FPGA-Based Accelerator Design from a Domain-Specific Language

M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich
Friedrich-Alexander University of Erlangen-Nürnberg (FAU), Germany
{akif.oezkan,oliver.reiche,hannig,teich}@fau.de

***Abstract*—A large portion of image processing applications often come with stringent requirements regarding performance, energy efficiency, and power. FPGAs have proven to be among the most suitable architectures for algorithms that can be processed in a streaming pipeline. Yet, designing imaging systems for FPGAs remains a very time consuming task. High-Level Synthesis, which has significantly improved due to recent advancements, promises to overcome this obstacle. In particular, Altera OpenCL is a handy solution for employing an FPGA in a heterogeneous system as it covers all device communication. However, to obtain efficient hardware implementations, extreme code modifications, contradicting OpenCL's data-parallel programming paradigm, are necessary.**

In this work, we explore the programming methodology that yields significantly better hardware implementations for the Altera Offline Compiler. We furthermore designed a compiler back end for a domain-specific source-to-source compiler to leverage the algorithm description to a higher level and generate highly optimized OpenCL code. Moreover, we advanced the compiler to support arbitrary bit width operations, which are fundamental to hardware designs. We evaluate our approach by discussing the resulting implementations throughout an extensive application set and comparing them with example designs, provided by Altera. In addition, as we can derive multiple implementations for completely different target platforms from the same domain-specific language source code, we present a comparison of the achieved implementations in contrast to GPU implementations.

I. Introduction

For image processing algorithms, many target domains, such as medical imaging or automotive computer vision, require stringent constraints regarding real-time capabilities, throughput, and energy efficiency. In particular, in mobile systems, the latter is of utmost importance, as those devices are usually battery-driven. To best possibly meet these requirements, Field Programmable Gate Arrays (FPGAs) have been proven to be among the most suitable architectures due to their fixed pipeline and low power consumption.

Yet, designing imaging systems for FPGAs is still very time consuming even for hardware experts. A remedy to overcome this obstacle is High-Level Synthesis (HLS), which has significantly improved due to recent advancements. Although, using high-level languages has considerably lowered the bar, an architecture expert is still required to obtain hardware implementations with high throughput while consuming a reasonable amount of resources. Furthermore, if the FPGA is used as a co-processor or dedicated accelerator in a heterogeneous system, handling data exchange and device communication still represent barriers for hardware designers.

To further lift the burden of hardware design, Open Computing Language (OpenCL) was proposed, whose Application Programming Interface (API) can be used to entirely cover host/device communication. Therefore, OpenCL proves to be very convenient

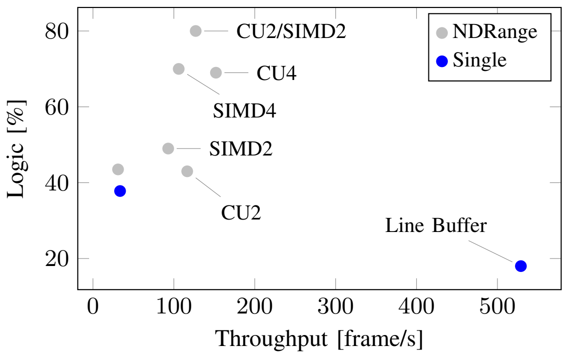


Figure 1: Design points of NDRange and single work-item kernels for the Gaussian blur filter. NDRange kernels have been varied in automatic replication of the entire accelerator (CU) and innermost kernel computation (SIMD).

for employing an FPGA in a heterogeneous system. However, as OpenCL per se is merely another incarnation of HLS, the right programming methodology still demands the skills of an architecture expert. The design space ranges from a variety of NDRange kernels, as used on Graphics Processing Unit (GPU) architectures, to highly optimized line-buffered implementations, as shown in Figure 1. A solution to this issue can be leveraging the algorithm description to an even higher language level, such as a Domain-Specific Language (DSL). Thereby, algorithms are described target-independently and the DSL compiler ensures the generation of efficient hardware implementations.

