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.

BEFORE COMPILING WITH CULBAS, CHOOSE WHICH .h FILE 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.

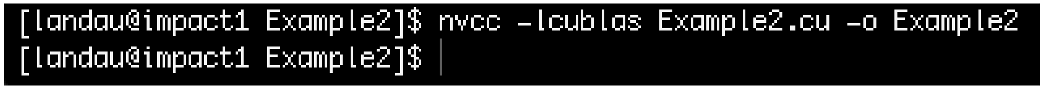
Things on the new API but not the old:

- `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()`.

COMPILING WITH CUBLAS

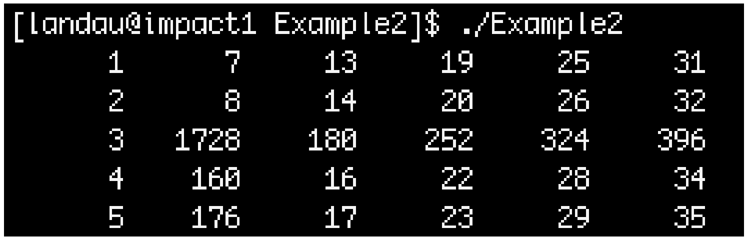
1. Include either “cublas_v2.h” or “cublas.h” in your source.
2. Compile with: **nvcc -lcublas your_source.cu -o your_binary**

Example:



```
[landau@impact1 Example2]$ nvcc -lcublas Example2.cu -o Example2
[landau@impact1 Example2]$
```

Then I can run the binary:



```
[landau@impact1 Example2]$ ./Example2
     1      7     13     19     25     31
     2      8     14     20     26     32
     3   1728    180    252    324    396
     4    160     16     22     28     34
     5    176     17     23     29     35
```

IMPLEMENTATION OF MATRICES

Matrices are implemented as linear arrays of memory. For example, CUBLAS thinks of this memory array:

1	1	2	3	5	8	13	21	34	55	89	144

as this matrix:

$$\begin{bmatrix} 1 & 2 & 5 & 13 & 34 & 89 \\ 1 & 3 & 8 & 21 & 55 & 144 \end{bmatrix} \qquad \text{or this matrix:} \qquad \begin{bmatrix} 1 & 5 & 34 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

depending on the number of rows and columns you specify.

NOTE: CUBLAS indexes matrices in column major format.

Let:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 \\ \hline\end{array}$$

$$B = \begin{bmatrix} 1 & 5 & 34 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

Then:

$$B[\text{row } i,\ \text{col } j] = A[j \cdot ld + i]$$

Where ld stands for "lead dimension". For column major order matrices, the lead dimension of a matrix is the number of elements in a column.

For indexing in your code, use a function or macro such as:

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

To go from matrix coordinates to the corresponding memory array index. [1]

[1] Note: use `#define IDX2F(i, j, ld) (j - 1) * ld + i-1` for 1-bases matrix coordinates

