab convention) does not have a *"sizeof(float)"*.

Calling a Matlab function within a mex function

It is often useful to be able to execute some of the nifty Matlab routines from within the mex function. While this is not particularly related to CUDA, it is still a useful technique. In my case, I needed to calculate a matrix division C=A/B, and no routine to do that is as yet available for CUDA (there are, however, a few recent research papers available on the subject). The combination of CUDA routines to do the bulk of the calculations with Matlab routines to easily compute specialized elements (although division is not entirely specialized, true) in a few places is attractive. Only the essential elements of calling a Matlab function, with matrix division used as the example, from within a mex function are given in this section.

The first step is to allocate and set up the pointers to the matrices

```
int dims0[2];
mxArray *rhs[1], *lhs[1];
float *A, *B, *C;
.
.
.
 /* rhs,lhs utility matrices set up - used for calling the matlab division.         */
      dims0[0]=I; dims0[1]=L;
      rhs[0] = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
      A = (float*) mxGetData(rhs[0]);

      dims0[0]=L; dims0[1]=L;
      rhs[1] = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
      B = (float*) mxGetData(rhs[1]);

      dims0[0]=I; dims0[1]=L;
      lhs[0] = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
```

Once A, B have been assigned values, the division can be executed ("mrdivide" for "matrix right division"). Any Matlab function, including any m-scripts written by the user, can be used this way. See the Matlab documentation for details.

```
// Get C = A/B
// We have to use matlab for this, since CUDA has no matrix division yet.
//     rhs[0] = A;
//     rhs[1] = B;
    mexCallMATLAB(1,&lhs[0],2,rhs,"mrdivide");
    C = (float*) mxGetData(lhs[0]);
```

The results can then be copied to the GPU for further calculations (in this example C is I×L, while gc is Ic×Lc with Ic, Lc being multiples of 32), and the lhs matrix can be destroyed when you are done with it to prevent memory leak (see section on memory below).

```
    // Copy C to GPU
    cublasSetMatrix (I, L, sizeof(float), C, I, (void*)gc, Ic);
    mxDestroyArray(lhs[0]);
```

Preserving variables on the GPU between mex function calls

It is useful to be able to calculate a result on the GPU using a mex function, and then sometime later be able to call another mex function (or the original mex funtion within a loop) and be able to access the results already stored on the GPU. As noted above, host↔device communications are frequently the bottleneck in a calculation, so to be able to avoid such communications is a good thing. The mex source code below shows how the matrix Arg can be defined and allocated such that subsequent calls to this mex function can recover the value of Arg from the GPU. The key element to this code appears to be the *"static float *Arg;"* at the top of the code. This preserves the values of Arg for subsequent calls of the mex function. It seems likely that Arg could also be preserved by returning Arg to Matlab, hence used by other mex functions. Once an array is allocated with cudaMalloc (...) that array is preserved on the GPU until a cudaFree (...) is called to free it, or until the mex function is cleared.

Calling *"persist"* from Matlab the first time will allocate a 1×1 matrix Arg on the GPU and set its value to 37. Subsequent calls to *"persist"* will read that value from the GPU, increment it by one, and write the new value back to the GPU. Hence, one is using a GPU to count to 10, as it were. The principle is useful, if not essential.

The source code for persist.cu:

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

static int initialized = 0;
static float *Arg;

void cleanup(void) {
  mexPrintf("MEX-file is terminating, destroying array\n");
  cudaFree(Arg);
  mexPrintf("Freed CUDA Array.\n");
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
  int N;
  int dims0[2];
  float *val;
  mxArray *rhs = NULL;

  if (!initialized) {
  // Create persistent array and register its cleanup.

      mexPrintf("MEX-file initializing, creating GPU array and setting a value.\n");
      N=1;
      cudaMalloc ((void **)&Arg, N * sizeof(Arg[0]));

      dims0[0]=1; dims0[1]=1;
      rhs = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
      val =  (float*) mxGetData(rhs);

  // Set the data of the array to some interesting value. (37 is a prime number, hence interesting)
      val[0] = 37.0;

      cudaMemcpy(Arg, val, N*sizeof(float), cudaMemcpyHostToDevice);

      mexAtExit(cleanup);
      initialized = 1;

  } else {

      dims0[0]=1; dims0[1]=1;
      rhs = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
      val = (float*) mxGetData(rhs);

