```
////////////////////////////////////////////////////////////////////////////
//
// Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
//
// Please refer to the NVIDIA end user license agreement (EULA) associated
// with this source code for terms and conditions that govern your use of
// this software. Any use, reproduction, disclosure, or distribution of
// this software and related documentation outside the terms of the EULA
// is strictly prohibited.
//
////////////////////////////////////////////////////////////////////////////

//
// Matrix multiplication: C = A * B.
// Host code.
//
// This sample implements matrix multiplication as described in Chapter 3
// of the programming guide and uses the CUBLAS library to demonstrate
// the best performance.

// SOME PRECAUTIONS:
// IF WE WANT TO CALCULATE ROW-MAJOR MATRIX MULTIPLY C = A * B,
// WE JUST NEED CALL CUBLAS API IN A REVERSE ORDER: cublasSegemm(B, A)!
// The reason is explained as follows:

// CUBLAS library uses column-major storage, but C/C++ use row-major storage.
// When passing the matrix pointer to CUBLAS, the memory layout alters from
// row-major to column-major, which is equivalent to an implicit transpose.

// In the case of row-major C/C++ matrix A, B, and a simple matrix multiplication
// C = A * B, we can't use the input order like cublasSgemm(A, B)  because of
// implicit transpose. The actual result of cublasSegemm(A, B) is A(T) * B(T).
// If col(A(T)) != row(B(T)), equal to row(A) != col(B), A(T) and B(T) are not
// multipliable. Moreover, even if A(T) and B(T) are multipliable, the result C
// is a column-based cublas matrix, which means C(T) in C/C++, we need extra
// transpose code to convert it to a row-based C/C++ matrix.

// To solve the problem, let's consider our desired result C, a row-major matrix.
// In cublas format, it is C(T) actually (because of the implicit transpose).
// C = A * B, so C(T) = (A * B) (T) = B(T) * A(T). Cublas matrice B(T) and A(T)
// happen to be C/C++ matrice B and A (still because of the implicit transpose)!
// We don't need extra transpose code, we only need alter the input order!
//
// CUBLAS provides high-performance matrix multiplication.
```

```
// See also:
// V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra,"
// in Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC '08),
// Piscataway, NJ: IEEE Press, 2008, pp. Art. 31:1-11.
//

// Utilities and system includes
#include <assert.h>
#include <helper_string.h>  // helper for shared functions common to CUDA
Samples

// CUDA runtime
#include <cuda_runtime.h>
#include <cublas_v2.h>

// CUDA and CUBLAS functions
#include <helper_functions.h>
#include <helper_cuda.h>

#ifndef min
#define min(a,b) ((a < b) ? a : b)
#endif
#ifndef max
#define max(a,b) ((a > b) ? a : b)
#endif

typedef struct _matrixSize      // Optional Command-line multiplier for matrix sizes
{
    unsigned int uiWA, uiHA, uiWB, uiHB, uiWC, uiHC;
} sMatrixSize;

////////////////////////////////////////////////////////////////////////////////
//! Compute reference data set matrix multiply on CPU
//! C = A * B
//! @param C          reference data, computed but preallocated
//! @param A          matrix A as provided to device
//! @param B          matrix B as provided to device
//! @param hA         height of matrix A
//! @param wB         width of matrix B
////////////////////////////////////////////////////////////////////////////////
void
matrixMulCPU(float *C, const float *A, const float *B, unsigned int hA, unsigned int
wA, unsigned int wB)
{
```

```
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j)
        {
            double sum = 0;

            for (unsigned int k = 0; k < wA; ++k)
            {
                double a = A[i * wA + k];
                double b = B[k * wB + j];
                sum += a * b;
            }

            C[i * wB + j] = (float)sum;
        }
}

// Allocates a matrix with random float entries.
