Agenda

- Detect platforms and devices and sum up two matrices (40m)
- Presentation: Optimization strategies (40m)
- Optimize matrix transposition (20m)
- Implement vector dot-product (30m)
- Try using NVIDIA SIMD instructions (20m)
- Compute Pi using monte-carlo (20m)
- Matrix multiplication (up to you...)
  - Support big matrices

S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Working with samples

- Go to ~/kseta/tutorials
- Enter directory with tutorial (0_sum first)
- Type cmake .
- Type make

S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Detect OpenCL platforms

- Detecting platforms
  - Find each platform name and version
  - Functions: **clGetPlatformIDs**, **clGetPlatformInfo**
  - Find devices
    - Find each device name, number of compute units, amount of memory, and maximum size of work-group
    - Functions: **clGetDevicesIDs**, **clGetDeviceInfo**

- There should be two platforms. NVIDIA graphic card has 16 compute units, 1535 MB of memory, and supports up to 1024 work items per group

Build sum.cl and print results

- Initialize OpenCL context
  - Functions: **clCreateContext**
- Load application from sum.cl into the C-string
- Build application
  - Functions: **clCreateProgramWithSource**, **clBuildProgram**
- Wait until build is finished
  - Functions: **clGetProgramBuildInfo**
- Print build log
  - Functions: **clGetProgramBuildInfo**
- You should see something like:

```
ptxas info    : Compiling entry function 'add' for 'sm_20'
ptxas info    : Function properties for add
 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 5 registers, 44 bytes cmem[0]
ptxas info    : Compiling entry function 'add_images' for 'sm_20'
ptxas info    : Function properties for add_images
 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 2 registers, 36 bytes cmem[0]
```

Sum 2 matrices on GPU

- Generate two single-precision square matrices (with a side multiple of 16) and fill them random numbers.
- Create a command queue (**clCreateCommandQueue**)
- Allocate memory on GPU and copy data
  - Functions: **clCreateBuffer**, **clEnqueueWriteBuffer**
- Create kernel and set the parameters
  - Functions: **clCreateKernel**, **clSetKernelArg**
- Enqueue kernel, wait for completion, and measure run time
  - Functions: **clEnqueueNDRangeKernel**, **clWaitForEvents, clGetEventProfilingInfo**
- Get results back
  - Functions: **clEnqueueReadBuffer**
- Sum matrices on CPU and measure maximal difference between values computed on CPU and GPU

Access input data using textures

Karlsruhe Institute of Technology

- Replace buffers with images and change allocation and copy functions
  - Functions: **clCreateImage2D**, **clEnqueueWriteImage**
- Write another kernel which is working with images instead of buffers and instantiate it in C-code
  - In the kernel the images have **image2d_t** type
  - To read from image use **read_imagef**
  - Sample may be set to: CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST
- Check if the results are still correct

S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Optimize matrix transposition

- Unoptimized version is in 1_transpose
  - Original performance is about 30 GB/s
- Use local memory to coalescence accesses to global memory
  - Kernel memory is allocated using clSetKernelArg with NULL passed as last argument. In the kernel the pointer to shared memory is declared with __local keyword.
  - The provided skeleton passes *2 * get_local_size(0) * get_local_size(0) * sizeof(float)* bytes of local memory to kernel
- Try to prevent local memory bank conflicts
- Expected performance is about 50 GB/s
  - Verify that stored results (result-transpose.out) have not changed

S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Optimize vector dot product

- Unoptimized version is in 2_dotproduct
  - Just uses a single work-item for computations
  - get_local_size(0) * sizeof(float) bytes of local memory is provided
  - size * sizeof(float) bytes of global memory is provided
  - Original performance is about 0.12 GB/s
- Use multiple work-items while there is enough independent data
  - Remember to coalescence accesses to global memory
- Sum up work-group results in shared memory
  - **barrier(CLK_LOCAL_MEM_FENCE)** is used to synchronize work-items in the group (all local memory writes completed before executing anything beyond this point in the code)
  - Remember about local memory bank conflicts
- Get final results in the global memory
  - **barrier(CLK_GLOBAL_MEM_FENCE)** is used to synchronize work-items in the group (all global memory writes completed before executing anything beyond this point in the code)
- Expected performance is about 100 GB/s
  - Verify printed result

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

NVIDIA Video SIMD instructions

The instruction was introduced in Kepler architecture and code will only work on ipepdvcompute2.ka.fzk.de

- Unoptimized version is in 3_simd
  - The goal is to compute a vector each element of which is absolute difference of two input vectors (uint8_t data type).
  - The provided version processes all bytes individually.
  - The performance is about 40 GB/s
- Modify the source to execute a single work-item per 4 elements.
  - Modify kernel to work with 32 bit integers
  - Verify that results are still correct
  - Modify kernel to use NVIDIA SIMD instruction (**vabsdiff4**)
  - Inline assembler is used for this purpose (gcc syntax)
  - You specify NVIDIA instruction along with considered data types and provided the list of input and output variables
- Verify that stored results (result-diff.out) have not changed
- Expected performance is 80 GB/s

```
asm("vabsdiff4.u32.u32.u32 %0, %1, %2, %0;" : "=r"(out) : "r"(in1), "r"(in2));
```


S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Compute pi with monte-carlo

- Skeleton is in 4_pi
  - There is kernel stub which provides random numbers using random123 library
  - At each iteration you will provide two pairs of random numbers in *a* and *b*
  - *get_local_size(0) * sizeof(long) bytes of local memory is provided*
  - get_num_groups(0) * sizeof(long) bytes of global memory is provided
- Compute monte-carlo hits in local variable (i.e. then points a and b are in inside circle with radius 1). Then, use the same reduction scheme as in vector dot-product. Return total number of hits by all work items in the result. If correct number of hits returned, the approximation of pi will be printed.
- Expected performance is about 6 giga-tries/s

S. Chilingaryan et. all

Institute for Data Processing and Electronics
Karlsruhe Institute of Technology

Matrix multiplication

- Stub is in 5_matrix
  - Intel MKL (CPU) version gives about 70 Gflop/s on ipepdvcompute1
  - clAMDBlas (GPU) produces about 280 Gflop/s
- Optimize
  - Consider square matrices with side multiple of whatever you like
  - Basic implementation
  - Use local memory
  - Process multiple points per work-item
  - Use pinned memory
  - Use queues to add parallelism (multiple matrices)
  - Try using textures to enhance cache hits
  - Support arbitrary matrix sizes
  - Support big matrices
  - Compute big matrices using multiple GPUs
    - Run application with environmental variable CUDA_VISIBLE_DEVICES set to “1,2,3,4,5,6,7,8” to have more GPUs
- The simple version will produce about 70 Gflop/s, the optimized (single GPU) may go as far as 700 Gflop/s excluding transfers. Interleaving transfers and computations may give about 500 Gflop/s (multiple matrix case).