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:

```
cublasStreamCreate()
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`

LEVEL 1 FUNCTIONS

In R:	**float**	**double**	**cuComplex**	**cuDoubleComplex**
which.max(**x**)	cublasIsamax()	cublasIdamax()	cublasIcamax()	cublasIzamax()
which.min(**x**)	cublasIsamin()	cublasIdamin()	cublasIcamin()	cublasIzamin()
sum(abs(**x**))	cublasSasum()	cublasDasum()	cublasScasum()	cublasDzasum()
α*x + y -> y	cublasSaxpy()	cublasDaxpy()	cublasCaxpy()	cublasZaxpy()
x -> y	cublasScopy()	cublasDcopy()	cublasCcopy()	cublasZcopy()

sum(x * y)	cublasSdot()	cublasDdot()	cublasCdotu()
			cublasCdotc()
				cublasZdotu()
				cublasZdotc()

sqrt(sum(x^2))	cublasSnrm2()	cublasDnrm2()	cublasScnrm2()	cublasDznrm2()

G %*% x; G %*% y	cublasSrot()	cublasDrot()	cublasCrot()
			cublasCsrot()
				cublasZrot()
				cublasZdrot()

H %*% x; H %*% y	cublasSrotm()	cublasDotm()		

α * x -> x	cublasSscal()	cublasDscal()	cublasCscal()
			cublasCsscal()
				cublasZscal()
				cublasZdscal()

x -> **m**; **y** -> **x**; **m** -> **y**	cublasSswap()	cublasDswap()	cublasCswap()	cublasZswap()

Where α is a scalar, `x` and `y` are vectors, `G` $= \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$, and `H` is some 2×2 matrix.

LEVEL 2 FUNCTIONS

$$\mathrm{op}(A)\mathbf{x} \to \mathbf{x}$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

type of matrix A	float	double	cuComplex	cuDoubleComplex
triangular, banded	cublasStbmv()	cublasDtbmv()	cublasCtbmv()	cublasZtbmv()
triangular, packed format	cublasStpmv()	cublasDtpmv()	cublasCtpmv()	cublasZtpmv()
triangular, upper or lower mode	cublasStrmv()	cublasDtrmv()	cublasCtrmv()	cublasZtrmv()

$$\text{Solve} \qquad \text{op}(A)\mathbf{x} = \mathbf{b} \qquad \text{for } \mathbf{x}$$

where:

$$\text{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

type of matrix A	float	double	cuComplex	cuDoubleComplex
triangular, banded	cublasStbsv()	cublasDtbsv()	cublasCtbsv()	cublasZtbsv()
triangular, packed format	cublasStpsv()	cublasDtpsv()	cublasCtpsv()	cublasZtpsv()
triangular, upper or lower mode	cublasStrsv()	cublasDtrsv()	cublasCtrsv()	cublasZtrsv()

$$\alpha \mathbf{x}\mathbf{x}^T + A \to A$$

type of matrix A	float	double
symmetric matrix	cublasSsyr()	cublasDsyr()
symmetric matrix in packed format	cublasSspr()	cublasDspr()

$$\alpha \mathbf{x}\mathbf{x}^H + A \to A$$

