Exploration of OpenCL for FPGAs using SDAccel and Comparison to GPUs and Multicore CPUs

Lester Kalms, Diana Göhringer
Technische Universität Dresden
Dresden, Germany
lester.kalms@tu-dresden.de, diana.goehringer@tu-dresden.de

Abstract— **Due to energy efficiency, heterogeneous computing is gaining more and more attention. Since FPGA implementations are time consuming, high-level synthesis (HLS) is used to close the productivity gap. OpenCL has become accepted as a good programming model for HLS, due to its portability, good capability of design verification and rich instruction set. This work implements different optimization strategies using OpenCL for a heterogeneous system containing CPU, integrated GPU, GPU and FPGA. Energy efficiency and performance of the architectures are compared using a feature detection algorithm. It is shown how to maximize performance while hitting the maximum memory bandwidth and keeping the resource utilization low for the SDAccel tool from Xilinx. The evaluation shows the great streaming capability of OpenCL for FPGAs. The FPGA achieves a speed up of 62.8 and consumes 49 times less energy for the application in comparison to an optimized single threaded CPU implementation in full HD.**

Keywords—OpenCL, SDAccel, GPU, CPU, FPGA, Energy Efficiency, Performance, Image Processing, Accelerators

I. INTRODUCTION

Due to the decreasing transistor size and power wall, there has been a significant growth towards parallel computing and heterogeneous systems. A variety of parallel architectures is used in these systems to increase computational throughput, while running at a slower clock speed, to increase energy efficiency. Using OpenCL, applications can be implemented for different architectures, like CPUs, GPUs, and FPGAs. Classically, FPGAs are programmed using Hardware Description Languages like VHDL or Verilog, which are time consuming and unfamiliar to most application developers. To ease programmability, vendors like Xilinx and Altera (now Intel) have introduced High-Level Synthesis (HLS) tools. Advantages are the easier and faster way of testing functional correctness, the portability of code and the shortened design cycles. Vivado HLS [1] from Xilinx uses pragmas in HLS code to improve hardware implementations. Based on this tool, Xilinx provides SDSoC and SDAccel, which further abstract the underlying hardware, by including Direct Memory Access (DMA), interconnections and hardware buffers. SDSoC [2] has been developed for heterogeneous embedded systems, which consist of an ARM and an FPGA and SDAccel [3] for x86 systems, to execute functions like on any other accelerator. Altera's OpenCL SDK also provides an environment that abstracts the underlying hardware details [4]. There are also university research HLS tools, like ROCCC [5] and LegUp [6]. This work compares different optimization strategies for OpenCL devices (CPU, integrated GPU (iGPU), GPU and FPGA). The AKAZE feature detection algorithm [15] has been chosen as use case, since it shows better repeatability than other state of the art algorithms like SURF [17]. It is very suitable as a benchmark for streaming applications, since it consists of various computational-intensive and streaming-capable functions. An exploration and evaluation of the computational throughput, resource utilization and memory bandwidth consumption has been done, to help developers to determine the performance of an application at design time. Streaming between functions is realized by using OpenCL pipes and parallel execution of functions by using OpenCL's out-of-order execution. In the following, Section II provides related work, Section III describes implementation and optimization strategies, Section IV compares achieved results and Section V contains conclusion and outlook.

