THE CUBLAS LIBRARY

Will Landau, Prof. Jarad Niemi

WHAT IS CUBLAS?

CUBLAS library is a CUDA C implementation of the C/Fortran library, BLAS (Basic Linear Algebra Subprograms).

3 "levels of functionality":

Level 1:	$\mathbf{y} \mapsto \alpha\mathbf{x} + \mathbf{y}$	and other vector-vector routines
Level 2:	$\mathbf{y} \mapsto \alpha A\mathbf{x} + \beta\mathbf{y}$	and other vector-matrix routines
Level 3:	$C \mapsto \alpha AB + \beta C$	and other matrix-matrix routines

where α and β are scalars, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and A, B, and C are matrices.

CAUTION

1. Like Fortran, CUBLAS uses column-major order.
2. Unlike C, CUBLAS uses 1-based indexing.

To avoid confusion in your own code, you can implement column-major order matrices as one-dimensional arrays.

Let `i` be a 0-based row index, `j` be a 0-based column index, and `ld` be the number of rows. Then you can calculate the 1D array index of matrix entry `(i,j)` by:

```
#define IDX2C(i,j,ld) ( ( j * ld ) + i )
```

For 1-based row and column indices:

```
#define IDX2F(i,j,ld) ( ( (j - 1) * ld ) + ( i - 1 ) )
```

INCLUDING CUBLAS: WHICH HEADER TO USE

CUBLAS version 4.0 and above has a different API, which is supposed to be better.

Include "cublas_v2.h", for the new API. Use this one for new programs.
Include "cublas.h" for the old API. Use this one for programs that depend on the old API.

Things on the new API but not the old:

- `cublasCreate` initializes the handle to the CUBLAS library context, allowing more user control.
- Scalars α and β can be passed by reference to host and device functions in addition to by value.
- Scalars can be returned by reference in addition to by value.
- All CUBLAS functions return an error status, `cublasStatus_t`.
- `cublasAlloc()` and `cublasFree()` are deprecated. Use `cudaMalloc()` and `cudaFree()` instead.
- `cublasSetKernelStream()` was renamed `cublasSetStream()`.

CUBLAS CONTEXT

For CUBLAS version 4.0 and beyond, you must wrap your code like this:

```
cublasHandle_t handle;
cublasCreate(&handle);

// your code

cublasDestroy(handle);
```

and pass `handle` to every CUBLAS function in your code.

This approach allows the user to use multiple host threads and multiple GPUs.

STREAMS

Streams provide a way to run multiple *kernels* simultaneously on the GPU.

For more information, look up the following functions:

`cudaStreamCreate()`
`cublasSetStream()`

CUBLAS HELPER FUNCTIONS

```
cublasSetVector()
cublasGetVector()
cublasSetMatrix()
cublasGetMatrix()
```

```
cublasStatus_t cublasSetVector(int n, int elemSize,
                               const void *x, int incx, void *devicePtr, int incy)
```

Copies a CPU vector `x` to a GPU vector `y` pointed to by `devicePtr`.

- `n`: number of elements copied from `x`
- `elemSize`: size, in bytes, of each element copies
- `incx`: storage spacing between consecutive elements of CPU vector, `x`.
- `incy`: storage spacing between consecutive elements of GPU vector, `y` (or `devicePtr`).

```
cublasStatus_t cublasGetVector(int n, int elemSize,
                               const void *x, int incx, void *y, int incy)
```

Copies a GPU vector `x` to a CPU vector `y` pointed to by `devicePtr`.

- `n`: number of elements copied from `x`
- `elemSize`: size, in bytes, of each element copies
- `incx`: storage spacing between consecutive elements of CPU vector, `x`.
- `incy`: storage spacing between consecutive elements of GPU vector, `y` (or `devicePtr`).

```
cublasStatus_t cublasSetMatrix(int rows, int cols, int elemSize,
                               const void *A, int lda, void *B, int ldb)
```

Copies a column-major CPU matrix A to a column-major GPU matrix B.

```
cublasStatus_t cublasGetMatrix(int rows, int cols, int elemSize,
                               const void *A, int lda, void *B, int ldb)
```

Copies a column-major GPU matrix A to a column-major CPU matrix B.

