```
1
2  // ================================================================================================
3  // Project:
4  // Exploring the performance of general matrix-multiplication on an NVIDIA Tesla K40m GPU.
5  //
6  // File information:
7  // Institution.... SURFsara <www.surfsara.nl>
8  // Author......... Cedric Nugteren <cedric.nugteren@surfsara.nl>
9  // Changed at..... 2014-11-17
10 // License........ MIT license
11 // Tab-size....... 4 spaces
12 // Line length.... 100 characters
13 //
14 // ================================================================================================
15
16 // Common include
17 #include "common.h"
18
19 // Include OpenCL
20 #include <CL/cl.h>
21
22 // Include kernel constants
23 #include "settings.h"
24
25 // Forward declaration of the OpenCL error checking function
26 void checkError(cl_int error, int line);
27
28 // ================================================================================================
29
30 // Set the locations of the OpenCL kernel files
31 #define CL_INCLUDE_FILE "src/settings.h"
32 #define CL_KERNEL_FILE "src/kernels.cl"
33
34 // Determine the location where to output the PTX code
35 #define CL_PTX_FILE "bin/myGEMM.cl.ptx"
36
37 // Define OpenCL compiler options, such as "-cl-nv-maxrregcount=127"
38 #define COMPILER_OPTIONS ""
39
40 // ================================================================================================
41
42 // Matrix-multiplication using a custom OpenCL SGEMM kernel. This function also copies the input
43 // matrices to the GPU, runs SGEMM, and copies the output matrix back to the CPU.
44 void myclblas(float* A, float* B, float* C,
45               int K, int M, int N,
46               int timerID) {
47
48     // In case of myGEMM10, compute matrix sizes K, M, N as rounded-up to form complete tiles
49     #if KERNEL == 10
50         int K_XL = CEIL_DIV(K, TSK) * TSK;
51         int M_XL = CEIL_DIV(M, TSM) * TSM;
52         int N_XL = CEIL_DIV(N, TSN) * TSN;
53     #else
54         int K_XL = K;
55         int M_XL = M;
56         int N_XL = N;
57     #endif
58
```

```
59      // Define OpenCL variables
60      cl_int err;
61      cl_platform_id platform = 0;
62      cl_device_id device = 0;
63      cl_device_id devices[MAX_NUM_DEVICES];
64      cl_uint numDevices = 0;
65      cl_context_properties props[3] = {CL_CONTEXT_PLATFORM, 0, 0};
66      cl_context context = 0;
67      cl_command_queue queue = 0;
68      cl_event event = NULL;
69      cl_program program = NULL;
70      char deviceName[MAX_DEVICE_NAME];
71
72      // Configure the OpenCL environment
73      err = clGetPlatformIDs(1, &platform, NULL);
74      err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
75      err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, numDevices, devices, NULL);
76      device = devices[CURRENT_DEVICE];
77      props[1] = (cl_context_properties)platform;
78      context = clCreateContext(props, 1, &device, NULL, NULL, &err);
79      queue = clCreateCommandQueue(context, device, 0, &err);
80      err = clGetDeviceInfo(device, CL_DEVICE_NAME, MAX_DEVICE_NAME, deviceName, NULL);
81      checkError(err,__LINE__);
82      //printf("## %d devices, running on %d: '%s'\n", numDevices, CURRENT_DEVICE, deviceName);
83
84      // Read the kernel file from disk
85      long sizeHeader, sizeSource;
86      char* header = readKernelFile(CL_INCLUDE_FILE, &sizeHeader);
87      char* source = readKernelFile(CL_KERNEL_FILE, &sizeSource);
88      long size = 2 + sizeHeader + sizeSource;
89      char* code = (char*)malloc(size*sizeof(char));
90      for (int c=0; c<size; c++) { code[c] = '\0'; }
91      strcat(code, header);
92      strcat(code, source);
93      const char* constCode = code;
94      free(header);
95      free(source);
96
97      // Compile the kernel file