II. RELATED WORK

OpenACC [7] is a pragma based alternative to OpenCL for accelerators with similar concepts as OpenMP. OpenARC [8] extends this for FPGAs, by source-to-source translating to OpenCL code and therefore reducing overhead. Whereas OpenCL has a rich instruction set, support of Task and Data-Level Parallelism and a high availability of different devices. In [9], Hill et al. compare Altera's VHDL and OpenCL design flow for image processing and show the good portability of OpenCL for FPGAs and a significantly reduced design time at cost of increased resource usage. In [10], Wang et al. present a performance analysis framework for OpenCL applications on Altera FPGAs, which predicts performance with different optimization combinations and identifies bottlenecks. Both proposals do not compare to other architectures, like GPUs. In [11], Ayat et al. explore Altera's OpenCL comparing a Sobel filter to GPU/CPU devices for different image and kernel sizes. They show that kernel size does not increase computation time for FPGAs as much as for GPUs/CPUs, but without energy measurements. In [12], Rethinagiri et al. show the potential and energy efficiency of platforms containing CPU, GPU and FPGA for high-performance and embedded systems using optimized applications in C/C++, CUDA and VHDL. Their OpenCL implementation for all platforms only achieves 1/8 performance. In [13], Vasiljevic et al present a pre-optimized stream memory component library for FPGAs using SDAccel showing benefits of streaming in OpenCL. Two applications are used to compare a naive with an optimized implementation, without comparing to other architectures.

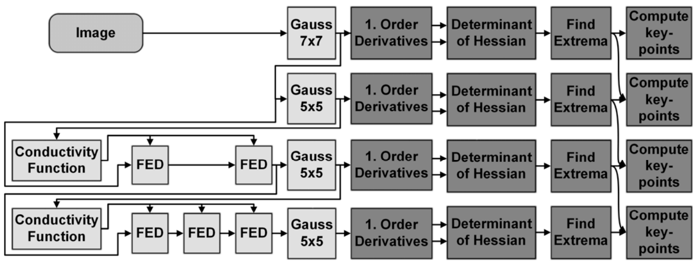

Figure 1: Task graph of the AKAZE feature detection implementation.

III. Implementation

A. System and Algorithm Overview

In OpenCL, host code initializes its context, moves data between host and devices and executes kernels (device functions). Work-items are kernel instances which execute concurrently over a grid and are grouped into work-groups. Device global memory is visible to host and device (e.g. GPU), which consists of compute units (e.g. GPU core). Local memory is visible to its work-group that is executed on a compute unit. We use SDAccel 2016.1, to pre-compile kernels and a Tcl script to create the hardware system, link all vendor libraries and execute all devices of the system (Table 1) in 1 application. AKAZE detects features by building a scale-space using nonlinear diffusion. The implemented design (Figure 1) consists of 2 parts: non-linear scale-space creation (light), feature detection (dark). Each FED step computes 5 multiply and 12 add/subtract operations and gets 5 pixels (3x3 window) of each input image. A conductivity function, computes derivatives using the Scharr operator getting 8 input pixels (3x3 window) and then computes 1 division, 2 multiply and 2 add operations. A Gaussian kernel of size 5x5 or 7x7 smooths the image at each level. The determinant of the Hessian (DoH) blob-detector needs 2 multiply, 1 subtract operation and first and second order derivatives (Scharr) at different scales (σ) (here 2, 3, 3 and 4). Key-points are extracted by comparing pixels in DoH images in a 3x3 window (Find Extrema). If a value is maximum, it is compared in σ radius to key-points in same, upper and lower level (Compute Keypoints). We dropped upper level comparison, since number of key-points differs by maximum 1%, and reduced complexity of the Compute Keypoints function from $O(n^2)$ to $O(n \cdot log(n))$ using adaptive searching pointers, since key-points are sorted.

