Open access • Proceedings Article • DOI:10.1109/RECOSOC.2017.8016152

High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier — Source link

R. Domingo, Ruben Salvador, Himar Fabelo, D. Madroñal ...+5 more authors

Institutions: University of Las Palmas de Gran Canaria

Published on: 12 Jul 2017 - Reconfigurable Communication-centric Systems-on-Chip

Topics: Performance per watt, Reconfigurable computing and Compiler

High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier

Conference Paper · July 2017

DOI: 10.1109/ReCoSoC.2017.8016152

9 authors, including:

Ruben Salvador
CentraleSupélec
48 PUBLICATIONS **373** CITATIONS
SEE PROFILE

Himar Fabelo
Universidad de Las Palmas de Gran Canaria
52 PUBLICATIONS **402** CITATIONS
SEE PROFILE

Daniel Madroñal
Universidad Politécnica de Madrid
33 PUBLICATIONS **174** CITATIONS
SEE PROFILE

Samuel Ortega
Universidad de Las Palmas de Gran Canaria
49 PUBLICATIONS **306** CITATIONS
SEE PROFILE

High-Level Design using Intel FPGA OpenCL: a Hyperspectral Imaging Spatial-Spectral Classifier

R. Domingo*, R. Salvador*, H. Fabelo†, D. Madroñal*, S. Ortega†, R. Lazcano*, E. Juárez,* G. Callicó†, C. Sanz*
*Centre on Software Technologies and Multimedia Systems (CITSEM), Universidad Politénica de Madrid (UPM), Spain
†Institute for Applied Microelectronics (IUMA), University of Las Palmas de Gran Canaria (ULPGC), Spain

***Abstract*—Current computational demands require increasing designer's efficiency and system performance per watt. A broadly accepted solution for efficient accelerators implementation is reconfigurable computing. However, typical HDL methodologies require very specific skills and a considerable amount of designer's time. Despite the new approaches to high-level synthesis like OpenCL, given the large heterogeneity in today's devices (manycore, CPUs, GPUs, FPGAs), there is no one-fits-all solution, so to maximize performance, platform-driven optimization is needed. This paper reviews some latest works using Intel FPGA SDK for OpenCL and the strategies for optimization, evaluating the framework for the design of a hyperspectral image spatial-spectral classifier accelerator. Results are reported for a Cyclone V SoC using Intel FPGA OpenCL Offline Compiler 16.0 *out-of-the-box*. From a common baseline C implementation running on the embedded ARM® Cortex®-A9, OpenCL-based synthesis is evaluated applying different generic and vendor specific optimizations. Results show how reasonable speedups are obtained in a device with scarce computing and embedded memory resources. It seems a great step has been given to effectively raise the abstraction level, but still, a considerable amount of HW design skills is needed.**

I. Introduction

Last years have witnessed a revamped approach to high-level synthesis (HLS) techniques for Reconfigurable Computing (RC) devices. In general, an improvement of high-level design (HLD) methodologies have occurred, pushed by the advancement of heterogeneous multicore/manycore devices.

The generalization of these devices has made researchers to investigate different strategies to express parallelism in the source code, to improve the interface among the different computational units of a system and to balance the processing workload accordingly at runtime. In fixed computing systems such as CPUs and GPUs, the interface among the processing units and memory is done at chip's design time; it is then the designer's task to efficiently exploit the available computing resources by using adequate specifications. Given the explosion of this multi/manycore devices, several different languages, programming models, compilers, and runtime engines have emerged to help programming and leveraging the amount of computing power available. This have effectively helped in rising the abstraction level at the specification phase.

HLD for FPGAs has traditionally suffered from two main issues. One is related to the compilation from high-level, non HDL-based descriptions, to dataflow RTL that FPGA synthesis tools understand. This first issue, the silicon compiler itself, has revealed a extremely complex task, with several commercial/research tools investigated. Secondly, a system level methodology and set of tools to help build out the whole system out of the different pieces of modern FPGAs, i.e., CPUs, memory and reconfigurable logic. During these years, these different parts had to be designed and integrated separately: (i) RTL design for the accelerators; (ii) system level C code for control tasks and offloading compute intensive tasks to the logic; and (iii) an interface logic able to sustain a shared memory model ensuring coherency and maximizing data transfers throughput.

Fortunately, the effort done for manycore devices and the interest in FPGAs given their good performance/watt figures, are paving the way to close the HLD gap for RC devices. Two main lines seem to be consolidating, both sharing C/C++ input specifications: a specific HLS compiler together with an automated tool for system level integration and the adaptation of previous architecture/memory models like OpenCL.

The first trusts on strict HLS for hardware accelerators design but without automated system level integration. This is the case with Xilinx Vivado, targeted at hardware designers and system integrators, composed of *Vivado HLS compiler* (formerly AutoESL) and *Vivado IP Integrator* for system level integration. Lately, SDSoC appeared to augment HLS bringing seamless automated system integration features, a highly appealing approach for software programmers. In the Intel FPGA case, no commercial tool is yet available. However, at the time of writing this paper, an HLS Compiler is available for beta access[1], although access seems restricted yet.

The other approach involves the use of a generic programming model for parallel and heterogeneous devices, OpenCL, which enables having a common codebase that can be *adapted* to different platforms. However OpenCL portability is just *functional*, since optimizations are needed to squeeze the target platform computing power as much as possible. This is a rather reasonable price to pay given the complexity of today's devices and considering the step ahead provided in terms of obtained performance vs. design effort. This has been the main bet from

[1]https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html

978-1-5386-3344-1/17/$31.00 ©2017 IEEE

Intel FPGA. Xilinx has also recently promoted its OpenCL tools from beta access to commercial availability.

