**Lab Exercise 9.1 – Using cuBLAS – A CUDA Library for Linear Algebra Operations**

**Objective:**

* Understand how to utilize cuBLAS (CUDA Basic Linear Algebra Subprograms) for GPU-accelerated matrix operations.
* Learn how to link and use cuBLAS in a CUDA C++ program.
* Execute and compare performance for matrix multiplication using cuBLAS.

**1. What is cuBLAS?**

cuBLAS is NVIDIA’s GPU-accelerated version of the BLAS library. It provides high-performance matrix and vector operations optimized for CUDA-enabled GPUs.

Key operations include:

* Vector addition/scaling
* Matrix-vector multiplication
* Matrix-matrix multiplication

**2. Requirements:**

* CUDA Toolkit installed
* A CUDA-enabled GPU
* g++ and nvcc (NVIDIA compiler)

**3. Program: Matrix Multiplication using cuBLAS**

#include <iostream>

#include <cublas\_v2.h>

#include <cuda\_runtime.h>

#define N 2 // Size of square matrices

int main() {

// Host matrices

float h\_A[N \* N] = {1.0, 2.0, 3.0, 4.0};

float h\_B[N \* N] = {5.0, 6.0, 7.0, 8.0};

float h\_C[N \* N]; // Result

// Device matrices

float \*d\_A, \*d\_B, \*d\_C;

cudaMalloc((void\*\*)&d\_A, N \* N \* sizeof(float));

cudaMalloc((void\*\*)&d\_B, N \* N \* sizeof(float));

cudaMalloc((void\*\*)&d\_C, N \* N \* sizeof(float));

// Copy host data to device

cudaMemcpy(d\_A, h\_A, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

// Create cuBLAS handle

cublasHandle\_t handle;

cublasCreate(&handle);

// Matrix multiplication: C = alpha\*A\*B + beta\*C

const float alpha = 1.0f;

const float beta = 0.0f;

// cuBLAS is column-major, so the order of A and B is swapped

cublasSgemm(handle,

CUBLAS\_OP\_N, CUBLAS\_OP\_N,

N, N, N,

&alpha,

d\_B, N, // B is passed first

d\_A, N,

&beta,

d\_C, N);

// Copy result back to host

cudaMemcpy(h\_C, d\_C, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Display result

std::cout << "Result matrix C = A x B:" << std::endl;

for (int i = 0; i < N \* N; i++) {

std::cout << h\_C[i] << " ";

if ((i + 1) % N == 0) std::cout << std::endl;

}

// Clean up

cublasDestroy(handle);

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

return 0;

}

**4. Explanation:**

* Matrices are stored in row-major order on the host but cuBLAS expects column-major.
* cublasSgemm performs single-precision matrix multiplication.
* alpha and beta control scaling (you can make C = A\*B directly with alpha = 1, beta = 0).

**5. Compilation:**

Use the following command to compile:

nvcc -o matrix\_mul\_cublas matrix\_mul\_cublas.cu -lcublas

Then run:

./matrix\_mul\_cublas

**6. Expected Output:**

Result matrix C = A x B:

19 22

43 50

Which corresponds to the multiplication:

[1 2] [5 6] [1×5+2×7 1×6+2×8] [19 22]

[3 4] × [7 8] = [3×5+4×7 3×6+4×8] = [43 50]

**7. Conclusion:**

This lab shows how to:

* Set up GPU memory
* Use cuBLAS for matrix multiplication
* Transfer results back to the host

**Detailed Explanation of the Code Using cuBLAS**

The code provided demonstrates how to perform matrix multiplication using **cuBLAS**, which is a GPU-accelerated BLAS (Basic Linear Algebra Subprograms) library provided by NVIDIA. Specifically, this code performs a matrix multiplication using the cublasSgemm function.

Let's break down each part of the code in detail:

**1. Create cuBLAS Handle**

cublasHandle\_t handle;

cublasCreate(&handle);

* **cublasHandle\_t handle**: This is a handle that represents a cuBLAS context. A context in cuBLAS is needed to manage and track operations.
* **cublasCreate(&handle)**: This function initializes the cuBLAS library and creates a handle that will be used to execute cuBLAS functions (like matrix multiplication). The handle needs to be passed as the first argument to each cuBLAS function.

**2. Matrix Multiplication Formula**

const float alpha = 1.0f;

const float beta = 0.0f;

* **alpha** and **beta** are scaling factors for the matrix multiplication operation. The matrix multiplication formula in cuBLAS is:

C=α⋅A⋅B+β⋅CC = \alpha \cdot A \cdot B + \beta \cdot CC=α⋅A⋅B+β⋅C

In this case:

* + **alpha = 1.0f**: This means the result of A \* B will be multiplied by 1.0.
  + **beta = 0.0f**: This means the existing values in matrix C will be scaled by 0.0, essentially ignoring the current values of C and overwriting them with the result of the matrix multiplication.