B. FPGA Bandwidth and Kernel Optimization

One approach for FPGA kernel design is to determine the maximum achievable bandwidth, by removing all code not related to memory access, and then increase computation speed in relation to the maximum bandwidth. The following, chronologically describes different optimizations for windowed functions. The baseline OpenCL implementation is a partially optimized function executed by 1 work-item with loops in x and y direction. First loop pipelining is applied to the inner loop and the complete code is moved into this loop. Data is pre-fetched from global to local memory (BRAM) in a burst operation using OpenCL's *async_work_group_copy* function, which can execute memory operations concurrently (asynchronous). Line buffers are used to reduce multiple global memory accesses to same data. If each line buffer row is stored in an own BRAM, a 7x7 kernel needs 4 clock cycles to read data, since each BRAM only has two data access ports. To reduce clock cycles to 1, registers are used for the window and combined with line buffers in a sliding window. Computation time now hardly depends on the window size and only a small overhead is needed to fill it with data. For parallelism, separate kernels are used for reading, writing and computation. These kernels are connected via FIFOs (OpenCL pipes) and executed concurrently (OpenCL out-of-order execution). To utilize the maximum available bandwidth, OpenCL vector operations are used for memory access (e.g. 16x32-bit vector for 512-bit interface) and computation. While GPUs are very good at floating point calculations, FPGAs have their strengths using fixed-point numbers, due to reduced resource utilization. We normalize all filters and reduce 32-bit floating point data to 16-bit fixed-point data, which also reduces the utilized memory bandwidth. To reduce the accuracy loss of the fixed-point numbers, also higher bit widths are used within the functions. Since the FPGAs 512-bit memory port is not fully utilized using OpenCL's maximum vector size of 16 and 16-bit fix point numbers, memory is accessed with a 32-bit data type containing 2 16-bit fixed-point numbers (packing). Data only needed by the next kernel, is streamed between them, to remove global memory access and reduce latency. This streaming capability also compensates the lower global memory bandwidth compared to the GPU of the test system.

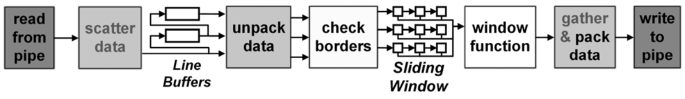

Figure 2: Windowed (3x3) function implementation on an FPGA.

Figure 1 shows the implemented algorithm. The input image is an 8-bit gray-scale image and each Find Extrema function gives a key-point vector as output. The chosen vector size is 4x16-bit for computations and 512-bit for memory access. All interim results are streamed between functions, which are combined, since SDAccel allows a maximum of 10 compute units. There is 1 unit for global memory access, 5 for nonlinear scale-space and 4 for detection. The Find Extrema function outputs maximum 2 key-points in 1 clock cycle, since 2 neighboring pixels can't be an extremum. The final implementation for a windowed function is shown in Figure 2. First data is read vector by vector from pipes. If a pipe is connected to a DMA, the input is a 16x32-bit vector and scattered into 8 2x32-bit vectors, which are written one after the other into line buffers, where data remains packed to reduce BRAM usage. Then data is unpacked from a 2x32-bit to an 4x16-bit vector. The example implementation duplicates border pixels if the window is outside of image boundaries. The sliding window moves 1 vector in each clock cycle from one register to the other and provides parallel access to the complete window. If results are written to DDR memory, 8 4x16-bit vectors must be gathered and packed to a 16x32-bit vector, otherwise data is packed to a 2x32-bit vector.

C. CPU, iGPU & GPU Kernel Optimization

There are 4 different implementations for windowed functions for a fair comparison to the FPGA. The first one (**OPT0**) is device independent, leaves work-group partitioning to the compiler and computes each output pixel in a single work-

Table 1: Test system components

Device	Vendor	Model	Fab (nm)	Bandwidth (GB/s)
FPGA	Xilinx	Virtex-7 XC7VX690T	28	10.67
GPU	NVIDIA	GTX 780	28	288
CPU	Intel	Core-i7 4770k	22	25,6
iGPU	Intel	HD Graphics 4600	22	25,6

Table 2: Resource utilization for 1 read and 1 write DMA kernel and the achieved memory bandwidth.

	32 bit	64 bit	128 bit	256 bit	512 bit
FF	2095	2205	2357	3141	4713
LUT	2537	3128	3122	3700	4850
BRAM	12	16	24	48	92
Bandwidth (MB/s)	740	1450	3170	6290	8899

Table 3: Computation time in ms for different memory port widths of a 5x5 Gaussian kernel.

