Программа перемножения двух векторов «matrixMul.cu» из Cuda Samples 8.0

```
/**
 * 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.
 * It has been written for clarity of exposition to illustrate various CUDA
 * programming principles, not with the goal of providing the most
 * performant generic kernel for 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.
 */

// System includes
#include <stdio.h>
#include <assert.h>
```

```
31
32  // CUDA runtime
33  #include <cuda_runtime.h>
34
35  // Helper functions and utilities to work with CUDA
36  #include <helper_functions.h>
37  #include <helper_cuda.h>
38
39  /**
40   * Matrix multiplication (CUDA Kernel) on the device: C = A * B
41   * wA is A's width and wB is B's width
42   */
43  template <int BLOCK_SIZE> __global__ void
44  matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
45  {
46      // Block index
47      int bx = blockIdx.x;
48      int by = blockIdx.y;
49
50      // Thread index
51      int tx = threadIdx.x;
52      int ty = threadIdx.y;
53
54      // Index of the first sub-matrix of A processed by the block
55      int aBegin = wA * BLOCK_SIZE * by;
56
57      // Index of the last sub-matrix of A processed by the block
58      int aEnd   = aBegin + wA - 1;
59
60      // Step size used to iterate through the sub-matrices of A
61      int aStep  = BLOCK_SIZE;
```

```
62
63      // Index of the first sub-matrix of B processed by the block
64      int bBegin = BLOCK_SIZE * bx;
65
66      // Step size used to iterate through the sub-matrices of B
67      int bStep  = BLOCK_SIZE * wB;
68
69      // Csub is used to store the element of the block sub-matrix
70      // that is computed by the thread
71      float Csub = 0;
72
73      // Loop over all the sub-matrices of A and B
74      // required to compute the block sub-matrix
75      for (int a = aBegin, b = bBegin;
76           a <= aEnd;
77           a += aStep, b += bStep)
78      {
79
80          // Declaration of the shared memory array As used to
81          // store the sub-matrix of A
82          __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
83
84          // Declaration of the shared memory array Bs used to
85          // store the sub-matrix of B
86          __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
87
88          // Load the matrices from device memory
89          // to shared memory; each thread loads
90          // one element of each matrix
91          As[ty][tx] = A[a + wA * ty + tx];
92          Bs[ty][tx] = B[b + wB * ty + tx];
```

```
93
94          // Synchronize to make sure the matrices are loaded
95          __syncthreads();
96
97          // Multiply the two matrices together;
98          // each thread computes one element
99          // of the block sub-matrix
100     #pragma unroll
101
102         for (int k = 0; k < BLOCK_SIZE; ++k)
103         {
104            Csub += As[ty][k] * Bs[k][tx];
105         }
106
107         // Synchronize to make sure that the preceding
108         // computation is done before loading two new
109         // sub-matrices of A and B in the next iteration
110         __syncthreads();
111     }
112
113     // Write the block sub-matrix to device memory;
114     // each thread writes one element
115     int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
116     C[c + wB * ty + tx] = Csub;
117 }
118
119 void constantInit(float *data, int size, float val)
120 {
121     for (int i = 0; i < size; ++i)
122     {
123         data[i] = val;
```

```
124         }
125     }
126
127     /**
128      * Run a simple test of matrix multiplication using CUDA
129      */
130     int matrixMultiply(int argc, char **argv, int block_size, dim3 &dimsA, dim3 &dimsB)
131     {
132         // Allocate host memory for matrices A and B
133         unsigned int size_A = dimsA.x * dimsA.y;
134         unsigned int mem_size_A = sizeof(float) * size_A;
135         float *h_A = (float *)malloc(mem_size_A);
136         unsigned int size_B = dimsB.x * dimsB.y;
137         unsigned int mem_size_B = sizeof(float) * size_B;
138         float *h_B = (float *)malloc(mem_size_B);
139
140         // Initialize host memory
141         const float valB = 0.01f;
142         constantInit(h_A, size_A, 1.0f);
143         constantInit(h_B, size_B, valB);
144
145         // Allocate device memory
146         float *d_A, *d_B, *d_C;
147
148         // Allocate host matrix C
149         dim3 dimsC(dimsB.x, dimsA.y, 1);
150         unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
151         float *h_C = (float *) malloc(mem_size_C);
152
153         if (h_C == NULL)
154         {
```

```
155         fprintf(stderr, "Failed to allocate host matrix C!\n");
156         exit(EXIT_FAILURE);
157     }
158
159     cudaError_t error;
160
161     error = cudaMalloc((void **) &d_A, mem_size_A);
162
163     if (error != cudaSuccess)
164     {
165         printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
166         exit(EXIT_FAILURE);
167     }
168
169     error = cudaMalloc((void **) &d_B, mem_size_B);
170
171     if (error != cudaSuccess)
172     {
173         printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
174         exit(EXIT_FAILURE);
175     }
176
177     error = cudaMalloc((void **) &d_C, mem_size_C);
178
179     if (error != cudaSuccess)
180     {
181         printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
182         exit(EXIT_FAILURE);
183     }
184
185     // copy host memory to device
```

```
186     error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
187
188     if (error != cudaSuccess)
189     {
190        printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
191        exit(EXIT_FAILURE);
192     }
193
194     error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);
195
196     if (error != cudaSuccess)
197     {
198        printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
199        exit(EXIT_FAILURE);
200     }
201
202     // Setup execution parameters
203     dim3 threads(block_size, block_size);
204     dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
205
206     // Create and start timer
207     printf("Computing result using CUDA Kernel...\n");
208
209     // Performs warmup operation using matrixMul CUDA kernel
210     if (block_size == 16)
211     {
212        matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
213     }
214     else
215     {
216        matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
```

```
217     }
218
219     printf("done\n");
220
221     cudaDeviceSynchronize();
222
223     // Allocate CUDA events that we'll use for timing
224     cudaEvent_t start;
225     error = cudaEventCreate(&start);
226
227     if (error != cudaSuccess)
228     {
229         fprintf(stderr, "Failed to create start event (error code %s)!\n", cudaGetErrorString(error));
230         exit(EXIT_FAILURE);
231     }
232
233     cudaEvent_t stop;
234     error = cudaEventCreate(&stop);
235
236     if (error != cudaSuccess)
237     {
238         fprintf(stderr, "Failed to create stop event (error code %s)!\n", cudaGetErrorString(error));
239         exit(EXIT_FAILURE);
240     }
241
242     // Record the start event
