Multiprocessor programming 521288S

Teemu Nyländen
University of Oulu
Department of Computer Science and Engineering
January 20, 2016

Mobile communication devices are becoming attractive platforms for multimedia applications as their display and imaging capabilities are improving together with the computational resources. Many of the devices have increasingly been equipped with several sensors i.e. motion sensors, environmental sensors and position sensors that allow the users to capture several signals at different rates.
To process these signals, with power and energy efficiency, reduced space in mind, most mobile device manufacturers integrate several chips and subsystems on a System on Chip (SoC). Figure 1 illustrates organization of a Snapdragon 805 system-on-chip (SoC) featured i.e. on Nexus 6 mobile phones. However, the same basic structure can be found on any high-end mobile phone or tablet. For long, it was a tedious task to develop programs that could exploit all available hardware. Ever since Open Computing Language (OpenCL) was introduced, the development work was simplified significantly. Although, the code still needs to be optimized using "brute force ", the program development itself has changed quite dramatically. The same code can be run and tested on multiple devices. In addition, the code can even be partioned to run on multiple devices simultaneously, providing true heterogeneous computing capabilities with a single programming framework.

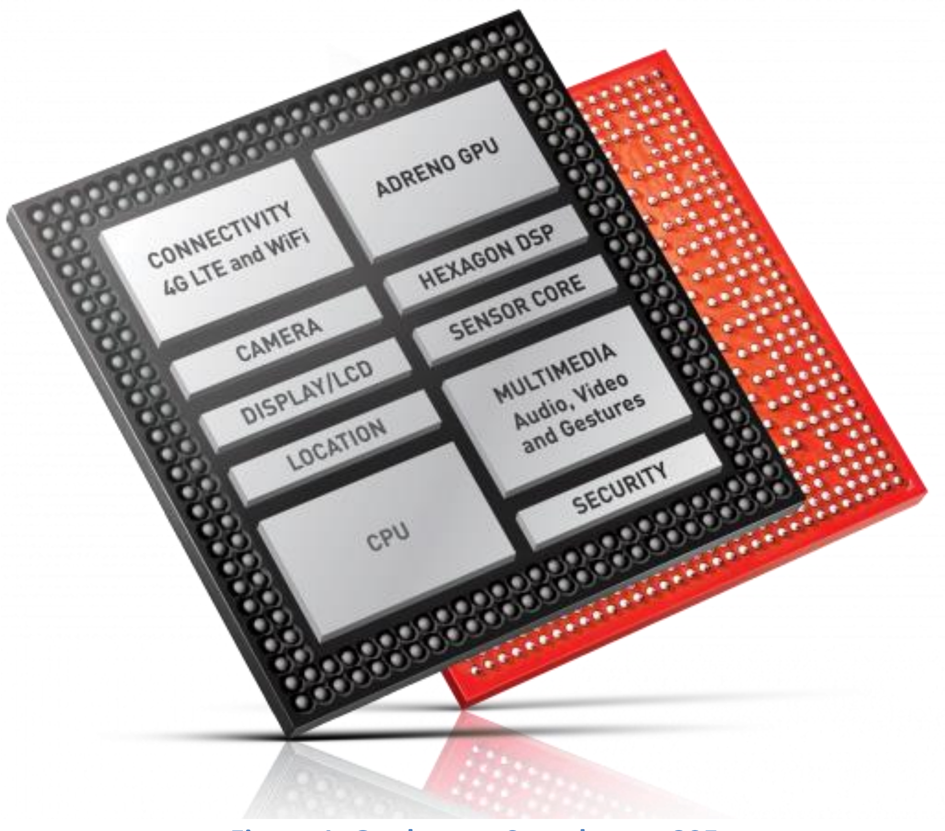


Figure 1: Qualcomm Snapdragon 805
https://www.qualcomm.com/products/snapdragon/processors/805

In this project, we will concentrate on a simple heterogeneous computing setup using only CPU and GPU. The implementation will be benchmarked on both devices individually as CPU and GPU only set-ups well as with a heterogeneous setup using both devices within the same context. The CPU works as the "host"-processor in all cases according to the OpenCL platform model. Based on the benchmarked performance of each setup, you will make the final optimization choice on which parts of the algorithms to offload to be executed on the GPU.

GETTING STARTED

STEP 1. Start your work by setting up the development environment. If you are using your own computer, you will need to first install a C/C++ compiler and if you want an integrated development environment (IDE) such as Eclipse or Visual Studio. In TS135 workstation class there are four Windows computers equipped with AMD GPUs and Visual Studio 2012. If you are doing the work in TS135 you can proceed to the next page.

For Windows users Visual Studio (you can get it with your O365 credentials).

For Linux users GCC/G++ compilers are enough, but if you want to use an IDE you can install i.e. Eclipse. GDB might also be useful for debugging your code.

For Mac users check https://developer.apple.com/opencl/

Note that you can use a virtual Linux machine running on your Windows/Mac machine, but there is no quarantee that you will be able to use the GPU, but CPU becnhmarking should work.

STEP 2. After you have the compiler and IDE installed, you will also have to install the OpenCL software development kit (SDK) or SDKs depending on your computer set-up. The SDK includes OpenCL headers, installable client driver (ICD) loader amongst other things needed to compile and run OpenCL programs. A short listing on SDKs for different set ups can be found below. If you are not sure which SDK you should install or if you encounter problems in Step 1. or 2. contact the assistant.

The workstation in TS135 will have also the AMD SDK pre-installed.

For a CPU only set-up:

AMD CPU : AMD SDK
Intel CPU: AMD or Intel SDK

For a CPU+GPU set-up (check that your GPU has OpenCL support from the vendor list)

Intel Integrated Graphics: Intel SDK
AMD GPU: AMD SDK
NVIDIA GPU: NVIDIA CUDA SDK + SDK (AMD or Intel) for the CPU. NVIDIA SDK does not provide support for running code on CPUs.

One should be able to use any SDK for all GPUs, but a vendor specific ICD and drivers are always required. The safe bet is to go with the processor vendors own SDK. The SDKs come with OpenCL samples. So you can check that your system is set up correctly by running a few of these samples.

THE ASSIGNMENT

Your task is to implement and accelerate a stereo disparity algorithm in OpenCL by offloading all or parts of the computation to the GPU. The same OpenCL implementation should be benchmarked with a CPU-only setup, a GPU-only setup and a heterogeneous setup using both CPU and GPU. You should decide which parts of the algorithm and post-processing to implement on GPU and CPU, based on the execution times of CPU-only and GPU-only set-ups. Taking into account both the data transfer and execution times of your implementation, report your final implementation choice in the final report.

The algorithm to be implemented is zero-mean normalized cross correlation (ZNCC) based depth estimation.

$$ZNCC(x,y) \triangleq \frac{\sum_{j=0}^{B-1}\sum_{i=0}^{B-1}\{[I_L(x+i,y+j) - \overline{I_L})] * [I_R(x+i-d,y+j) - \overline{I_R}]\}}{\sqrt{\sum_{j=0}^{B-1}\sum_{i=0}^{B-1}(I_L(x+i,y+j) - \overline{I_L})^2} * \sqrt{\sum_{j=0}^{B-1}\sum_{i=0}^{B-1}(I_R(x+i-d,y+j) - \overline{I_R})^2}}$$

B = block size, I_L, I_R= left and right images, $\overline{I_L}, \overline{I_R}$ = mean value of the block NOTE! You need to take the disparity into account, when calculating the block mean , d = disparity value (0 to d)

NOTE that there is no disparity in y-axis, since the images are already *rectified*.

The final disparity image / depth map is obtained using winner take all (WTA) approach

$Depth\ map_{(x,y)}(d) \triangleq \text{argmax}(ZNCC_{(x,y)}(d))$

There is now ideal block size. The block size needs to be large enough to ensure proper matching, but small enough to avoid effects of perspective distortion. Start with 9x9 blocks. The blocks don't have to be squares.

The images used in this laboratory work are from the Middlebure Stereo Vision datasets [1][2] at http://vision.middlebury.edu/stereo/data/scenes2014/datasets/Backpack-perfect/. Download images im0.png and im1.png. If you want you can download more image sets from the webpage and try your solution on more demanding sets. Notice that the disparity values are not constant but are determined per image set. The maximum disparity value for the two images included in the package can be found on the calib.txt file and is 260 (ndisp) for the above mentioned two images. Notice that you need to scale this value when you downscale the image sizes.

Start with the C-implementation. Use LodePNG to read the images. You will only need to include lodepng.c- or lodepng.cpp-source file and the lodepng.h-header file to your project and use the appropriate functions provided in these files to read and write the image files.

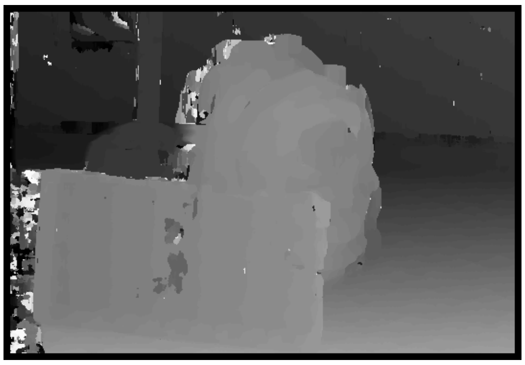
To help you get started with the algorithm implementation you can find one possible pseudo-code on the final page of the instructions. Notice, however, that this is not the only solution. You are free to experiment and use other approaches. Once you get the C-code working, you can start parallelizing the code and programming the OpenCL kernel.

Exercise instructions in brief:

Since the execution time of your implementation has an effect on your grade, implement the code in a way that the parameters, such as block size, are easily changed.

1st phase (C-implementation):

- Read/decode the images using LodePNG. (You need to download and add the appropriate .c or .cpp files and the header file to you project). Use the `lodepng_decode32_file`-function to decode the images into 32-bit RGBA raw images and `lodepng_encode_file`-function to encode the output image(s).
- Resize the images to $\frac{1}{16}$ of the original size (From 2940x2016 to 735x504). For this work it is enough to simply take pixels from every fourth row and column. You are free to use more advanced approach if you want to.
- Transform the images to grey scale images (8-bits per pixel). E.g. $Y=0.2126R+0.7152G+0.0722B$.
- Implement the ZNCC algorithm, start with 9x9 block size. Since the image size has been downscaled, you also need to scale the dmin-value. Use the resized grayscale images as input. You are free to choose how your process the image borders.
- Compute the disparity images for both input images in order to get two disparity maps for post-processing. After post-processing one final disparity map sufficient in this work.
- Check that your outputs resembles the one presented below (Output the result image to depthmap.png Notice that you need to normalize the pixel values from 0-dmin to 0-255. However, once you have checked that the image looks ok remove the normalization and implement it after the post-processing.)



- Implement the post-processing including cross-check and occlusion filling. (Instructions for post-processing after the pseudo code on the last page)
- Check again that the disparity map looks right
- Measure the total execution time of your implementation including the scaling and grey scaling using QueryPerformanceCounter-function in Windows and gettimeofday-function in Linux

1st CHECKPOINT!

2nd phase (OpenCL implementation):

- Read the original images (im0.png and im1.png) Use the ready C-implementation as the basis.
    - You are free to use existing host-codes from SDK-example projects or from the internet freely. Just cite where your implementation is from. NOTE that you still need to understand what you are using
        - E.g. there is a vast amount of examples in AMDAPPSDK-folder in the workstations in TS135
    - Find out at least the following parameters of your GPU using the clDeviceInfo-functions
        - `CL_DEVICE_LOCAL_MEM_TYPE`
        - `CL_DEVICE_LOCAL_MEM_SIZE`
        - `CL_DEVICE_MAX_COMPUTE_UNITS`
        - `CL_DEVICE_MAX_CLOCK_FREQUENCY`
        - `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE`
        - `CL_DEVICE_MAX_WORK_GROUP_SIZE`
        - `CL_DEVICE_MAX_WORK_ITEM_SIZES`
- Read the images (original png-files) into the device as image-objects.
- Implement a kernel or kernels for resizing and grey scaling the images
- After resizing and grey scaling, using buffer-objects can be more convenient. NOTE! You only need to read the buffers/images back to the host, if you plan to use them on the host or another device, otherwise keep them on the device in order to avoid unnecessary data transfers.
- Implement the actual ZNCC algorithm. Once the algorithm works proceed to implementing the post processing. You are free to divide the work into multiple kernels. Whatever in your opinion is the best approach, but justify the chosen approach in the final report.
    - Notice that transferring data between host and device is costly, so minimize data transfers
- Report the execution times of each kernel using *clGetEventProfilingInfo* and the total execution time using the appropriate C-functions on the host-side.
- Make a copy of the implementation before proceeding to the optimization phase! Use this implementation as the basis for optimization for both desktop and mobile implementations. The same approaches will probably not work for both.

3rd phase (OpenCL optimization desktop):

- Optimize your code. All vendors have their own optimization guides with good examples.
    - Vectorization
    - Full memory architecture utilization (local memory)
    - Memory coalescing
    - Reducing the register usage
    - …
- Report the execution times of the optimized code

4th phase (OpenCL mobile implementation and optimization):

- Mobile implementation on Odroid-XU4
    - The Odroid has full Kubuntu OS
    - Compile and execute your codes on Odroid (Script provided on the desktop)

- Benchmark the original implementation and report the results
- Optimize the code based on device characteristics and report the results
- Architecture differs quite significantly from the desktop AMD GPU, so same optimization strategies do not necessarily apply

PSEUDO CODE:

```
FOR J = 0 to height
    FOR I = 0 to width
        FOR d = 0 to MAX_DISP
            FOR WIN_Y = 0 to WIN_SIZE
                FOR WIN_X = 0 to WIN_SIZE
                    CALCULATE THE MEAN VALUE FOR EACH WINDOW
                END FOR
            END FOR

            FOR WIN_Y = 0 to WIN_SIZE
                FOR WIN_X = 0 to WIN_SIZE
                    CALCULATE THE ACTUAL ZNCC VALUE FOR EACH WINDOW
                END FOR
            END FOR

            IF WINDOW SUM > CURRENT MAXIMUM SUM THEN
                UPDATE CURRENT MAXIMUM SUM
                UPDATE BEST DISPARITY VALUE
            END IF
        END FOR

    DISPARITY_IMAGE_PIXEL = BEST DISPARITY VALUE
    END_FOR
END FOR
```

Post-processing:

Post-processing consists of two phases: cross-check and oclusion filling.

CROSS CHECK: (Two input images, one output image)
In this phase you simply check that the corresponding pixels in the left and right disparity images are consistent by comparing their absolute difference to a threshold value. Start with threshold value of 8 and by experimenting find the best threshold value for your implementation. If the absolute difference is larger than the threshold then replace the pixel value with zero. Otherwise the pixel value remains unchanged.

OCLUSION FILLING: (One input image, one output image)
In the simplest form: Replace each pixel with zero value with the nearest non-zero pixel value.

You are free to experiment with more complex post-processing approaches.

References:

[1]
D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms.
International Journal of Computer Vision, 47(1/2/3):7-42, April-June 2002.
Microsoft Research Technical Report MSR-TR-2001-81, November 2001.

[2]
D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms.
In Workshop on Stereo and Multi-Baseline Vision (in conjunction with IEEE CVPR 2001), pages 131-140, Kauai, Hawaii, December 2001.