This paper investigates the design of OpenCL-based accelerators for Intel Cyclone V SoCs, in a Hyperspectral Image (HI) processing application. The rest of the paper is organized as follows. Section II introduces the general OpenCL model and the specifics of the Intel FPGA implementation. Following, Section III reviews related work with emphasis on optimizations for accelerators using this model, before presenting the test case in Section V. The paper continues with the implementation, the obtained results and analysis in Sections VI and VII before concluding in Section VIII.

II. The OpenCL Model

OpenCL appeared mostly to help leveraging the available computing power of GPUs. Since then, it has evolved to support a wide spectrum of heterogeneous platforms like CPUs, GPUs, DSPs and FPGAs. An OpenCL application consists of a *host* program running on the host CPU that triggers the execution of *kernels* in an accelerator device, managing all required data transfers in a master-slave model. The architectural model of OpenCL conceives a system as a set of *Compute Devices* (CDs) attached to a host CPU. Each CD have a number of *Compute Units* (CUs), which are in turn composed of another number of *Processing Elements* (PEs). The system memory model is divided among a shared *global memory*, *local memory*, *constant memory* and *private* memory.

This model, inherited from the massive parallelism of GPUs, gives support to a data-parallel approach in which a complete computational work can be divided into several non-dependent, concurrent execution threads. Therefore, loop iterations are replaced by parallel executing instances of the kernel. Each of them, basically a thread, is known as *Work-Item* (WI) and is mapped onto a PE. WIs are grouped in so called *Work-Groups* (WGs) (mapped to CUs), the total of which conforms the complete kernel. The host CPU and the CUs share the global and constant memories, accessible by all WIs but usually with high latency access.Local memory, shared among the WIs of each WG, and private memory, restricted for each WI, are both on-chip, so accesses are usually low latency. Additionally, Instruction Level Parallelism (ILP) is possible in OpenCL through its task-parallel model, known as single WI (SWI), which enables concurrent kernel execution. Fig. 1 shows the generic OpenCL model.

A. OpenCL for FPGAs

In the specific case of FPGAs, the OpenCL model brings a twofold advantage. On one side, it enhances the generic model by allowing the designer to create application specific architectures instead of dealing with a fixed datapath. And on the other side, it allows a high-level programming approach to hardware design, which considers the heterogeneity of today's SoC-based FPGAs. The Intel FPGA SDK for OpenCL supports the embedded profile of the OpenCL Specification version 1.0. Up to the newest revisions with the introduction of *Pipes*, the standard memory model required the host to coordinate all data transfers. To overcome this, Intel *Altera Offline Compiler* (AOC) introduced *Channels* to allow for concurrent kernel execution enabling kernel-kernel and kernel-I/O communications through FIFO buffers.

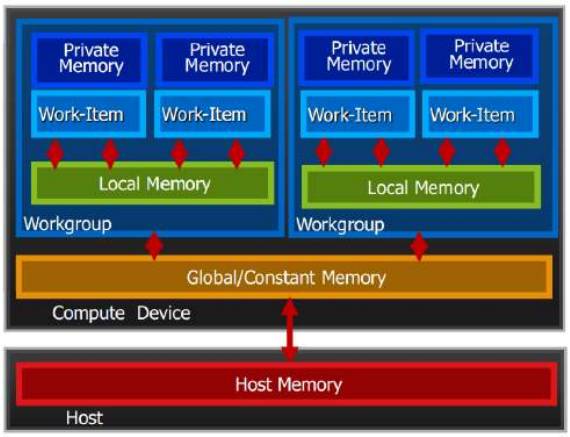


Fig. 1. OpenCL model. From https://www.khronos.org/

III. Related Work

Despite its recent appearance around 2011, an interesting amount of work to explore Intel FPGA OpenCL have been reported. Most deal with implementation results and possible optimizations for different applications or with modified flows in the form of compiler backends for code transformation toward optimized synthesis and performance results.

A source-to-source compiler was proposed to transform higher level C++-based Domain-Specific Language (DSL) specifications into optimized OpenCL descriptions for Vivado HLS [1] and Intel FPGAs [2]. The work extends the Hipacc framework[2], based on Clang/LLVM for image processing applications, targeting code generation such as CUDA/OpenCL for GPUs, Renderscript for Android and C++ for Vivado HLS. The specific optimizations for FPGAs include (as in typical RTL-based flows): (i) generation of streaming pipelines, better than buffer-based communications in the case of GPUs; (ii) operator replication, either by kernel vectorization or CU replication; (iii) generation of specific bitwidth datapaths by automatically transforming the DSL code with masking operations to direct the AOC compiler.

Regarding parallelism exploitation it has been shown that replicating internal kernel operators applying *loop coarsening* [3] yields better results than replicating the whole accelerator and using loop tiling. Loop coarsening basically requires prefetching and storing data in local memories to reduce communication overhead and to replicate the datapath to increase throughput. This results in a single, more complex control structure, but whose extra size compensates for the replicated control structures in case of tiling and kernel replication. Results were reported for local image operators for both Vivado HLS [3] and Intel FPGA OpenCL [4].

[2]Hipacc: http://hipacc-lang.org

Works focused on accelerator implementations have been reported for different applications like machine learning, scientific computing, computer vision, etc. One of them [5] implements and analyzes performance and power consumption of a k-means clustering algorithm for various data, cluster, and dimension sizes. The popularity of Deep Learning (DL) has pushed research on the implementation of Convolutional Neural Networks (CNNs) in FPGAs [6], [7], which is typically memory bound. A Deep Learning Accelerator (DLA) that maximizes data reuse and minimizes external memory bandwidth to overcome this issue, has been reported [6]. On-chip memory is used for vectorization and the Winograd transform [8] to further boost performance reducing the number of required multiply-accumulate operations in convolutions.