In this work, our contributions are as follows:

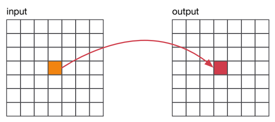
- We investigate the appropriate programming methodology for the Altera SDK for OpenCL (AOCL) by analyzing its synthesis and providing details on the effects of applying optimizations recommended by Altera.
- We propose a novel code generation for AOCL from a DSL by applying the programming methodology resulting from our investigations.

II. Background

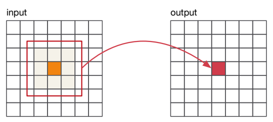
The aim of this section is to give a brief overview of the algorithmic domain, we focus on, as well as to introduce architectures and tools relevant for the remainder of this work.

A. *Domain of Image Processing*

Many image processing algorithms can be described as an image processing pipeline, a concatenation of *point*, *local*, or *global* operators. In this work, we focus on point and local operators. Point operators process only a single input pixel for every single output pixel they produce, as illustrated by Figure 2a. Typical applications for point operators are color space conversions. On the other hand, local operators process



(a) Point operator



(b) Local operator

Figure 2: Memory access patterns for point and local operators in image processing.

neighboring pixels within a static region, the local window, in order to produce a single output pixel. Pixels within that region are usually weighted by mask coefficients, such as those of the Gaussian filter, before aggregating them to a single output pixel. The process performed by local operators is depicted in Figure 2b.

B. *Open Computing Language (OpenCL)*

OpenCL was originally developed for massively data-parallel architectures, mostly inspired by GPUs. Therefore, the applied programming concept also follows a highly data-parallel programming paradigm, meaning source codes, so-called *kernels*, usually only describe computations that are processed by a single thread. All computations are data-independent and the creation and scheduling of threads is initiated and managed by the OpenCL runtime. How many threads are spawned depends on a range (1D, 2D, or 3D), specified by the developer. Kernels that are meant to perform across a specified range are called *NDRange kernels*.

C. *Altera Software Development Kit (SDK) for OpenCL*

The Altera SDK for OpenCL promises three fundamental advantages over conventional hardware design: (a) Simple structuring of Single Program Multiple Data (SPMD) applications; (b) portability to other architectures; and (c) a high level of abstraction for describing hardware.

Due to OpenCL's data-parallel programming paradigm, the first claim can be confirmed. For the second claim, as OpenCL is an industry standard, one can rely on functional portability across different architectures. However, performance portability, in the sense of achieving efficient results on other architectures, cannot be ensured. Developers have to choose between an SPMD implementation, which is suitable for many-core architectures, and a pipelined implementation, which we discuss in Section III-A. Nevertheless, employing OpenCL for FPGAs proves to be very attractive considering the third claim. Besides offering an abstract programming interface through HLS, OpenCL goes even further by strictly distinguishing between host, the system's CPU, and the device, the FPGA accelerator, on heterogeneous systems.

The Altera Offline Compiler (AOC) offers the *unroll* pragma to further guide synthesis. There, loops, without loop-carried data dependencies, can be unrolled either completely or by a specific factor. Altera also provides other keywords to increase overall throughput by applying automatic replication of either the entire accelerator or just the innermost kernel computation. A visualization of the former and the latter can be found in Figures 3 and 4, respectively.

III. Programming Methodology

Besides Altera being able to synthesize plain OpenCL code, significantly better hardware implementations can be achieved

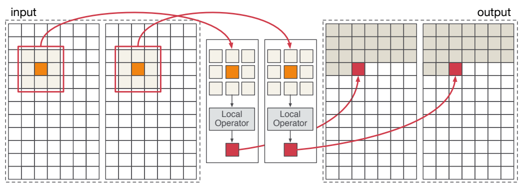


Figure 3: Automatic replication of the entire accelerator by specifying `num_compute_units(`n`)`.

