```
1  // header files
2  // standard headers
3  #include <stdio.h>
4  #include <math.h> // fabs()
5
6  // cuda headers
7  // OpenCL headers
8  #include <CL/opencl.h>
9
10 #include "helper_timer.h"
11
12 // macros
13 #define BLOCK_WIDTH 64
14
15 // global variables
16 cl_platform_id oclPlatformID;
17 cl_device_id oclDeviceID;
18
19 cl_context oclContext;
20 cl_command_queue oclCommandQueue;
21
22 cl_program oclProgram;
23 cl_kernel oclKernel;
24
25 int *hostA=NULL;
26 int *hostB=NULL;
27 int *hostC=NULL;
28 int *gold=NULL;
29
30 cl_mem deviceA=NULL;
31 cl_mem deviceB=NULL;
32 cl_mem deviceC=NULL;
33
34 float timeOnCPU = 0.0f;
35 float timeOnGPU = 0.0f;
36
37 // OpenCL kernel
38 const char *oclSourceCode =
39 "__kernel void matMulGPU(__global int *A, __global int *B, __global int *C,int
     numARows,int numAColumns,int numBColumns,int numCColumns)" \
40 "{" \
41 "int row=get_global_id(0);" \
42 "int column=get_global_id(1);" \
43 "if((row < numARows) && (column < numBColumns))" \
44 "{" \
45 "int value=0;" \
46 "for(int k=0; k < numAColumns; k++)"
47 "{" \
48 "int a=A[row * numAColumns + k];" \
49 "int b=B[k * numBColumns + column];" \
50 "value += a*b;" \
51 "}" \
52 "C[row * numCColumns + column]=value;" \
53 "}" \
54 "}";
55
```

```
56  int main(int argc,char *argv[])
57  {
58      // function declarations
59      void InitA(int *data,int,int);
60      void InitB(int *data,int,int);
61      void matMulCPU(int*, int*, int*, int, int, int, int);
62      void cleanup(void);
63
64      // variable declarations
65      int numARows=BLOCK_WIDTH;
66      int numAColumns=BLOCK_WIDTH;
67      int numBRows=BLOCK_WIDTH;
68      int numBColumns=BLOCK_WIDTH;
69
70      int numCRows=numARows;
71      int numCColumns=numBColumns;
72
73      int numGoldRows=numARows;
74      int numGoldColumns=numBColumns;
75
76      int sizeA = numARows * numAColumns * sizeof(int);
77      int sizeB = numBRows * numBColumns * sizeof(int);
78      int sizeC = numCRows * numCColumns * sizeof(int);
79      int sizeGold = numGoldRows * numGoldColumns * sizeof(int);
80
81      cl_int result;
82
83      // code
84      // host memory allocation
85      hostA=(int *)malloc(sizeA);
86      if(hostA==NULL)
87      {
88          printf("Host Memory allocation is failed for hostA matrix.\n");
89          cleanup();
90          exit(EXIT_FAILURE);
91      }
92
93      hostB=(int *)malloc(sizeB);
94      if(hostB==NULL)
95      {
96          printf("Host Memory allocation is failed for hostB matrix.\n");
97          cleanup();
98          exit(EXIT_FAILURE);
99      }
100
101     hostC=(int *)malloc(sizeC);
102     if(hostC== NULL)
103     {
104         printf("Host Memory allocation is failed for hostC matrix.\n");
105         cleanup();
106         exit(EXIT_FAILURE);
107     }
108
109     gold=(int *)malloc(sizeGold);
110     if(gold== NULL)
111     {
```

```
112         printf("Host Memory allocation is failed for gold matrix.\n");
113         cleanup();
114         exit(EXIT_FAILURE);
115     }
116
117     // printing matrix dimensions and sizes
118     printf("The Dimensions Of Matrix 'hostA' Are : %d x %d
          \n",numARows,numAColumns);
119     printf("The Dimensions Of Matrix 'hostB' Are : %d x %d
          \n",numBRows,numBColumns);
120     printf("The Dimensions Of Matrix 'hostC' Are : %d x %d
          \n",numCRows,numCColumns);
121
122     printf("The Dimensions Of Matrix 'gold' Are : %d x %d
          \n",numGoldRows,numGoldColumns);
123
124     printf("Size Of Matrix hostA = %d\n",sizeA);
125     printf("Size Of Matrix hostB = %d\n",sizeB);
126     printf("Size Of Matrix hostC = %d\n",sizeC);
127
128     printf("Size Of Matrix gold = %d\n",sizeGold);
129
130     // fill source matrices
131     InitA(hostA,numARows,numAColumns);