Another work [7] proposes an analytical performance model defining two metrics, *machine balance* and *code balance*. This model is able to "quantify the difference between available resources provided by native hardware (FPGA devices) and actual resources demanded by the application", i.e., resource requirements vs. availability. Hence, the performance bottleneck in a given kernel for a given device can be found. Authors apply the model to a CNN implementation, finding the bottleneck to be on-chip memory bandwidth, so they propose: (i) a 2D connection scheme among PEs for efficient data sharing without memory replication; (ii) a 2D dispatcher to support the proposed interconnection and a WI scheduling technique to further optimize memory usage; and (iii) a shared buffer technique to further reduce the external memory bandwidth.

Optimization techniques for Partial Differential Equations (PDE) solvers to improve performance and energy efficiency, are also reported [9]. Comparisons are done among an Arria 10 device, a low performance Intel HD Graphics GPU and a power hungry NVIDIA GTX 980 GPU. Authors show how having a $10\times$ higher memory bandwidth allows the NVIDIA GPU to obtain higher energy efficiency (MB/Joule).

Lastly, a GA has been used [10] to find suitable approximate versions of OpenCL kernels to reduce area and hence increase data-level parallelism by increasing the number of kernels. The core of the work is a source-to-source compiler that optimizes the code by finding *safe-to-approximate* operations and data elements and reduces their precision.

IV. Optimizing OpenCL Kernels for Intel FPGAs

Generic OpenCL ensures functional portability among different platforms. However, in order to optimize performance in a given platform, specific tweaks need to be introduced in the code to better direct the compiler. In the case of FPGAs, this is probably a bit more relevant since the code itself will be used to infer the architecture. It is important to keep in mind that bad optimizations might produce implementations with a lower performance compared to an unoptimized version.

Most of the works introduced previously analyze optimizations somehow. Specific major *in-house* optimizations, such as source-to-source compilers, or any other not available in the AOC that involves changing the default methodology, are left out of the following compilation. The following review list of optimizations is highly inspired in other previous works [2], [9] and in Intel FPGA SDK for OpenCL guides [11], [12]; this is just a compilation with some extra comments based on authors' own experience. For more specific details, `#pragmas` and `__attributes__` please check [11], [12]:

- ***Loop unrolling***. Improves loop performance by running various loop iterations in parallel given no data dependencies are found inside loop execution. Hence an $N\times$ speedup might be ideally achieved if N iterations are dispatched in parallel. The unroll factor can be manually set using `#pragma unroll N` before a given loop; if `N` is omitted a full loop unroll is tried. Unrolling is also recommended for register inference since it helps determining static data access to arrays.
- ***Loop pipelining***. A form of data parallelism in which a sequential chain of operations is split-up in different stages by introducing intermediate storage elements (flip-flops/BRAMs). This increases resource usage but allows to feed the blocks of the different processing stages with new data at (ideally) each clock cycle. The AOC applies this optimization automatically.
- ***Kernel vectorization***. SIMD-like vectorization replicates the kernel datapath creating computing vector lanes and distributing the WIs among the available lanes. This way, each WI instance does effectively perform more computations. Automatic kernel vectorization is possible (and recommended over manual) using attributes to indicate the AOC the pair (number of WIs, WG size) without host/kernel code modifications, even coalescing memory accesses when possible. To ensure coalescing, vector loads/stores should be manually indicated in kernel parameters (float2, float4...) and write the code so that sequential access patterns can be identified at compile time. Ideally, $N\times$ throughput speedups for N-sized vectorizations (limited to 2, 4, 8 or 16) could be achieved.
- ***CU replication***. By using multiple CUs, the complete accelerator is replicated so parallel WG execution is enabled. Ideally, the speedup should be $N\times$ for N replications, effectively increasing data throughput; however, memory bandwidth contention among CUs will exist. Compared to SIMD vectorization, logic resources usage usually increases to arbitrate memory accesses and dispatch data to the CUs. CU replication can be combined with kernel vectorization, but the optimal factors have to be carefully analyzed.
- ***Hardware inspired code.***. In order to maximize performance, typical control structures such as `if-else` should be limited to reduce delay by avoiding multiplexers. Besides, the required control logic to implement jumps and branches will not consume additional FPGA resources. Ozkan et al. [2] report how using MUX-like conditional assignments yields better results than standard `if-else` code. As opposed to SW practices, instead of doing computations inside each branch of a conditional statement to avoid unnecessary operations,

moving computations outside the condition and leaving just the resulting assignments for the inside, improves results. Arrays in `__private` memory are inferred as registers as long as the AOC is able to solve accesses statically at compile time, the code uses hardcoded accesses and loops are fully unrolled. This is indeed the way to infer shift registers, which must avoid conditional shifting for an efficient implementation. For large arrays with dynamic accesses, the use of `__local` memory is recommended over `__private` since it creates more efficient hardware.

- ***Regular memory accesses***. Random memory accesses usually penalize processing systems with small cache memories. Hence, random, i.e., irregular and unaligned memory accesses tend to penalize performance greatly in FPGAs. Some works [9] propose the use of stencils for matrix-like operations, which forces to hard-code the vector elements and associated coefficients inside kernels. A typical approach generally followed is *memory coalescing*, which refers to combining multiple, aligned memory accesses into a single transaction to optimize efficiency by using the full bus datawidth.
- ***Bitwidths optimization***. Automatically applied by AOC if data ranges are known at compile time (loop trip counts, pointers for memory accesses, etc.). Besides, specific attributes are available to further guide the compiler, for instance indicating local memory pointer sizes.