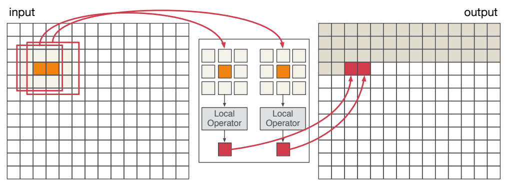


Figure 4: Automatic replication of the innermost kernel computation by specifying `num_simd_work_items(`n`)`.

with a certain programming style. The intention of this section is to provide a brief summary of best practices for Altera OpenCL. Some of them are recommended by Altera themselves, for which we will detail their effects with results. Others are original findings that have been revealed throughout our investigations.

A. *Single Work-Item Kernel with Line Buffering*

NDRange kernels are highly portable, as they are written in a style very similar to GPU programming, and therefore would also perform well on such architectures. However, for synthesis, Altera recommends employing *single work-item kernels* in combination with line buffering instead of NDRange kernels.

Single work-item kernels are not automatically executed across a specified range by the OpenCL runtime. Instead, the developer has to explicitly describe the range and the order in which that range is processed in the kernel's source code. Consequently, the number of threads that are spawned by the OpenCL runtime is exactly one. Thereby, performance portability to massively parallel many-core architectures, such as GPUs, is entirely negated. However, single work-item kernels open up the possibility for further FPGA-specific optimizations, as data locality can be explicitly exploited.

Line buffering is a well-known design methodology that is particularly beneficial for local operators in image processing, such as convolution, consisting of local operators, as shown in Figure 2b. A local operator of size $n \times m$ might require the pixel at $(x - n/2, y - m/2)$ to calculate the output for (x, y). However, images are mostly captured and stored in raster order. Furthermore, reading data from DRAM memory is far faster for consecutive access. In order to process one pixel at each read without compromising memory access speed, the minimum number of pixels satisfying the dependencies of the first pixel are stored to on-chip memory, M10K, through $m - 1$ many First In First Out (FIFO) buffers for reuse by later iterations. An $n \times m$ dimensional sliding window traverses over the image in raster order by reading one pixel from the off-chip memory, while the $m - 1$ many pixels are loaded from row buffers.

Utilizing a single work-item kernel in combination with line buffering yields significantly better hardware implementations than NDRange kernels, as shown in Figure 1. However, as a downside, the automatic replication of entire accelerators or innermost kernel computations through Altera-specific keywords

Table I: Comparison of different boundary conditions, implemented in software (SW) and hardware (HW) manner for a 5×5 Gaussian blur on a 1024×1024 image.

Kernel	II	ALUTs	Registers	Logic (%)	M10K	DSP	Freq [MHz]
SW-UNDEF	1	8012	8981	16.15	51	2	148.31
SW-CONST	1	8989	9486	17.55	52	2	142.27
SW-CLAMP	1	18595	11369	41.11	52	12	136.41
SW-MIRROR	1	19004	11488	41.42	52	12	135.68
HW-UNDEF	1	8012	8983	16.18	51	2	148.82
HW-CONST	1	8364	9232	16.71	52	2	152.35
HW-CLAMP	1	8570	9276	17.03	52	2	132.07
HW-MIRROR	1	8586	9431	17.21	52	2	148.55

can no longer be applied anymore. Contrary to expectations, for some cases, we even noticed minor benefits by moving from the NDRange to single work-item kernels even without line buffering.

B. *Developing Software in Hardware Manner*

The most beneficial methodology for AOC is to develop single-item line-buffered kernels as already explained in Section III-A. Programming a loop including shift registers for row buffers and sliding window is enough for implementing a basic line-buffered pipeline. Yet, we discovered that obeying hardware development mentality instead of classical software programming yields far better results. In this context we propose two main principles: (I) make the assignments as static as possible by first maintaining all corner cases and deciding which one to choose at the end; and (II) exploit the temporal locality in the program to avoid unnecessary corner cases.