These values are passed as pointers to the cublasSgemm function, and they control how the result matrix C is computed.

The constants alpha and beta in the cublasSgemm() function are **scalars** that control how the result matrix C is computed:

C=α⋅A⋅B+β⋅CC = \alpha \cdot A \cdot B + \beta \cdot CC=α⋅A⋅B+β⋅C

While alpha = 1.0f and beta = 0.0f are commonly used for a pure matrix multiplication (C = A \* B), **you can set them to other values** depending on your use case.

**Examples of Other Values and Their Effects**

| **alpha** | **beta** | **Resulting Formula** | **Use Case** |
| --- | --- | --- | --- |
| 1.0f | 1.0f | C = A \* B + C | Accumulate new product result into existing C |
| 2.0f | 0.0f | C = 2 \* A \* B | Scale result by a factor of 2 |
| 1.0f | -1.0f | C = A \* B - C | Subtract old C from the new product result |
| 0.5f | 0.5f | C = 0.5 \* A \* B + 0.5 \* C | Average of the new result and the old content of C |
| 0.0f | 1.0f | C = C | No change – this skips the multiplication and keeps the current C |
| 1.0f | 2.0f | C = A \* B + 2 \* C | Adds the current result to twice the previous value in C |

**3. Matrix Multiplication Using cublasSgemm**

cublasSgemm(handle,

CUBLAS\_OP\_N, CUBLAS\_OP\_N,

N, N, N,

&alpha,

d\_B, N, // B is passed first

d\_A, N,

&beta,

d\_C, N);

This line calls the **cublasSgemm** function, which is used for matrix multiplication in cuBLAS. Here's a detailed breakdown of the parameters:

**Function Signature:**

cublasStatus\_t cublasSgemm(cublasHandle\_t handle,

cublasOperation\_t transa, cublasOperation\_t transb,

int m, int n, int k,

const float \*alpha,

const float \*A, int lda,

const float \*B, int ldb,

const float \*beta,

float \*C, int ldc);

**Parameters:**

1. **handle**: The handle created earlier, representing the cuBLAS context.
2. **CUBLAS\_OP\_N**: This indicates that the matrices A and B will **not** be transposed before the multiplication. The first argument (CUBLAS\_OP\_N) corresponds to matrix A, and the second (CUBLAS\_OP\_N) corresponds to matrix B.
   * CUBLAS\_OP\_N means **no transpose**.
   * CUBLAS\_OP\_T would indicate that the matrix is to be transposed.
   * CUBLAS\_OP\_C means the conjugate transpose.
3. **N**: This is the dimension of the square matrices A, B, and C. Specifically, A is an N x N matrix, B is also N x N, and C will be an N x N matrix.
4. **alpha**: This is the scalar multiplier for the result of the matrix multiplication A \* B. In this case, alpha = 1.0f, so the result of A \* B will be scaled by 1.
5. **d\_B, N**: This is the pointer to matrix B, and N is the leading dimension of matrix B. In cuBLAS, matrices are stored in **column-major order**, so N represents the stride between successive columns of B in memory.
   * d\_B is a pointer to the device memory where matrix B is stored.
   * The second argument N represents the leading dimension (lda) of matrix B, which is typically the number of rows in B. Since B is N x N, lda is simply N.
6. **d\_A, N**: This is the pointer to matrix A and its leading dimension. Again, d\_A points to the device memory where matrix A is stored, and N is the leading dimension (lda) of A.
7. **beta**: This is the scalar multiplier for the matrix C. In this case, beta = 0.0f, so the existing values in C are ignored and replaced by the result of the multiplication.
8. **d\_C, N**: This is the pointer to matrix C, which will store the result of the matrix multiplication. The leading dimension (ldc) of matrix C is also N.

**Matrix Dimensions:**

* **Matrix A**: N x N
* **Matrix B**: N x N
* **Matrix C**: N x N (output matrix)

**Order of Matrices in the Operation:**

cuBLAS uses **column-major** order, meaning matrices are stored such that the elements of each column are stored contiguously in memory. Therefore, the order of multiplication is slightly different compared to other libraries that may use row-major order. In cuBLAS, matrix B is passed **first** and matrix A is passed **second** in the function call. This is because of the way the library is optimized for column-major storage.

**4. Result of Matrix Multiplication**

Once the cublasSgemm function is executed, the result of the matrix multiplication A \* B is stored in matrix C. Since beta = 0.0f, the initial values of C are ignored and replaced by the result of alpha \* A \* B.

**Summary of Key Points:**

* **cuBLAS handle**: Required for cuBLAS functions to track the cuBLAS context.
* **Matrix multiplication formula**: C = alpha \* A \* B + beta \* C, with alpha = 1.0f and beta = 0.0f, meaning the result of A \* B will be directly stored in C.
* **Column-major order**: cuBLAS uses column-major order, so matrix B is passed **first** in the multiplication operation.
* **Matrix dimensions**: The matrices are all N x N, and the multiplication is performed using cublasSgemm, which is optimized for matrix-matrix multiplication on the GPU.