243     error = cudaEventRecord(start, NULL);
244
245     if (error != cudaSuccess)
246     {
247         fprintf(stderr, "Failed to record start event (error code %s)!\n", cudaGetErrorString(error));
```

```
248         exit(EXIT_FAILURE);
249     }
250
251     // Execute the kernel
252     int nIter = 300;
253
254     for (int j = 0; j < nIter; j++)
255     {
256         if (block_size == 16)
257         {
258             matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
259         }
260         else
261         {
262             matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
263         }
264     }
265
266     // Record the stop event
267     error = cudaEventRecord(stop, NULL);
268
269     if (error != cudaSuccess)
270     {
271         fprintf(stderr, "Failed to record stop event (error code %s)!\n", cudaGetErrorString(error));
272         exit(EXIT_FAILURE);
273     }
274
275     // Wait for the stop event to complete
276     error = cudaEventSynchronize(stop);
277
278     if (error != cudaSuccess)
```

```
279     {
280         fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n", cudaGetErrorString(error));
281         exit(EXIT_FAILURE);
282     }
283
284     float msecTotal = 0.0f;
285     error = cudaEventElapsedTime(&msecTotal, start, stop);
286
287     if (error != cudaSuccess)
288     {
289         fprintf(stderr, "Failed to get time elapsed between events (error code %s)!\n", cudaGetErrorString(error));
290         exit(EXIT_FAILURE);
291     }
292
293     // Compute and print the performance
294     float msecPerMatrixMul = msecTotal / nIter;
295     double flopsPerMatrixMul = 2.0 * (double)dimsA.x * (double)dimsA.y * (double)dimsB.x;
296     double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
297     printf(
298         "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops, WorkgroupSize= %u threads/block\n",
299         gigaFlops,
300         msecPerMatrixMul,
301         flopsPerMatrixMul,
302         threads.x * threads.y);
303
304     // Copy result from device to host
305     error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);
306
307     if (error != cudaSuccess)
308     {
309         printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
```

```
310         exit(EXIT_FAILURE);
311     }
312
313     printf("Checking computed result for correctness: ");
314     bool correct = true;
315
316     // test relative error by the formula
317     //     |<x, y>_cpu - <x,y>_gpu|/<|x|, |y|>  < eps
318     double eps = 1.e-6 ; // machine zero
319
320     for (int i = 0; i < (int)(dimsC.x * dimsC.y); i++)
321     {
322         double abs_err = fabs(h_C[i] - (dimsA.x * valB));
323         double dot_length = dimsA.x;
324         double abs_val = fabs(h_C[i]);
325         double rel_err = abs_err/abs_val/dot_length ;
326
327         if (rel_err > eps)
328         {
329             printf("Error! Matrix[%05d]=%.8f, ref=%.8f error term is > %E\n", i, h_C[i], dimsA.x*valB, eps);
330             correct = false;
331         }
332     }
333
334     printf("%s\n", correct ? "Result = PASS" : "Result = FAIL");
335
336     // Clean up memory
337     free(h_A);
338     free(h_B);
339     free(h_C);
340     cudaFree(d_A);
```

```
341     cudaFree(d_B);
342     cudaFree(d_C);
343
344     printf("\nNOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.\n");
345
346     if (correct)
347     {
348         return EXIT_SUCCESS;
349     }
350     else
351     {
352         return EXIT_FAILURE;
353     }
354 }
355
356
357 /**
358  * Program main
359  */
360 int main(int argc, char **argv)
361 {
362     printf("[Matrix Multiply Using CUDA] - Starting...\n");
363
364     if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
365         checkCmdLineFlag(argc, (const char **)argv, "?"))
366     {
367         printf("Usage -device=n (n >= 0 for deviceID)\n");
368         printf("      -wA=WidthA -hA=HeightA (Width x Height of Matrix A)\n");
369         printf("      -wB=WidthB -hB=HeightB (Width x Height of Matrix B)\n");
370         printf("  Note: Outer matrix dimensions of A & B matrices must be equal.\n");
371
```

```
372         exit(EXIT_SUCCESS);
373     }
374
375     // By default, we use device 0, otherwise we override the device ID based on what is provided at the command line
376     int devID = 0;
377
378     if (checkCmdLineFlag(argc, (const char **)argv, "device"))
379     {
380         devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
381         cudaSetDevice(devID);
382     }
383
384     cudaError_t error;
385     cudaDeviceProp deviceProp;
386     error = cudaGetDevice(&devID);
387
388     if (error != cudaSuccess)
389     {
390         printf("cudaGetDevice returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
391     }
392
393     error = cudaGetDeviceProperties(&deviceProp, devID);
394
395     if (deviceProp.computeMode == cudaComputeModeProhibited)
396     {
397         fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no threads can use ::cudaSetDevice().\n");
398         exit(EXIT_SUCCESS);
399     }
400
401     if (error != cudaSuccess)
402     {
```

```
403         printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n", cudaGetErrorString(error), error, __LINE__);
404     }
405     else
406     {
407         printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID, deviceProp.name, deviceProp.major, deviceProp.minor);
408     }
409
410     // Use a larger block size for Fermi and above
411     int block_size = (deviceProp.major < 2) ? 16 : 32;
412
413     dim3 dimsA(5*3*block_size, 5*4*block_size, 1);
414     dim3 dimsB(5*2*block_size, 5*3*block_size, 1);
415
416     // width of Matrix A
417     if (checkCmdLineFlag(argc, (const char **)argv, "wA"))
418     {
419         dimsA.x = getCmdLineArgumentInt(argc, (const char **)argv, "wA");
420     }
421
422     // height of Matrix A
423     if (checkCmdLineFlag(argc, (const char **)argv, "hA"))
424     {
425         dimsA.y = getCmdLineArgumentInt(argc, (const char **)argv, "hA");
426     }
427
428     // width of Matrix B
429     if (checkCmdLineFlag(argc, (const char **)argv, "wB"))
430     {
431         dimsB.x = getCmdLineArgumentInt(argc, (const char **)argv, "wB");
432     }
433
```

```
    // height of Matrix B
    if (checkCmdLineFlag(argc, (const char **)argv, "hB"))
    {
        dimsB.y = getCmdLineArgumentInt(argc, (const char **)argv, "hB");
    }

    if (dimsA.x != dimsB.y)
    {
        printf("Error: outer matrix dimensions must be equal. (%d != %d)\n",
               dimsA.x, dimsB.y);
        exit(EXIT_FAILURE);
    }

    printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", dimsA.x, dimsA.y, dimsB.x, dimsB.y);

    int matrix_result = matrixMultiply(argc, argv, block_size, dimsA, dimsB);

    exit(matrix_result);
}
```

```
==========================================================================
MatrixA(480,640), MatrixB(320,480)
Computing result using CUDA Kernel...
Performance= 12.32 GFlop/s, Time= 15.958 msec, Size= 196608000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
==========================================================================
```