      N=1;
      cudaMemcpy(val, Arg, N*sizeof(float), cudaMemcpyDeviceToHost);
      mexPrintf("Pulling value of first array element off GPU; it is %g\n", val[0]);

   // Increment the value by one to show one can do something with the values.
      val[0]=val[0]++;
   // Copy the new value to the GPU
      cudaMemcpy(Arg, val, N*sizeof(float), cudaMemcpyHostToDevice);

  }
}
```

Defining a CUDA kernel in a mex function

The BLAS routine *cublasSgemm* is a CUDA kernel that has been developed by the Nvidia CUDA team. (It is carefully tuned for efficiency on the GPU.) In this section, a simple

example kernel that calculates the sine and cosine of the elements of a matrix will be described. Beyond this, we are in an area out my expertise! But hopefully this example will show you how you can get started with developing your own CUDA applications that can be called from a mex function.

The top of the code shows the kernel - consult the CUDA documents for the technical description of things here. A critical value is "COS_THREAD_CNT 200" which defines the number of threads CUDA employs for the calculation. This value affects how efficient the calculation is performed, hence it is worth either calculating (don't know how) or testing out various values to see what works best (See Appendix). I believe that the kernel here must be completely independent; this is a calculation in parallel that does not have access to other threads of the loop or other data. The index "i" is not incremented linearly in a particular thread for example; one cannot use it as a counter. Computations in parallel require a different mindset.

The lines:

```
for (i = threadIdx.x; i < parms.n; i += COS_THREAD_CNT)
```

```
cos_main<<<1,COS_THREAD_CNT>>>(funcParams);
```

contain the only elements of this code specific to the *"nvcc"* compiler.

Naturally, a kernel does not have to be defined at the top of the mex function this way; it can be its own stand alone subroutine so long as it is linked by the Makefile.

A Matlab script to test the routine:

```
clear all
amp=[ 1.1 2.5; 4.0 6.0];
phase = 2*pi*[0.2 0.5; 0.3 0.8];
ANS1=[amp.*cos(phase) amp.*sin(phase)]
ANS2=cosine(single(amp),single(phase))
```

The source code for cosine.cu :

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

#define COS_THREAD_CNT 200
/* ------------------------- target code (the kernel) ----------------*/
struct cosParams {
float *arg;
float *res;
float *a;
int n;
};

__global__ void cos_main(struct cosParams parms)
{
int i;
float sint, cost;

for (i = threadIdx.x; i < parms.n; i += COS_THREAD_CNT)
  {
  sincosf(parms.arg[i], &sint, &cost);
  parms.res[i]         = parms.a[i]*cost ;
  parms.res[i+parms.n] = parms.a[i]*sint ;
  }
}

/* -------------------------- host code ------------------------------*/
/*  cosine.cu - Gateway function for [A.*cos A.*sin]
 Called by:

  [A] = cosine(amp,phase);

  amp, phase should be dimensioned the same.
  Returns A = [amp.*cos(phase) amp.*sin(phase)];
*/

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
      int dims0[2];
      int NX,NY,NN;

      float* cosArg =0;
      float* aaa = 0;
      float* cosRes = 0;;

      float *phase, *amp;
      float *A;

      struct cosParams funcParams;

      if (nrhs != 2) {
          mexErrMsgTxt("engine requires 2 input arguments");
      } else if (nlhs != 1) {
          mexErrMsgTxt("engine requires 1 output argument");
      }

      if ( !mxIsSingle(prhs[0]) || !mxIsSingle(prhs[1]) ) {
           mexErrMsgTxt("All input arrays must be single precision.");
      }

// Get the various dimensions to this problem.
      NY = mxGetM(prhs[0]);   /* Number of rows of data.             */
      NX = mxGetN(prhs[0]);   /* Number of columns of data.          */
```

```
// Get the input data.
      amp =    (float*) mxGetData(prhs[0]);
      phase =  (float*) mxGetData(prhs[1]);

// Left hand side matrices set up for the result.
      dims0[0]=NY;
      dims0[1]=2*NX;
      plhs[0] = mxCreateNumericArray(2,dims0,mxSINGLE_CLASS,mxREAL);
      A = (float*) mxGetData(plhs[0]);

// Allocate variables on the GPU for calculating A.
      NN=NX*NY;
      cudaMalloc ((void **)&cosArg, NN*sizeof(cosArg[0]));
      cudaMalloc ((void **)&aaa, NN*sizeof(aaa[0]));
      cudaMalloc ((void **)&cosRes, 2*NN*sizeof(cosRes[0]));

// Copy amp, phase to the GPU.
      cudaMemcpy (cosArg, phase, NN*sizeof(phase[0]), cudaMemcpyHostToDevice);
      cudaMemcpy (aaa, amp, NN*sizeof(aaa[0]), cudaMemcpyHostToDevice);

      funcParams.res = cosRes;
      funcParams.arg = cosArg;
      funcParams.a = aaa;
      funcParams.n = NN;

      cos_main<<<1,COS_THREAD_CNT>>>(funcParams);
// "A" should now be in the array pointer "cosRes" on the device.
//  We'll need to copy it to A
// "aaa", "cosArg" are NY by NX, while "cosRes" is NY by 2*NX
//  (although everything here is stored in linear memory)

// Copy the result, which is A, from the device to the host
      cudaMemcpy (A, cosRes, 2*NN*sizeof(A[0]), cudaMemcpyDeviceToHost);
// Done!

// Free up the allocations on the GPU
      cudaFree(cosArg);
      cudaFree(aaa);
      cudaFree(cosRes);

}
```

Memory issues to look out for

Allocating GPU memory

When allocating memory on the GPU, one should program conservatively and test whether the allocation was successful. Code that allocates more memory than the GPU has can often run "successfully," but with indeterminant results. While overallocating GPU memory can cause a variety of problems - a crashing desktop or computer - often the code will just quietly run for however long the routine takes, but give completely erroneous results.

Code snippets to test for a successful memory allocation are:

For cudaMalloc:

```
cudaError_t cudaError;
.
.
.
cudaError=cudaMalloc ((void **)&A, ND*NN * sizeof(A[0]));
if (cudaError != cudaSuccess) { mexErrMsgTxt("Out of Nvidia device memory."); }
```

For cublasAlloc:

```
cublasStatus status;
.
.
.
status=cublasAlloc (I*I, sizeof(float), (void**)&G);
if (status != CUBLAS_STATUS_SUCCESS) { mexErrMsgTxt("Out of Nvidia device memory."); }
```

The "mexErrMsgTxt" is used because this call has the handy property that it stops matlab when the mex file returns. Using just a "return;" does not stop matlab, which is usually not the desired behaviour.

Memory leaks in Matlab loops

When using loops in Matlab that call a mex function, or loops within the mex function, there is the possibility of "memory leaks". One memory leak can occur when calling a Matlab function from a mex function like:

```
mexCallMATLAB(1,&lhs[0],2,rhs,"mrdivide");
```

Matlab does not overwrite the lhs pointer with each call, but just keeps allocating more space. If you loop enough, you'll fill up all your computer's memory. So, after calling this from a CUDA-using mex function to do the matrix division, when you are done with the result (e.g., copy the result to the GPU) the array should be destroyed:

```
mxDestroyArray(lhs[0]);
```

This destruction does not make lhs go away, but clears out the allocated memory. No more host memory leak. Example pseudo-code:

```
for (i = 0; i < LargeN; i++)
    {
      <some stuff>
      mexCallMATLAB(1,&lhs[0],2,rhs,"mrdivide");
      <copy result to GPU>
      mxDestroyArray(lhs[0]);
      <more stuff>
    }
```

Memory leaks in allocating space on the GPU

Another memory leak can occur when allocating space on the GPU from a mex function, e.g.,

```
cudaMalloc ((void **)&A, N * sizeof(A[0]));
```

The allocated space does not clear by itself when the mex call terminates. (This property allows the data in arrays on the GPU to be recovered on subsequent mex file calls, as described above.) So, if the mex function is repeatedly called from a loop in Matlab, the GPU memory will fill up. You have to be sure to clear it with:

```
cudaFree(A);
```

at the end of the mex function. If a call to a mex function is made repeatedly from a loop in Matlab without clearing the CUDA allocations, you may get erroneous results if the GPU runs out of memory. Clearing such allocations means no more GPU memory leak.