132     InitB(hostB,numBRows,numBColumns);
133
134     // get OpenCL supporting platform's ID
135     result = clGetPlatformIDs(1, &oclPlatformID, NULL);
136     if (result != CL_SUCCESS)
137     {
138         printf("clGetPlatformIDs() Failed : %d\n", result);
139         cleanup();
140         exit(EXIT_FAILURE);
141     }
142
143     // get OpenCL supporting CPU device's ID
144     result = clGetDeviceIDs(oclPlatformID, CL_DEVICE_TYPE_GPU, 1,
          &oclDeviceID, NULL);
145     if (result != CL_SUCCESS)
146     {
147         printf("clGetDeviceIDs() Failed : %d\n", result);
148         cleanup();
149         exit(EXIT_FAILURE);
150     }
151
152     // create OpenCL compute context
153     oclContext = clCreateContext(NULL, 1, &oclDeviceID, NULL, NULL, &result);
154     if (result != CL_SUCCESS)
155     {
156         printf("clCreateContext() Failed : %d\n", result);
157         cleanup();
158         exit(EXIT_FAILURE);
159     }
160
161     // create command queue
162     oclCommandQueue = clCreateCommandQueue(oclContext, oclDeviceID, 0,
```

```
        &result);
163     if (result != CL_SUCCESS)
164     {
165         printf("clCreateCommandQueue() Failed : %d\n", result);
166         cleanup();
167         exit(EXIT_FAILURE);
168     }
169
170     // create OpenCL program from .cl
171     oclProgram = clCreateProgramWithSource(oclContext, 1, (const char **)
          &oclSourceCode, NULL, &result);
172     if (result != CL_SUCCESS)
173     {
174         printf("clCreateProgramWithSource() Failed : %d\n", result);
175         cleanup();
176         exit(EXIT_FAILURE);
177     }
178
179     // build OpenCL program
180     result = clBuildProgram(oclProgram, 0, NULL, NULL, NULL, NULL);
181     if (result != CL_SUCCESS)
182     {
183         size_t len;
184         char buffer[2048];
185         clGetProgramBuildInfo(oclProgram, oclDeviceID, CL_PROGRAM_BUILD_LOG,
            sizeof(buffer), buffer, &len);
186         printf("Program Build Log : %s\n", buffer);
187         printf("clBuildProgram() Failed : %d\n", result);
188         cleanup();
189         exit(EXIT_FAILURE);
190     }
191
192     // create OpenCL kernel by passing kernel function name that we used
          in .cl file
193     oclKernel = clCreateKernel(oclProgram, "matMulGPU", &result);
194     if (result != CL_SUCCESS)
195     {
196         printf("clCreateKernel() Failed : %d\n", result);
197         cleanup();
198         exit(EXIT_FAILURE);
199     }
200
201     // device memory allocation
202     deviceA=clCreateBuffer(oclContext,CL_MEM_READ_ONLY,sizeA,NULL,&result);
203     if(result!=CL_SUCCESS)
204     {
205         printf("clCreateBuffer() Failed For 1st Input Matrix : %d\n",result);
206         cleanup();
207         exit(EXIT_FAILURE);
208     }
209
210     deviceB=clCreateBuffer(oclContext,CL_MEM_READ_ONLY,sizeB,NULL,&result);
211     if(result!=CL_SUCCESS)
212     {
213         printf("clCreateBuffer() Failed For 2nd Input Matrix : %d\n",result);
214         cleanup();
```

```
215         exit(EXIT_FAILURE);
216     }
217
218     deviceC=clCreateBuffer(oclContext,CL_MEM_WRITE_ONLY,sizeC,NULL,&result);
219     if(result!=CL_SUCCESS)
220     {
221         printf("clCreateBuffer() Failed For Output Matrix : %d\n",result);
222         cleanup();
223         exit(EXIT_FAILURE);
224     }
225
226     // set 0 based 0th argument i.e. deviceA
227     result=clSetKernelArg(oclKernel,0,sizeof(cl_mem),(void *)&deviceA);
228     if(result != CL_SUCCESS)
229     {
230         printf("clSetKernelArg() Failed For 1st Argument : %d\n",result);
231         cleanup();
232         exit(EXIT_FAILURE);
233     }
234
235     // set 0 based 1st argument i.e. deviceB
236     result=clSetKernelArg(oclKernel,1,sizeof(cl_mem),(void *)&deviceB);