Программа перемножения двух векторов «matrixMulCUBLAS» из Cuda Samples 8.0

```
////////////////////////////////////////////////////////////////////////////
//
// Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
```

```
4   //
5   // Please refer to the NVIDIA end user license agreement (EULA) associated
6   // with this source code for terms and conditions that govern your use of
7   // this software. Any use, reproduction, disclosure, or distribution of
8   // this software and related documentation outside the terms of the EULA
9   // is strictly prohibited.
10  //
11  ////////////////////////////////////////////////////////////////////////////
12
13  //
14  // Matrix multiplication: C = A * B.
15  // Host code.
16  //
17  // This sample implements matrix multiplication as described in Chapter 3
18  // of the programming guide and uses the CUBLAS library to demonstrate
19  // the best performance.
20
21  // SOME PRECAUTIONS:
22  // IF WE WANT TO CALCULATE ROW-MAJOR MATRIX MULTIPLY C = A * B,
23  // WE JUST NEED CALL CUBLAS API IN A REVERSE ORDER: cublasSegemm(B, A)!
24  // The reason is explained as follows:
25
26  // CUBLAS library uses column-major storage, but C/C++ use row-major storage.
27  // When passing the matrix pointer to CUBLAS, the memory layout alters from
28  // row-major to column-major, which is equivalent to an implicit transpose.
29
30  // In the case of row-major C/C++ matrix A, B, and a simple matrix multiplication
31  // C = A * B, we can't use the input order like cublasSgemm(A, B)  because of
32  // implicit transpose. The actual result of cublasSegemm(A, B) is A(T) * B(T).
33  // If col(A(T)) != row(B(T)), equal to row(A) != col(B), A(T) and B(T) are not
34  // multipliable. Moreover, even if A(T) and B(T) are multipliable, the result C
```

```
35  // is a column-based cublas matrix, which means C(T) in C/C++, we need extra
36  // transpose code to convert it to a row-based C/C++ matrix.
37
38  // To solve the problem, let's consider our desired result C, a row-major matrix.
39  // In cublas format, it is C(T) actually (because of the implicit transpose).
40  // C = A * B, so C(T) = (A * B) (T) = B(T) * A(T). Cublas matrice B(T) and A(T)
41  // happen to be C/C++ matrice B and A (still because of the implicit transpose)!
42  // We don't need extra transpose code, we only need alter the input order!
43  //
44  // CUBLAS provides high-performance matrix multiplication.
45  // See also:
46  // V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra,"
47  // in Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC '08),
48  // Piscataway, NJ: IEEE Press, 2008, pp. Art. 31:1-11.
49  //
50
51  // Utilities and system includes
52  #include <assert.h>
53  #include <helper_string.h>  // helper for shared functions common to CUDA Samples
54
55  // CUDA runtime
56  #include <cuda_runtime.h>
57  #include <cublas_v2.h>
58
59  // CUDA and CUBLAS functions
60  #include <helper_functions.h>
61  #include <helper_cuda.h>
62
63  #ifndef min
64  #define min(a,b) ((a < b) ? a : b)
65  #endif
```

```
66  #ifndef max
67  #define max(a,b) ((a > b) ? a : b)
68  #endif
69
70  typedef struct _matrixSize      // Optional Command-line multiplier for matrix sizes
71  {
72      unsigned int uiWA, uiHA, uiWB, uiHB, uiWC, uiHC;
73  } sMatrixSize;
74
75  ////////////////////////////////////////////////////////////////////////////////
76  //! Compute reference data set matrix multiply on CPU
77  //! C = A * B
78  //! @param C          reference data, computed but preallocated
79  //! @param A          matrix A as provided to device
80  //! @param B          matrix B as provided to device
81  //! @param hA         height of matrix A
82  //! @param wB         width of matrix B
83  ////////////////////////////////////////////////////////////////////////////////
84  void
85  matrixMulCPU(float *C, const float *A, const float *B, unsigned int hA, unsigned int wA, unsigned int wB)
86  {
87      for (unsigned int i = 0; i < hA; ++i)
88          for (unsigned int j = 0; j < wB; ++j)
89          {
90              double sum = 0;
91
92              for (unsigned int k = 0; k < wA; ++k)
93              {
94                  double a = A[i * wA + k];
95                  double b = B[k * wB + j];
96                  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]);

```

```
128             if (fDiff > fListTol)
129             {
130                 if (error_count < iListLength)
131                 {
132                     printf("    Loc(%d,%d)\tCPU=%.5f\tGPU=%.5f\tDiff=%.6f\n", i, j, data1[k], data2[k], fDiff);
133                 }
134
135                 error_count++;
136             }
137         }
138     }
139
140     printf(" \n  Total Errors = %d\n", error_count);
141 }
142
143 void initializeCUDA(int argc, char **argv, int &devID, int &iSizeMultiple, sMatrixSize &matrix_size)
144 {
145     // By default, we use device 0, otherwise we override the device ID based on what is provided at the command line
146     cudaError_t error;
147     devID = 0;
148
149     if (checkCmdLineFlag(argc, (const char **)argv, "device"))
150     {
151         devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
152         error = cudaSetDevice(devID);
153
154         if (error != cudaSuccess)
155         {
156             printf("cudaSetDevice returned error code %d, line(%d)\n", error, __LINE__);
157             exit(EXIT_FAILURE);
158         }
```

```
159     }
160
161     // get number of SMs on this GPU
162     error = cudaGetDevice(&devID);
163
164     if (error != cudaSuccess)
165     {
166        printf("cudaGetDevice returned error code %d, line(%d)\n", error, __LINE__);
167        exit(EXIT_FAILURE);
168     }
169
170
171     if (checkCmdLineFlag(argc, (const char **)argv, "sizemult"))
172     {
173        iSizeMultiple = getCmdLineArgumentInt(argc, (const char **)argv, "sizemult");
174     }
175
176     iSizeMultiple = min(iSizeMultiple, 10);
177     iSizeMultiple = max(iSizeMultiple, 1);
178
179     cudaDeviceProp deviceProp;
180
181     error = cudaGetDeviceProperties(&deviceProp, devID);
182
183     if (error != cudaSuccess)
184     {
185        printf("cudaGetDeviceProperties returned error code %d, line(%d)\n", error, __LINE__);
186        exit(EXIT_FAILURE);
187     }
188
189     printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID, deviceProp.name, deviceProp.major, deviceProp.minor);
```

```
190
191         // use a larger block size for Fermi and above
192         int block_size = (deviceProp.major < 2) ? 16 : 32;
193
194         matrix_size.uiWA = 3 * block_size * iSizeMultiple;
195         matrix_size.uiHA = 4 * block_size * iSizeMultiple;
196         matrix_size.uiWB = 2 * block_size * iSizeMultiple;
197         matrix_size.uiHB = 3 * block_size * iSizeMultiple;
198         matrix_size.uiWC = 2 * block_size * iSizeMultiple;
199         matrix_size.uiHC = 4 * block_size * iSizeMultiple;
200
201         printf("MatrixA(%u,%u), MatrixB(%u,%u), MatrixC(%u,%u)\n",
202                matrix_size.uiHA, matrix_size.uiWA,
203                matrix_size.uiHB, matrix_size.uiWB,
204                matrix_size.uiHC, matrix_size.uiWC);
205
206         if( matrix_size.uiWA != matrix_size.uiHB ||
207             matrix_size.uiHA != matrix_size.uiHC ||
208             matrix_size.uiWB != matrix_size.uiWC)
209         {
210             printf("ERROR: Matrix sizes do not match!\n");
211             exit(-1);
212         }
213     }
214
215     ////////////////////////////////////////////////////////////////////////////
216     //! Run a simple test matrix multiply using CUBLAS
217     ////////////////////////////////////////////////////////////////////////////
218     int matrixMultiply(int argc, char **argv, int devID, sMatrixSize &matrix_size)
219     {
220         cudaDeviceProp deviceProp;
```

```
221
222     checkCudaErrors(cudaGetDeviceProperties(&deviceProp, devID));
223
224     // use a larger block size for Fermi and above
225     int block_size = (deviceProp.major < 2) ? 16 : 32;
226
227     // set seed for rand()
228     srand(2006);
229
230     // allocate host memory for matrices A and B
231     unsigned int size_A = matrix_size.uiWA * matrix_size.uiHA;
232     unsigned int mem_size_A = sizeof(float) * size_A;
233     float *h_A = (float *)malloc(mem_size_A);
234     unsigned int size_B = matrix_size.uiWB * matrix_size.uiHB;
235     unsigned int mem_size_B = sizeof(float) * size_B;
236     float *h_B = (float *)malloc(mem_size_B);
237
238     // set seed for rand()
239     srand(2006);
240
241     // initialize host memory
242     randomInit(h_A, size_A);
243     randomInit(h_B, size_B);
244
245     // allocate device memory
246     float *d_A, *d_B, *d_C;
247     unsigned int size_C = matrix_size.uiWC * matrix_size.uiHC;
248     unsigned int mem_size_C = sizeof(float) * size_C;
249
250     // allocate host memory for the result
251     float *h_C      = (float *) malloc(mem_size_C);
```

```
252     float *h_CUBLAS = (float *) malloc(mem_size_C);
253
254     checkCudaErrors(cudaMalloc((void **) &d_A, mem_size_A));
255     checkCudaErrors(cudaMalloc((void **) &d_B, mem_size_B));
256     checkCudaErrors(cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));
257     checkCudaErrors(cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice));
258     checkCudaErrors(cudaMalloc((void **) &d_C, mem_size_C));
259
260     // setup execution parameters
261     dim3 threads(block_size, block_size);
262     dim3 grid(matrix_size.uiWC / threads.x, matrix_size.uiHC / threads.y);
263
264     // create and start timer
265     printf("Computing result using CUBLAS...");
266
267     // execute the kernel
268     int nIter = 30;
269
270     // CUBLAS version 2.0
271     {
272         const float alpha = 1.0f;
273         const float beta  = 0.0f;
274         cublasHandle_t handle;
275         cudaEvent_t start, stop;
276
277         checkCudaErrors(cublasCreate(&handle));
278
279         //Perform warmup operation with cublas
280         checkCudaErrors(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, matrix_size.uiWB, matrix_size.uiHA, matrix_size.uiWA,
281 &alpha, d_B, matrix_size.uiWB, d_A, matrix_size.uiWA, &beta, d_C, matrix_size.uiWB));
282
```

```
283         // Allocate CUDA events that we'll use for timing
284         checkCudaErrors(cudaEventCreate(&start));
285         checkCudaErrors(cudaEventCreate(&stop));
286 
287         // Record the start event
288         checkCudaErrors(cudaEventRecord(start, NULL));
289 
290         for (int j = 0; j < nIter; j++)
291         {
292            //note cublas is column primary!
293            //need to transpose the order
294            checkCudaErrors(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, matrix_size.uiWB, matrix_size.uiHA, matrix_size.uiWA,
295    &alpha, d_B, matrix_size.uiWB, d_A, matrix_size.uiWA, &beta, d_C, matrix_size.uiWB));
296 
297         }
298 
299         printf("done.\n");
300 
301         // Record the stop event
302         checkCudaErrors(cudaEventRecord(stop, NULL));
303 
304         // Wait for the stop event to complete
305         checkCudaErrors(cudaEventSynchronize(stop));
306 
307         float msecTotal = 0.0f;
308         checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));
309 
310         // Compute and print the performance
311         float msecPerMatrixMul = msecTotal / nIter;
312         double flopsPerMatrixMul = 2.0 * (double)matrix_size.uiHC * (double)matrix_size.uiWC * (double)matrix_size.uiHB;
313         double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
```

```
314         printf(
315             "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops\n",
316             gigaFlops,
317             msecPerMatrixMul,
318             flopsPerMatrixMul);
319 
320         // copy result from device to host
321         checkCudaErrors(cudaMemcpy(h_CUBLAS, d_C, mem_size_C, cudaMemcpyDeviceToHost));
322 
323         // Destroy the handle
324         checkCudaErrors(cublasDestroy(handle));
325     }
326 
327     // compute reference solution
328     printf("Computing result using host CPU...");
329     float *reference = (float *)malloc(mem_size_C);
330     matrixMulCPU(reference, h_A, h_B, matrix_size.uiHA, matrix_size.uiWA, matrix_size.uiWB);
331     printf("done.\n");
332 
333     // check result (CUBLAS)
334     bool resCUBLAS = sdkCompareL2fe(reference, h_CUBLAS, size_C, 1.0e-6f);
335 
336     if (resCUBLAS != true)
337     {
338         printDiff(reference, h_CUBLAS, matrix_size.uiWC, matrix_size.uiHC, 100, 1.0e-5f);
339     }
340 
341     printf("Comparing CUBLAS Matrix Multiply with CPU results: %s\n", (true == resCUBLAS) ? "PASS" : "FAIL");
342 
343     printf("\nNOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.\n");
344 
```

```
345         // clean up memory
346         free(h_A);
347         free(h_B);
348         free(h_C);
349         free(reference);
350         checkCudaErrors(cudaFree(d_A));
351         checkCudaErrors(cudaFree(d_B));
352         checkCudaErrors(cudaFree(d_C));
353
354         if (resCUBLAS == true)
355         {
356             return EXIT_SUCCESS;    // return value = 1
357         }
358         else
359         {
360             return EXIT_FAILURE;     // return value = 0
361         }
362     }
363
364     ////////////////////////////////////////////////////////////////////////////
365     // Program main
366     ////////////////////////////////////////////////////////////////////////////
367     int main(int argc, char **argv)
368     {
369         printf("[Matrix Multiply CUBLAS] - Starting...\n");
370
371         int devID = 0, sizeMult = 5;
372         sMatrixSize matrix_size;
373
374         initializeCUDA(argc, argv, devID, sizeMult, matrix_size);
375
```

```
376     int matrix_result = matrixMultiply(argc, argv, devID, matrix_size);
377
378     return matrix_result;
379 }
```

```
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "GeForce GTX 570" with compute capability 2.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 743.68 GFlop/s, Time= 0.264 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

Пример программ перемножения векторов с использованием библиотеки NVRTC (NVIDIA Run Time Compilation) из Cuda Samples 8.0

Программа перемножения двух векторов «matrixMul.cpp»

```
1  /**
2   * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
3   *
4   * Please refer to the NVIDIA end user license agreement (EULA) associated
5   * with this source code for terms and conditions that govern your use of
6   * this software. Any use, reproduction, disclosure, or distribution of
7   * this software and related documentation outside the terms of the EULA
8   * is strictly prohibited.
9   *
10  */
11
12
13 /**
14  * Matrix multiplication: C = A * B.
15  * Host code.
16  *
17  * This sample implements matrix multiplication as described in Chapter 3
```

```
 * of the programming guide.
 * It has been written for clarity of exposition to illustrate various CUDA
 * programming principles, not with the goal of providing the most
 * performant generic kernel for 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.
 */


// System includes
#include <stdio.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include "nvrtc_helper.h"

// Helper functions and utilities to work with CUDA
#include <helper_functions.h>


void constantInit(float *data, int size, float val)
{
    for (int i = 0; i < size; ++i)
    {
        data[i] = val;
    }
}


/**
 * Run a simple test of matrix multiplication using CUDA
 */
int matrixMultiply(int argc, char **argv, int block_size, dim3 &dimsA, dim3 &dimsB)
{
    // Allocate host memory for matrices A and B
    unsigned int size_A = dimsA.x * dimsA.y;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = dimsB.x * dimsB.y;
    unsigned int mem_size_B = sizeof(float) * size_B;
```