A. Single Work-Item Kernels vs. NDRange Kernels

As introduced in Section II, two types of kernels are possible. The approach followed in manycore architectures such as GPUs is the use of ***NDRange kernels***. In this case, the complete processing work is divided among a number of WIs so kernels execute over the specified range. It is the OpenCL runtime that is in charge of dispatching WIs to the kernel. How the complete work is split greatly determines the obtained performance, since it directly impacts the created memory access patterns. As shown in Fig.II, local memory is shared by all WIs in a WG. Therefore, data reuse has to be maximized so WIs can obtain from local memory the maximum amount of shared data among them as possible. In the specific case of Intel FPGA OpenCL, the AOC would not synthesize a given kernel if the required amount of memory (as per variable declaration) exceeds the available resources. The WG size must be specified whenever possible to avoid excessive logic resources derived from using default WG sizes, which might vary from 1, to 256 to the global NDRange size.

In the case of ***SWI kernels***, which is the approach recommended by Intel FPGA, the complete workload is processed in a single call to a single kernel, i.e., one thread. This makes this type of kernels not suited to massively parallel architectures like GPUs, which greatly benefit from many threads. For the case of FPGAs, task-based kernels allow exploiting deep pipelines and data locality very explicitly (if allowed by the application), so data buffering as typical in local operators for image processing is recommended. Some works have confirmed this recommendation [2] for local image operators in image processing. However, other works have reported a contradicting behaviour in the case of search-based kernels (k-means clustering) [5] or for a Conjugated Gradient method for PDEs (basically, vector scale-and-adds, dot-products and sparse, Laplacian Matrix-vector multiplications) [9].

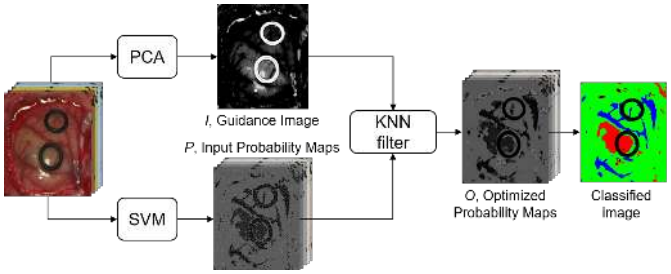


Fig. 2. Block diagram of the KNN based spatial-spectral classification

V. Test Case Description

A spatial-spectral classifier for Hyperspectral Images (HI) is used as test case in this work. An HI [13] consists of a collection of 2D images, so-called data cubes, captured at different (hundred) wavelengths, which makes them contain a huge amount of information. A vector of reflectance or emittance values is measured for each pixel of the image, obtaining what is known as the spectral signature of the pixel. Hence, from this captured data cube it is possible to analyze the type of material a given object is made of [14].

The combination of HI with different machine learning techniques, has allowed this technology to be used as an automatic detection mechanism for different purposes. Among different applications, HI has been used for remote sensing of the Earth surface [15], astronomy [16], forensics [17], target detection [18] or medicine [19].

The test case considered uses pixel-wise results from a Support Vector Machine (SVM) classifier combined with the spectral information derived from a Principal Components Analysis (PCA) algorithm, to tune up the class probability maps obtained with the SVM [20], as shown in Fig. 2.

The probability map smoothing is achieved with a K Nearest Neighbors (KNN) filter, which has two inputs: a set of probability maps derived from the SVM classification process and a one-band PCA guidance image that contains the spectral information. These inputs are mingled in a feature space that consists of the normalized pixel value of the guidance image and the weighted (λ) normalized coordinates of the pixel as shown in (1), which defines the feature vector $F(i)$.

$$F(i) = ((I(i), \lambda r(i), \lambda c(i)) \tag{1}$$

where $I(i)$ is the normalized pixel value of the guidance image I (first PCA band), and $r(i)$ and $c(i)$ refer to the normalized coordinates of pixel i. λ controls the balance between the spectral and spatial domains in the KNN searching process. As in previous works [20], a value of $\lambda = 1$ and $K < 40$ was

determined to be a good compromise, given that high values of λ and K tend to oversmooth the resulting image.

In the first stage of the KNN algorithm, to find the K nearest neighbors of a pixel, the euclidean distance (within the feature space) to all the pixels in the image is computed. Note that the weight parameter λ controls the importance of the spatial coordinates with regard to the spectral components. Once the K nearest pixels are found, i.e., those pixels of the image whose distances to the current pixel are the smallest ones, the second stage of the KNN algorithm averages (low-pass filters) the pixel probabilities of those nearest pixels to compute the current class pixel probability, repeating this action for each of the available classes, as shown in (2):

$$O(i) = \frac{\sum P(j)}{K}, j \in w_i \quad (2)$$

where O contains the optimized probability maps, P the original probability maps from the SVM and w_i indicates the K nearest neighbours of pixel i found in the feature space $F(i)$. Please note how there will be as many O output classification maps as classes are considered in the SVM. From these optimized probability maps in O, a final classification map is obtained by assigning each pixel the label of the class with the maximum probability.

To reduce the computational complexity of the search process, a band of 14 rows has been experimentally selected as the search space. The rationale behind this decision is that the results produced are exactly the same as those obtained when the whole image is employed as search space. The band around the processed pixel is selected to be symmetrical. However, for the top-most and bottom-most rows, the size of the band is selected in such a way that no search is conducted in the image further than half of the band size down and up, respectively. For these rows, the band size is increased or decreased as each new pixel is processed until the steady state is reached.