An abstract representation of the first principle is given below. Assume that a program consists of some operations followed by an assignment in a conditional scope. Splitting the scope into two parts by moving all operations, except the assignment, to outside of the conditional scope strictly resembles a Multiplexer (MUX), its source and the selection.

```
a = calculate_a(...);
if (condition){
  /* calculate_a(...) has been moved from here */
  out = a;
}
```

Applying the principle above leads to significant improvements, especially when the excluded operation, represented as `calculate_a`, consists of an inner condition. A recursive employment of the principle through all inner conditions results in a program that maintains valid data for all corner cases and selects the correct case at the end. This paradigm contradicts good software development practice since it burdens the program with redundant operations and additional variable assignments.

Loops that consist of one-level deep conditional assignments can become a useful tool for applying the presented principle when they are used with AOC's unroll pragma. On the other hand, we observed that they must always be programmed to have constant bounds, even at the cost of additional outer loops or inner conditions. For instance, in the example below, instead of choosing `i` as the inner loop's upper bound, we specify a constant and insert an additional condition.

```
for (uint i = 0; i < 5; ++i)
  for (uint j = 0; j < 5; ++j)
    if (j < i) { /* operations */ }
```

Most of the image processing algorithms have great potential for the second principle type of optimizations. A very basic illustration is given below.

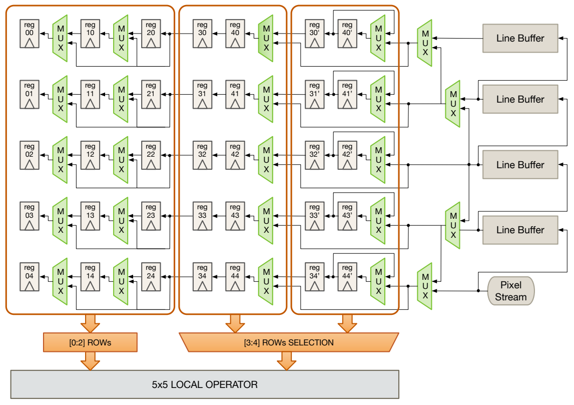


Figure 5: Described hardware architecture of sliding window for CLAMP boundary handling.

```
x = y;      // x = in[i-2];
y = z;      // y = in[i-1];
z = in[i];
out = compute(x, y, z);
```

Any knowledge of the schedule should be used to simplify the assignments, conditions or arithmetic operations. A few examples are: (a) Minimizing arithmetic operations by using intermediate results; and (b) having 1-bit control signals updated in corner cases instead of having multiple checks.

Besides the mentality discussed so far, strength reduction, replacing expensive operations like multiplications with shifts and adds, makes a considerable difference as already recommended by Altera. In our experience, writing a balanced tree for the integer arithmetic operations does not lead to any better results.

It is worth mentioning that the quality of results highly depend on the coding style. Using an additional intermediate variable instead of reading an element of a local array multiple times proved to perform well for most of the cases. Avoiding the else branch of an if-statement, i. e., removing the else branch and moving its content to the preceding improved the area.

Implementing boundary conditions for a basic line-buffered convolution filter is a good proof of concept for discussing the principles we propose in this section. A common software approach to writing the conditions for clamp is given below, in which `W`/`H`, `w`/`h` are the image and half of the filter width/height, respectively.

```
sliding_window[ (y + j < h) ? (h - y) :
            (y+j < H-1 + h) ? j : (H-1+h-y)
][              (x + i < w) ? (w - x) :
            (x+i < W-1 + w) ? i : (W-1+w-x) ]
```

Implementation results for all boundary conditions following a similar approach is given in Table I (SW). Considering that the logic used only for Input/Output (IO) management is around 14 %, as discussed in Section III-C, additional logic for undefined boundary implementation can be recognized as severe.