	No compute kernel	Gaussian vector 8	Gaussian vector 4
128 bit	0.829	1.686	1.688
256 bit	1.107	1.101	1.175
512 bit	0.829	0.836	1.171
Estimated		0.581	1.159

item. Different optimizations are applied (e.g. compiler optimizations), the amount of kernel computations is reduced and branches are replaced by built-in functions. Qualifiers like *restrict*, *const*, *read_only* and *write_only* are used to optimize memory access. The **OPT1** approach is related to Intel's optimization guide [14]. Every work-item processes an 8x8 grid of pixels, using a vectorization of 8 and processing 8 rows in a sliding window approach, which achieved the best results. The work-group size is left to the compiler. **OPT2** loads data into local memory, to reduce global memory access. Data is read into local memory of size $m\ x\ m = (n + 2 \cdot r)x(n + 2 \cdot r)$, which depends on the work-group size n x n and the radius r of the input window. The input is split into 4 areas, which are read by the work-items in 4 separate operations (if $(2 \cdot n \geq m)$). **OPT3** is like *OPT1* without vectorization. It loads the input window into registers, which are used as sliding window. Since input data is marked as read only, constant and restrict, the read only cache of the device can be used. Due to the SIMD structure in GPUs, the cache creates a line buffer and each input pixel of a work-group is only loaded once. In comparisons to *OPT2*, no extra commands are needed to prefetch data into local memory. The implementation of the **Find Extrema** function contains 3 stages. The 1. stage writes detected key-points of each row in an own vector using a global counter for each row, since key-points only need to be sorted by their y coordinate. The 2. stage computes the parallel prefix sum to sum the resulting row counters, to determine the position of a key-point in the result vector. It is computed for each work-group (3 – 4 groups) independently. The 3. stage gets the results of all parallel prefix sum and writes the key-points to the correct position in memory. The CPU version computes the prefix sum in simple loops.

IV. Results

Table 2 shows estimated resource utilization and the memory bandwidth of a data copy between two locations in global memory using different port widths, *async_work_group_copy* and separate read/write kernels. Compared to the theoretical bandwidth (Table 1), a high consumption can be achieved (resolution 2048x2048 and 32-bit width). Table 3 shows the computation time of a Gaussian convolution filter for different port widths and vectorization degrees (720p and 16-bit width). The estimated computation time of the kernel (eq. 1) and a memory copy serve as roofline (height (H), width (W), kernel radius (r), vectorization degree (v), pipeline stages (x)).

$$\frac{(H + r) \cdot \left\lceil \frac{W + r}{v} \right\rceil + x}{frequency} \quad (1)$$

The measured computation approaches the roofline as expected, which makes computation time of functions predictable. The resource utilization increases with the bit width, since packing/unpacking of data is done within the compute kernel and not the DMAs, to maximize memory throughput. Executing kernels from host requires about 15 µs for kernel with no functionality and 1 kernel parameter and increases by about 15% for each further parameter. Executing more than one function within one loop reduces the resource utilization, but increases the difficulty to meet timing requirements. SDAccel also provides the Vivado HLS tool flow, which allows executing several functions in parallel within one kernel by activating loop level parallelism using the HLS DATAFLOW pragma. Figure 3 shows the computation time for different devices, calculated by taking the average time of 3072 executions. The resolution is 720p and every function is labeled with its best optimization strategy. The effect of an increasing window size is lower in comparison to the single threaded CPU implementation. For the CPU, OPT1 achieves best performance for larger kernels. For a small kernels (FED) or a sparse input (DoH, Scharr), the baseline method, which benefits from auto-vectorization and compiler optimizations, is best. For GPU and iGPU, OPT3 is best, since caching data into read only memory saves computation in comparison to OPT2, where data is loaded explicitly into local memory. OPT0 achieves a bandwidth of 409 GB/s on the GPU for a 7x7 kernel, which proves that data is cached, since bandwidth is higher than the theoretical one (Table 1). The sliding window in OPT3/OPT1 additionally reduces memory access. The Find Extrema function achieves a speed-up of 3.88, 1.76, and 13.32 on CPU, iGPU and GPU in comparison to the single threaded method.