VI. Implementation

A low-cost and small device (compared to current Arria 10 accelerators) was used for implementation purposes: a Cyclone V A5 SoC with 85 000 Logic Elements, 32 075 ALMs, 397 M10K memory blocks (3970 Mb), 480Kb as MLABs (Memory Logic Array Block) and 174 18-by-19 multipliers/87 variable-precision DSP blocks.

The basic structure of the KNN-based method for spatial-spectral classification is shown in Fig.3, which gathers a pseudocode description. Both the search & ordering (lines 18–38) and filtering (lines 39–58) stages are shown. Different experiments have been done to evaluate the different options and optimizations available with the AOC. A baseline software implementation running in one of the two ARM Cortex-A9 processors is selected for comparing the acceleration possibilities within the platform. The following sections describe the specific details and initial experiments accomplished.

A. Single-Work Item Kernel

The optimization of SWI kernels is guided by the obtained pipeline quality found in the synthesis report. Several **#pragma unroll** can be seen in the code as result of the first tests. They correspond to the innermost loop in the ordering phase (lines 32–38) so the neighbours array was implemented as a shift register. Besides, all the loops with a low number of iterations are also unrolled, as shown in the filtering stage loops. The synthesis report was able to pipeline the outer search loop with succesive iterations every 2 cycles. Although it is not the optimum, which should be just one, it is still a good result given the amount of conditional branches needed for the search & ordering process (lines 29–38) involving floating point numbers. In particular, the dependency is on the floating point compare operations for the variable knn (and $dist$) at lines 27, 29 and 34. A version with 2 separate kernels, one for each part of the algorithm communicating the intermediate results through the host, was also evaluated.

B. NDRange Kernel

The kernel structure for this case, shown in Fig. 5, besides the specific keywords required to make it such (the attributes `(reqd_work_group_size())` and `(num_simd_work_items())`, is mostly the same as in the SWI case. Differences with SWI case are shown in purple color in the code. Given the resource limitations, as will be shown in Section VII, the filtering stage has been moved to the ARM processor for all these kernels but one. Now, each WI processes a different pixel in parallel, so there is no need for the outer loop at line 18. Also, the update of the search band limits has been moved to the very beginning of the kernel before line 19, since each pixel (WI) needs a different search band. Loop indexes were modified accordingly to use now the required global and local indexes of each WI in which the whole computing work was divided.

Several combinations of WG sizes were evaluated, as well as two versions with two separate kernels, one for each part of the algorithm communicating the intermediate results through the host and through a channel.

Lastly, a version which makes specific use of local memory to buffer a block of pixels from the image/search band, was also assessed. Since the search band moves in a row-wise fashion, all pixels from a row will always share the same search band. Hence, to optimize processing, loading blocks of pixels (sized Blockx, as in Fig. 5) in local memory allows these to be used by various WIs of the WG. This enables the creation of WGs with as many WIs as pixels in a row that share a search band and hence might benefit from local memory sharing and loop unrolling. Due to resource limitations, the whole search band does not fit in device's memory, so various iterations are needed. This case requires barriers in order to allow each WI to bring its pixels from global memory and be synchronized with the others (so the other pixels needed from the search band are already there), hopefully letting a smart cache strategy in the runtime exploit this data locality. These barriers are placed after loading pixels to local memory and right before finishing with a given block of the search band, to join all the threads.

```
1:  I: input PCA image (1D linearized indexes)
2:  r: row-wise pixel 2-D coordinates
3:  c: column-wise pixel 2-D coordinates
4:  O: output optimized classification map
5:
6:  Nrows: number of rows
7:  Nsamples: number of samples (pixels) per row, i.e., columns
8:  P: total pixels (Nrows × Nsamples)
9:  BL, BU: search band upper/lower pixel index
10: Brows: number of search band rows
11: K: number of neighbours to find
12: knn: temporary array, stores K nearest neighbours of a given pixel
13: coords: temporary array, 1-D linearized coordinates (r × c)
14: NClasses: number of classes
15: SVM: input SVM maps
16: prob: temp array, probabilities per class
17:
18: for i ← 0, P do                                   ▷ For each PCA (I) pixel
19:     #pragma unroll
20:     for k ← 0, K do                               ▷ Initialize temp arrays
21:         knn[k] ← 1000000
22:         coords[k] ← 0
23:     for j ← BL, BU do                             ▷ For each search band pixel
24:         distp ← (I[i] − I[j])                     ▷ Spectral distance
25:         distr ← (r[i] − r[j])                     ▷ Row distance
26:         distc ← (c[i] − c[j])                     ▷ Column distance
27:         dist ← (distp × distp) + (distr × distr) + (distc × distc)
28:         linear_index ← (r[j] × Nsamples) + c[j]
29:         if dist < knn[K-1] then                   ▷ If distance larger than maximum
30:             knn[K − 1] ← dist
31:             coords[K − 1] ← linear_index
32:             #pragma unroll
33:             for k ← [K − 2], 0 do                 ▷ Ordered insertion
34:                 if dist < knn[k] then
35:                     knn[k + 1] ← knn[k]
36:                     coords[k + 1] ← linear_index
37:                     knn[k] ← dist
38:                     coords[k] ← linear_index
39:     #pragma unroll
40:     for k ← 0, K do                               ▷ Neighbours array initialization
41:         neighbours[k] ← knn[k]
42:     #pragma unroll
43:     for c ← 0, NClasses do                        ▷ Class probability initialization
44:         prob[c] ← 0
45:     for k ← 0, K do                               ▷ For each neighbour
46:         SVMidx ← neighbours(k)
47:         #pragma unroll
48:         for c ← 0, NClasses do                    ▷ Accumulate each class probability
49:             prob[c] ← prob[c] + SVM[c + NClasses × SVMidx]
50:     #pragma unroll
51:     for c ← 0, NClasses do                        ▷ Average class probability
52:         prob[c] ← prob[c] × (1/K)
53:     maxProbC ← 0                                  ▷ Assume maximum probability is in class 0
54:     #pragma unroll
55:     for c ← 1, NClasses do                        ▷ Iterate on remaining classes
56:         if prob[c] > prob[maxProbC] then
57:             maxProbC ← c
58:     O[i] ← maxProbC + 1
59:     BL, BU ← Update according to search band strategy
```

Fig. 3. KNN-based Spatial-Spectral Classification Algorithm. Baseline code with parallelization pragmas indicated in blue.