- `lda`: number of rows in `A`
- `ldv`: number of rows in `B`

EXAMPLE1: CODE WITH CUBLAS

```
//Example 1. Application Using C and CUBLAS: 1-based indexing
//-----------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define M 6
#define N 5
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))

static __inline__ void modify (cublasHandle_t handle, float *m, int ldm, int n, int↩
    p, int q, float alpha, float beta){
    cublasSscal (handle, n-p+1, &alpha, &m[IDX2F(p,q,ldm)], ldm);
    cublasSscal (handle, ldm-p+1, &beta, &m[IDX2F(p,q,ldm)], 1);
}
```

```
int main (void){
    cudaError_t cudaStat;
    cublasStatus_t stat;
    cublasHandle_t handle;
    int i, j;
    float* devPtrA;
    float* a = 0;
    a = (float *)malloc (M * N * sizeof (*a));
    if (!a) {
        printf ("host memory allocation failed");
        return EXIT_FAILURE;
    }
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            a[IDX2F(i,j,M)] = (float)((i-1) * M + j);
        }
    }
    cudaStat = cudaMalloc ((void**)&devPtrA, M*N*sizeof(*a));
    if (cudaStat != cudaSuccess) {
        printf ("device memory allocation failed");
        return EXIT_FAILURE;
    }
    stat = cublasCreate(&handle);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("CUBLAS initialization failed\n");
        return EXIT_FAILURE;
    }
```

```
    stat = cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("data download failed");
        cudaFree (devPtrA);
        cublasDestroy(handle);
        return EXIT_FAILURE;
    }
    modify (handle, devPtrA, M, N, 2, 3, 16.0f, 12.0f);
    stat = cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("data upload failed");
        cudaFree (devPtrA);
        cublasDestroy(handle);
        return EXIT_FAILURE;
    }
    cudaFree (devPtrA);
    cublasDestroy(handle);
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            printf ("%7.0f", a[IDX2F(i,j,M)]);
        }
        printf ("\n");
    }
    return EXIT_SUCCESS;
}
```