type of matrix A	cuComplex	cuDoubleComplex
Hermitian	cublasCher()	cublasZher()
Hermitian, packed format	cublasChpr()	cublasZhpr()

$$\alpha \mathbf{x}\mathbf{y}^T + A \to A$$

type of matrix A	float	double	cuComplex	cuDoubleComplex
any $m \times n$ matrix	cublasSger()	cublasDger()	cublasCgeru()	cublasZgeru()

$$\alpha \mathbf{x}\mathbf{y}^H + A \to A$$

type of matrix A	cuComplex	cuDoubleComplex
any $m \times n$ matrix	cublasCgerc()	cublasZgerc()

$$\alpha(\mathbf{x}\mathbf{y}^T + \mathbf{y}\mathbf{x}^T) + A \rightarrow A$$

type of matrix A	float	double
symmetric matrix	cublasSsyr2()	cublasDsyr2()
symmetric matrix in packed format	cublasSspr2()	cublasDspr2()

$$\alpha(\mathbf{x}\mathbf{y}^H + \mathbf{y}\mathbf{x}^H) + A \rightarrow A$$

type of matrix A	cuComplex	cuDoubleComplex
Hermitian	cublasCher2()	cublasZher2()
Hermitian, packed format	cublasChpr2()	cublasZhpr2()

$$\alpha \cdot \mathrm{op}(A)\mathbf{x} + \beta\mathbf{y} \to \mathbf{y}$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

type of matrix A	float	double	cuComplex	cuDoubleComplex
any $m \times n$	cublasSgemv()	cublasDgemv()	cublasCgemv	cublasZgemv()
$m \times n$, banded	cublasSgbmv()	cublasDgbmv()	cublasCgbmv	cublasZgbmv()
symmetric, banded [2]	cublasSsbmv()	cublasDsbmv()	-	-
symmetric, packed format [1]	cublasSspmv()	cublasDspmv()	-	-
symmetric, lower/upper mode [1]	cublasSsymv()	cublasDsymv()	-	-
Hermitian [1]	-	-	cublasChemv()	cublasZhemv()
Hermitian, banded [1]	-	-	cublasChbmv()	cublasZhbmv()

[2] Here, op(A) = A with no `transa` option.

LEVEL 3 FUNCTIONS

$$\alpha \cdot \mathrm{op}(A)\mathrm{op}(B) + \beta C \to C$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
any with compatible sizes	cublasSgemm()	cublasDgemm()	cublasCgemm()	cublasZgemm()

Batch of `batchCount` matrices:

$$\alpha \cdot \mathrm{op}(A[i])\mathrm{op}(B[i]) + \beta C[i] \to C[i]$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices types $A[i]$, $B[i]$, $C[i]$	any with compatible sizes
float function	cublasSgemmBatched()
double function	cublasDgemmBatched()
cuComplex function	cublasCgemmBatched()
cuDoubleComplex function	cublasZgemmBatched()

$$\left.\begin{array}{ll}\alpha AB+\beta C & : \texttt{side == CUBLAS\_SIDE\_LEFT} \\ \alpha BA+\beta C & : \texttt{side == CUBLAS\_SIDE\_RIGHT}\end{array}\right\} \rightarrow C$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
