```
1  // headers
2  #include <stdio.h>
3
4  #include <cuda.h> // for CUDA
5
6  // from NVIDIA CUDA SDK [ REMEBER : Some header file changes are done to the
     original file ]
7  #include "helper_timer.h"
8
9  #define BLOCK_WIDTH 4
10
11 // variable declarations
12 float *hostA=NULL;
13 float *hostB=NULL;
14 float *hostC=NULL;
15 float *CHost=NULL;
16
17 float *deviceA=NULL;
18 float *deviceB=NULL;
19 float *deviceC=NULL;
20
21 float timeOnCPU;
22 float timeOnGPU;
23
24 // global kernel function definition
25 __global__ void matrixMultiply(float *A,float *B,float *C,int numARows,int
     numAColumns,int numBRows,int numBColumns,int numCRows,int numCColumns)
26 {
27     // variable declarations
28     int row=blockIdx.y * blockDim.y + threadIdx.y;
29     int col=blockIdx.x * blockDim.x + threadIdx.x;
30     // code
31     if((row < numARows) && (col < numBColumns))
32     {
33         float Cvalue=0.0;
34         for(int k=0; k < numAColumns; k++)
35         {
36             Cvalue +=A[row * numAColumns + k] * B[k * numBColumns + col];
37         }
38         C[row * numCColumns + col]=Cvalue;
39     }
40 }
41
42 int main(int argc,char *argv[])
43 {
44     // function declarations
45     void fillFloatArrayWithRandomNumbers(float *, int);
46     void matMulHost(float *,float *,float *,int,int,int);
47     void cleanup(void);
48
49     // variable declarations
50     int numARows;
```

```
51      int numAColumns;
52      int numBRows;
53      int numBColumns;
54      int numCRows;
55      int numCColumns;
56      int numCHostRows;
57      int numCHostColumns;
58
59      // code
60      numARows=4;
61      numAColumns=4;
62      numBRows=4;
63      numBColumns=4;
64
65      numCRows=numARows;
66      numCColumns=numBColumns;
67
68      numCHostRows=numARows;
69      numCHostColumns=numBColumns;
70
71      int sizeA= numARows * numAColumns * sizeof(float);
72      int sizeB= numBRows * numBColumns * sizeof(float);
73      int sizeC= numCRows * numCColumns * sizeof(float);
74      int sizeCHost= numCHostRows * numCHostColumns * sizeof(float);
75
76      // allocate host-memory
77      hostA=(float *)malloc(sizeA);
78      if(hostA==NULL)
79      {
80          printf("CPU Memory Fatal Error = Can Not Allocate Enough Memory For Host
              Input Matrix A.\nExitting ...\n");
81          exit(EXIT_FAILURE);
82      }
83
84      hostB=(float *)malloc(sizeB);
85      if(hostB==NULL)
86      {
87          printf("CPU Memory Fatal Error = Can Not Allocate Enough Memory For Host
              Input Matrix B.\nExitting ...\n");
88          cleanup();
89          exit(EXIT_FAILURE);
90      }
91
92      hostC=(float *)malloc(sizeC);
93      if(hostC== NULL)
94      {
95          printf("CPU Memory Fatal Error = Can Not Allocate Enough Memory For Host
              Output Matrix C.\nExitting ...\n");
96          cleanup();
97          exit(EXIT_FAILURE);
98      }
99
```

```
100     CHost=(float *)malloc(sizeCHost);
101     if(hostC== NULL)
102     {
103         printf("CPU Memory Fatal Error = Can Not Allocate Enough Memory For Host
              Output Matrix C.\nExitting ...\n");
104         cleanup();
105         exit(EXIT_FAILURE);
106     }
107
108     // fill above input host vectors with arbitary but hard-coded data
109     fillFloatArrayWithRandomNumbers(hostA,numARows * numAColumns);
110     fillFloatArrayWithRandomNumbers(hostB,numBRows * numBColumns);
111
112     // allocate device-memory
113     cudaError_t err=cudaSuccess;
114     err=cudaMalloc((void **)&deviceA,sizeA);
115     if(err!=cudaSuccess)
116     {
117         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
118         cleanup();
119         exit(EXIT_FAILURE);
120     }
121
122     err=cudaMalloc((void **)&deviceB,sizeB);
123     if(err!=cudaSuccess)
124     {
125         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
126         cleanup();
127         exit(EXIT_FAILURE);
128     }
129
130     err=cudaMalloc((void **)&deviceC,sizeC);
131     if(err!=cudaSuccess)
132     {
133         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
134         cleanup();
135         exit(EXIT_FAILURE);
136     }
137
138     // copy host memory contents to device memory
139     err=cudaMemcpy(deviceA,hostA,sizeA,cudaMemcpyHostToDevice);
140     if(err!=cudaSuccess)
141     {
142         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
143         cleanup();
144         exit(EXIT_FAILURE);
145     }
146
```

```
147     err=cudaMemcpy(deviceB,hostB,sizeB,cudaMemcpyHostToDevice);
148     if(err!=cudaSuccess)
149     {
150         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
151         cleanup();
152         exit(EXIT_FAILURE);
153     }
154
155     // cuda kernel configuration
156     dim3 DimGrid=dim3(ceil((int)numCColumns/(int)BLOCK_WIDTH),ceil((int)numCRows/
          (int)BLOCK_WIDTH),1);
157     dim3 DimBlock=dim3(BLOCK_WIDTH,BLOCK_WIDTH,1);
158
159     // start timer
160     StopWatchInterface *timer = NULL;
161     sdkCreateTimer(&timer);
162     sdkStartTimer(&timer);
163
164     matrixMultiply<<<DimGrid,DimBlock>>>
          (deviceA,deviceB,deviceC,numARows,numAColumns,numBRows,numBColumns,numCRows
          ,numCColumns);
165
166     // stop timer
167     sdkStopTimer(&timer);
168     timeOnGPU = sdkGetTimerValue(&timer);
169     sdkDeleteTimer(&timer);
170
171     // copy device memory to host memory
172     err=cudaMemcpy(hostC,deviceC,sizeC,cudaMemcpyDeviceToHost);
173     if(err!=cudaSuccess)
174     {
175         printf("GPU Memory Fatal Error = %s In File Name %s At Line No. %d.