Following software approaches also lead to costly implementation results: (a) Applying the upper conditions to variables instead of indices; (b) describing all corners in 1-level conditions and having all assignments without any loops; and (c) partitioning the image to regions according to boundary conditions, storing pre-computed indices for all regions to a constant array, and fetching the indices using address variables that are changed only when the current iteration enters a new region.

To achieve better results, all conditions written using the software approach are strictly transformed according to the

Table II: Impact of channels on line-buffered single-work item implementations.

Kernel	Type	II	ALUTs	Registers	Logic (%)	M10K	DSP	Freq [MHz]
Copy Kernel	single	1	6821	7812	14.04	43	0	152.89
Mean Filter	single	1	8248	9415	16.78	51	0	151.52
2 Mean Filter	separate	1	12104	15132	25.36	100	0	145.78
2 Mean Filter	channel	1	10439	12466	21.47	60	0	154.14
2 Mean Filter	fused	1	9480	11715	19.47	64	0	122.47
3 Mean Filter	channel	1	12829	15451	26.56	67	0	145.57
3 Mean Filter	fused	1	10899	13651	22.24	79	0	125.00
Luma+Mean	single	1	8244	9694	16.84	66	2	132.56
Luma+Mean	channel	1	11281	13653	23.75	64	3	148.75
Luma+Mean	fused	1	9635	11846	20.09	72	2	123.53
Harris corner	channel	1	43127	54334	89.87	186	9	142.52
Harris corner	fused	1	33263	43319	70.87	177	9	107.35

first principle. Recursively applying this principle leads to the assignments below with respect to the current y coordinate.

```
for (uint j = 0; j < h; ++j){
  if (y<=j)       wnd[h-1-j][2*w] = in[h-j];
  if (y>=H-1-j)   wnd[h+1+j][2*w] = in[h+j];
}
```

The code consists of redundant assignments without nested conditions. For a single column in a 5×5 window, this leads to a chain of 9 MUXs, the green logic between the row buffer and the window in Figure 5 on the right. Considering 5 columns leads to 45 MUXs, just to satisfy the conditions depending on the y coordinate. In the next step, the same conditions need to be applied for the x coordinate, in total resulting in 90 MUXs.

Exploiting the temporal locality of the raster order process, the second principle, by inserting register assignments degrades the number of MUXs to 49 in total. This is achieved by deploying two additional registers per row in order to calculate and maintain valid data for all corner cases depending on the x coordinate. At the end, the correct case is selected, which we denote as *window-selection*. The execution of the corresponding program can be best described with the hardware architecture given in Figure 5. The results of our implementations for different boundary conditions following the proposed principles are presented in Table I (HW). Implementing boundary conditions for a 5×5 Gaussian kernel only employs minor supplementary resources at just 0.85 % and 1.03 % additional logic for clamping and mirroring on the contrary of a regular software approach (SW).

C. Altera OpenCL System Level Interfaces

Communication between kernels is essential for hardware design to harness the benefits of pipelining, unlike GPUs whereby data is transferred to global memory before and after kernel execution. Yet, this has only become available with the OpenCL 2.0 release [1] through *pipe* memory objects. Besides offering a preliminary support for OpenCL pipe structures, Altera SDK for OpenCL provides an extension, *channels*, for not only passing data between kernels but also synchronizing them. Channels are basically FIFO buffers with an additional feature of stalling the kernel that attempts to read data while it is empty or write when it is full. The data written to a channel is preserved as long as its program remains loaded on the device. The concurrent execution of different kernels communicating by channels are facilitated by separate command queues on the host side.