Memory comments

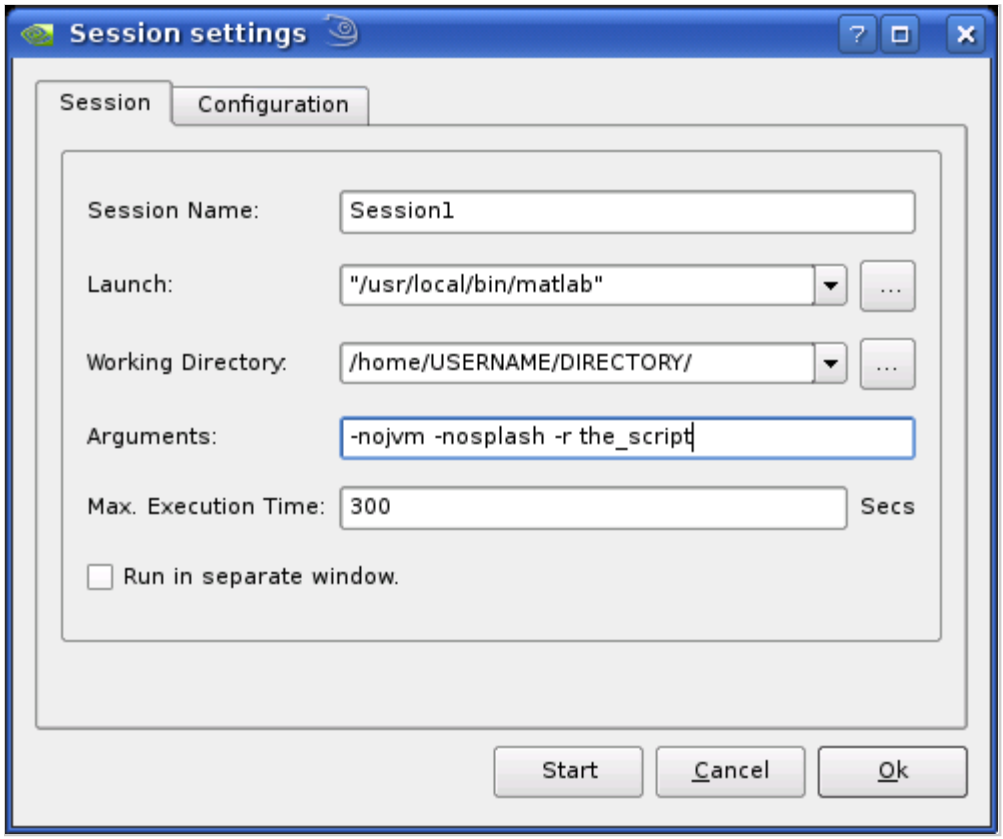
In all cases, memory is freed when either Matlab is exited or the mex function is cleared as a variable in Matlab.

One might be tempted to use the routine "cudaMallocHost" in a mex file to speed up the host-device communications. Allocating memory this way uses "pinned memory," which is faster because it gives CUDA direct access to memory without going through the paged memory of the computer's kernel. Matlab is fairly memory sensitive, however, so it is not generally possible to allocate memory this way. I suspect, however, that one could actually sometimes get away with using cudaMallocHost() in a mex file - it is a matter of avoiding any calls to matlab while such memory is allocated. So one could cudaMallocHost(), compute away, and then clear out the CUDA variables before Matlab is aware of what is going on and complain/crash (its the "don't ask, don't tell" memory

policy).

Using the CUDA Visual Profiler

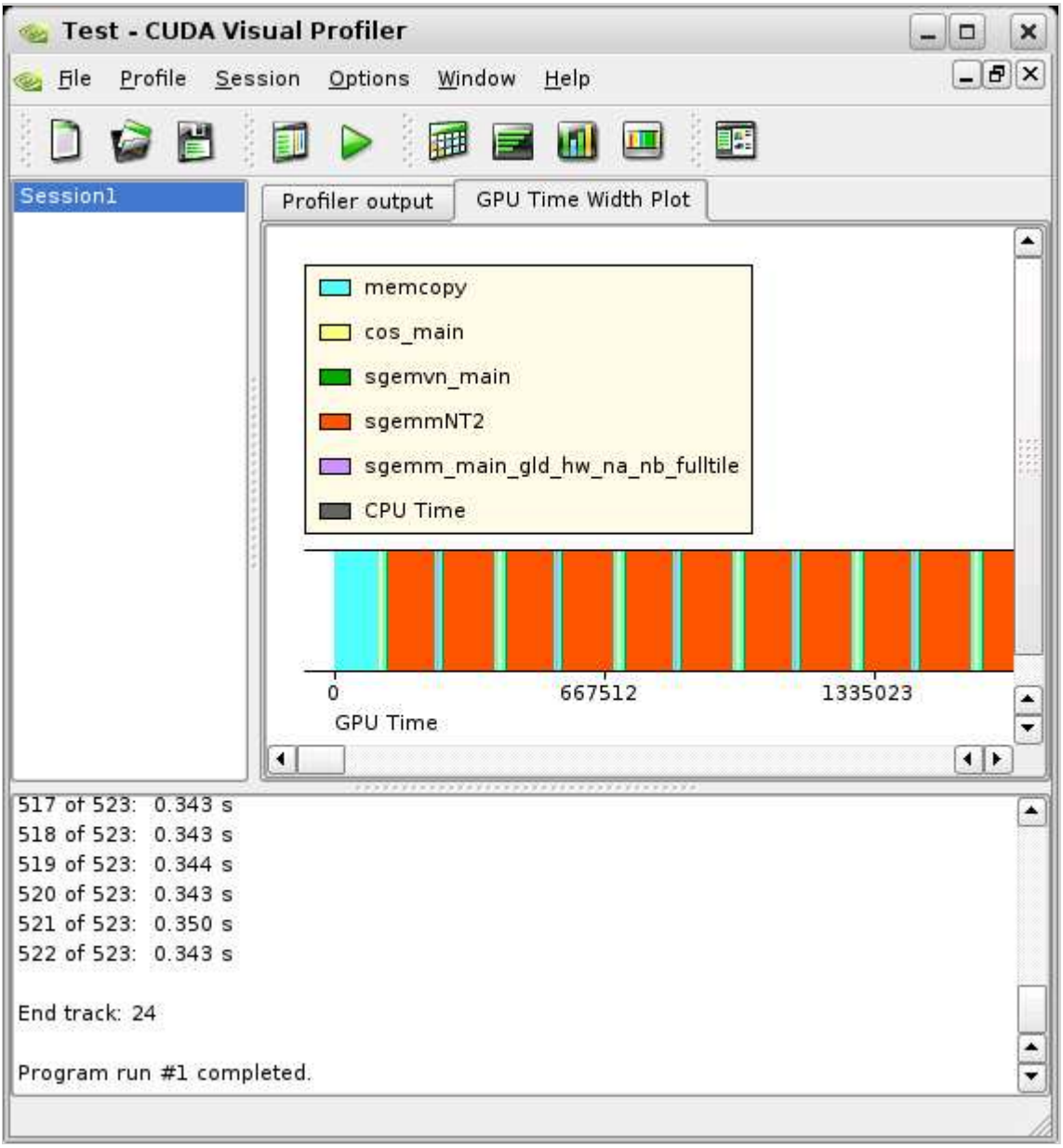
The Profiler allows one to run a Matlab script while timing the various calls to CUDA routines. This gives one a good look at where the program is spending time, hence can give some important clues for identifying how to speed up calculations. In general, one want to minimize the memory copies between host and device.



Configuration file for using the CUDA Visual Profiler in Matlab. The script to be run is preceded by a "-r" and the ".m" is not included.

The figure above shows how to start a Matlab script from the Profiler. The call to Matlab requires the quotes, and I believe the full path name. Note that java is turned off. The script is started using the "-r" option, and no ".m" should be put on the script name. Be sure to set the "Max. Execution Time" to be a number larger than the time it takes your application to complete. Graphics will display properly using the Profiler. The Matlab script has to "quit" at the end for the Profiler to work properly, i.e., be able to save the timed values. With these settings, hit the Start button and at the completion of

the script, you'll be presented with a table of the timing results.



Results from the CUDA Visual Profiler in Matlab. An initial large memcopy is followed by mostly sgemmNT2; this calculation is efficiently calculating on the GPU, rather than being bogged down with memory copies.

The figure above shows the results of the table presented in a graphical format. After the initial large transfer of data to the GPU, the calculation stays mostly on the GPU, so it appears to be fairly efficient. The blocks are repeated because the calculations are within a loop. The interruptions in the GPU calculations occur because I needed to do a matrix

division, hence you'll see the three memcopies - copy two arrays from the GPU, calculate the division, copy the results back to the GPU and continue the calculation. At the end, unseen way off to the right, there is another large memcopy when the results of the loop are copied back to Matlab from the GPU.