98      program = clCreateProgramWithSource(context, 1, &constCode, NULL, &err);
99      checkError(err,__LINE__);
100     err = clBuildProgram(program, 0, NULL, COMPILER_OPTIONS, NULL, NULL);
101
102     // Check for compilation errors
103     size_t logSize;
104     err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &logSize);
105     checkError(err,__LINE__);
106     char* messages = (char*)malloc((1+logSize)*sizeof(char));
107     err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, logSize, messages, NULL);
108     checkError(err,__LINE__);
109     messages[logSize] = '\0';
110     //if (logSize > 10) { printf("## Compiler message: %s\n", messages); }
111     free(messages);
112
113     // Retrieve the PTX code from the OpenCL compiler and output it to disk
114     size_t binSize;
115     err = clGetProgramInfo(program, CL_PROGRAM_BINARY_SIZES, sizeof(size_t), &binSize, NULL);
116     checkError(err,__LINE__);
117     unsigned char *bin = (unsigned char *)malloc(binSize);
118     err = clGetProgramInfo(program, CL_PROGRAM_BINARIES, sizeof(unsigned char *), &bin, NULL);
119     checkError(err,__LINE__);
120     FILE* file = fopen(CL_PTX_FILE, "wb");
121     fwrite(bin, sizeof(char), binSize, file);
122     fclose(file);
123     free(bin);
124
125     // Prepare OpenCL memory objects
126     cl_mem bufA    = clCreateBuffer(context, CL_MEM_READ_ONLY,  M*K*sizeof(*A), NULL, &err);
127     cl_mem bufB    = clCreateBuffer(context, CL_MEM_READ_ONLY,  K*N*sizeof(*B), NULL, &err);
128     cl_mem bufB_TR = clCreateBuffer(context, CL_MEM_READ_ONLY,  N*K*sizeof(*B), NULL, &err);
129     cl_mem bufC    = clCreateBuffer(context, CL_MEM_READ_WRITE, M*N*sizeof(*C), NULL, &err);
130     checkError(err,__LINE__);
131
132     // Copy matrices to the GPU (also C to erase the results of the previous run)
133     err = clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0, M*K*sizeof(*A), A, 0, NULL, NULL);
```

```
134     err = clEnqueueWriteBuffer(queue, bufB, CL_TRUE, 0, K*N*sizeof(*B), B, 0, NULL, NULL);
135     err = clEnqueueWriteBuffer(queue, bufC, CL_TRUE, 0, M*N*sizeof(*C), C, 0, NULL, NULL);
136     checkError(err,__LINE__);
137
138     // Create extra objects for rounded-up sizes (only needed in case of myGEMM10)
139     cl_mem bufA_XL    = clCreateBuffer(context, CL_MEM_READ_ONLY,  M_XL*K_XL*sizeof(*A), NULL, &err);
140     cl_mem bufB_TR_XL = clCreateBuffer(context, CL_MEM_READ_ONLY,  N_XL*K_XL*sizeof(*B), NULL, &err);
141     cl_mem bufC_XL    = clCreateBuffer(context, CL_MEM_READ_WRITE, M_XL*N_XL*sizeof(*C), NULL, &err);
142     checkError(err,__LINE__);
143
144     // Configure the myGEMM kernel
145     char kernelname[100];
146     sprintf(kernelname, "myGEMM%d", KERNEL);
147     cl_kernel kernel1 = clCreateKernel(program, kernelname, &err);
148     checkError(err,__LINE__);
149
150     // Set the arguments of the myGEMM kernel
151     #if KERNEL == 10
152         err = clSetKernelArg(kernel1, 0, sizeof(int), (void*)&M_XL);
153         err = clSetKernelArg(kernel1, 1, sizeof(int), (void*)&N_XL);
154         err = clSetKernelArg(kernel1, 2, sizeof(int), (void*)&K_XL);
155         err = clSetKernelArg(kernel1, 3, sizeof(cl_mem), (void*)&bufA_XL);
156         err = clSetKernelArg(kernel1, 4, sizeof(cl_mem), (void*)&bufB_TR_XL);
157         err = clSetKernelArg(kernel1, 5, sizeof(cl_mem), (void*)&bufC_XL);
158     #else
159         err = clSetKernelArg(kernel1, 0, sizeof(int), (void*)&M);
160         err = clSetKernelArg(kernel1, 1, sizeof(int), (void*)&N);
161         err = clSetKernelArg(kernel1, 2, sizeof(int), (void*)&K);
162         err = clSetKernelArg(kernel1, 3, sizeof(cl_mem), (void*)&bufA);
163         #if KERNEL == 5 || KERNEL == 6 || KERNEL == 7 || KERNEL == 8 || KERNEL == 9
164             err = clSetKernelArg(kernel1, 4, sizeof(cl_mem), (void*)&bufB_TR);
165         #else
166             err = clSetKernelArg(kernel1, 4, sizeof(cl_mem), (void*)&bufB);
167         #endif
168         err = clSetKernelArg(kernel1, 5, sizeof(cl_mem), (void*)&bufC);
169     #endif
170     checkError(err,__LINE__);
171
172     // Configure the supporting transpose kernel and set its arguments (only for certain myGEMMs)
173     #if KERNEL == 5 || KERNEL == 6 || KERNEL == 7 || KERNEL == 8 || KERNEL == 9 || KERNEL == 10
174         cl_kernel kernel2 = clCreateKernel(program, "transpose", &err);
175         checkError(err,__LINE__);
176         err = clSetKernelArg(kernel2, 0, sizeof(int), (void*)&K);
177         err = clSetKernelArg(kernel2, 1, sizeof(int), (void*)&N);
178         err = clSetKernelArg(kernel2, 2, sizeof(cl_mem), (void*)&bufB);
179         err = clSetKernelArg(kernel2, 3, sizeof(cl_mem), (void*)&bufB_TR);
180         checkError(err,__LINE__);
181         const size_t tLocal[2] = { TRANSPOSEX, TRANSPOSEY };
182         const size_t tGlobal[2] = { (size_t)K, (size_t)N };
183     #endif
184
185     // Configure the supporting padding kernels and set their arguments (only for myGEMM10)
186     #if KERNEL == 10
187         cl_kernel kernel3a = clCreateKernel(program, "paddingAddZeroes", &err);
188         checkError(err,__LINE__);
189         err = clSetKernelArg(kernel3a, 0, sizeof(int), (void*)&M);
190         err = clSetKernelArg(kernel3a, 1, sizeof(int), (void*)&K);
191         err = clSetKernelArg(kernel3a, 2, sizeof(cl_mem), (void*)&bufA);
192         err = clSetKernelArg(kernel3a, 3, sizeof(int), (void*)&M_XL);
193         err = clSetKernelArg(kernel3a, 4, sizeof(int), (void*)&K_XL);
194         err = clSetKernelArg(kernel3a, 5, sizeof(cl_mem), (void*)&bufA_XL);
195         checkError(err,__LINE__);
196         cl_kernel kernel3b = clCreateKernel(program, "paddingAddZeroes", &err);
197         checkError(err,__LINE__);
198         err = clSetKernelArg(kernel3b, 0, sizeof(int), (void*)&N);
199         err = clSetKernelArg(kernel3b, 1, sizeof(int), (void*)&K);
200         err = clSetKernelArg(kernel3b, 2, sizeof(cl_mem), (void*)&bufB_TR);
201         err = clSetKernelArg(kernel3b, 3, sizeof(int), (void*)&N_XL);
202         err = clSetKernelArg(kernel3b, 4, sizeof(int), (void*)&K_XL);
203         err = clSetKernelArg(kernel3b, 5, sizeof(cl_mem), (void*)&bufB_TR_XL);
204         checkError(err,__LINE__);
205         cl_kernel kernel3c = clCreateKernel(program, "paddingRemoveZeroes", &err);
206         checkError(err,__LINE__);
207         err = clSetKernelArg(kernel3c, 0, sizeof(int), (void*)&M_XL);
208         err = clSetKernelArg(kernel3c, 1, sizeof(int), (void*)&N_XL);
```

```
209         err = clSetKernelArg(kernel3c, 2, sizeof(cl_mem), (void*)&bufC_XL);
210         err = clSetKernelArg(kernel3c, 3, sizeof(int), (void*)&M);
211         err = clSetKernelArg(kernel3c, 4, sizeof(int), (void*)&N);
212         err = clSetKernelArg(kernel3c, 5, sizeof(cl_mem), (void*)&bufC);
213         checkError(err,__LINE__);
214         const size_t pLocal[2] = { PADDINGX, PADDINGY };
215         const size_t pAGlobal[2] = { (size_t)M_XL, (size_t)K_XL };
216         const size_t pBGlobal[2] = { (size_t)N_XL, (size_t)K_XL };
217         const size_t pCGlobal[2] = { (size_t)M, (size_t)N };
218     #endif
219
220     // Configure the thread/work-group dimensions of the myGEMM kernel
221     #if KERNEL == 1 || KERNEL == 2
222         const size_t local[2] = { TS, TS };
223         const size_t global[2] = { (size_t)M, (size_t)N };
224     #elif KERNEL == 3 || KERNEL == 5
225         const size_t local[2] = { TS, TS/WPT };
226         const size_t global[2] = { (size_t)M, (size_t)(N/WPT) };
227     #elif KERNEL == 4
228         const size_t local[2] = { TS/WIDTH, TS };
229         const size_t global[2] = { (size_t)(M/WIDTH), (size_t)N };
230     #elif KERNEL == 6 || KERNEL == 7 || KERNEL == 8 || KERNEL == 9
231         const size_t local[2] = { TSM/WPTM, TSN/WPTN };
232         const size_t global[2] = { (size_t)(M/WPTM), (size_t)(N/WPTN) };
233     #elif KERNEL == 10
234         const size_t local[2] = { TSM/WPTM, TSN/WPTN };
235         const size_t global[2] = { (size_t)(M_XL/WPTM), (size_t)(N_XL/WPTN) };
236     #elif KERNEL == 11
237         const size_t local[2] = { THREADSX, THREADSY };
238         const size_t global[2] = { (size_t)(M/RX), (size_t)(N/RY) };
239     #endif
240
241     // Start the timed loop
242     double startTime = timer();
243     for (int r=0; r<NUM_RUNS; r++) {
244
245         // Run the transpose kernel first
246         #if KERNEL == 5 || KERNEL == 6 || KERNEL == 7 || KERNEL == 8 || KERNEL == 9 || KERNEL == 10
247             err = clEnqueueNDRangeKernel(queue, kernel2, 2, NULL, tGlobal, tLocal, 0, NULL, &event);
248         #endif
249
250         // Make the inputs extra large with padded zeros
251         #if KERNEL == 10
252             err = clEnqueueNDRangeKernel(queue, kernel3a, 2, NULL, pAGlobal, pLocal, 0, NULL, &event);
253             err = clEnqueueNDRangeKernel(queue, kernel3b, 2, NULL, pBGlobal, pLocal, 0, NULL, &event);
254         #endif
255
256         // Run the myGEMM kernel
257         err = clEnqueueNDRangeKernel(queue, kernel1, 2, NULL, global, local, 0, NULL, &event);
258
259         // Remove padded zeroes from the larger output
260         #if KERNEL == 10
261             err = clEnqueueNDRangeKernel(queue, kernel3c, 2, NULL, pCGlobal, pLocal, 0, NULL, &event);
262         #endif
263
264         // Wait for calculations to be finished
265         checkError(err,__LINE__);
266         err = clWaitForEvents(1, &event);
267     }
268
269     // End the timed loop
270     timers[timerID].t += (timer() - startTime) / (double)NUM_RUNS;
271     timers[timerID].kf += ((long)K * (long)M * (long)N * 2) / 1000;
272
273     // Copy the output matrix C back to the CPU memory
274     err = clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0, M*N*sizeof(*C), C, 0, NULL, NULL);
275     checkError(err,__LINE__);
276
277     // Free the memory objects
278     free(code);
279     clReleaseMemObject(bufA);
280     clReleaseMemObject(bufB);
281     clReleaseMemObject(bufB_TR);
282     clReleaseMemObject(bufC);
283     clReleaseMemObject(bufA_XL);
```

```
284     clReleaseMemObject(bufB_TR_XL);
285     clReleaseMemObject(bufC_XL);
286 
287     // Clean-up OpenCL
288     clReleaseCommandQueue(queue);
289     clReleaseContext(context);
290     clReleaseProgram(program);
291     clReleaseKernel(kernel1);
292     #if KERNEL == 5 || KERNEL == 6 || KERNEL == 7 || KERNEL == 8 || KERNEL == 9 || KERNEL == 10
293         clReleaseKernel(kernel2);
294     #endif
295     #if KERNEL == 10
296         clReleaseKernel(kernel3a);
297         clReleaseKernel(kernel3b);
298         clReleaseKernel(kernel3c);
299     #endif
300 }
301 
302 // =================================================================================================
303 
304 // Print an error message to screen (only if it occurs)
305 void checkError(cl_int error, int line) {
306     if (error != CL_SUCCESS) {
307         switch (error) {
308             case CL_DEVICE_NOT_FOUND:                 printf("-- Error at %d:  Device not found.\n", line); break;
309             case CL_DEVICE_NOT_AVAILABLE:             printf("-- Error at %d:  Device not available\n", line); break;
310             case CL_COMPILER_NOT_AVAILABLE:           printf("-- Error at %d:  Compiler not available\n", line); break;
311             case CL_MEM_OBJECT_ALLOCATION_FAILURE:    printf("-- Error at %d:  Memory object allocation failure\n", line); break
312             case CL_OUT_OF_RESOURCES:                 printf("-- Error at %d:  Out of resources\n", line); break;
313             case CL_OUT_OF_HOST_MEMORY:               printf("-- Error at %d:  Out of host memory\n", line); break;
314             case CL_PROFILING_INFO_NOT_AVAILABLE:     printf("-- Error at %d:  Profiling information not available\n", line); br
315             case CL_MEM_COPY_OVERLAP:                 printf("-- Error at %d:  Memory copy overlap\n", line); break;
316             case CL_IMAGE_FORMAT_MISMATCH:            printf("-- Error at %d:  Image format mismatch\n", line); break;
317             case CL_IMAGE_FORMAT_NOT_SUPPORTED:       printf("-- Error at %d:  Image format not supported\n", line); break;
318             case CL_BUILD_PROGRAM_FAILURE:            printf("-- Error at %d:  Program build failure\n", line); break;
319             case CL_MAP_FAILURE:                      printf("-- Error at %d:  Map failure\n", line); break;
320             case CL_INVALID_VALUE:                    printf("-- Error at %d:  Invalid value\n", line); break;
321             case CL_INVALID_DEVICE_TYPE:              printf("-- Error at %d:  Invalid device type\n", line); break;
322             case CL_INVALID_PLATFORM:                 printf("-- Error at %d:  Invalid platform\n", line); break;
323             case CL_INVALID_DEVICE:                   printf("-- Error at %d:  Invalid device\n", line); break;
324             case CL_INVALID_CONTEXT:                  printf("-- Error at %d:  Invalid context\n", line); break;
325             case CL_INVALID_QUEUE_PROPERTIES:         printf("-- Error at %d:  Invalid queue properties\n", line); break;
326             case CL_INVALID_COMMAND_QUEUE:            printf("-- Error at %d:  Invalid command queue\n", line); break;
327             case CL_INVALID_HOST_PTR:                 printf("-- Error at %d:  Invalid host pointer\n", line); break;
328             case CL_INVALID_MEM_OBJECT:               printf("-- Error at %d:  Invalid memory object\n", line); break;
329             case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:  printf("-- Error at %d:  Invalid image format descriptor\n", line); break;
330             case CL_INVALID_IMAGE_SIZE:               printf("-- Error at %d:  Invalid image size\n", line); break;
331             case CL_INVALID_SAMPLER:                  printf("-- Error at %d:  Invalid sampler\n", line); break;
332             case CL_INVALID_BINARY:                   printf("-- Error at %d:  Invalid binary\n", line); break;
333             case CL_INVALID_BUILD_OPTIONS:            printf("-- Error at %d:  Invalid build options\n", line); break;
334             case CL_INVALID_PROGRAM:                  printf("-- Error at %d:  Invalid program\n", line); break;
335             case CL_INVALID_PROGRAM_EXECUTABLE:       printf("-- Error at %d:  Invalid program executable\n", line); break;
336             case CL_INVALID_KERNEL_NAME:              printf("-- Error at %d:  Invalid kernel name\n", line); break;
337             case CL_INVALID_KERNEL_DEFINITION:        printf("-- Error at %d:  Invalid kernel definition\n", line); break;
338             case CL_INVALID_KERNEL:                   printf("-- Error at %d:  Invalid kernel\n", line); break;
339             case CL_INVALID_ARG_INDEX:                printf("-- Error at %d:  Invalid argument index\n", line); break;
340             case CL_INVALID_ARG_VALUE:                printf("-- Error at %d:  Invalid argument value\n", line); break;
341             case CL_INVALID_ARG_SIZE:                 printf("-- Error at %d:  Invalid argument size\n", line); break;
342             case CL_INVALID_KERNEL_ARGS:              printf("-- Error at %d:  Invalid kernel arguments\n", line); break;
343             case CL_INVALID_WORK_DIMENSION:           printf("-- Error at %d:  Invalid work dimensionsension\n", line); break;
344             case CL_INVALID_WORK_GROUP_SIZE:          printf("-- Error at %d:  Invalid work group size\n", line); break;
345             case CL_INVALID_WORK_ITEM_SIZE:           printf("-- Error at %d:  Invalid work item size\n", line); break;
346             case CL_INVALID_GLOBAL_OFFSET:            printf("-- Error at %d:  Invalid global offset\n", line); break;
347             case CL_INVALID_EVENT_WAIT_LIST:          printf("-- Error at %d:  Invalid event wait list\n", line); break;
348             case CL_INVALID_EVENT:                    printf("-- Error at %d:  Invalid event\n", line); break;
349             case CL_INVALID_OPERATION:                printf("-- Error at %d:  Invalid operation\n", line); break;
350             case CL_INVALID_GL_OBJECT:                printf("-- Error at %d:  Invalid OpenGL object\n", line); break;
351             case CL_INVALID_BUFFER_SIZE:              printf("-- Error at %d:  Invalid buffer size\n", line); break;
352             case CL_INVALID_MIP_LEVEL:                printf("-- Error at %d:  Invalid mip-map level\n", line); break;
353             case -1024:                               printf("-- Error at %d:  *clBLAS* Functionality is not implemented\n", lir
354             case -1023:                               printf("-- Error at %d:  *clBLAS* Library is not initialized yet\n", line)
355             case -1022:                               printf("-- Error at %d:  *clBLAS* Matrix A is not a valid memory object\n"
356             case -1021:                               printf("-- Error at %d:  *clBLAS* Matrix B is not a valid memory object\n"
357             case -1020:                               printf("-- Error at %d:  *clBLAS* Matrix C is not a valid memory object\n"
358             case -1019:                               printf("-- Error at %d:  *clBLAS* Vector X is not a valid memory object\n"
```

```
359             case -1018:                             printf("-- Error at %d:  *clBLAS* Vector Y is not a valid memory object\n'
360             case -1017:                             printf("-- Error at %d:  *clBLAS* An input dimension (M,N,K) is invalid\n'
361             case -1016:                             printf("-- Error at %d:  *clBLAS* Leading dimension A must not be less tha
362             case -1015:                             printf("-- Error at %d:  *clBLAS* Leading dimension B must not be less tha
363             case -1014:                             printf("-- Error at %d:  *clBLAS* Leading dimension C must not be less tha
364             case -1013:                             printf("-- Error at %d:  *clBLAS* The increment for a vector X must not be
365             case -1012:                             printf("-- Error at %d:  *clBLAS* The increment for a vector Y must not be
366             case -1011:                             printf("-- Error at %d:  *clBLAS* The memory object for Matrix A is too sm
367             case -1010:                             printf("-- Error at %d:  *clBLAS* The memory object for Matrix B is too sm
368             case -1009:                             printf("-- Error at %d:  *clBLAS* The memory object for Matrix C is too sm
369             case -1008:                             printf("-- Error at %d:  *clBLAS* The memory object for Vector X is too sm
370             case -1007:                             printf("-- Error at %d:  *clBLAS* The memory object for Vector Y is too sm
371             case -1001:                             printf("-- Error at %d:  Code -1001: no GPU available?\n", line); break;
372             default:                                printf("-- Error at %d:  Unknown with code %d\n", line, error);
373         }
374         exit(1);
375     }
376 }
377
378 // ================================================================================================
```