simpleCUBLAS: MORE CODE WITH CUBLAS

```
/* Includes, system */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Includes, cuda */
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <shrQATest.h>

/* Matrix size */
#define N  (275)

/* Host implementation of a simple version of sgemm */
static void simple_sgemm(int n, float alpha, const float *A, const float *B,
                         float beta, float *C)
{
    int i;
    int j;
    int k;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            float prod = 0;
            for (k = 0; k < n; ++k) {
                prod += A[k * n + i] * B[j * n + k];
            }
            C[j * n + i] = alpha * prod + beta * C[j * n + i];
        }
    }
}
```

```
/* Main */
int main(int argc, char** argv)
{
    cublasStatus_t status;
    float* h_A;
    float* h_B;
    float* h_C;
    float* h_C_ref;
    float* d_A = 0;
    float* d_B = 0;
    float* d_C = 0;
    float alpha = 1.0f;
    float beta = 0.0f;
    int n2 = N * N;
    int i;
    float error_norm;
    float ref_norm;
    float diff;
    cublasHandle_t handle;

    shrQAStart(argc, argv);

    /* Initialize CUBLAS */
    printf("simpleCUBLAS test running..\n");

    status = cublasCreate(&handle);
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! CUBLAS initialization error\n");
        return EXIT_FAILURE;
    }
```

```
/* Allocate host memory for the matrices */
h_A = (float*)malloc(n2 * sizeof(h_A[0]));
if (h_A == 0) {
    fprintf (stderr, "!!!! host memory allocation error (A)\n");
    return EXIT_FAILURE;
}
h_B = (float*)malloc(n2 * sizeof(h_B[0]));
if (h_B == 0) {
    fprintf (stderr, "!!!! host memory allocation error (B)\n");
    return EXIT_FAILURE;
}
h_C = (float*)malloc(n2 * sizeof(h_C[0]));
if (h_C == 0) {
    fprintf (stderr, "!!!! host memory allocation error (C)\n");
    return EXIT_FAILURE;
}

/* Fill the matrices with test data */
for (i = 0; i < n2; i++) {
    h_A[i] = rand() / (float)RAND_MAX;
    h_B[i] = rand() / (float)RAND_MAX;
    h_C[i] = rand() / (float)RAND_MAX;
}
```

```
/* Allocate device memory for the matrices */
if (cudaMalloc((void**)&d_A, n2 * sizeof(d_A[0])) != cudaSuccess) {
    fprintf (stderr, "!!!! device memory allocation error (allocate A)\n");
    return EXIT_FAILURE;
}
if (cudaMalloc((void**)&d_B, n2 * sizeof(d_B[0])) != cudaSuccess) {
    fprintf (stderr, "!!!! device memory allocation error (allocate B)\n");
    return EXIT_FAILURE;
}
if (cudaMalloc((void**)&d_C, n2 * sizeof(d_C[0])) != cudaSuccess) {
    fprintf (stderr, "!!!! device memory allocation error (allocate C)\n");
    return EXIT_FAILURE;
}

/* Initialize the device matrices with the host matrices */
status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! device access error (write A)\n");
    return EXIT_FAILURE;
}
status = cublasSetVector(n2, sizeof(h_B[0]), h_B, 1, d_B, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! device access error (write B)\n");
    return EXIT_FAILURE;
}
status = cublasSetVector(n2, sizeof(h_C[0]), h_C, 1, d_C, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! device access error (write C)\n");
    return EXIT_FAILURE;
}
```

```
/* Performs operation using plain C code */
simple_sgemm(N, alpha, h_A, h_B, beta, h_C);
h_C_ref = h_C;

/* Performs operation using cublas */
status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N, d_B, N, &beta, d_C,
 N);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! kernel execution error.\n");
    return EXIT_FAILURE;
}

/* Allocate host memory for reading back the result from device memory */
h_C = (float*)malloc(n2 * sizeof(h_C[0]));
if (h_C == 0) {
    fprintf (stderr, "!!!! host memory allocation error (C)\n");
    return EXIT_FAILURE;
}

/* Read the result back */
status = cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! device access error (read C)\n");
    return EXIT_FAILURE;
}
```

```
/* Check result against reference */
error_norm = 0;
ref_norm = 0;
for (i = 0; i < n2; ++i) {
    diff = h_C_ref[i] - h_C[i];
    error_norm += diff * diff;
    ref_norm += h_C_ref[i] * h_C_ref[i];
}
error_norm = (float)sqrt((double)error_norm);
ref_norm = (float)sqrt((double)ref_norm);
if (fabs(ref_norm) < 1e-7) {
    fprintf (stderr, "!!!! reference norm is 0\n");
    return EXIT_FAILURE;
}

/* Memory clean up */
free(h_A);
free(h_B);
free(h_C);
free(h_C_ref);
if (cudaFree(d_A) != cudaSuccess) {
    fprintf (stderr, "!!!! memory free error (A)\n");
    return EXIT_FAILURE;
}
if (cudaFree(d_B) != cudaSuccess) {
    fprintf (stderr, "!!!! memory free error (B)\n");
    return EXIT_FAILURE;
}
if (cudaFree(d_C) != cudaSuccess) {
    fprintf (stderr, "!!!! memory free error (C)\n");
    return EXIT_FAILURE;
}
```

```
    /* Shutdown */
    status = cublasDestroy(handle);
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! shutdown error (A)\n");
        return EXIT_FAILURE;
    }

    shrQAFinish(argc, (const char **)argv, (error_norm / ref_norm < 1e-6f) ? QA_PASSED : QA_FAILED );

    return EXIT_SUCCESS;
}
```