A: symmetric, lower or upper mode	cublasSsymm()	cublasDsymm()	cublasCsymm()	cublasZsymm()
A: Hermitian, lower or upper mode	-	-	cublasChemm()	cublasZhemm()

$$\left.\begin{array}{ll} \alpha A A^T + \beta C & : \texttt{trans == CUBLAS\_OP\_N} \\ \alpha A^T A + \beta C & : \texttt{trans == CUBLAS\_OP\_T} \end{array}\right\} \rightarrow C$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
C: symmetric, lower or upper mode	cublasSsyrk()	cublasDsyrk()	cublasCsyrk()	cublasZsyrk()

$$\left.\begin{array}{ll}\alpha(AB^T + BA^T) + \beta C & : \texttt{trans == CUBLAS\_OP\_N} \\ \alpha(A^T B + B^T A) + \beta C & : \texttt{trans == CUBLAS\_OP\_T}\end{array}\right\} \to C$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
C: symmetric, lower or upper mode	cublasSsyr2k()	cublasDsyr2k()	cublasCsyr2k()	cublasZsyr2k()

$$\left.\begin{array}{ll} \alpha \mathrm{op}(A)B & : \texttt{trans == CUBLAS\_SIDE\_LEFT} \\ \alpha B\mathrm{op}(A) & : \texttt{trans == CUBLAS\_SIDE\_RIGHT} \end{array}\right\} \to C$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices A, B	float	double	cuComplex	cuDoubleComplex
A: triangular, lower or upper mode	cublasStrmm()	cublasDtrmm()	cublasCtrmm()	cublasZtrmm()

Solve for X:

$$\begin{cases} \mathrm{op}(A)X = \alpha B & : \texttt{trans == CUBLAS\_SIDE\_LEFT} \\ X\mathrm{op}(A) = \alpha B & : \texttt{trans == CUBLAS\_SIDE\_RIGHT} \end{cases}$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^T & : \texttt{transa == CUBLAS\_OP\_T} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices A, B, X	float	double	cuComplex	cuDoubleComplex
A: triangular, lower or upper mode	cublasStrsm()	cublasDtrsm()	cublasCtrsm()	cublasZtrsm()

$$\alpha \cdot \mathrm{op}(A)\mathrm{op}(A)^H + \beta C \to C$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
C: Hermitian, lower or upper mode	-	-	cublasCherk()	cublasZherk()

$$\alpha \cdot \mathrm{op}(A)\mathrm{op}(B)^H + \overline{\alpha}\mathrm{op}(B)\mathrm{op}(A)^H + \beta \cdot C \to C$$

where:

$$\mathrm{op}(A) = \begin{cases} A & : \texttt{transa == CUBLAS\_OP\_N} \\ A^H & : \texttt{transa == CUBLAS\_OP\_C} \end{cases}$$

matrices A, B, C	float	double	cuComplex	cuDoubleComplex
C: Hermitian, lower or upper mode	-	-	cublasCher2k()	cublasZher2k()

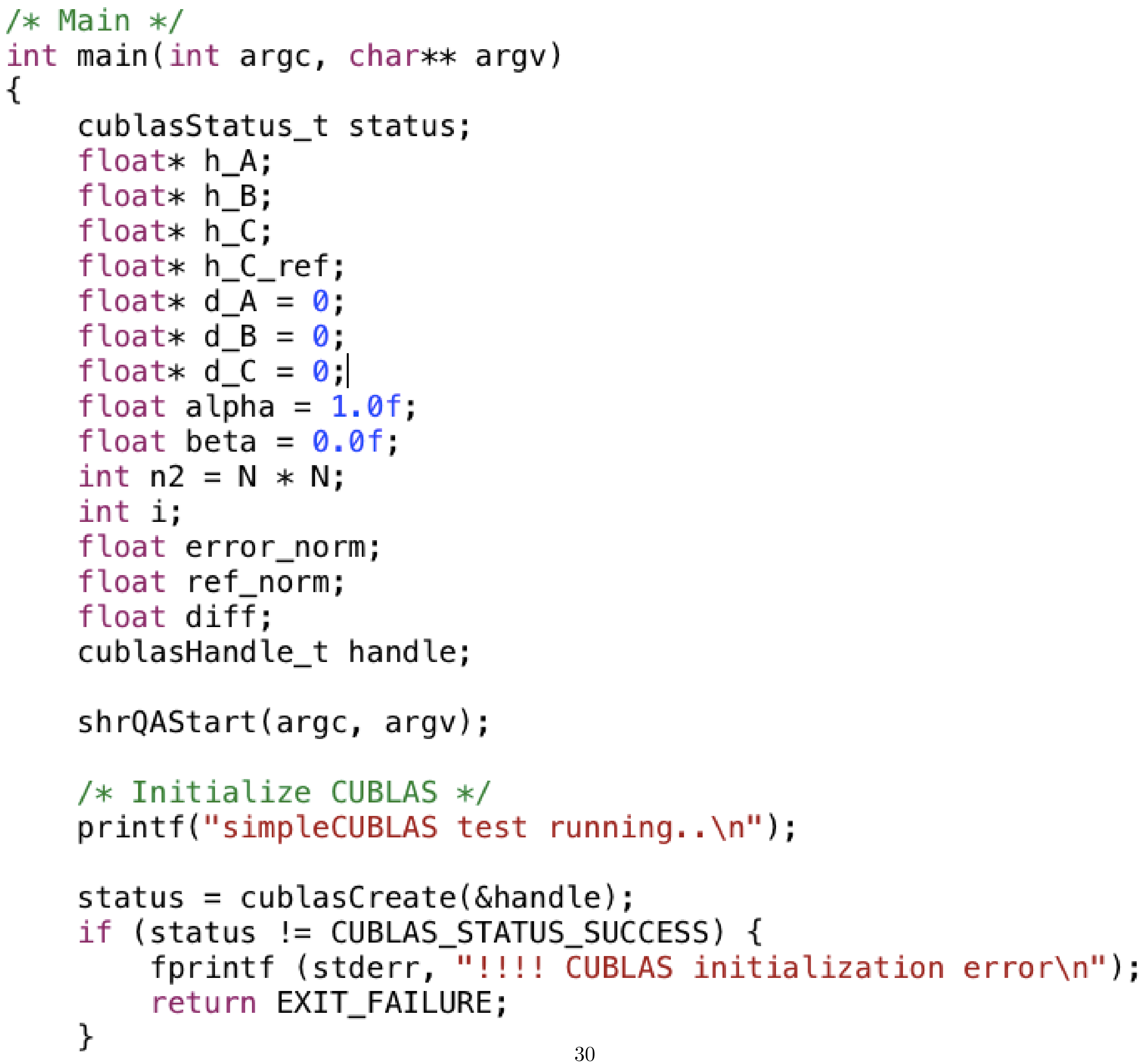
simpleCUBLAS: EXAMPLE CUBLAS CODE