237     if(result != CL_SUCCESS)
238     {
239         printf("clSetKernelArg() Failed For 2nd Argument : %d\n",result);
240         cleanup();
241         exit(EXIT_FAILURE);
242     }
243
244     // set 0 based 2nd argument i.e. deviceC
245     result=clSetKernelArg(oclKernel,2,sizeof(cl_mem),(void *)&deviceC);
246     if(result != CL_SUCCESS)
247     {
248         printf("clSetKernelArg() Failed For 3rd Argument : %d\n",result);
249         cleanup();
250         exit(EXIT_FAILURE);
251     }
252
253     // set 0 based 3rd argument i.e. numARows
254     result=clSetKernelArg(oclKernel,3,sizeof(cl_int),(void *)&numARows);
255     if(result != CL_SUCCESS)
256     {
257         printf("clSetKernelArg() Failed For 4th Argument : %d\n",result);
258         cleanup();
259         exit(EXIT_FAILURE);
260     }
261
262     // set 0 based 4th argument i.e. numAColumns
263     result=clSetKernelArg(oclKernel,4,sizeof(cl_int),(void *)&numAColumns);
264     if(result != CL_SUCCESS)
265     {
266         printf("clSetKernelArg() Failed For 5th Argument : %d\n",result);
267         cleanup();
268         exit(EXIT_FAILURE);
269     }
270
```

```
271     // set 0 based 5th argument i.e. numBColumns
272     result=clSetKernelArg(oclKernel,5,sizeof(cl_int),(void *)&numBColumns);
273     if(result != CL_SUCCESS)
274     {
275         printf("clSetKernelArg() Failed For 6th Argument : %d\n",result);
276         cleanup();
277         exit(EXIT_FAILURE);
278     }
279
280     // set 0 based 6th argument i.e. numCColumns
281     result=clSetKernelArg(oclKernel,6,sizeof(cl_int),(void *)&numCColumns);
282     if(result != CL_SUCCESS)
283     {
284         printf("clSetKernelArg() Failed For 7th Argument : %d\n",result);
285         cleanup();
286         exit(EXIT_FAILURE);
287     }
288
289     // write abve 'input' device buffer to device memory
290     result=clEnqueueWriteBuffer
          (oclCommandQueue,deviceA,CL_FALSE,0,sizeA,hostA,0,NULL,NULL);
291     if(result != CL_SUCCESS)
292     {
293         printf("clEnqueueWriteBuffer() Failed For 1st Input Device Buffer : %d
            \n",result);
294         cleanup();
295         exit(EXIT_FAILURE);
296     }
297
298     result=clEnqueueWriteBuffer
          (oclCommandQueue,deviceB,CL_FALSE,0,sizeB,hostB,0,NULL,NULL);
299     if(result != CL_SUCCESS)
300     {
301         printf("clEnqueueWriteBuffer() Failed For 2nd Input Device Buffer : %d
            \n",result);
302         cleanup();
303         exit(EXIT_FAILURE);
304     }
305
306     // kernel configuration
307     size_t globalWorkSize[2];
308     globalWorkSize[0] = BLOCK_WIDTH;
309     globalWorkSize[1] = BLOCK_WIDTH;
310
311     // start timer
312     StopWatchInterface *timer = NULL;
313     sdkCreateTimer(&timer);
314     sdkStartTimer(&timer);
315
316     result=clEnqueueNDRangeKernel
          (oclCommandQueue,oclKernel,2,NULL,globalWorkSize,NULL,0,NULL,NULL);
317     if(result != CL_SUCCESS)
318     {
319         printf("clEnqueueNDRangeKernel() Failed : %d\n",result);
320         cleanup();
321         exit(EXIT_FAILURE);
```

```
322     }
323
324     // finish OpenCL command queue
325     clFinish(oclCommandQueue);
326
327     // stop timer
328     sdkStopTimer(&timer);
329     timeOnGPU = sdkGetTimerValue(&timer);
330     sdkDeleteTimer(&timer);
331
332     // read back result from the device (i.e from deviceOutput) into cpu
          variable (i.e hostOutput)
333     result=clEnqueueReadBuffer
          (oclCommandQueue,deviceC,CL_TRUE,0,sizeC,hostC,0,NULL,NULL);
334     if(result != CL_SUCCESS)
335     {
336         printf("clEnqueueReadBuffer() Failed : %d\n",result);
337         cleanup();
338         exit(EXIT_FAILURE);
339     }
340
