Abstract — article spotlights process of porting big video processing application to the OpenCL and challenges which team has faced during this process. Implemented tricks and techniques which served for reaching better performance are described.

1. Introduction

Specifics of the computer vision tasks imply the existence of the computationally expensive operations. All of these tasks involve interactions with images that basically are matrix operations.

So, control flow contains huge amount of consecutive operations, where result of the previous one doesn’t affect result of the next one. It means that such kind of applications contain a huge amount of the parallelism. Hence, execute it in a serial manner would be ineffective and slow. Our goal was to take advantage of the natural parallelism of the task and implement the parallel version of the system.

Original system is a library which able to find scene boundary cuts on the given video. This library could be used in the wide range of the domains like indexing of video content or video editing.

Hence, we had a goal to pick parallel programming technology which is suitable for servers and desktops. Most of them are highly specialized. For example, MPI oriented for execution on clusters and OpenMP devoted for creating parallel code for SMP systems. Finally, OpenCL was chosen. This decision is supported with following reasons:

- General purpose GPU is the approach acceptable for desktops and servers both
- OpenCL is oriented to solving mathematical tasks
- OpenCL technology supported with all vendors (nVidia, ATI, Intel, etc)

2. Architecture

Architecture of the system was slightly reconsidered and rewritten in purpose to develop clear and supportable parallel code. Writing of parallel code with use of any technology has lots of pitfalls and dramatically slower than writing serial code. Description of the scene boundary detection system which we are optimizing could be found in [1]. So, rewriting of the architecture allowed to deliver following features:

First feature is that OpenCL and C++ versions are interchangeable. They represented as classes PyraProbe and OCLPyraProbe. These classes encapsulate all business-logic related to pyramidal analysis of the frame (all computationally expensive work). Figure 1 illustrates that at the highest level of application call stack code operates with IBasicPyraProbe objects without even knowing which implementation it is.

![Figure 1. UML Class diagram of the parallel system](image)

Also, both versions are kept in the same project in purpose of easier building and execution in both ways. The library is being built with or without OpenCL support based on preprocessor definitions, and being executed serial or parallel way based on given settings.

Another architecture feature is namespaces Processors and OCLProcessors which shown on Figure 1. Both of them contain image processing functions which are exploited by PyraProbe and OCLPyraProbe respectively. Some examples of these functions are listed for better insight of the idea:

- generation of the 2D Gaussian distribution
- 2D convolution
- normalizing color image
- creating image pyramid

This design allowed to develop parallel version iteratively and enabled unit-testing of these functions. Class Validator serves for this functionality. So developers were able to prove that some function ported successfully by comparison of the results of C++ and OpenCL version of the function. Besides visual comparison, perceptualDiff tool was used.

OCLManager serves for handling all SDK-related calls. It builds and stores kernels, keeps OpenCL context and other SDK objects required for execution.

Other aspects of the architecture:

- Minimized interaction between RAM and videomemory. Copying of image...
occurs only twice: delivering source image to OCLPyraProbe and then acquiring result back to RAM. All computation in between is being performed on GPU.

- Modularity of the system enables easy code adaptation for execution on the GPU farm.

3. Optimization

When development of the system with architecture described above was finished, performance tests have been performed. These tests revealed slight improvement at the high-resolution video, but parallel version didn’t outperform serial one on the low-resolution videos.

Profiling results of the parallel version before optimization are shown on the Table 1.

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Total Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter2D</td>
<td>2701.98992</td>
</tr>
<tr>
<td>buildPyramidGaussSmooth</td>
<td>725.59914</td>
</tr>
<tr>
<td>upStart</td>
<td>432.86879</td>
</tr>
<tr>
<td>IterativeInteraction</td>
<td>216.81900</td>
</tr>
<tr>
<td>buildPyramidlevel0</td>
<td>112.02099</td>
</tr>
<tr>
<td>naiveSummationMethodGetDenom</td>
<td>82.59578</td>
</tr>
<tr>
<td>buildPyramidReducingCopy</td>
<td>56.41125</td>
</tr>
</tbody>
</table>

This table summarizes most time-demanding kernels of the application. As it is obvious from table, most of the application running time was taken by convolution calculation. Filtering is such computationally expensive because model uses big kernels (25x25, 21x21). So, filter2D is the most promising target for optimizations. Following sections describe techniques which made OpenCL code run faster.

3.1 Divergence of the code

One of the properties of the efficient GPGPU code is low divergence. This means that all threads of kernel have approximately same execution time. If control flow of the kernel has a lot of conditional statements, most likely that code will be slow.

Removing of the conditionals gave significant speed gain for our case.

3.2 Advanced memory usage

OpenCL global memory has huge capacity but the slowest access speed. So, the only way to implement something efficient is to put an effort on extracting advantage of the local and constant memory zones. For the better insight of tricks described in this subsection, OpenCL memory model is shown at Figure 2. More information about OpenCL memory model could be found at [2] or [3].

There is possibility to define memory buffers as __constant if buffers are read only. Also size of the constant buffer is bounded by value which depends on computing device. So, first thing to do was to move all filter buffers from global memory to constant. Another way to speed up convolution operation is to avoid reading neighbor pixel values from the global memory [4]. So, filter2D kernel execution model was changed to use workgroups of size 16x16. Items of the same workgroup load part of the image to the local memory and then use this data for convolution calculation.

It is good practices to keep amount of kernel arguments as small as possible. As it obvious from Table 2, filter2D was separated to two kernels. This action enables us to not pass size of kernel (and some other variables like that) as argument, but define it as a constant. These values are used in the loop code; consequently, compiler will be able to implicitly unroll loops [5].

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Total Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicFilter2Dimg</td>
<td>9861.60287</td>
</tr>
<tr>
<td>filter2DGabor</td>
<td>1382.35945</td>
</tr>
<tr>
<td>IterativeInteraction</td>
<td>662.88315</td>
</tr>
<tr>
<td>buildPyramidGaussSmooth</td>
<td>396.98843</td>
</tr>
<tr>
<td>upStart</td>
<td>242.95238</td>
</tr>
<tr>
<td>buildPyramidlevel0</td>
<td>65.73823</td>
</tr>
<tr>
<td>naiveSummationMethodGetDenom</td>
<td>46.39889</td>
</tr>
<tr>
<td>downStart</td>
<td>28.78435</td>
</tr>
</tbody>
</table>

4. Experiments

Performance tests and profiling were repeated after implementation of the optimized version. Kernel summary is represented in the Table 2. Timings for process one frame are spotlighted in the Table 3. Our optimization work is not finished yet, but even now it is obvious that GPGPU solution achieved performance on this problem of image analysis which is way more effective than serial CPU solution. Tests which are represented in Table 3 ran at desktop with Intel i5-430m processor and AMD Radeon HD 5850M.
5. Conclusion

Article illustrates set of techniques which was put on the most often used kernel of the application and highlights achieved speedup. But optimization work is not over yet. Other slow kernels will be improved in similar way. So, plans for future development are:

- Continuation of the OpenCL code optimization
- Work on the computational model for increasing precision and decreasing computational complexity

6. References