CULA and MAGMA; BLAS and LAPACK

Nvidia presently provides a subset of BLAS routines, some of which have been discussed above. Not all BLAS routines are presently available, nor are the LAPACK routines. There are several projects underway to implement BLAS and LAPACK routines for the GPU. Two notable projects are CULA (http://www.culatools.com/), a quasi-commercial enterprise that is working closely with Nvidia, and MAGMA (Matrix Algebra on GPU and Multicore Architectures; http://icl.cs.utk.edu/magma/) a multi-institutional academic program that also works closely with Nvidia. MAGMA is led by Jack Dongarra of the University of Tennessee. Both of these projects have initial releases of routines that implement elements of BLAS and LAPACK. Notably, both have released versions of SGESV which computes the solution to a real system of linear equations, A*X=B, i.e., matrix division. Implementing these routines within a mex file is straightforward, after the usual debugging, but as of this writing, the routines are not quite operational within matlab. The MAGMA binary libraries cause mex files to crash. CULA's implementation of SGESV appears to interact with matlab in ways that are not quite understood, causing the routine to run rather slowly. In any case, it is worth monitoring these projects as they work towards a full implementation of efficient BLAS and LAPACK routines.

Appendix: Grids, blocks, threads, and all that

Multiple processors

A CUDA device consists of a number of multiprocessors (MP's) each with a number of stream processors. For example, a GTX 260 consists of 24 MP's, with 8 stream processors per MP. That gives a total of 24×8=192 stream processors (aka "cores"). (And for the GTX 200 series, only one of the stream processors per MP is double precision.)

Grid and block size

When a kernel is called, one must specify a grid size and a block size. The grid size sets the number of blocks. The block size sets the number of threads per block. MP's do not

run; rather, blocks run on MP's. A block will run on one MP, and an MP can run up to 8 blocks "at the same time". An MP can handle at most 768 threads at the same time (1024 on GTX 200), but this depends heavily on the resource usage.

When a kernel is called, all one has to do is set the desired number of threads per block and the number of blocks. CUDA takes care of running all those blocks in the most efficient way on the MP's. If resources permit, several blocks will run on one MP. The grid and block sizes can often be adjusted to optimize the calculation - this optimization maximizes the computational load of the stream processors, or, equivalently, minimizes their idle time.

What is a warp?

The warp size is the number of instructions that are run at any one time for a given MP. The purpose of warp size, which is specified by hardware, is to maximize the work of the stream processors. The reason for warp size is something like this: Each MP has just one instruction decoder, so each of the 8 (on current cards) stream processors needs to run the same instruction. Hence, the minimum warp size is 8. The stream processors are also pipelined, so for maximum efficiency 2 instructions need to be in flight to keep the pipeline stages busy. The same decoded instruction is therefore run again, but for another set of threads, which doubles the warp size up to 16. There is also a clock rate difference between the instruction decoder and the stream processors, which requires some extra time to decode the next instruction. The warp size is therefore be doubled again to 32, which is the warp size for most CUDA devices.

The size of the warp could not be reduced without causing parts of the chip to be underused. For a lot of calculations, a large warp size is an advantage, not a disadvantage. The warp size allows the GPU to maximize the execution of instructions, without spending too much time in instruction decoding. The warp design is why one generally wants to have many more threads than stream processors.

The term "warp" originates from weaving: "In weaving, the warp is the set of lengthwise yarns through which the weft is woven." (Wikipedia)

Threads

A "thread" is short for "thread of execution," according to Wikipedia, and it is the "lightest" unit of kernel scheduling. The core processors of a GPU are unlike the processors of an ordinary symmetric multiprocessor (SMP) computer. The warp strategy which manages the threads is employed so that the hardware stays as busy as possible during the cycles of the calculations. In general, one wants many more threads

than stream processors because:

1. At least 4 threads per Arithmetic Logic Unit (ALU) are run for a warp.
2. At least 6 warps (giving 4 threads×8 ALU's×6 warps = 192 total threads) are needed to have no read after write register dependencies.
3. More threads can be employed to hide the memory-access latency.

If one just ran a number of threads equal to the number of core processors, most of the time the ALU's would be idly waiting for more instructions to execute.

Acknowledgement

Special thanks to the Nvidia forums from where much of this material was gleaned.