The constants CUBLAS\_OP\_N, CUBLAS\_OP\_T, and CUBLAS\_OP\_C in cuBLAS are used to **control how each input matrix is interpreted** before being used in operations like matrix multiplication (e.g., in cublasSgemm). These flags **do not modify the matrix data** itself; they only tell cuBLAS how to **treat the matrix** internally.

**Purpose:**

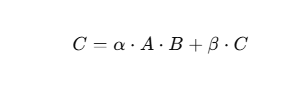
When calling a cuBLAS routine like:

cublasSgemm(handle, transA, transB, m, n, k, &alpha, A, lda, B, ldb, &beta, C, ldc);

The transA and transB parameters can be:

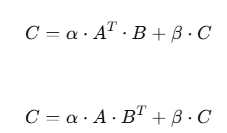
**1. CUBLAS\_OP\_N — No Transpose**

* **Meaning**: Use matrix A or B as-is.
* **Effect**: cuBLAS will treat the input matrix as **not transposed**.
* **Example**:



**2. CUBLAS\_OP\_T — Transpose**

* **Meaning**: Use the **transpose** of matrix A or B.
* **Effect**: cuBLAS will **interpret** the matrix as transposed (without modifying the actual matrix).



* **Use Case**: You want to multiply a matrix by the **transpose of another**, or align shapes for multiplication.

**3. CUBLAS\_OP\_C — Conjugate Transpose (Hermitian)**

* **Meaning**: Use the **conjugate transpose** of matrix A or B.
* **Effect**: For **complex matrices**, it conjugates and transposes.
* **Example**:



where A^H is the Hermitian (conjugate transpose) of A.

* **Use Case**: Required when dealing with **complex numbers** in quantum simulations, signal processing, etc.

**Why Use These?**

1. **Save Memory**: You don’t need to create a separate transposed matrix in memory.
2. **Performance**: cuBLAS internally handles transposition efficiently, avoiding manual transpose operations.
3. **Shape Flexibility**: Helps match dimensions during multiplication:
   * If A is 3x2 and B is 3x4, you can’t multiply directly. But with:

CUBLAS\_OP\_T for A // Now A becomes 2x3

the multiplication becomes valid.

**Matrix Shapes for Examples:**

Let's say we have:

* A: a **2×3** matrix
* B: a **3×4** matrix
* C: will be a **2×4** matrix (as output)

**1. Example: No Transpose (CUBLAS\_OP\_N, CUBLAS\_OP\_N)**

// C = A \* B

cublasSgemm(handle,

CUBLAS\_OP\_N, // A: 2×3

CUBLAS\_OP\_N, // B: 3×4

2, 4, 3, // m, n, k (A: m×k, B: k×n)

&alpha,

d\_A, 2, // lda = number of rows in A

d\_B, 3, // ldb = number of rows in B

&beta,

d\_C, 2); // ldc = number of rows in C

**Result**: Standard matrix multiplication:



**2. Example: Transpose A (CUBLAS\_OP\_T, CUBLAS\_OP\_N)**

// A is 3×2 (we treat it as transposed)

cublasSgemm(handle,

CUBLAS\_OP\_T, // Transpose A: becomes 3×2

CUBLAS\_OP\_N, // B: 3×4

3, 4, 2, // m = rows of A^T, n = cols of B, k = rows of A

&alpha,

d\_A, 2, // lda = leading dim of original A

d\_B, 3,

&beta,

d\_C, 3); // ldc = m

**Result**:



* **Use case**: You want to use the transpose of A without allocating new memory.

**3. Example: Transpose B (CUBLAS\_OP\_N, CUBLAS\_OP\_T)**

// B is 4×3, but we use its transpose (i.e., 3×4)

cublasSgemm(handle,

CUBLAS\_OP\_N, // A: 2×3

CUBLAS\_OP\_T, // Transpose B: becomes 3×4

2, 3, 4, // A: 2×4, B^T: 4×3

&alpha,

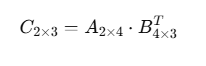
d\_A, 2,

d\_B, 4,

&beta,

d\_C, 2);

**Result**:



**4. Example: Conjugate Transpose (CUBLAS\_OP\_C)**

For **complex matrices**, this is especially useful.

Let’s say cuComplex \*d\_A is a matrix of complex numbers. Then:

cublasCgemm(handle,

CUBLAS\_OP\_C, // conjugate transpose A

CUBLAS\_OP\_N,

m, n, k,

&alpha,

d\_A, lda,

d\_B, ldb,

&beta,

d\_C, ldc);

**Result**:



where AHA^HAH is the **conjugate transpose** of A.

* **Use case**: Signal processing, quantum computing, FFTs — where complex math is standard.