Unfortunately, Altera OpenCL SDK still lacks the generation of a detailed report that shows individual results for the kernels or the overhead of system management, i. e., IO logic. Although all system integration projects and the generated Hardware Description Language (HDL) files are left after the synthesis, distinguishing the data path of a kernel from generated HDL files is not simple and error-prone. Therefore, we conducted a brief analysis on channel implementations coupling single work-item kernels. The keywords single, separate, channel and fused in Table II are used to refer the type of implementations with only one kernel, multiple independent kernels with host control, multiple kernels coupled with channels and one kernel generated by the fusion of multiple kernels, respectively.

The single-item copy kernel consists of two IO buffers. It reads and writes 1-byte in a loop with a constant bound, and therefore it can be treated as a minimal kernel having only IO logic. The mean filter (5×5) consists of a row buffer, a sliding window, and convolution logic, fully parallel addition preceded by a division. The resource usage of the convolution in mean filter can be approximately calculated by subtracting the implementation results of the mean filter and copy kernel, resulting 2.04 % of logic and 8 M10Ks. On the other hand, implementing a channel instead of an IO buffer reduces the required logic as can be derived with a comparison of the separate and channel implementations of 2 mean filters. The linear increase in resource consumption according to the number of channels can be observed in the results from channel type implementations of 2 and 3 mean filters. The results reveal that channel implementations have deterministic overhead.

Despite channels eliminating IO overhead, fusing kernels to remove channels yields significantly better results in terms of area. Luma+Mean in Table II consists of two point operators for color conversion and thresholding together with a mean filter in between. The version using channels is formed by splitting the single version into three kernels without any code modifications. We argue that the difference between the two implementations, logic utilization of 6.91 %, are reserved for two one-element integer channels.

We developed a tool, simply for the analyses, to automatically fuse kernels that are coupled by one-element channels. The fusion protects all variable declarations and the operations through the process. As a result, we expect to obtain an implementation with the only difference that channels are replaced with registers. The fusion performs three steps to unify all kernels into one without further optimizations: (a) For every channel a new local variable and a validity flag is described. The validity is enabled when the corresponding channel is written or read for the first time. (b) All read/write statements of channels and their conditions are replaced with corresponding register assignments and validity checks, respectively. (c) Kernel programs are sequentially ordered under the last kernel after they are encapsulated in private scopes.

Implementation results illustrate that the opportunity cost of a channel becomes more noticeable when it is used to couple pixel operators, as in Luma+Mean, since all synchronization becomes redundant. As it is presented through harris corner implementation, channels can consume a considerable amount of area when an application is split into small kernels. Finally, we observed that keeping the size of a kernel code short significantly improves the clock frequency. This might be due to the optimization heuristics of AOC. Through these experiments we also revealed that major improvements can be achieved by automatically transforming channels to simple registers where they are used only to send one element without complicated handshake protocols. Moreover, the number of required channels decreases in this way since multiple channels must be utilized when multiple kernels need to read the same data from a preceding kernel through FIFO buffers.

In light of acquired knowledge, we chose to use channels in our DSL-based code generation since communicating through channels shortens the code and increases the clock frequency. Not only is the generated code less dependent on compiler optimizations in this way, but also it can benefit from AOC's future improvements.

IV. Code Generation for Altera OpenCL

In this work, we present a novel code generation for Altera OpenCL. Here, the aforementioned optimizations have been applied to generate kernel codes and provide appropriate building blocks. For code generation, we employ the Hipacc framework for which we extended its OpenCL back end accordingly to support Altera-specific peculiarities.

A. Heterogeneous Image Processing Acceleration (Hipacc)

The Hipacc framework (hipacc-lang.org) was originally designed to target GPUs [2]. It consists of a DSL for image processing and a source-to-source compiler, based on the compiler infrastructure Clang/LLVM 3.6. The DSL is embedded into C++ and its use strongly resembles that of using a regular image processing library. However, with the aid of the source-to-source compiler, large portions of the code are rewritten and replaced with target-specific generated code. Supported target languages are Compute Unified Device Architecture (CUDA), OpenCL for GPUs, Renderscript, and a specific kind of C++, which is suitable for synthesis with Xilinx' Vivado HLS.