```
62      float *h_B = (float *)malloc(mem_size_B);
63
64      // Initialize host memory
65      const float valB = 0.01f;
66      constantInit(h_A, size_A, 1.0f);
67      constantInit(h_B, size_B, valB);
68
69      // Allocate device memory
70      CUdeviceptr d_A, d_B, d_C;
71
72      char *ptx, *kernel_file;
73      size_t ptxSize;
74
75      kernel_file = sdkFindFilePath("matrixMul_kernel.cu", argv[0]);
76      compileFileToPTX(kernel_file, 0, NULL, &ptx, &ptxSize);
77
78      CUmodule module = loadPTX(ptx, argc, argv);
79
80
81
82      // Allocate host matrix C
83      dim3 dimsC(dimsB.x, dimsA.y, 1);
84      unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
85      float *h_C = (float *) malloc(mem_size_C);
86
87      if (h_C == NULL)
88      {
89          fprintf(stderr, "Failed to allocate host matrix C!\n");
90          exit(EXIT_FAILURE);
91      }
92
93      checkCudaErrors(cuMemAlloc(&d_A, mem_size_A));
94      checkCudaErrors(cuMemAlloc(&d_B, mem_size_B));
95      checkCudaErrors(cuMemAlloc(&d_C, mem_size_C));
96
97      // copy host memory to device
98      checkCudaErrors(cuMemcpyHtoD(d_A, h_A, mem_size_A));
99      checkCudaErrors(cuMemcpyHtoD(d_B, h_B, mem_size_B));
100
101     // Setup execution parameters
102     dim3 threads(block_size, block_size);
103     dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
104
105     // Create and start timer
```

```
106       printf("Computing result using CUDA Kernel...\n");
107
108       CUfunction kernel_addr;
109       if (block_size == 16)
110       {
111         checkCudaErrors(cuModuleGetFunction(&kernel_addr, module, "matrixMulCUDA_block16"));
112       }
113       else
114       {
115         checkCudaErrors(cuModuleGetFunction(&kernel_addr, module, "matrixMulCUDA_block32"));
116       }
117
118       void *arr[] = { (void *)&d_C, (void *)&d_A, (void *)&d_B, (void *)&dimsA.x, (void *)&dimsB.x };
119
120       // Execute the kernel
121       int nIter = 300;
122
123       for (int j = 0; j < nIter; j++)
124       {
125           checkCudaErrors(cuLaunchKernel(kernel_addr,
126                                          grid.x, grid.y, grid.z, /* grid dim */
127                                          threads.x, threads.y, threads.z, /* block dim */
128                                          0,0, /* shared mem, stream */
129                                          &arr[0], /* arguments */
130                                          0));
131
132           checkCudaErrors(cuCtxSynchronize());
133       }
134
135       // Copy result from device to host
136       checkCudaErrors(cuMemcpyDtoH(h_C, d_C, mem_size_C));
137
138       printf("Checking computed result for correctness: ");
139
140       bool correct = true;
141
142       // test relative error by the formula
143       //     |<x, y>_cpu - <x,y>_gpu|/<|x|, |y|>  < eps
144
145       double eps = 1.e-6 ; // machine zero
146
147       for (int i = 0; i < (int)(dimsC.x * dimsC.y); i++)
148       {
149           double abs_err = fabs(h_C[i] - (dimsA.x * valB));
```

```
150             double dot_length = dimsA.x;
151             double abs_val = fabs(h_C[i]);
152             double rel_err = abs_err/abs_val/dot_length ;
153 
154             if (rel_err > eps)
155             {
156                 printf("Error! Matrix[%05d]=%.8f, ref=%.8f error term is > %E\n", i, h_C[i], dimsA.x*valB, eps);
157                 correct = false;
158             }
159         }
160 
161         printf("%s\n", correct ? "Result = PASS" : "Result = FAIL");
162 
163         printf("\nNOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.\n");
164 
165         // Clean up memory
166         free(h_A);
167         free(h_B);
168         free(h_C);
169 
170         checkCudaErrors(cuMemFree(d_A));
171         checkCudaErrors(cuMemFree(d_B));
172         checkCudaErrors(cuMemFree(d_C));
173 
174         if (correct)
175         {
176             return EXIT_SUCCESS;
177         }
178         else
179         {
180             return EXIT_FAILURE;
181         }
182     }
183 
184     /**
185      * Program main
186      */
187 
188     int main(int argc, char **argv)
189     {
190 
191         printf("[Matrix Multiply Using CUDA] - Starting...\n");
192 
193         if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
```

```
194         checkCmdLineFlag(argc, (const char **)argv, "?"))
195     {
196 
197         printf("Usage -device=n (n >= 0 for deviceID)\n");
198         printf("      -wA=WidthA -hA=HeightA (Width x Height of Matrix A)\n");
199         printf("      -wB=WidthB -hB=HeightB (Width x Height of Matrix B)\n");
200         printf("  Note: Outer matrix dimensions of A & B matrices must be equal.\n");
201 
202         exit(EXIT_SUCCESS);
203     }
204 
205     // Use a larger block size for Fermi and above
206     int block_size = 32;
207 
208     //original:
209     dim3 dimsA(5*2*block_size, 5*2*block_size, 1);
210     dim3 dimsB(5*4*block_size, 5*2*block_size, 1);
211 
212     // reduce sizes to avoid running out of memory
213     //dim3 dimsA(32,32, 1);
214     //dim3 dimsB(32,32,1);
215 
216     // width of Matrix A
217     if (checkCmdLineFlag(argc, (const char **)argv, "wA"))
218     {
219         dimsA.x = getCmdLineArgumentInt(argc, (const char **)argv, "wA");
220     }
221 
222     // height of Matrix A
223     if (checkCmdLineFlag(argc, (const char **)argv, "hA"))
224     {
225         dimsA.y = getCmdLineArgumentInt(argc, (const char **)argv, "hA");
226     }
227 
228     // width of Matrix B
229     if (checkCmdLineFlag(argc, (const char **)argv, "wB"))
230     {
231         dimsB.x = getCmdLineArgumentInt(argc, (const char **)argv, "wB");
232     }
233 
234     // height of Matrix B
235     if (checkCmdLineFlag(argc, (const char **)argv, "hB"))
236     {
237         dimsB.y = getCmdLineArgumentInt(argc, (const char **)argv, "hB");
```