```
/* 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];
        }
    }
}
```

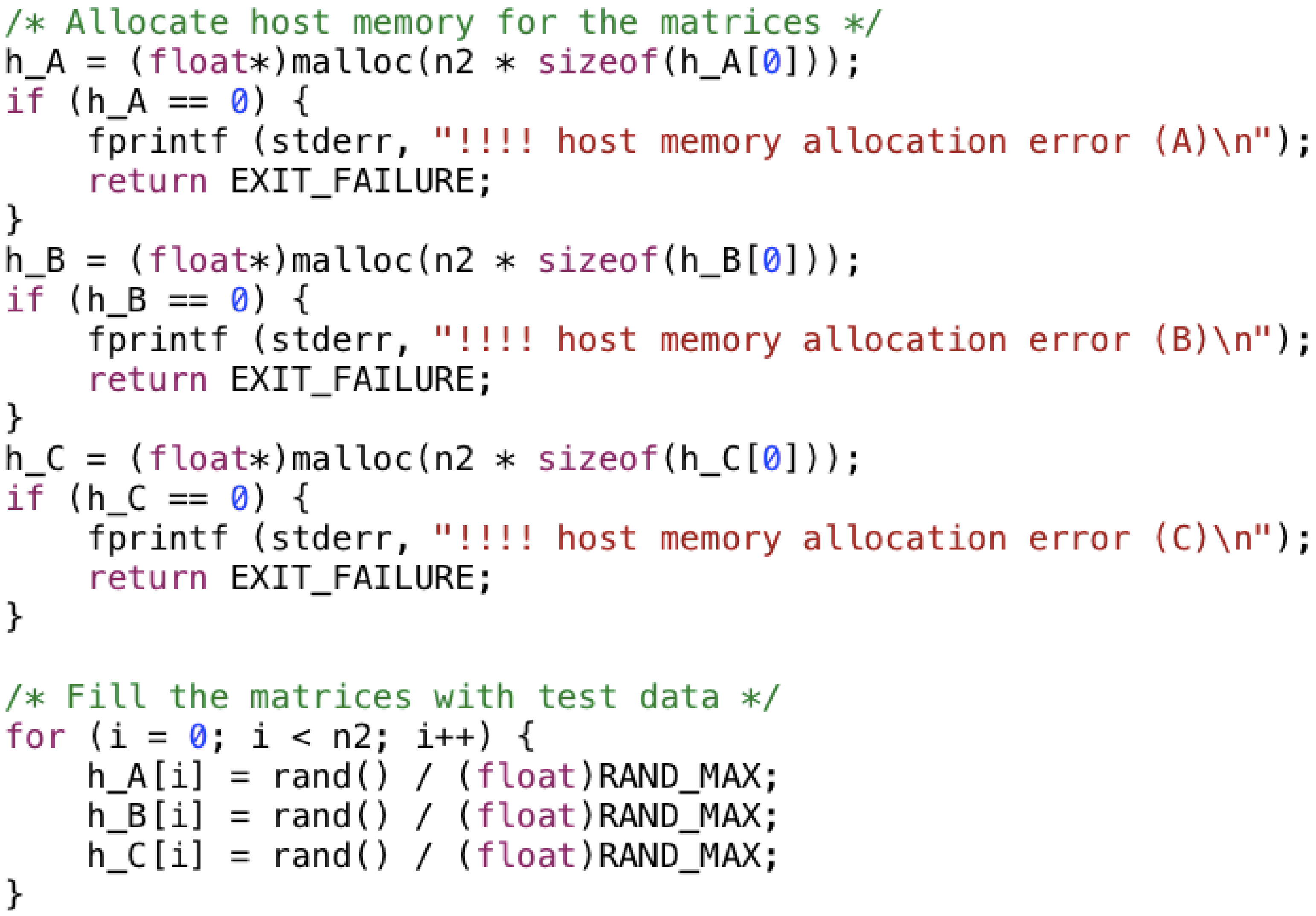