              \nExitting ...\n",cudaGetErrorString(err),__FILE__,__LINE__);
176         cleanup();
177         exit(EXIT_FAILURE);
178     }
179
180     // results
181     matMulHost(hostA,hostB,CHost,numAColumns,numCHostRows,numCHostColumns);
182
183     // compare results for golden-host
184     const float epsilon = 0.000001f;
185     bool bAccuracy=true;
186     int breakValue=0;
187     int i;
188     for(i=0;i<numARows * numAColumns;i++)
189     {
190         float val1 = CHost[i];
191         float val2 = hostC[i];
192         if(fabs(val1-val2) > epsilon)
193         {
```

```
194             bAccuracy = false;
195             breakValue=i;
196             break;
197         }
198     }
199
200     if(bAccuracy==false)
201     {
202         printf("Break Value = %d\n",breakValue);
203     }
204
205     char str[125];
206     if(bAccuracy==true)
207         sprintf(str,"%s","Comparison Of Output Arrays On CPU And GPU Are Accurate
            Within The Limit Of 0.000001");
208     else
209         sprintf(str,"%s","Not All Comparison Of Output Arrays On CPU And GPU Are
          Accurate Within The Limit Of 0.000001");
210
211     printf("1st Matrix Is From 0th Element %.6f To %dth Element %.6f\n",hostA[0],
        (numARows * numAColumns)-1, hostA[(numARows * numAColumns)-1]);
212     printf("2nd Matrix Is From 0th Element %.6f To %dth Element %.6f\n",hostB[0],
        (numBRows * numBColumns)-1, hostB[(numBRows * numBColumns)-1]);
213     printf("Grid Dimension = (%d,1,1) And Block Dimension = (%d,1,1)
        \n",DimGrid.x,DimBlock.x);
214     printf("Multiplication Of Above 2 Matrices Creates 3rd Matrix As :\n");
215     printf("3nd Matrix Is From 0th Element %.6f To %dth Element %.6f\n",hostC[0],
        (numCRows * numCColumns)-1, hostC[(numCRows * numCColumns)-1]);
216     printf("The Time Taken To Do Above Addition On CPU = %.6f (ms)\n",timeOnCPU);
217     printf("The Time Taken To Do Above Addition On GPU = %.6f (ms)\n",timeOnGPU);
218     printf("%s\n",str);
219
220     // total cleanup
221     cleanup();
222
223     return(0);
224 }
225
226 void cleanup(void)
227 {
228     // code
229
230     // free allocated device-memory
231     if(deviceA)
232     {
233         cudaFree(deviceA);
234         deviceA=NULL;
235     }
236
237     if(deviceB)
238     {
239         cudaFree(deviceB);
```

```
240         deviceB=NULL;
241     }
242
243     if(deviceC)
244     {
245         cudaFree(deviceC);
246         deviceC=NULL;
247     }
248
249     // free allocated host-memory
250     if(hostA)
251     {
252         free(hostA);
253         hostA=NULL;
254     }
255
256     if(hostB)
257     {
258         free(hostB);
259         hostB=NULL;
260     }
261
262     if(hostC)
263     {
264         free(hostC);
265         hostC=NULL;
266     }
267
268     if(CHost)
269     {
270         free(CHost);
271         CHost=NULL;
272     }
273 }
274
275 void fillFloatArrayWithRandomNumbers(float *pFloatArray, int iSize)
276 {
277     // code
278     int i;
279     const float fScale = 1.0f / (float)RAND_MAX;
280     for (i = 0; i < iSize; ++i)
281     {
282         pFloatArray[i] = fScale * rand();
283     }
284 }
285
286 void matMulHost(float *A,float *B,float* C,int iAColumns,int iCRows,int
      iCColumns)
287 {
288     // code
289     // start timer
290     StopWatchInterface *timer = NULL;
```

```
291     sdkCreateTimer(&timer);
292     sdkStartTimer(&timer);
293 
294     for(int i=0;i<iCRows;++i)
295     {
296         for(int j=0;j<iCColumns;++j)
297         {
298             float sum=0.0f;
299             for(int k=0;k<iAColumns;++k)
300             {
301                 float a=A[i * iAColumns + k];
302                 float b=B[k * iCColumns + j];
303                 sum += a * b;
304             }
305             C[i * iCColumns + j] = sum;
306         }
307     }
308 
309     // stop timer
310     sdkStopTimer(&timer);
311     timeOnCPU = sdkGetTimerValue(&timer);
312     sdkDeleteTimer(&timer);
313 }
314 
```