341     // matrix multiplication on host
342     matMulCPU(hostA, hostB, gold, numARows, numAColumns, numBColumns,
          numCColumns);
343
344     // comparison
345     int breakValue = -1;
346     bool bAccuracy = true;
347     for (int i = 0; i < numCRows * numCColumns; i++)
348     {
349         int val1 = gold[i];
350         int val2 = hostC[i];
351         if (val1 != val2)
352         {
353             bAccuracy = false;
354             breakValue = i;
355             break;
356         }
357     }
358
359     char str[128];
360     if (bAccuracy == false)
361         sprintf(str, "Comparison of CPU and GPU Matrix Multiplication is not
              accurate at array index %d", breakValue);
362     else
363         sprintf(str, "Comparison of CPU and GPU Matrix Multiplication is
              accurate");
364
365     printf("Time taken for Matrix Multiplication on CPU = %.6f\n", timeOnCPU);
366     printf("Time taken for Matrix Multiplication on GPU = %.6f\n", timeOnGPU);
367     printf("%s\n", str);
368
369     // cleanup
370     cleanup();
371
372     return(0);
```

```
373 }
374
375 void InitA(int *data,int row,int col)
376 {
377     int num=1;
378     // code
379     for(int i=0;i<row;i++)
380     {
381         for(int j=0;j<col;j++)
382         {
383             *(data + i * col + j)=num;
384             num++;
385         }
386     }
387 }
388
389 void InitB(int *data,int row,int col)
390 {
391     int num=BLOCK_WIDTH;
392     // code
393     for(int i=0;i<row;i++)
394     {
395         for(int j=0;j<col;j++)
396         {
397             *(data + i * col + j)=num;
398             num--;
399         }
400     }
401 }
402
403 void matMulCPU(int* A, int* B, int* C, int numARows, int numAColumns, int
      numBColumns, int numCColumns)
404 {
405     // code
406     StopWatchInterface* timer = NULL;
407     sdkCreateTimer(&timer);
408     sdkStartTimer(&timer);
409
410     for (int i = 0; i < numARows; ++i)
411     {
412         for (int j = 0; j < numBColumns; ++j)
413         {
414             int value = 0.0f;
415             for (int k = 0; k < numAColumns; ++k)
416             {
417                 int a = A[i * numAColumns + k];
418                 int b = B[k * numBColumns + j];
419                 value += a * b;
420             }
421             C[i * numCColumns + j] = value;
422         }
423     }
424
425     sdkStopTimer(&timer);
426     timeOnCPU = sdkGetTimerValue(&timer);
427     sdkDeleteTimer(&timer);
```

```
428     timer = NULL;
429 }
430
431 void cleanup(void)
432 {
433     // code
434     if(deviceC)
435     {
436         clReleaseMemObject(deviceC);
437         deviceC=NULL;
438     }
439
440     if(deviceB)
441     {
442         clReleaseMemObject(deviceB);
443         deviceB=NULL;
444     }
445
446     if(deviceA)
447     {
448         clReleaseMemObject(deviceA);
449         deviceA=NULL;
450     }
451
452     if(oclKernel)
453     {
454         clReleaseKernel(oclKernel);
455         oclKernel=NULL;
456     }
457
458     if(oclProgram)
459     {
460         clReleaseProgram(oclProgram);
461         oclProgram=NULL;
462     }
463
464     if(oclCommandQueue)
465     {
466         clReleaseCommandQueue(oclCommandQueue);
467         oclCommandQueue=NULL;
468     }
469
470     if(oclContext)
471     {
472         clReleaseContext(oclContext);
473         oclContext=NULL;
474     }
475
476     if(gold)
477     {
478         free(gold);
479         gold=NULL;
480     }
481
482     if(hostC)
483     {
```

```
484         free(hostC);
485         hostC=NULL;
486     }
487 
488     if(hostB)
489     {
490         free(hostB);
491         hostB=NULL;
492     }
493 
494     if(hostA)
495     {
496         free(hostA);
497         hostA=NULL;
498     }
499 }
500 
```