VII. Results

This section contains the comparison results among the different kernels implemented. These are shown in Fig. 6 for both the processing time and the consumed resources of the device (in %). The timing results shown in the figure are for four different image sizes 256×256, 512×256, 256×512 and

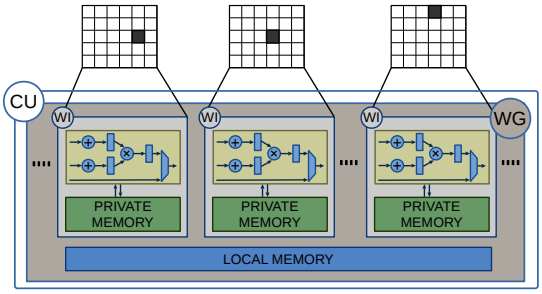


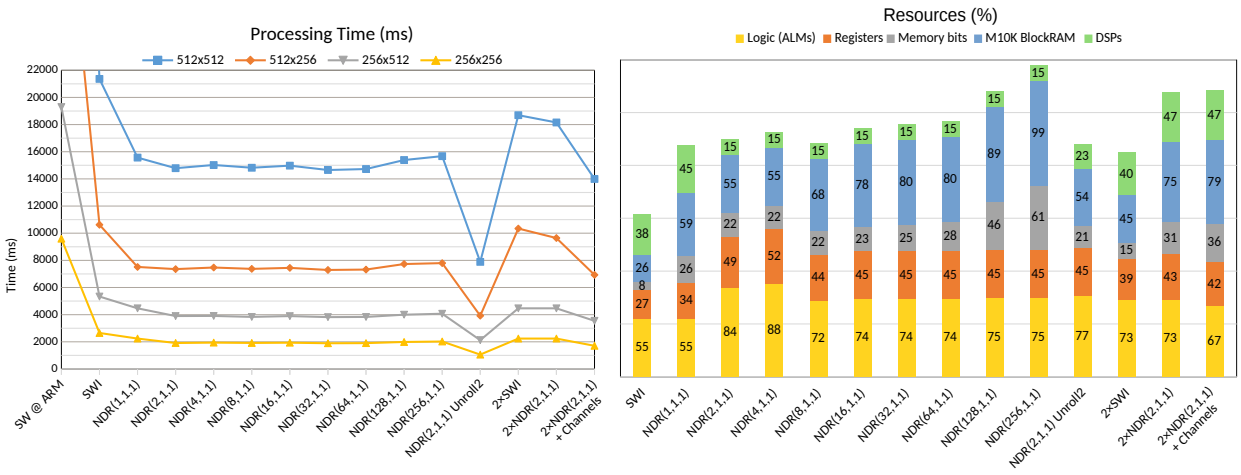
Fig. 4. Representation of the design with one CU and N WIs per WG. The architecture shown corresponds with the case of an NDRange kernel with multiple WIs in a WG, in which each WI processes one pixel.

```
1:  block_x, block_y ← get_group_id(0), get_group_id(1)   ▷ Global dim. index
2:  local_x, local_y ← get_local_id(0), get_local_id(1)   ▷ Local dim. index
3:  BL, BU ← Update according to search band strategy
4:  #pragma unroll
5:  for k ← 0, K do                                       ▷ Initialize temp arrays
6:      knn[k] ← 1000000
7:      coords[k] ← 0
8:  for j ← BL, BU do                                     ▷ For each search band pixel
9:      local_mem ← I(i + local_x)
10:     barrier(CLK_LOCAL_MEM_FENCE)
11:     #pragma unroll
12:     for b ← 0, Blockx do                              ▷ For each pixel in the block
13:         distp ← (I[i] − I[j])                         ▷ Spectral distance
14:         distr ← (r[i] − r[j])                         ▷ Row distance
15:         distc ← (c[i] − c[j])                         ▷ Column distance
16:         dist ← (distp × distp) + (distr × distr) + (distc × distc)
17:         linear_index ← (r[j] × Nsamples) + c[j]
18:         if dist < knn[K-1] then                       ▷ If distance larger than maximum
19:             knn[K − 1] ← dist
20:             coords[K − 1] ← linear_index
21:             #pragma unroll
22:             for k ← [K − 2], 0 do                     ▷ Ordered insertion
23:                 if dist < knn[k] then
24:                     knn[k + 1] ← knn[k]
25:                     coords[k + 1] ← linear_index
26:                     knn[k] ← dist
27:                     coords[k] ← linear_index
28:     barrier(CLK_LOCAL_MEM_FENCE)
```

Fig. 5. KNN-based Spatial-Spectral Classification Algorithm. Purple code indicates the differences with the SWI kernel.

512×512. The processing time results from the image sized 512×256 are the ones to be used in the following discussion. Regarding the ARM, it takes 38.9 seconds to compute the whole algorithm for this image. In general, the obtained speedup is of at least $5\times$ for any of the HW implementations as compared to SW (38.9 vs. 10.6 for the SWI result, which is the slowest). In general, the times obtained for the different kernels are highly similar (although not as much as could seem given the graph scale).