```
238         }
239 
240         if (dimsA.x != dimsB.y)
241         {
242             printf("Error: outer matrix dimensions must be equal. (%d != %d)\n", dimsA.x, dimsB.y);
243             exit(EXIT_FAILURE);
244         }
245 
246         printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", dimsA.x, dimsA.y, dimsB.x, dimsB.y);
247 
248         int matrix_result = matrixMultiply(argc, argv, block_size, dimsA, dimsB);
249 
250         exit(matrix_result);
251     }
```

Программа ядра для перемножения двух векторов «matrixMul_kernel.cu»

```
 1   /**
 2    * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 3    *
 4    * Please refer to the NVIDIA end user license agreement (EULA) associated
 5    * with this source code for terms and conditions that govern your use of
 6    * this software. Any use, reproduction, disclosure, or distribution of
 7    * this software and related documentation outside the terms of the EULA
 8    * is strictly prohibited.
 9    *
10    */
11 
12   /**
13    * Matrix multiplication: C = A * B.
14    * Host code.
15    *
16    * This sample implements matrix multiplication as described in Chapter 3
17    * of the programming guide.
18    * It has been written for clarity of exposition to illustrate various CUDA
19    * programming principles, not with the goal of providing the most
20    * performant generic kernel for matrix multiplication.
21    *
22    * See also:
23    * V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra,"
24    * in Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC '08),
25    * Piscataway, NJ: IEEE Press, 2008, pp. Art. 31:1-11.
26    */
```

```
27
28  /**
29   * Matrix multiplication (CUDA Kernel) on the device: C = A * B
30   * wA is A's width and wB is B's width
31   */
32
33  template <int BLOCK_SIZE> __device__ void
34
35  matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
36  {
37      // Block index
38      int bx = blockIdx.x;
39      int by = blockIdx.y;
40
41      // Thread index
42      int tx = threadIdx.x;
43      int ty = threadIdx.y;
44
45      // Index of the first sub-matrix of A processed by the block
46      int aBegin = wA * BLOCK_SIZE * by;
47
48      // Index of the last sub-matrix of A processed by the block
49      int aEnd   = aBegin + wA - 1;
50
51      // Step size used to iterate through the sub-matrices of A
52      int aStep  = BLOCK_SIZE;
53
54      // Index of the first sub-matrix of B processed by the block
55      int bBegin = BLOCK_SIZE * bx;
56
57      // Step size used to iterate through the sub-matrices of B
58      int bStep  = BLOCK_SIZE * wB;
59
60      // Csub is used to store the element of the block sub-matrix
61      // that is computed by the thread
62      float Csub = 0;
63
64      // Loop over all the sub-matrices of A and B
65      // required to compute the block sub-matrix
66      for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
67      {
68          // Declaration of the shared memory array As used to
69          // store the sub-matrix of A
70          __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
```

```
71
72          // Declaration of the shared memory array Bs used to
73          // store the sub-matrix of B
74          __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
75
76          // Load the matrices from device memory
77          // to shared memory; each thread loads
78          // one element of each matrix
79          As[ty][tx] = A[a + wA * ty + tx];
80          Bs[ty][tx] = B[b + wB * ty + tx];
81
82          // Synchronize to make sure the matrices are loaded
83          __syncthreads();
84
85          // Multiply the two matrices together;
86          // each thread computes one element
87          // of the block sub-matrix
88  #pragma unroll
89          for (int k = 0; k < BLOCK_SIZE; ++k)
90          {
91              Csub += As[ty][k] * Bs[k][tx];
92          }
93
94          // Synchronize to make sure that the preceding
95          // computation is done before loading two new
96          // sub-matrices of A and B in the next iteration
97          __syncthreads();
98      }
99
100     // Write the block sub-matrix to device memory;
101     // each thread writes one element
102     int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
103     C[c + wB * ty + tx] = Csub;
104 }
105
106 extern "C" __global__ void  matrixMulCUDA_block16(float *C, float *A, float *B, int wA, int wB)
107 {
108     matrixMulCUDA<16>(C,A,B,wA,wB);
109 }
110
111 extern "C" __global__ void  matrixMulCUDA_block32(float *C, float *A, float *B, int wA, int wB)
112 {
113     matrixMulCUDA<32>(C,A,B,wA,wB);
114 }
```