Table 4 shows the final results for a 720p (1080p) resolution. The Compute Keypoints functions remains on host and needs 130 (255) µs using OpenMP. They can be processed on the CPU in parallel to GPU/FPGA. The avg. time to send an image to the device and read back the ~4464 key-points is 430 (770) µs for the GPU/FPGA. The FPGA achieves the highest speed-up and is 1.46 times faster than the GPU, due to efficient use of memory bandwidth enabled by streaming. The entire system power is measured, since host and components like DDR affect the efficiency. Energy is measured by processing the algorithm for 10 minutes and taking the average power consumption, subtracting idle from total consumption. The iGPU is only 1.17 times faster than the CPU, but 2.68 times more energy efficient, due to its parallel structure and lower frequency. The speed-up of the FPGA increases with increasing resolution, showing a good scalability. The estimated computation time without memory access can be determined by the time it takes to start all 10 kernels (165.5 µs) and the time of the last kernel. The total estimated time for a 200

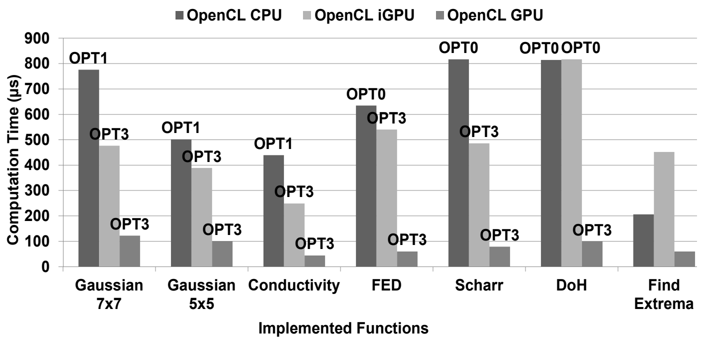

Figure 3: Computation time of the different functions implemented in OpenCL for CPU, iGPU and GPU for an image resolution of 1280x720.

MHz is 1343 (2780) µs. Writing back key-points adds the remaining time. An extra write DMA kernel would optimize this, but exceed the limit of 10 compute units. The final implementation consumes 46.1% LUTMs, 19.6% FFs, 8.1% DSPs and 44.8% BRAMs. Our FPGA design achieves 342 fps for a 1080p resolution. If copying data between host and device is not parallelized to computation it would still be 271 fps. A low-level AKAZE implementation [16] achieved 127 fps for a 1080p image. They additionally compute 4 sub-levels on quarter sized images and a descriptor. We use an accelerated OpenMP version of the FREAK descriptor [18]. The Compute Keypoints function and the FREAK descriptor can be executed in parallel on the host with 324 fps.

V. Conclusion and Outlook

We have implemented a feature detection algorithm for comparison on different OpenCL devices and SDAccel has been evaluated for the FPGA. Different ways of implementing a streaming application containing windowed functions have been explored. This makes it easier to estimate the final performance, to maximize the possible bandwidth consumption and to determine if an application is memory or computation bound. To have a fair comparison, different optimization strategies have been implemented in OpenCL for CPU, GPU and iGPU. The FPGA implementation is 12.89, 10.59 and 1.47 times faster than the one of the CPU, iGPU and GPU and 62.8 times faster than a single threaded CPU one for a 1080p resolution. Additionally, it consumes 23.3, 8.7 and 10.6 times less energy than the CPU, iGPU and GPU OpenCL implementations for a 720p resolution. In future, we extend the benchmark by adding more functions, make the implementation open source and distribute the algorithm on different devices.

Acknowledgment

The authors gratefully acknowledge for the FPGA board and SDAccel tool provided by XILINX. This work is part of the TULIPP project, which is funded by the European Commission under the H2020 Framework Programme for Research and Innovation under grant agreement No 688403.

Table 4: Comparison of the implemented algorithm with a 720p (upper numbers) and a 1080p (lower numbers) resolution for different devices.

	Computation Time (ms)	Speed Up	Power (watts)	Energy (mJ)
Single Thread	80.22 183.48	1 1	23.77	1906.9
CPU OpenMP	23.60 53.60	3.40 3.42	65.74	1527.9
CPU OpenCL	15.57 37.69	5.15 4.87	66.86	1040.7
iGPU OpenCL	13.26 30.92	6.05 5.93	29.31	388.6
GPU OpenCL	2.135 4.290	37.58 42.77	222.69	475.4
FPGA OpenCL	1.467 2.923	55.14 62.77	30.45	44.67

References

[1] F. Winterstein, S. Bayliss and G. A. Constantinides, “High-level synthesis of dynamic data structures: A case study using Vivado HLS,” In Proc. of the Int. Conference on Field-Programmable Technology (FPT), pp. 362-365., Dec. 2013

[2] SDSoC Environment User Guide, 2015.

[3] Loring Wirbel. Xilinx SDAccel A Unified Development Environment for Tomorrows Data Center. 2014.

[4] I. Janik, Q. Tang and M. Khalid, “An overview of Altera SDK for OpenCL: A user perspective,” In Proc. of the 28th Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 559-564., May 2015

[5] J. Villarreal, A. Park, W. Najjar and R. Halstead, “Designing Modular Hardware Accelerators in C with ROCCC 2.0,” In Proc. of the 18th Int. Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 127-134., May 2010

[6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, T. Czajkowski, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” In Proc. of the Int. Symposium on Field Programmable Gate Arrays (FPGA), pp. 33-36, Feb. 2011

[7] The OpenACC Application Programming Interface, 2011.

[8] S. Lee, J. Kim and J. S. Vetter, “OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing,” In Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS), pp. 544-554., May 2016

[9] K. Hill, S. Craciun, A. George and H. Lam, “Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA,” In Proc. of the 26th Int. Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 189-193., July 2015

[10] Z. Wang, B. He, W. Zhang and S. Jiang, “A performance analysis framework for optimizing OpenCL applications on FPGAs,” In Proc. of the Int. Symposium on High Performance Computer Architecture (HPCA), pp. 114-125., March 2016

[11] S. O. Ayat, M. Khalil-Hani and R. Bakhteri, “OpenCL-based hardware-software co-design methodology for image processing implementation on heterogeneous FPGA platform,” In Proc. of the Int. Conference on Control System, Computing and Engineering (ICCSCE), pp. 36-41., Nov. 2015

[12] S. K. Rethinagiri, O. Palomar, J. A. Moreno, O. Unsal and A. Cristal, “Trigeneous Platforms for Energy Efficient Computing of HPC Applications,” In Proc. of the 22nd Int. Conference on High Performance Computing (HiPC), pp. 264-274., Dec. 2015

[13] J. Vasiljevic et al., “OpenCL library of stream memory components targeting FPGAs,” In Proc. of the Int. Conference on Field Programmable Technology (FPT), pp. 104-111, Dec. 2015.

[14] Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization, 2014

[15] Pablo F. Alcantarilla, J. Nuevo and Adrien Bartoli, “Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces”, In British Machine Vision Conference (BMVC), Sept. 2013

[16] G. Jiang, L. Liu, Wenping Zhu, Shouyi Yin and Shaojun Wei, “A 127 fps in full HD accelerator based on optimized AKAZE with efficiency and effectiveness for image feature extraction,” In Proc. of the 52nd Design Automation Conference (DAC), pp. 1-6., June 2015

[17] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features" Computer Vision-ECCV, pp 404-417, 2006

[18] A. Alahi, R. Ortiz and P. Vandergheynst, "FREAK: Fast Retina Keypoint," In Proc. of the Conference on Computer Vision and Pattern Recognition, pp. 510-517, 2012