```
/* Main */
int main(int argc, char** argv)
{
    cublasStatus_t status;
    float* h_A;
    float* h_B;
    float* h_C;
    float* h_C_ref;
    float* d_A = 0;
    float* d_B = 0;
    float* d_C = 0;
    float alpha = 1.0f;
    float beta = 0.0f;
    int n2 = N * N;
    int i;
    float error_norm;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

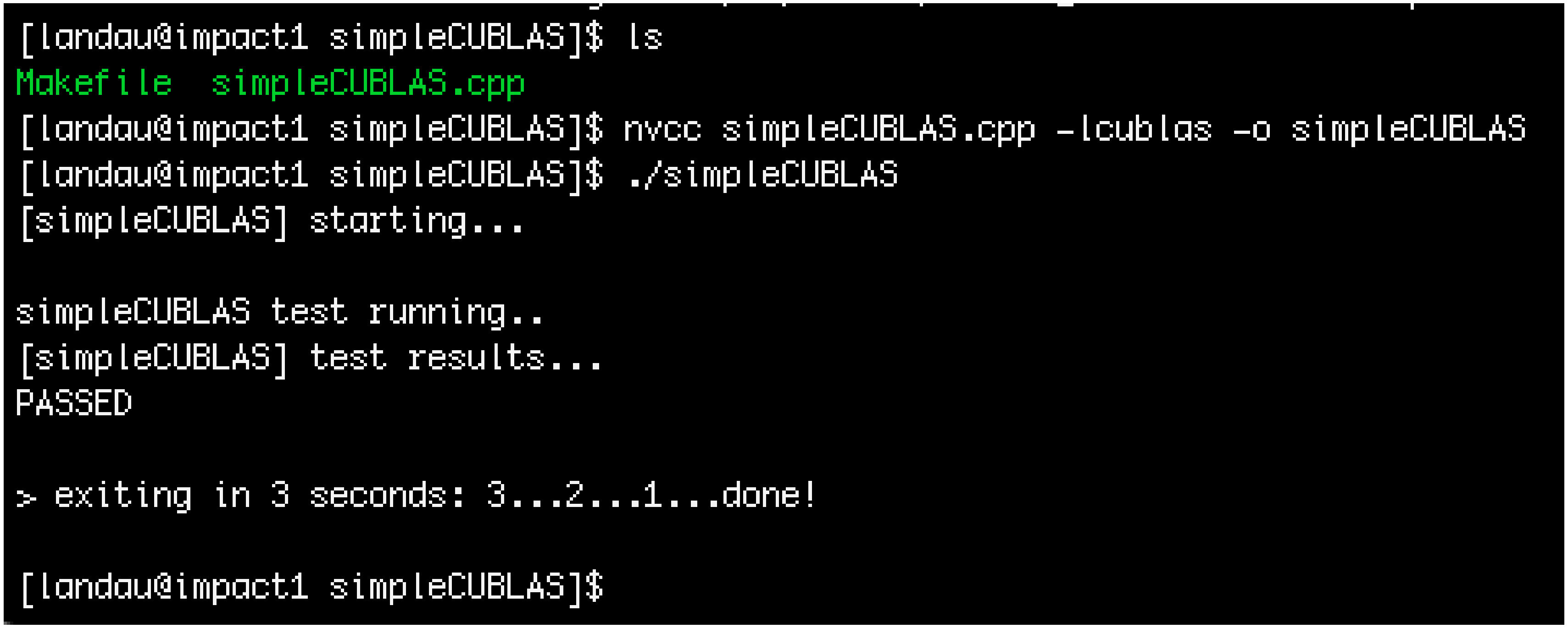
```
/* Check result against reference */
error_norm = 0;
ref_norm = 0;
for (i = 0; i < n2; ++i) {
    diff = h_C_ref[i] - h_C[i];
    error_norm += diff * diff;
    ref_norm += h_C_ref[i] * h_C_ref[i];
}
error_norm = (float)sqrt((double)error_norm);
ref_norm = (float)sqrt((double)ref_norm);
if (fabs(ref_norm) < 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;
}
```



```
[landau@impact1 simpleCUBLAS]$ ls
Makefile  simpleCUBLAS.cpp
[landau@impact1 simpleCUBLAS]$ nvcc simpleCUBLAS.cpp -lcublas -o simpleCUBLAS
[landau@impact1 simpleCUBLAS]$ ./simpleCUBLAS
[simpleCUBLAS] starting...

simpleCUBLAS test running..
[simpleCUBLAS] test results...
PASSED

> exiting in 3 seconds: 3...2...1...done!

[landau@impact1 simpleCUBLAS]$
```