For all NDR kernels, with the exception of the NDR(1,1,1), the filtering stage has been moved to the ARM processor (there is just a difference of 100ms in this stage) to dedicate the logic resources to speedup the search process. All the NDRange kernel versions behave very similar in terms of speed, with the exception of NDR(2,1,1) with an unroll factor of 2 and local memory to store parts of the band, which halves the time of the same kernel version without unrolling. This was expected,



(a) (b)

Fig. 6. (a) contains the timing results for different image sizes) and (b) the use of resources of the device for each kernel.

and shows the effectiveness of SIMD kernel vectorization. Sadly, as seen in Fig. 6b, the use of logic resources (77%) and memory (54%) made it impossible to evaluate higher unroll factors since they would not fit in the device. It is worth noting how for the NDR(x,1,1), as x increases, the amount of resources also increases, but the time remains mostly the same. This is reasonable since no explicit vectorization has been possible by doing loop unrolling or using vector data types. Hence, the extra amount of logic resources are mostly those required to manage more WIs in the WG, but there were no more logic resources available to leverage the extra parallelization capability.

For the case with hardware replication, NDR(2,1,1)-Unroll2, DSP usage increases. However, it is interesting to see how logic resources and registers used are less than for the NDR(2,1,1) case. This might be due to the fact that completely unrolling the loop (a factor of 2 for 2 WIs, as per Blockx) helps optimizing the area required to manage the WIs and saving resources for the loop control logic.

The last three tests have evaluated the impact of dividing the algorithm in two different kernels, one for searching and another for filtering. As expected, in those using the host to communicate data between both kernels (2×SWI and 2×NDR(2,1,1) computing time increases. However, when using channels (last experiment in the graph) time is reduced, even more than doing all the processing in a single kernel. However, duplicating kernels introduces an overhead in the use of resources that does not justify this slight time reduction.

It can be concluded, that for this application and this device, and with some tests remaining to be done, the case of an NDRange kernel with a local dimension of 2 WIs, local memory, and an unroll factor of 2 is the most balanced and fastest solution.

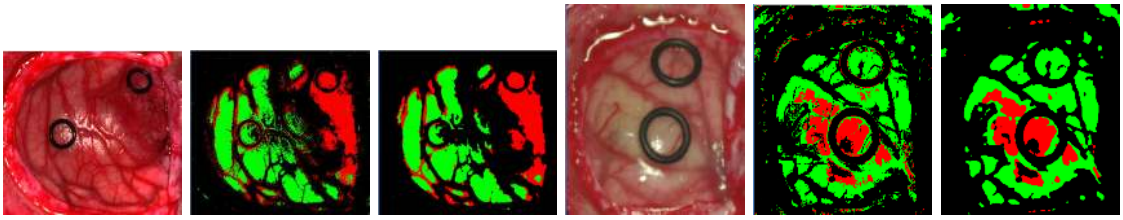
Finally, Fig. 7 contains a comparison of the resulting probability maps obtained after both the SVM classification and the spatial-spectral filtering using the KNN-based filtering process. The HI images shown correspond with a test case for cancer detection purposes in human brain tissues. It shows how after the KNN filtering process, the classification map from the SVM has been smoothed to better detect tumour margins.

VIII. Conclusion

This work has reviewed some of the latest works in the use of OpenCL as a HLS tool, analyzing the specific case of Intel FPGAs. The model and the methodology for optimizing kernels have been presented. Besides, the different techniques to synthesize well-performing kernels were also introduced, including different forms of parallelization by vectorization and replication, some design patterns to direct the compiler for efficient hardware synthesis and the importance of optimized memory accesses. Regarding the experiments reported, out-of-the-box AOC compiler optimizations were applied to evaluate the state of the technology. As a use case application, a KNN-based filtering stage for combined spatial-spectral classification in hyperspectral imaging for cancer detection, has been selected given the amount of processing power required.

Several experiments have been done with SWI and NDRange kernels, applying different parallelization strategies. Contrary to expectations and as advised by Intel and part of the literature, NDRange kernels have achieved higher speedups compared to SWI kernels in this case. However, for a similar application, a k-means algorithm, authors also reported the same findings as analyzed. This might be due to the ordering process itself, which involves conditional logic to be synthesized, penalizing this way the pipeline efficiency. Hence, further studies and experiments with different sorting methods will be performed. Besides, the SWI kernels will be reviewed, since they are not properly using the available local memory to cache shared data, which should improve performance.

Sadly, the size of the FPGA has turned to be insufficient to study different degrees of vectorization and kernel and CU replications. Just 2 kernels, no CU replications and WGs sized 2 WIs with complete unroll were possible. Anyway, the



(a) RGB, 12C1 (b) SVM, 12C1 (c) KNN, 12C1 (d) RGB, 20C1 (e) SVM, 20C1 (f) KNN, 20C1

Fig. 7. Comparison of the resulting classification maps. The figure shows the results from two different patients. The images show the RGB representations (a) and (d), the SVM classification maps (b) and (e), and the KNN-based spatial-spectral classification maps (c) and (f), respectively.

increase in logic derived from larger WGs has been analyzed, showing the importance of a good balance among WG sizes and unrolling factors for efficient data-parallel processing.

As future work also remains comparisons of latest RC devices and associated HLS tools with other low power consumption embedded platforms like the NVIDIA Jetson TX2 GPU, the Massively Parallel Processor Array (MMPA) from Kalray or the new Epiphany manycore processors.

Although a learning curve is needed, the introduction of OpenCL as an HLS tool means a great step ahead. However, better reporting tools are still needed to better understand the synthesized datapath and optimize performance. Fortunately, the latest version of the Intel FPGA SDK for OpenCL has introduced some tools to analyze the architecture.

ACKNOWLEDGMENT

This work has been supported in part by the European Commission through the FP7 FET Open programme ICT- 2011.9.2, European Project HELICoiD "HypErspectral Imaging Cancer Detection" under Grant Agreement 618080.

REFERENCES

[1] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich, "Code generation from a domain-specific language for c-based hls of hardware accelerators," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis - CODES '14*, ACM Press, 2014, pp. 1–10, ISBN: 9781450330503. DOI: 10.1145/2656075.2656081.

[2] M. A. Ozkan, O. Reiche, F. Hannig, and J. Teich, "Fpga-based accelerator design from a domain-specific language," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, Aug. 2016, pp. 1–9, ISBN: 978-2-8399-1844-2. DOI: 10.1109/FPL.2016.7577357.

[3] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in c-based high-level synthesis," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, Jul. 2015, pp. 166–173. DOI: 10.1109/ASAP.2015.7245730.

[4] O. Reiche, M. A. Özkan, F. Hannig, J. Teich, and M. Schmid, "Loop parallelization techniques for fpga accelerator synthesis," *Journal of Signal Processing Systems*, pp. 1–25, Feb. 2017. DOI: 10.1007/s11265-017-1229-7.

[5] Q. Y. Tang and M. A. S. Khalid, "Acceleration of k-means algorithm using altera sdk for opencl," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 1, pp. 1–19, Sep. 2016, ISSN: 19367406. DOI: 10.1145/2964910.

[6] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™ deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, ser. FPGA'17, Monterrey, CA: ACM Press, Feb. 2017, pp. 55–64, ISBN: 9781450343541. DOI: 10.1145/3020078.3021738.

[7] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, ser. FPGA '17, Monterrey, CA: ACM Press, Feb. 2017, pp. 25–34, ISBN: 9781450343541. DOI: 10.1145/3020078.3021698.

[8] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[9] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, "Energy efficient scientific computing on fpgas using opencl," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, ser. FPGA'17, Monterrey, CA: ACM Press, Feb. 2017, pp. 247–256, ISBN: 9781450343541. DOI: 10.1145/3020078.3021730.

[10] A. Lotfi, A. Rahimi, A. Yazdanbakhsh, H. Esmaeilzadeh, and R. K. Gupta, "Grater: an approximation workflow for exploiting data-level parallelism in fpga acceleration," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Mar. 2016, pp. 1279–1284, ISBN: 978-3-9815370-7-9. DOI: 10.3850/9783981537079_0805.

[11] Intel Corporation, *Intel fpga sdk for opencl, Programming guide*, UG-OCL002, version 2016.10.31, Oct. 31, 2016.

[12] ——, *Intel fpga sdk for opencl, Best practices guide*, UG-OCL003, version 2016.12.2, Dec. 2, 2016.

[13] C.-I. Chang, *Hyperspectral imaging: Techniques for spectral detection and classification*. Springer Science & Business Media, 2003, vol. 1.

[14] D. Manolakis and G. Shaw, "Detection algorithms for hyperspectral imaging applications," *IEEE Signal Processing Magazine*, vol. 19, no. 1, pp. 29–43, Jan. 2002, ISSN: 1053-5888. DOI: 10.1109/79.974724.

[15] J. M. Bioucas-Dias, A. Plaza, G. Camps-Valls, P. Scheunders, N. Nasrabadi, and J. Chanussot, "Hyperspectral remote sensing data analysis and future challenges," *IEEE Geoscience and Remote Sensing Magazine*, vol. 1, no. 2, pp. 6–36, Jun. 2013, ISSN: 2473-2397. DOI: 10.1109/MGRS.2013.2244672.

[16] E. K. Hege, D. O'Connell, W. Johnson, S. Basty, and E. L. Dereniak, "Hyperspectral imaging for astronomy and space surveillance," in *Proc. SPIE 5159, Imaging Spectrometry IX*, vol. 5159, 2004, pp. 380–391. DOI: 10.1117/12.506426.

[17] G. Edelman, E. Gaston, T. van Leeuwen, P. Cullen, and M. Aalders, "Hyperspectral imaging for non-contact analysis of forensic traces," *Forensic Science International*, vol. 223, no. 1–3, pp. 28–39, 2012, ISSN: 0379-0738. DOI: 10.1016/j.forsciint.2012.09.012.

[18] M. Fauvel, Y. Tarabalka, J. A. Benediktsson, J. Chanussot, and J. C. Tilton, "Advances in spectral-spatial classification of hyperspectral images," *Proceedings of the IEEE*, vol. 101, no. 3, pp. 652–675, Mar. 2013, ISSN: 0018-9219. DOI: 10.1109/JPROC.2012.2197589.

[19] M. E. Martin, M. B. Wabuyele, K. Chen, P. Kasili, M. Panjehpour, M. Phan, B. Overholt, G. Cunningham, D. Wilson, R. C. DeNovo, and T. Vo-Dinh, "Development of an advanced hyperspectral imaging (hsi) system with applications for cancer detection," *Annals of Biomedical Engineering*, vol. 34, no. 6, pp. 1061–1068, 2006, ISSN: 1573-9686. DOI: 10.1007/s10439-006-9121-9.

[20] K. Huang, S. Li, X. Kang, and L. Fang, "Spectral-spatial hyperspectral image classification based on knn," *Sensing and Imaging*, vol. 17, no. 1, pp. 1–13, 2015, ISSN: 1557-2072. DOI: 10.1007/s11220-015-0126-z.