void randomInit(float *data, int size)
{
    for (int i = 0; i < size; ++i)
        data[i] = rand() / (float)RAND_MAX;
}

void printDiff(float *data1, float *data2, int width, int height, int iListLength, float
fListTol)
{
    printf("Listing first %d Differences > %.6f...\n", iListLength, fListTol);
    int i,j,k;
    int error_count=0;

    for (j = 0; j < height; j++)
    {
        if (error_count < iListLength)
        {
            printf("\n  Row %d:\n", j);
        }

        for (i = 0; i < width; i++)
        {
            k = j * width + i;
            float fDiff = fabs(data1[k] - data2[k]);

            if (fDiff > fListTol)
            {
```

```
            if (error_count < iListLength)
            {
                printf("    Loc(%d,%d)\tCPU=%.5f\tGPU=%.5f\tDiff=%.6f\n", i, j,
data1[k], data2[k], fDiff);
            }

            error_count++;
        }
    }
}

printf(" \n  Total Errors = %d\n", error_count);
}

void initializeCUDA(int argc, char **argv, int &devID, int &iSizeMultiple, sMatrixSize
&matrix_size)
{
    // By default, we use device 0, otherwise we override the device ID based on
what is provided at the command line
    cudaError_t error;
    devID = 0;

    if (checkCmdLineFlag(argc, (const char **)argv, "device"))
    {
        devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
        error = cudaSetDevice(devID);

        if (error != cudaSuccess)
        {
            printf("cudaSetDevice returned error code %d, line(%d)\n", error, __LINE__);
            exit(EXIT_FAILURE);
        }
    }

    // get number of SMs on this GPU
    error = cudaGetDevice(&devID);

    if (error != cudaSuccess)
    {
        printf("cudaGetDevice returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }
```

```
    if (checkCmdLineFlag(argc, (const char **)argv, "sizemult"))
    {
        iSizeMultiple = getCmdLineArgumentInt(argc, (const char **)argv, "sizemult");
    }

    iSizeMultiple = min(iSizeMultiple, 10);
    iSizeMultiple = max(iSizeMultiple, 1);

    cudaDeviceProp deviceProp;

    error = cudaGetDeviceProperties(&deviceProp, devID);

    if (error != cudaSuccess)
    {
        printf("cudaGetDeviceProperties returned error code %d, line(%d)\n", error,
__LINE__);
        exit(EXIT_FAILURE);
    }

    printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);

    // use a larger block size for Fermi and above
    int block_size = (deviceProp.major < 2) ? 16 : 32;

    matrix_size.uiWA = 3 * block_size * iSizeMultiple;
    matrix_size.uiHA = 4 * block_size * iSizeMultiple;
    matrix_size.uiWB = 2 * block_size * iSizeMultiple;
    matrix_size.uiHB = 3 * block_size * iSizeMultiple;
    matrix_size.uiWC = 2 * block_size * iSizeMultiple;
    matrix_size.uiHC = 4 * block_size * iSizeMultiple;

    printf("MatrixA(%u,%u), MatrixB(%u,%u), MatrixC(%u,%u)\n",
           matrix_size.uiHA, matrix_size.uiWA,
           matrix_size.uiHB, matrix_size.uiWB,
           matrix_size.uiHC, matrix_size.uiWC);

    if( matrix_size.uiWA != matrix_size.uiHB ||
        matrix_size.uiHA != matrix_size.uiHC ||
        matrix_size.uiWB != matrix_size.uiWC)
    {
       printf("ERROR: Matrix sizes do not match!\n");
       exit(-1);
    }
```

```
}

////////////////////////////////////////////////////////////////////////////////
//! Run a simple test matrix multiply using CUBLAS
////////////////////////////////////////////////////////////////////////////////
int matrixMultiply(int argc, char **argv, int devID, sMatrixSize &matrix_size)
{
    cudaDeviceProp deviceProp;

    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, devID));

    // use a larger block size for Fermi and above
    int block_size = (deviceProp.major < 2) ? 16 : 32;

    // set seed for rand()
    srand(2006);

    // allocate host memory for matrices A and B
    unsigned int size_A = matrix_size.uiWA * matrix_size.uiHA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = matrix_size.uiWB * matrix_size.uiHB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float *h_B = (float *)malloc(mem_size_B);

    // set seed for rand()
    srand(2006);

    // initialize host memory
    randomInit(h_A, size_A);
    randomInit(h_B, size_B);

    // allocate device memory
    float *d_A, *d_B, *d_C;
    unsigned int size_C = matrix_size.uiWC * matrix_size.uiHC;
    unsigned int mem_size_C = sizeof(float) * size_C;

    // allocate host memory for the result
    float *h_C      = (float *) malloc(mem_size_C);
    float *h_CUBLAS = (float *) malloc(mem_size_C);

    checkCudaErrors(cudaMalloc((void **) &d_A, mem_size_A));
    checkCudaErrors(cudaMalloc((void **) &d_B, mem_size_B));
    checkCudaErrors(cudaMemcpy(d_A, h_A, mem_size_A,
```

```
cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_B, h_B, mem_size_B,
cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMalloc((void **) &d_C, mem_size_C));

    // setup execution parameters
    dim3 threads(block_size, block_size);
    dim3 grid(matrix_size.uiWC / threads.x, matrix_size.uiHC / threads.y);

    // create and start timer
    printf("Computing result using CUBLAS...");

    // execute the kernel
    int nIter = 30;

    // CUBLAS version 2.0
    {
        const float alpha = 1.0f;
        const float beta  = 0.0f;
        cublasHandle_t handle;
        cudaEvent_t start, stop;

        checkCudaErrors(cublasCreate(&handle));

        //Perform warmup operation with cublas
        checkCudaErrors(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
matrix_size.uiWB, matrix_size.uiHA, matrix_size.uiWA, &alpha, d_B,
matrix_size.uiWB, d_A, matrix_size.uiWA, &beta, d_C, matrix_size.uiWB));

        // Allocate CUDA events that we'll use for timing
        checkCudaErrors(cudaEventCreate(&start));
        checkCudaErrors(cudaEventCreate(&stop));

        // Record the start event
        checkCudaErrors(cudaEventRecord(start, NULL));

        for (int j = 0; j < nIter; j++)
        {
            //note cublas is column primary!