QUICK REVIEW: IMPLEMENTATION OF MATRICES

Matrices are implemented as linear arrays of memory. For example, CUBLAS thinks of this memory array:

1	1	2	3	5	8	13	21	34	55	89	144

as this matrix:

$$\begin{bmatrix} 1 & 2 & 5 & 13 & 34 & 89 \\ 1 & 3 & 8 & 21 & 55 & 144 \end{bmatrix} \qquad \text{or this matrix:} \qquad \begin{bmatrix} 1 & 5 & 34 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

depending on the number of rows and columns you specify.

NOTE: CUBLAS indexes matrices in column major format.

Let:

$$A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 \\ \hline \end{array}$$

$$B = \begin{bmatrix} 1 & 5 & 34 \\ 1 & 8 & 55 \\ 2 & 13 & 89 \\ 3 & 21 & 144 \end{bmatrix}$$

Then:

$$B[\text{row } i,\ \text{col } j] = A[j \cdot ld + i]$$

Where ld stands for "lead dimension". For column major order matrices, the lead dimension of a matrix is the number of elements in a column.

For indexing in your code, use a function or macro such as:

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

To go from matrix coordinates to the corresponding memory array index. [3]

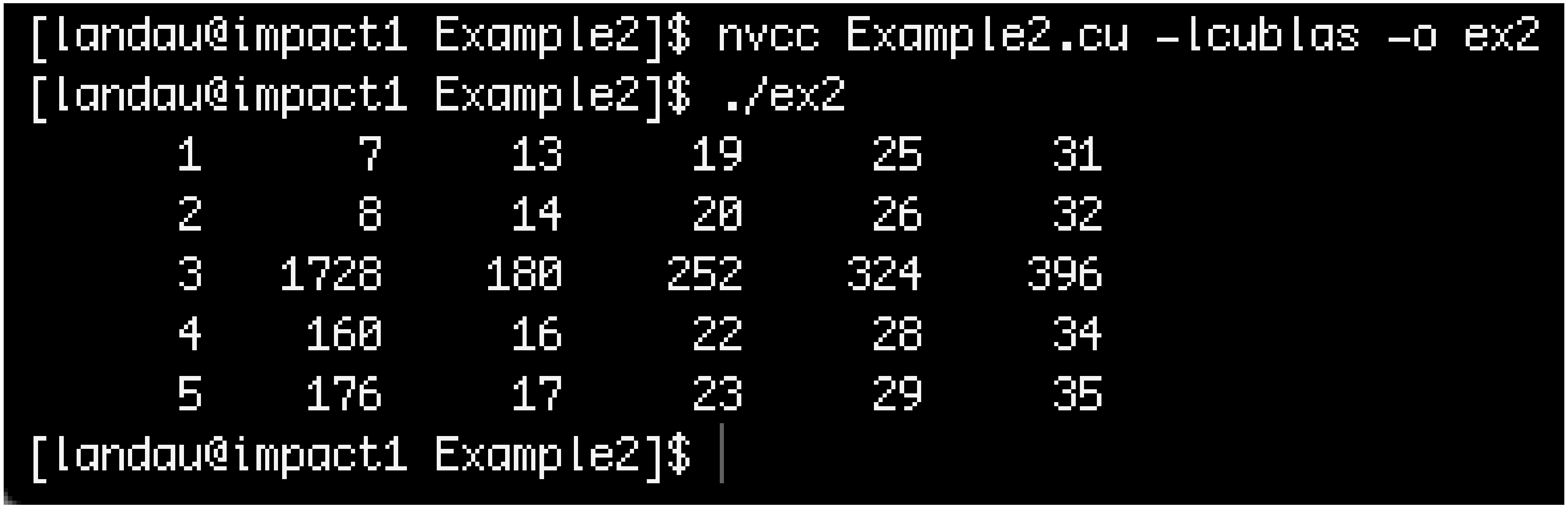
[3] Note: use `#define IDX2F(i, j, ld) (j - 1) * ld + i-1` for 1-bases matrix coordinates

EXAMPLE2: MORE CUBLAS CODE

```
#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;
}
```



```
[landau@impact1 Example2]$ nvcc Example2.cu -lcublas -o ex2
[landau@impact1 Example2]$ ./ex2
      1      7     13     19     25     31
      2      8     14     20     26     32
      3   1728    180    252    324    396
      4    160     16     22     28     34
      5    176     17     23     29     35
[landau@impact1 Example2]$ 
```

GPU SERIES MATERIALS

These slides, a tentative syllabus for the whole series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the materials.

REFERENCES

”CUDA Toolkit 4.2 CUBLAS Library”.
http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf