Parallel Processing in CUDA/OpenCL Laboratories

Tomasz Boiński
Paweł Czarnul
Paweł Rościszewski
Michał Wójcik

January 2014

Contents

Chapter 1

Laboratory 1 - Basics

1.1 CUDA and OpenCl enabled hosts

CUDA And OpenCL applications can be run on the computers located in room 527 (des01 to des18) and on two servers: apl09 (IP: 153.19.50.41) and apl10 (IP: 153.19.50.42). All those 20 machines are integrated with Department's authentication system so each student that has a KASK account can access all of those machines. Also the home directories are shared accros all those machines.

Aforementioned machines are not publicly visible. The access is limited to the faculty's network. Access form outside of the Faculty is possible using faculty VPN or through laboratory's firewall kask.eti.pg.gda.pl (recommended). When accessing the laboratory using the VPN users should use IP addresses of CUDA server instead of their names.

Students that does not have KASK account should log in do computers in room 527 using `student` account with password `student`. The servers are than available via SSH using `studentXX` accounts, where XX is a number form 01 to 18. All those accounts have `student` password. The user should select the account that has the same number as the hos he or she is currently working on. This will prevent mixing up files with code of other students. Be aware that all aforementioned computers are only available from Facultys network. If you want to work from home you have to connect using VPN as described on `http://starter.eti.pg.gda.pl/` web page.

When writing and OpenCL code students should use only functions defined by the standard and defined in `CL/cl.h` header file. Especially students should not use `ocl.h` provided by NVIDIA SDK.

1.2 Compiling

In general the code can be compiled using the commands as described later in this section.

On computers in room 527 (desXX), apl09 and apl10 CUDA code can be compiled by running:

```
$ nvcc program.cu -o program
```

On in room 527 (desXX), apl09 and apl10 OpenCL code can be compiled by running:

```
$ nvcc -lOpenCL program.cpp -o program
```

1.3 NVidia SDK

Read only CUDA SDK is available on all machines. On computers in room 527 it is available in `/opt/NVIDIA_GPU_Computing_SDK` directory and on apl09 and apl10 servers on `/CUDA_SDK/NVIDIA_GPU_Computing_SDK` directory. In all cases users should copy those directories to their home directories and use their own copy.

1.4 The Task

During this laboratory a program in either CUDA or OpenCL should be written, compiled and executed. The program should present in graphical form a result of an algorithm provided by the lecturer.

The program can be based on examples described in "CUDA by Example: An Introduction to General-Purpose GPU Programming" [3] book that is available in University's library and at Department of Computer Architecture. The source code of the examples can be downloaded from `https://developer.nvidia.com/content/cuda-example-introduction-general-purpose-gpu-programming-0`. The samples for this laboratory can be found in file `chapter04/julia_gpu.cu` and on Listing 1.1

1.5 The Example

Listing 1.1: Listing of julia_gpu.cu

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement (EULA)
 * associated with this source code for terms and conditions that govern
 * your use of this NVIDIA software.
 *
 */


#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 1000

struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
```

```
    }
29  __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
31  }
    __device__ cuComplex operator+(const cuComplex& a) {
33      return cuComplex(r+a.r, i+a.i);
    }
35  };

37  __device__ int julia( int x, int y ) {
    const float scale = 1.5;
39  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
41
    cuComplex c(-0.8, 0.156);
43  cuComplex a(jx, jy);

45  int i = 0;
    for (i=0; i<200; i++) {
47      a = a * a + c;
        if (a.magnitude2() > 1000)
49          return 0;
    }
51
    return 1;
53  }

55  __global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
57  int x = blockIdx.x;
    int y = blockIdx.y;
59  int offset = x + y * gridDim.x;

61  // now calculate the value at that position
    int juliaValue = julia( x, y );
63  ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
65  ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
67  }

69  // globals needed by the update routine
    struct DataBlock {
71      unsigned char   *dev_bitmap;
    };
73
    int main( void ) {
75      DataBlock   data;
        CPUBitmap bitmap( DIM, DIM, &data );
77      unsigned char    *dev_bitmap;

79      HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
        data.dev_bitmap = dev_bitmap;
81
        dim3    grid(DIM,DIM);
83      kernel<<<grid,1>>>( dev_bitmap );

85      HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                                  bitmap.image_size(),
87                                cudaMemcpyDeviceToHost ) );

89      HANDLE_ERROR( cudaFree( dev_bitmap ) );

91      bitmap.display_and_exit();
    }
```

The example uses CUDA kernel (lines 55 – 67) to calculate the points that should be displayed by checking whether they are inside of Julia set or not. For displaying the picture itself the code uses external library (`freeglut`, line 91). The Julia set is a complex number set so a proper structure for storing the data and perform calculations on the set elements needs to be implemented. Such tasks are thus implemented as `cuComplex` structure (lines 22 – 35) which is a

standard C language structure with added keywords allowing storing data and running operators on the device (the `__device__` keyword).

1.6 Compiling the example

The example presented on Listing 1.1 and available within provided examples contains an error preventing it form compiling. The `cuComplex` structure will be placed on the device and not in main memory of the host machine. To initialize the structure a pseudo-class constructor implemented in the structure and than called later in the code. The constructor lacks `__device__` keyword preventing it from being run on the device. So the line 25 of the code should read:

```
__device__ cuComplex( float a, float b ) : r(a), i(b) {}
```

After the modification we can compile the code using the following command:

```
$ nvcc -lglut -lpthread julia_gpu.cu -o julia_gpu
```

The code requires that the `freeglut` library is present. It also requires proper version of GCC. Usually modern versions are not supported so in KASK Laboratory version 4.4.6 was installed just for CUDA compiling.

1.7 Running the example locally and on apl09 and apl10 servers

After compiling the code can be just run using command:

```
$ ./julia_gpu
```

When it is being run remotely on one of the two CUDA servers X Window forwarding is needed for displaying of the results. For that, from Linux client, you can login to one of the servers using SSH with forwarding enabled using the -X switch:

```
$ ssh -X user@apl09
```

When a graphical application is run using X Window forwarding it is executed on the remote host but the graphical window is displayed locally using local X Window Server, that can run on any compatible host (including Windows with CygWin and Linux in virtual machines). It is possible to transfer such window through multiple Linux hosts. The user should log in to each one in the chain using the -X switch. For example when running application on apl09 from home (from where apl09 is not directly accessible), the user can first log in to the firewall:

```
$ ssh -X user@kask.eti.pg.gda.pl
```

and from there to apl09 server:

```
$ ssh -X user@apl09
```

The application should be started on cuda1.server and the output will be first forwarded do kask.eti.pg.gda.pl server and than to the local machine. It's worth noting that such app will only see files and folders residing on apl09 server, not the local machine.

Chapter 2

Laboratory 2 - Matrix Multiplication

2.1 The Task

Students should write and optimize a code for matrix multiplication both in CUDA and OpenCL. The code should allow multiplication of square matrices of any size and printing of the results.

The final code should:

- allow multiplication of square matrices of any size,
- allow selection of the device it will be run on (i.e. one of the GPU's available on each host),
- allow selection of the size of the grid,
- include basic optimizations (usage of shared memory, data prefetching, coalescing),
- allow comparison of execution time between application written in CUDA and in OpenCL.

Students should compare times of execution dependent on technology used, input size, graphic card and grid size.

For completion of the tasks student can get up to 5 points:

- 2 points for writing, compiling and running CUDA version of the code,
- 2 points for writing, compiling and running OpenCL version of the code,
- 1 point for comparison of the results.

2.2 Sample Code

The basic code in CUDA for matrix multiplication is shown on Listing 2.1. For OpenCl the kernel is shown on Listing 2.2 and the program itself on Listing 2.3. See comments for explanation of crucial parts of the code.

Listing 2.1: Listing of basic CUDA program

```
#include "cuda.h"
#include "cuda_runtime.h"
#include "stdio.h"
#include "stdlib.h"

#define TILE_WIDTH 2

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  float Pvalue = 0;

  for (int k = 0; k < Width; ++k) {
    Pvalue += Md[ty * Width + k] * Nd[k * Width + tx];
  }
  Pd[ty * Width + tx] = Pvalue;
}

void MatrixMultiplication(float* M, float* N, float* P, int Width) {
        int size = Width * Width * sizeof(float);
        float* Md;
        float* Nd;
        float* Pd;

        cudaMalloc((void**) &Md, size);
        cudaMalloc((void**) &Nd, size);
        cudaMalloc((void**) &Pd, size);

        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

        //dim3 dimBlock(Width, Width, 1);
        //dim3 dimGrid(1,1,1);
        dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
        dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

        MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

        cudaFree(Md);
        cudaFree(Nd);
        cudaFree(Pd);
}

void ReadMatrix(float* M, int Width) {
        int i = 0;
        int j = 0;
        for (i = 0; i < Width; i++) {
                for (j = 0; j < Width; j++) {
                        scanf("%f", &M[i * Width + j]);
                }
        }
}

void PrintMatrix(float* M, int Width) {
        int i;
        int j;
        for (i = 0; i < Width; i++) {
                for (j = 0; j < Width; j++) {
                        printf("%f␣", M[i * Width + j]);
                }
                printf("\n");
        }
}

int main(void) {
        float* M;
        float* N;
        float* P;
        int Width;

```

```
        scanf("%d", &Width);

        M = (float*) malloc(Width * Width * sizeof(float));
        N = (float*) malloc(Width * Width * sizeof(float));
        P = (float*) malloc(Width * Width * sizeof(float));

        ReadMatrix(M, Width);
        ReadMatrix(N, Width);

        MatrixMultiplication(M, N, P, Width);

        printf("\n");
        PrintMatrix(M, Width);
        printf("\n");
        PrintMatrix(N, Width);
        printf("\n");
        PrintMatrix(P, Width);
        free(M);
        free(N);
        free(P);
}
```

Listing 2.2: Listing of basic OpenCL kernel

```
__kernel void matrixMul(__global float* P, __global float* M, __global float*
      N,  int Width) {

   // 2D Thread ID
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   // value stores the element
   // that is computed by the thread
   float value = 0;
   for (int k = 0; k < Width; ++k) {
      float elementA = M[ty * Width + k];
      float elementB = N[k * Width + tx];
      value += elementA * elementB;
   }

   // Write the matrix to device memory each
   // thread writes one element
   P[ty * Width + tx] = value;
}
```

Listing 2.3: Listing of basic OpenCL program

```
#include "stdio.h"
#include "stdlib.h"
#include "oclUtils.h"
#include "iostream"

#define TILE_WIDTH 2

using namespace std;

void ReadMatrix(float* M, int Width) {
        int i = 0;
        int j = 0;
        for (i = 0; i < Width; i++) {
                for (j = 0; j < Width; j++) {
                        scanf("%f", &M[i * Width + j]);
                }
        }
}

void PrintMatrix(float* M, int Width) {
        int i = 0;
        int j = 0;
        for (i = 0; i < Width; i++) {
                for (j = 0; j < Width; j++) {
                        printf("%f␣", M[i * Width + j]);
                }
```

```
27                  printf("\n");
            }
29  }

31  void pfn_notify(const char *errinfo, const void *private_info, size_t cb,
        void *user_data) {
            printf("Error␣%s\n", errinfo);
33  }

35  void pfn_notify2(cl_program program, void* user_data) {
            size_t param_value_size;
37          clGetProgramBuildInfo(program, NULL, CL_PROGRAM_BUILD_LOG, 0, NULL, &
                param_value_size);
            char* param_value = new char[param_value_size + 1];
39          //char* param_value = (char*) malloc(param_value_size+1);
            clGetProgramBuildInfo(program, NULL, CL_PROGRAM_BUILD_LOG,
                param_value_size, param_value, NULL);
41          param_value[param_value_size] = '\0';
            cout << param_value;
43          //printf("%s\n", param_value);
            delete param_value;
45  }

47  int main(void) {
            float* M;
49          float* N;
            float* P;
51          int Width;

53          scanf("%d", &Width);

55          M = (float*) malloc(Width * Width * sizeof(float));
            N = (float*) malloc(Width * Width * sizeof(float));
57          P = (float*) malloc(Width * Width * sizeof(float));

59          ReadMatrix(M, Width);
            ReadMatrix(N, Width);
61
            //OpenCL specific variables
63          cl_context clGPUContext;
            cl_command_queue clCommandQue;
65          cl_program clProgram;
            cl_kernel clKernel;
67
            size_t dataBytes;
69          size_t kernelLength;
            cl_int errcode;
71
            // OpenCL device memory for matrices
73          cl_mem Md;
            cl_mem Nd;
75          cl_mem Pd;

77          // Get platforms
            cl_uint num_platforms;
79          errcode = clGetPlatformIDs(0, NULL, &num_platforms);
            shrCheckError(errcode, CL_SUCCESS);
81          cl_platform_id* platforms = (cl_platform_id*) malloc(num_platforms *
                sizeof(cl_platform_id));
            errcode = clGetPlatformIDs(num_platforms, platforms, NULL);
83          shrCheckError(errcode, CL_SUCCESS);
            // Initialize context
85          cl_context_properties props[3];
            props[0] = CL_CONTEXT_PLATFORM;
87          props[1] = (cl_context_properties) platforms[0];
            props[2] = 0;
89          clGPUContext = clCreateContextFromType(props, CL_DEVICE_TYPE_GPU,
                                                   &pfn_notify, NULL, &errcode);
91          shrCheckError(errcode, CL_SUCCESS);

93          // Check for devices
            errcode = clGetContextInfo(clGPUContext, CL_CONTEXT_DEVICES,
95                                     0, NULL, &dataBytes);
            shrCheckError(errcode, CL_SUCCESS);
```

```
97          cl_device_id *clDevices = (cl_device_id *) malloc(dataBytes);
            errcode = clGetContextInfo(clGPUContext, CL_CONTEXT_DEVICES,
99                                     dataBytes, clDevices, NULL);
            shrCheckError(errcode, CL_SUCCESS);
101
            // Create command queue
103         clCommandQue = clCreateCommandQueue(clGPUContext, clDevices[0],
                                                0, &errcode);
105         shrCheckError(errcode, CL_SUCCESS);

107         // Set up device memory
            Pd = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE,
109                             Width*Width*sizeof(float), NULL, &errcode);
            shrCheckError(errcode, CL_SUCCESS);
111         Nd = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE |
                CL_MEM_COPY_HOST_PTR,
                                Width*Width*sizeof(float), N, &errcode);
113         shrCheckError(errcode, CL_SUCCESS);
            Md = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE |
                CL_MEM_COPY_HOST_PTR,
115                             Width*Width*sizeof(float), M, &errcode);
            shrCheckError(errcode, CL_SUCCESS);
117
            // Create program
119         char* kernelCode = oclLoadProgSource("kernel.cl", "// Some comment\n"
                ,
                                                &kernelLength);
121
            // Create kernel
123         clProgram = clCreateProgramWithSource(clGPUContext, 1, (const char**)
                 &kernelCode,
                                                &kernelLength, &errcode);
125         shrCheckError(errcode, CL_SUCCESS);

127         // Build program
            errcode = clBuildProgram(clProgram, 0, NULL, NULL, &pfn_notify2, NULL
                );
129         shrCheckError(errcode, CL_SUCCESS);

131         // Create kernel
            clKernel = clCreateKernel(clProgram, "matrixMul", &errcode);
133         shrCheckError(errcode, CL_SUCCESS);

135         // Set kernel arguments
            errcode = clSetKernelArg(clKernel, 0, sizeof(cl_mem), (void *)&Pd);
137         errcode |= clSetKernelArg(clKernel, 1, sizeof(cl_mem), (void *)&Md);
            errcode |= clSetKernelArg(clKernel, 2, sizeof(cl_mem), (void *)&Nd);
139         errcode |= clSetKernelArg(clKernel, 3, sizeof(int), (void *)&Width);
            shrCheckError(errcode, CL_SUCCESS);
141
            // Set dimensions
143         size_t localWorkSize[2], globalWorkSize[2];
            localWorkSize[0] = TILE_WIDTH;
145         localWorkSize[1] = TILE_WIDTH;
            globalWorkSize[0] = Width;
147         globalWorkSize[1] = Width;

149         cl_event *events = (cl_event*) malloc(3 * sizeof(cl_event));

151         // Start execution
            errcode = clEnqueueNDRangeKernel(clCommandQue, clKernel, 2, NULL,
153                                          globalWorkSize, localWorkSize, 0,
                                                 NULL, &events[0]);
            shrCheckError(errcode, CL_SUCCESS);
155
            // Get results
157         errcode = clEnqueueReadBuffer(clCommandQue, Pd, CL_FALSE, 0, Width*
                Width*sizeof(float),
                                          P, 1, events, &events[1]);
159         shrCheckError(errcode, CL_SUCCESS);

161         clWaitForEvents(2, events);

163         printf("\n");
```

```
        PrintMatrix(M, Width);
165     printf("\n");
        PrintMatrix(N, Width);
167     printf("\n");
        PrintMatrix(P, Width);
169
        free(M);
171     free(N);
        free(P);
173
        // Free open cl objects
175     free(events);
        clReleaseMemObject(Md);
177     clReleaseMemObject(Nd);
        clReleaseMemObject(Pd);
179     free(clDevices);
        free(kernelCode);
181     clReleaseContext(clGPUContext);
        clReleaseKernel(clKernel);
183     clReleaseProgram(clProgram);
        clReleaseCommandQueue(clCommandQue);
185 }
```

Chapter 3

Laboratory 3 - Debugging of CUDA applications

3.1 How to debug applications

Debugging allows finding non obvious bugs in applications. Usually the process of debugging requires to go through the application line by one and for simple input check what is happening in the code. Unfortunately for multi threaded application this process can be very complicated.

To be able to debug software we need to compile it with -g -G flags:

```
$ nvcc -g -G bitreverse.cu -o bitreverse
```

What it does:

- forces `-O0` (mostly unoptimized) compilation,
- makes the compiler include symbolic debugging information in the executable.

Than we start debugging by running:

```
$ cuda-gdb bitreverse
```

Cuda-dbg is used the same way as gdb, so function symbol names and source file line numbers can be used as break-point symbols. Breakpoints can be added both in host and device functions using commands:

- `(cuda-gdb) break mykernel_main`,
- `(cuda-gdb) b mykernel_main`;

The above command will set a breakpoint at a particular device location (the address of `mykernel_main`), and it will force all resident GPU threads to stop at this location, there is currently no method to stop only certain threads or warps at any given breakpoint.

To query info about devices one of the functions can be used:

- `(cuda-gdb) info cuda devices` – all the devices,
- `(cuda-gdb) info cuda sm` – all SMs in the current device,

- `(cuda-gdb) info cuda warp` – all warps in the current SM,
- `(cuda-gdb) info cuda lane` – all lanes in the current warp,
- `(cuda-gdb) info cuda kernels` – all active kernels,
- `(cuda-gdb) info cuda blocks` – active block in the current kernel,
- `(cuda-gdb) info cuda threads` – active threads in the current kernel.

Those options vary between cuda-gdb versions!

Inspecting the coordinates (while in kernel function) can be done using (cuda-gdb) cuda with:

- device,
- sm,
- warp,
- lane,
- block,
- thread,
- kernel.

To change the physical coordinates use:

```
(cuda-dbg) cuda device 0 sm 1 warp 2 lane 3
```

To change the logical coordinates use:

```
(cuda-dbg) cuda thread (15,0,0)
(cuda-dbg) cuda block (1,0) thread (3,0,0)
```

- providing only the CUDA thread coordinates will maintain the current block of focus while switching to the specified thread,
- providing only thread (10) will switch to the CUDA thread with X coordinates 10 and Y and Z coordinates 0;

To change kernel focus use:

```
(cuda-dbg) cuda kernel 0
```

Furthermore when debugging a kernel that appear to be hanging or looping indefinitely, the Ctrl-C signal can be used, it will freeze the GPU and report back the source code location.

3.2 Example

The usage of the `cuda-gdb` will be presented on simple code shown on Listing 3.1. The program performs simple 8-bit bit reversal operations.

Listing 3.1: Listing of bitreverse.cu

```
#include <stdio.h>
#include <stdlib.h>

// Simple 8-bit bit reversal Compute test

#define N 256

__global__ void bitreverse(void *data) {
        unsigned int *idata = (unsigned int*)data;
        extern __shared__ int array[];

        array[threadIdx.x] = idata[threadIdx.x];

        array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
                ((0x0f0f0f0f & array[threadIdx.x]) << 4);
        array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
                ((0x33333333 & array[threadIdx.x]) << 2);
        array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
                ((0x55555555 & array[threadIdx.x]) << 1);

        idata[threadIdx.x] = array[threadIdx.x];
}

int main(void) {
        void *d = NULL; int i;
        unsigned int idata[N], odata[N];

        for (i = 0; i < N; i++)
                idata[i] = (unsigned int)i;

        cudaMalloc((void**)&d, sizeof(int)*N);
        cudaMemcpy(d, idata, sizeof(int)*N,
                cudaMemcpyHostToDevice);

        bitreverse<<<1, N, N*sizeof(int)>>>(d);

        cudaMemcpy(odata, d, sizeof(int)*N,
                cudaMemcpyDeviceToHost);

        for (i = 0; i < N; i++)
                printf("%u -> %u\n", idata[i], odata[i]);

        cudaFree((void*)d);
        return 0;
}
```

Part of the output is presented on Listing 3.2

Listing 3.2: Output of bitreverse.cu

```
0 -> 0
1 -> 128
2 -> 64
3 -> 192
4 -> 32
5 -> 160
6 -> 96
7 -> 224
8 -> 16
9 -> 144
10 -> 80
11 -> 208
```

```
12 -> 48
13 -> 176
```

To start debugging:

- compile code with appropriate flags,
- run cuda-gdb,
- put breakpoint at host function,
- put breakpoint at device function,
- put breakpoint according to line number,

```
$ nvcc -g -G bitreverse.cu -o bitreverse
$ cuda-gdb bitreverse

(cuda-gdb) b main //breakpoint
  Breakpoint 1 at 0x400950: file bitreverse.cu, line 25
(cuda-gdb) b bitreverse //breakpoint
  Breakpoint 2 at 0x400b06: file bitreverse.cu, line 8
(cuda-gdb) b bitreverse.cu:21 //breakpoint
  Breakpoint 3 at 0x400b12: file bitreverse.cu, line21
```

- run program,
- new process and thread information will be shown,
- the code will stop at first brakpoint,

```
(cuda-gdb) r //run
  Starting program: /macierz/home/psysiu/dydaktyka/cuda/debugging/bitreverse
  [Thread debugging using libthread_db enabled]
  [New process 26269 ]
  [New Thread 140057761130272 (LWP 2 6 2 6 9)]
  [Switching to Thread 140057761130272 (LWP 2 6 2 6 9)]

  Breakpoint 1, main() at bitreverse.cu:25
  25 void *d = NULL; int i;
```

At this point, commands can be entered to advance execution and/or print program state. For this walkthrough we will continue to the device kernel:

- creating new context on the first device,
- launching kernel,
- breakpoint in kernel,
- cuda-gdb has detected that we have reached a CUDA device kernel, so it prints the current CUDA thread and focus.

```
(cuda-gdb) c //continue
  Continuing.
  [Context Create of context 0x1a53e10 on Device 0]
  Breakpoint 3 at 0x1b92070: file bitreverse.cu, line 21.
  [Launch of CUDA Kernel 0 (bitreverse<<<(1,1,1),(256,1,1)>>>) on Device 0]
  [Switching focus to CUDA kernel 0, grid 1, block(0,0,0), thread (0,0,0),
       device 0, sm 2, warp 0]

  Breakpoint 2, bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x200100000) at
       bitreverse.cu:9
  warning: Sourcefile is more recent than executable.
  9 unsigned int *idata = (unsigned int*) data;
```

The output bellow tells us that the host thread of focus has LWP ID of 26176 and the current CUDA thread has block coordinates of (0,0,0) and thread coordinates (0,0,0):

```
(cuda-gdb) info threads
  2 Thread 139666865731360 (LWP 26716) 0x00007f06b7f1ee89 in ?? () from /usr/
      lib/libcuda.so
  1 process 26716 0x00007f06b7f1ee89 in ?? () from /usr/lib/libcuda.so

(cuda-gdb) info cuda threads
    BlockIdx ThreadIdx To BlockIdx ThreadIdx Count VirtualPC
        Filename       Line
  Kernel 0
  * (0,0,0)  (0,0,0)       (0,0,0) (255,0,0)  256   0x00000000020df868
      bitreverse.cu  9
```

Than:

- get info about cuda (device) threads,
- switch to host thread,
- switch to host tread 2,
- show backtrace of the host thread,

```
(cuda-gdb) info cuda threads
    BlockIdx ThreadIdx To BlockIdx ThreadIdx Count VirtualPC
        Filename       Line
  Kernel 0
  * (0,0,0)  (0,0,0)       (0,0,0)  (255,0,0) 256   0x0000000001b91868
      bitreverse.cu  9

(cuda-gdb) thread
  [Current thread is 2 (Thread 140057761130272 (LWP 26269))]

(cuda-gdb) thread 2
  [Switching to thread 2 (Thread 140057761130272 (LWP 26269))]#0 0
      x00007f61bbe16f05 in pthread_mutex_lock

(cuda-gdb )bt //backtrace
  #0 0x00007f61bbe16f05 in pthread_mutex_lock() from /lib/libpthread.so.0
  #1 0x00007f61bb1d59b7 in ?? () from /usr/lib/libcuda.so
  #2 0x00007f61bb20391e in ?? () from /usr/lib/libcuda.so
  #3 0x00007f61bb2098d5 in ?? () from /usr/lib/libcuda.so
  #4 0x00007f61bb209ab6 in ?? () from /usr/lib/libcuda.so
  #5 0x00007f61bb1f495b in ?? () from /usr/lib/libcuda.so
  #6 0x00007f61bb1eae9c in ?? () from /usr/lib/libcuda.so
  #7 0x00007f61bb1cae41 in ?? () from /usr/lib/libcuda.so
  #8 0x00007f61bb1cebc8 in ?? () from /usr/lib/libcuda.so
  #9 0x00007f61bb1c1244 in ?? () from /usr/lib/libcuda.so
  #10 0x00007f61bcd48de2 in ?? () from /usr/local/cuda/lib64/libcudart.so.4
  #11 0x00007f61bcd6c824 in cudaMemcpy () from /usr/local/cuda/lib64/
      libcudart.so.4
  #12 0x0000000000400a3d in main () at bitreverse.cu:37
```

And:

- switch to thread number 170,
- run through the code,

```
(cuda-gdb) cuda thread 170
  [Switching focus to CUDA kernel 0 , grid 1, block (0,0,0), thread (170,0,0)
      , device 0, sm 2 , warp 0
  9 unsigned int *idata=(unsigned int*) data;
(cuda-gdb) n \\next
  12 array[threadIdx.x] = idata[threadIdx.x];
```

```
(cuda-gdb) n
  14 array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
(cuda-gdb) n
  16 array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
(cuda-gdb) n
  18 array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
(cuda-gdb) n
  Breakpoint 3, bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x200100000)
  21 idata[threadIdx.x] = array[threadIdx.x];
```

Finally we can:

- check shared memory content,
- delete breakpoints and continue program.

```
(cuda-gdb) p &array
$13 = (@shared int(*)[0]) 0x0
(cuda-gdb) p array[0]@256
$8 = {0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208, 48, 176, 112, 240,
      8, 136, 72, 200, 40, 168, 104, 232, 24, 152, 88, 216, 56, 184, 120,
     248, 4, 132, 68, 196, 36, 164, 100, 228, 20, 148, 84, 212, 52, 180, 116,
      244, 12, 140, 76, 204, 44, 172, 108, 236, 28, 156, 92, 220, 60, 188,
     124, 252, 2, 130, 66, 194, 34, 162, 98, 226, 18, 146, 82, 210, 50, 178,
     114, 242, 10, 138, 74, 202, 42, 170, 106, 234, 26, 154, 90, 218, 58,
     186, 122, 250, 6, 134, 70, 198, 38, 166, 102, 230, 22, 150, 86, 214, 54,
      182, 118, 246, 14, 142, 78, 206, 46, 174, 110, 238, 30, 158, 94, 222,
     62, 190, 126, 254, 1, 129, 65, 193, 33, 161, 97, 225, 17, 145, 81, 209,
     49, 177, 113, 241, 9, 137, 73, 201, 41, 169, 105, 233, 25, 153, 89, 217,
      57, 185, 121, 249, 5, 133, 69, 197, 37, 165, 101, 229, 21, 149, 85,
     213, 53, 181, 117, 245, 13, 141, 77, 205, 45, 173, 109, 237, 29, 157,
     93, 221, 61, 189, 125, 253, 3, 131, 67, 195, 35, 163, 99, 227...}
(cuda-gdb) p &data
$15 = (@generic void * @parameter *) 0x20
(cuda-gdb) p *(@generic void * @parameter *) 0x20
$16 = (@generic void * @parameter) 0x200100000
(cuda-gdb) delete b
Delete all breakpoints? (y or n) y
(cuda-gdb) continue
Continuing.
```

3.3 The Task

In this laboratory students will be presented with a task of finding 5 bugs in the code delivered by the lecturer. Occurrence of each bug have to be verified using `cuda-gdb` even if it can be found without it by analysing the code. For each bug found and verified the student will be awarded by 1 point with total up to 5 points.

Chapter 4

Laboratory 4 - Mixing CUDA and MPI

4.1 CUDA with MPI

Typical CUDA program runs on a single host and utilizes some or all of the locally available CUDA devices. Some more complex problems require further distribution of program execution on multiple CUDA enabled hosts. The hosts should than communicate with each other and pass intermediate results. Thus the program should be run in parallel on two levels:

1. cluster nodes – on the higher level the calculations should be distributed using chosen MPI (Message Passing Interface) implementation,
2. CUDA devices – each process on MPI node runs a CUDA kernel to further parallelize the calculations.

Every process on a given cluster node behaves as a standard MPI application. Each process initializes and closes MPI environment by running `MPI_Init(...)` and `MPI_Finalize()` functions. Each process can check the total number of MPI hosts using `MPI_Comm_size` function using `MPI_COMM_WORLD` communicator. Using this info and the process rank the code can determine what part of the problem should be calculated on given host. The process should than further distribute the calculations among local CUDA processors. Every MPI host is thus treated as a standard CUDA application with added communication between other hosts.

4.2 Sample Application

This application calculates π using a XVII century equation:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots \tag{4.1}$$

Each thread sums given subset of components of the aforementioned sum. The size of the subset is dependent on the number of CUDA threads and the number of MPI hosts. Sample allocation of threads is shown below:

$\frac{\pi}{4} =$		$\frac{1}{1}$	$-\frac{1}{3}$	$+\frac{1}{5}$	$-\frac{1}{7}$	$+\frac{1}{9}$	...
CUDA id/MPI rank	0/0	1/0	0/1	1/1	0/0	...

The code of the MPI part of the application is presented on Listing 4.1 and the kernel on Listing 4.2.

Listing 4.1: Listing of main.c

```
/* P. Czarnul
   KASK, ETI Politechnika Gdanska
*/

#include <stdio.h>
#include <mpi.h>

#define MAXITER 100000000

void launch_picuda(double max, int nodecount, int noderank, double *
    cudanoderesult);

int main (int argc, char **argv)
{
        int rank, nprocs;



    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);

    double result=0,pi=0;

    launch_picuda(MAXITER, nprocs, rank, &result);

    printf("\nNode %d result=%f\n",rank,result);
    fflush(stdout);

    MPI_Reduce(&result,&pi,1,
               MPI_DOUBLE,MPI_SUM,0,
               MPI_COMM_WORLD);

    if (rank==0) {
      pi*=4;
      printf("\nPi computed = %f\n",pi);
    }

    MPI_Finalize();
       return 1;
}
```

Listing 4.2: Listing of pi.cu

```
/* P. Czarnul
   KASK, ETI Politechnika Gdanska
*/

#include <cuda.h>
#include <cuda_runtime.h>

#define CUDATHREADCOUNT 64

__global__ void picuda(double max, int nodecount, int noderank, double *
    cudathreadresults)
{
 const int i = threadIdx.x;


 int mine=(threadIdx.x + CUDATHREADCOUNT*noderank)*2+1;
```

```
16   int sign=(((mine-1)/2)%2)?-1:1;

18   cudathreadresults[i]=0;

20   for (;mine<max;) {

22     cudathreadresults[i]+=sign/(double)mine;
       mine+=2*nodecount*CUDATHREADCOUNT;
24     sign=(((mine-1)/2)%2)?-1:1;
   }
26

28 }

30 extern "C" void launch_picuda(double max, int nodecount, int noderank, double
       *cudanoderesult)
   {
32
    double nodememresults[CUDATHREADCOUNT];
34
    double *cudathreadresults;
36  int memsize=sizeof(double)*CUDATHREADCOUNT;

38  cudaMalloc((void**)&cudathreadresults, memsize);

40  picuda<<<1, CUDATHREADCOUNT>>>(max, nodecount, noderank, cudathreadresults);

42  cudaMemcpy(nodememresults, cudathreadresults, memsize,
        cudaMemcpyDeviceToHost);

44  for(int i=0;i<CUDATHREADCOUNT;i++)
      *cudanoderesult+=nodememresults[i];
46
    cudaFree(cudathreadresults);
48 }
```

To compile the code the following commands should be run:

```
$ nvcc --gpu-architecture sm_13 -c pi.cu -o pi.o
$ mpicc -c main.c -o main.o
$ mpicc main.o pi.o -o mpicudapi -L/usr/local/cuda/lib64 -
    lcudart
```

Note the `-gpu-architecture` flag that will allow usage of double precision - single precision is not enough. Even the order of calculations can have impact on the results.

And than execution on one host:

```
$ mpirun -np 1 ./mpicudapi
```

The result should be as follows:

```
Node 0 result=0.785398

Pi computed = 3.141593
```

To use more than one host the value of `-np` flag should be increased:

```
$ mpirun -np 4 ./mpicudapi
```

The result should be as follows:

```
Node 1 result=0.002211

Node 0 result=0.781834

Node 2 result=0.000857
```

```
Pi computed = 3.141593

Node 3 result=0.000497
```

When running the application the user can point the machines that should run MPI processes using `-machinefile` flag. Sample content of a text file pointing to apl09 and apl10 servers looks like presented below:

```
$ cat machinefile
apl09.eti.pg.gda.pl
apl10.eti.pg.gda.pl
```

Than to run the code on the aforementioned hosts the command should look like presented below:

```
$ mpirun -machinefile ./machinefile -x LD_LIBRARY_PATH -np 4
    ./mpicudapi
```

4.3 The Task

During this laboratory students should write a parallel program solving a problem stated by the lecturer. The program should work on multiple CUDA enabled hosts. The hosts should communicate with each other and pass intermediate results. The program should be thus run in parallel on two levels:

1. cluster nodes – on the higher level the calculations should be distributed using chosen MPI (Message Passing Interface) implementation,
2. CUDA devices – each process on MPI node runs a CUDA kernel to further parallelize the calculations.

The application should use all available GPU on given MPI host to perform calculations. The algorithm should be able to run on heterogeneous cluster of CUDA devices - every node can have different number and type of CUDA devices. The algorithm needs to be stable, the result returned by running the program should be identical no matter how the cluster is configured. It should be thus independent from the number of MPI hosts, CUDA threads, gird/block size etc.

Chapter 5

Laboratory 5 - Overlapping

5.1 Introduction

One of the reasons to utilize GPUs for our applications is that they allow good performance to energy consumption ratio. However, while measurement of devices' energy consumption is straightforward, their performance vastly depends on the application. There are well established benchmark applications (e.g. LINPACK benchmark), that allow to estimate the devices general performance, which is usually measured in FLOPS (Floating-point Operations Per Second). Such estimates can be misleading for a few reasons, especially in case of GPUs. One of the reasons is that the GPUs allow to execute thousands of threads in parallel, but in most cases, the threads need some data to operate on. For the sample application discussed further in this instruction, the Tesla K20m GPU is able to process 320GB of data per second, whereas it is only possible to copy 6GB/s from the host memory to the GPU. This shows, that it is not so easy to exploit the full potential of the GPUs.

Having this in mind, designers of the contemporary GPUs enabled concurrent data transfers and computations. This allows to apply one of the basic optimization schemes called overlapping. The scheme is used with many different parallel technologies and involves partitioning the input data into chunks and then computing each chunk concurrently with the transmission of the following chunk. This way, only the first data transfer adds to the execution time of the application. In case of Quadro and Tesla GPUs with compute capability higher than 2.0 it is even possible to overlap two data transfers with a kernel execution. During this laboratory, the students should explore this capabilities to optimize a chosen application and get the most out of the available GPUs.

To learn the corresponding CUDA API, we will work with the CUDA samples, that can be easily installed on the computers in the lab, by executing the following command:

```
$ cuda-install-samples-6.0.sh <directory>
```

5.2 Overlapping support levels

The capability of a certain CUDA-enabled GPU to perform concurrent data copying and computation depends on the value of the property `asyncEngineCount`:

- 0 – not supported
- 1 – copying and computing supported
- 2 – two copys and computing supported

In order to check this property of our available GPU, we can modify the `deviceQuery` sample from the previously installed CUDA samples. Listing 5.1 is a fragment from the source in `1_Utilities/deviceQuery/deviceQuery.cpp`:

Listing 5.1: Code fragment of the DeviceQuery utility from CUDA Examples

```
        printf("␣␣Run␣time␣limit␣on␣kernels:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.kernelExecTimeoutEnabled ? "Yes" : "No");
        printf("␣␣Integrated␣GPU␣sharing␣Host␣Memory:␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.integrated ? "Yes" : "No");
        printf("␣␣Support␣host␣page-locked␣memory␣mapping:␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.canMapHostMemory ? "Yes" : "No");
        printf("␣␣Alignment␣requirement␣for␣Surfaces:␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.surfaceAlignment ? "Yes" : "No");
        printf("␣␣Device␣has␣ECC␣support:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.ECCEnabled ? "Enabled" : "Disabled");
#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
        printf("␣␣CUDA␣Device␣Driver␣Mode␣(TCC␣or␣WDDM):␣␣␣␣␣␣␣␣␣␣%s\n",
            deviceProp.tccDriver ? "TCC␣(Tesla␣Compute␣Cluster␣Driver)" : "
            WDDM␣(Windows␣Display␣Driver␣Model)");
#endif
        printf("␣␣Device␣supports␣Unified␣Addressing␣(UVA):␣␣␣␣␣␣␣%s\n",
            deviceProp.unifiedAddressing ? "Yes" : "No");
        printf("␣␣Device␣PCI␣Bus␣ID␣/␣PCI␣location␣ID:␣␣␣␣␣␣␣␣␣␣␣%d␣/␣%d\n",
            deviceProp.pciBusID, deviceProp.pciDeviceID);
```

After this fragment we can add the following line:

```
    printf("␣␣AsyncEngineCount:␣%d\n", deviceProp.asyncEngineCount);
```

Then build and execute the program:

```
$ cd NVIDIA_CUDA-6.0_Samples/1_Utilities/deviceQuery
$ make
$ ./deviceQuery
```

Among other parameters of the GPU we should see the `asyncEngineCount` capability value.

5.3 The `simpleMultiCopy` CUDA sample

The `simpleMultiCopy` sample in `0_Simple/simpleMultiCopy/simpleMultiCopy.cu` is an implementation of a simple incrementation application that measures the copying and computing capabilities of the GPU and compares the execution times with and without overlapping. This section discusses the crucial fragments of this sample.

5.3.1 Measuring execution times on CUDA devices

Measuring the execution time of a GPU operation is possible using system utilities (e.g. gettimeofday C function). However, a more precise way would be to use on–board GPU counters. In CUDA it is possible to measure the time with sub-microsecond resolution, using `cudaEvent` timers. Listing 5.2 highlights the usage of the event API in the `simpleMultiCopy` sample:

Listing 5.2: `simpleMultiCopy` fragments regarding execution time measurement

```
cudaEvent_t start, stop;

...


    cudaEventCreate(&start);
    cudaEventCreate(&stop);

...

    cudaEventRecord(start,0);
    // perform copying/computations
    cudaEventRecord(stop,0);
    cudaEventSynchronize(stop);

    float memcpy_h2d_time;
    cudaEventElapsedTime(&<memcpy/kernel>_time, start, stop);
```

We are able to record the precise moment of our execution using the `cudaEventRecord` function. We can use the `cudaEventElapsedTime` function to obtain the time between two recorded events. The way of time measurement is especially useful in case of asynchronous operations, since CUDA events are aware of the simultaneous executions.

5.3.2 CUDA streams

The concurrent execution of one or more kernels and data copying is done using CUDA Streams. The operations in each stream are invoked in parallel. In the `simpleMultiCopy` sample, the streams are used as shown in listing 5.3.

Listing 5.3: Fragments of the `simpleMultiCopy` sample regarding execution time measurement

```
cudaEvent_t cycleDone[STREAM_COUNT];
cudaStream_t stream[STREAM_COUNT];

...

    for (int i =0; i<STREAM_COUNT; ++i)
    {
        ...

        checkCudaErrors(cudaStreamCreate(&stream[i]));
        checkCudaErrors(cudaEventCreate(&cycleDone[i]));

        cudaEventRecord(cycleDone[i], stream[i]);
    }

    ...

    for (int i=0; i<nreps; ++i)
    {
        int next_stream = (current_stream + 1) % streams_used;

        ...

        // Ensure that processing and copying of the last cycle has finished
```

```
        cudaEventSynchronize(cycleDone[next_stream]);

        // Process current frame
        incKernel<<<grid, block, 0, stream[current_stream]>>>(
            d_data_out[current_stream],
            d_data_in[current_stream],
            N,
            inner_reps);

        // Upload next frame
        checkCudaErrors(cudaMemcpyAsync(
                            d_data_in[next_stream],
                            h_data_in[next_stream],
                            memsize,
                            cudaMemcpyHostToDevice,
                            stream[next_stream]));

        // Download current frame
        checkCudaErrors(cudaMemcpyAsync(
                            h_data_out[current_stream],
                            d_data_out[current_stream],
                            memsize,
                            cudaMemcpyDeviceToHost,
                            stream[current_stream]));

        checkCudaErrors(cudaEventRecord(
                            cycleDone[current_stream],
                            stream[current_stream]));

        current_stream = next_stream;
    }

    ...

    cudaDeviceSynchronize();
```

Each stream is described by a `cudaStream_t` structure. After initializing the streams using `cudaStreamCreate`, we can:

- Invoke a kernel in this stream — `kernelName<<<..., stream>>>`
- Issue a memory copy in this stream — `cudaMemcpyAsync(..., stream)`

In the sample, the streams are synchronized explicitly through CUDA events, using `cudaEventSynchronize` function. After the stream executions, `cudaDeviceSynchronize` is used, which blocks until all issued CUDA calls are complete. It is also possible to wait for all operations in a particular stream, using `cudaStreamSynchronize` function.

5.3.3 Results

Listing 5.4 shows exemplary results of the `simpleMultiCopy` sample, run on Tesla K20M GPU:

Listing 5.4: Fragments of the `simpleMultiCopy` sample regarding execution time measurement

```
[simpleMultiCopy] - Starting...
> Using CUDA device [0]: Tesla K20m
[Tesla K20m] has 13 MP(s) x 192 (Cores/MP) = 2496 (Cores)
> Device name: Tesla K20m
> CUDA Capability 3.5 hardware with 13 multi-processors
> scale_factor = 1.00
> array_size   = 4194304


Relevant properties of this CUDA device
```

```
(X) Can overlap one CPU<>GPU data transfer with GPU kernel execution (device
    property "deviceOverlap")
(X) Can overlap two CPU<>GPU data transfers with GPU kernel execution
    (Compute Capability >= 2.0 AND (Tesla product OR Quadro 4000/5000/6000/
        K5000)

Measured timings (throughput):
 Memcpy host to device  : 2.783200 ms (6.028031 GB/s)
 Memcpy device to host  : 2.523808 ms (6.647580 GB/s)
 Kernel                 : 0.535648 ms (313.213460 GB/s)

Theoretical limits for speedup gained from overlapped data transfers:
No overlap at all (transfer-kernel-transfer): 5.842656 ms
Compute can overlap with one transfer: 5.307008 ms
Compute can overlap with both data transfers: 2.783200 ms

Average measured timings over 10 repetitions:
 Avg. time when execution fully serialized      : 5.806851 ms
 Avg. time when overlapped using 4 streams      : 2.978979 ms
 Avg. speedup gained (serialized - overlapped)  : 2.827872 ms

Measured throughput:
 Fully serialized execution             : 5.778421 GB/s
 Overlapped using 4 streams             : 11.263735 GB/s
```

5.4 The task

During the laboratory, each student should write a CUDA program, solving a problem provided by the lecturer. The program should use overlapping capabilities of the GPU by means of streams. It should be possible to run the program with or without overlapping, measure and compare the execution times on GPU. In this lab student can get up to 5 points:

- showing the GPU capabilities — 1 point
- running and describing results of the `simpleMultiCopy` sample — 1 point
- writing the CUDA program — 2 points
- comparison of overlapping/non–overlapping execution times — 1 point

Chapter 6

Laboratory 6 - running CUDA from Java

6.1 JCuda

JCuda is a library that allows execution of standard CUDA functions and CUDA extensions available in CUDA SDK using JNI (Java Native Interface).

JCuda and its documentation can be found at `http://www.jcuda.de/`. For proper execution of JCuda code all jar files should be visible through `CLASSPATH` both during compilation and execution of the code. Native `.so` libraries should be added to `java.library.path` during execution of the code. It can be done using `-D` switch to `java` command. It is also recommended to download additional JCuda libraries allowing simpler kernel execution. Those libraries are available at `http://www.jcuda.de/utilities/utilities.html`.

The environment should be configured as follows:

1. (only on aplXY) Latest JDK in proper architecture should be downloaded and installed from `http://java.oracle.com`,
2. (only on aplXY) `JAVA_HOME`, `JRE_HOME` and `PATH` variables should be set correctly so proper Java will be used (using bash `export` command),
3. JCuda of the same architecture as Java should be downloaded and installed. JCuda classes should be added to `CLASSPATH` variable and the `.so` libraries to `LD_LIBRARY_PATH`. The can also be passed at runtime as an option to `java` command, i.e.: java -Djava.library.path=/path/to/so.

6.2 Sample Application

A sample application for vector adding is presented on Listing 6.1 and the kernel on Listing 6.2.

Listing 6.1: Listing of JCudaTest.java

```
import jcuda.*;
import jcuda.runtime.*;
import jcuda.driver.*;
import jcuda.utils.KernelLauncher;
```

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.lang.StringBuilder;
import java.io.IOException;
import java.util.Arrays;

public class JCudaTest {

  public static void main(String args[]) {
    int size = 10;
    float[] result = new float[size];
    float[] a = new float[size];
    float[] b = new float[size];
    for (int i = 0; i < size; i++) {
      a[i] = i;
      b[i] = i;
    }

    Pointer dA = new Pointer();
    Pointer dB = new Pointer();
    Pointer dResult = new Pointer();

    JCuda.cudaMalloc(dA, size * Sizeof.FLOAT);
    JCuda.cudaMalloc(dB, size * Sizeof.FLOAT);
    JCuda.cudaMalloc(dResult, size * Sizeof.FLOAT);

    JCuda.cudaMemcpy(dA, Pointer.to(a), size * Sizeof.FLOAT,
            cudaMemcpyKind.cudaMemcpyHostToDevice);
    JCuda.cudaMemcpy(dB, Pointer.to(b), size * Sizeof.FLOAT,
            cudaMemcpyKind.cudaMemcpyHostToDevice);

    KernelLauncher kernel = KernelLauncher.create("add.cu", "add",
             "-arch sm_21");
    dim3 gridDim = new dim3(size, 1, 1);
    dim3 blockDim = new dim3(1, 1, 1);
    kernel.setup(gridDim, blockDim).call(dResult, dA, dB);

    JCuda.cudaMemcpy(Pointer.to(result), dResult, size * Sizeof.FLOAT,
            cudaMemcpyKind.cudaMemcpyDeviceToHost);

    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
    System.out.println(Arrays.toString(result));

    JCuda.cudaFree(dA);
    JCuda.cudaFree(dB);
    JCuda.cudaFree(dResult);
  }
}
```

Listing 6.2: Listing of add.cu

```
extern "C" __global__ void add(float *result, float *a, float *b)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    {
        result[i] = a[i] + b[i];
    }
}
```

When writing the code please note that the C code needs to be preceded by `extern "C"` declaration. The kernel itself can be compiled separately using the command:

```
$ nvcc -cubin -arch sm_21 add.cu -o add.cubin
```

During manual compilation user should explicitly point the version of CUDA for which the code is compiled by using `-arch` switch. Without it JCuda can rise a `CUDA_ERROR_INVALID_SOURCE` error at runtime.

To compile and run the code on apl09 and apl10 servers the following commands should be run:

```
# compilation
javac -cp "lib/*" JCudaTest.java
# execution
java -Djava.library=lib -cp ".:lib/*" JCudaTest
```

When the code is being executed on the desktops in room 527, the commands remain the same but the code needs to be changed to reflect the change in default GCC binary. The line 36 of `JCudaTest.java` presented on Listing 6.1 should be changed to:

```
KernelLauncher kernel = KernelLauncher.create("add.cu", "add
    ", "-arch␣sm_21");
```

6.3 The Task

During the laboratory students should write two kernels solving the problem provided by the lecturer. The kernels should be run from code in Java language. The program should adjust the grd and block size to the size of input provided at runtime. The output of one kernel may not be compatible with input of second kernel. The conversion should be done in Java code.

Bibliography

[1] Adam Jasińsk. Programowanie procesorów gpgpu. Master's thesis, 2009.

[2] W Hwu Kirk, David B Wen-mei. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann, 2010.

[3] Edward Sanders, Jason Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.