            //need to transpose the order
            checkCudaErrors(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
matrix_size.uiWB, matrix_size.uiHA, matrix_size.uiWA, &alpha, d_B,
matrix_size.uiWB, d_A, matrix_size.uiWA, &beta, d_C, matrix_size.uiWB));
```

```
        }

        printf("done.\n");

        // Record the stop event
        checkCudaErrors(cudaEventRecord(stop, NULL));

        // Wait for the stop event to complete
        checkCudaErrors(cudaEventSynchronize(stop));

        float msecTotal = 0.0f;
        checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));

        // Compute and print the performance
        float msecPerMatrixMul = msecTotal / nIter;
        double flopsPerMatrixMul = 2.0 * (double)matrix_size.uiHC *
(double)matrix_size.uiWC * (double)matrix_size.uiHB;
        double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul /
1000.0f);
        printf(
            "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops\n",
            gigaFlops,
            msecPerMatrixMul,
            flopsPerMatrixMul);

        // copy result from device to host
        checkCudaErrors(cudaMemcpy(h_CUBLAS, d_C, mem_size_C,
cudaMemcpyDeviceToHost));

        // Destroy the handle
        checkCudaErrors(cublasDestroy(handle));
    }

    // compute reference solution
    printf("Computing result using host CPU...");
    float *reference = (float *)malloc(mem_size_C);
    matrixMulCPU(reference, h_A, h_B, matrix_size.uiHA, matrix_size.uiWA,
matrix_size.uiWB);
    printf("done.\n");

    // check result (CUBLAS)
    bool resCUBLAS = sdkCompareL2fe(reference, h_CUBLAS, size_C, 1.0e-6f);

    if (resCUBLAS != true)
```

```
    {
        printDiff(reference, h_CUBLAS, matrix_size.uiWC, matrix_size.uiHC, 100, 
1.0e-5f);
    }

    printf("Comparing CUBLAS Matrix Multiply with CPU results: %s\n", (true == 
resCUBLAS) ? "PASS" : "FAIL");

    printf("\nNOTE: The CUDA Samples are not meant for performance 
measurements. Results may vary when GPU Boost is enabled.\n");

    // clean up memory
    free(h_A);
    free(h_B);
    free(h_C);
    free(reference);
    checkCudaErrors(cudaFree(d_A));
    checkCudaErrors(cudaFree(d_B));
    checkCudaErrors(cudaFree(d_C));

    // cudaDeviceReset causes the driver to clean up all state. While
    // not mandatory in normal operation, it is good practice.  It is also
    // needed to ensure correct operation when the application is being
    // profiled. Calling cudaDeviceReset causes all profile data to be
    // flushed before the application exits
    cudaDeviceReset();

    if (resCUBLAS == true)
    {
        return EXIT_SUCCESS;    // return value = 1
    }
    else
    {
        return EXIT_FAILURE;     // return value = 0
    }
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    printf("[Matrix Multiply CUBLAS] - Starting...\n");
```

```
    int devID = 0, sizeMult = 5;
    sMatrixSize matrix_size;

    initializeCUDA(argc, argv, devID, sizeMult, matrix_size);

    int matrix_result = matrixMultiply(argc, argv, devID, matrix_size);

    return matrix_result;
}
```