B. Generating a Streaming Pipeline

As Hipacc was originally designed to target GPUs, it employs a buffer-wise execution model. In a larger image processing pipeline, which embodies multiple kernels, each kernel reads from and writes to buffers sequentially. In doing so, every succeeding kernel starts only after all previous kernels in the pipeline have finished their execution. Data exchange between kernels is accomplished through buffers that have the size of the entire image to be processed. This is a valid approach for GPUs but rather inappropriate for FPGAs.

For FPGAs, the buffer-wise execution model must be transformed into a streaming pipeline. This issue addressed in a previous work [3] in order to support code generation for Vivado HLS. Thereby, the use of every buffer is translated to an internal representation in Static Single Assignment (SSA) manner, meaning that reused buffers are treated as new ones. To construct a streaming pipeline, the buffers are then replaced by streaming objects, implementing FIFO semantics. However, for buffers that are read by n many kernels, $n+1$ many streams will be instantiated. This is because a single stream is dedicated to the output of the previous kernel and n many streams are inserted as input for the n many succeeding kernels. In between those streams, glue logic (which we call *splitStream*) is generated to read a single data element from the output stream and maintains its value through propagation to multiple input streams.

This is a practical approach for Vivado HLS, as the insertion of an additional output stream does not have any significant impact on the synthesis results. For Altera OpenCL, however, a single additional channel, which is Altera's implementation of streaming objects, does have severe impact on synthesis results, as already shown in Section III-C. Furthermore, an additional *splitStream* kernel is needed to implement the necessary glue logic for maintaining the output value and propagating it to the succeeding kernels' input channels. This additional *splitStream* kernel invocation would introduce a serious overhead, which is considerably noticeable in synthesis results.

For Altera OpenCL, we modified Hipacc's internal representation of the streaming pipeline to support writing to multiple streams, even though Hipacc itself does not support writing to multiple buffers. To achieve this, we analyze the existing internal representation, locate the occurrence of any *splitStream* logic, and transform it to the modified internal representation by merging it with the previous kernel. Thereby, reads with n many kernels result in n many channels without the need for an additional *splitStream* kernel, which maintains the generation of an efficient streaming pipeline for Altera OpenCL.

C. Kernel Vectorization

Altera's automatic replication keywords cannot be used for vectorization because of the line buffers in the generated streaming pipeline. Nevertheless, replication can be applied through code generation to still be able to increase overall throughput. Innermost kernel replication has already been shown to be favored over accelerator replication for local operators [4]. This is achieved by partitioning input data for accelerator replication, as indicated by Figure 3, which eliminates redundant computations on overlapping data in a larger image processing pipeline. Furthermore, the HLS compiler can optimize and eliminate redundant computations within multiple merged local windows when replicating only the innermost kernel. Therefore, for code generation, we employ kernel vectorization through by the innermost kernel and merging their local windows, which can be enabled by specifying a *vectorization factor* as a compiler switch. Results show that the growth in hardware utilization is sublinear to the increase in throughput, as discussed in Section V-D.

D. Bit Width Reduction

Altera OpenCL provides the ability to reduce integer data types to an arbitrary bit width. The OpenCL standard, however, only contains primitive data types of fixed width, such as `char`/`short`/`int` with respective bit widths of 8/16/32. As Altera claims that standard-conform OpenCL codes can be used for hardware synthesis, demanding the use of newly introduced arbitrary bit width data types would contradict this claim. Therefore, to guide the AOC to reduce the bit width of a variable, a bitwise AND operation with a specific bit mask must be applied. This way, bit width reduction can be transparently specified without functional changes and without losing portability.

For instance, to reduce the variable `x` to a width of 3 bits, a masking operation such as `(x & 0x7)` must be applied to any occurrence of that variable. Moreover, if such a variable is modified through an assignment operator, the entire Right Hand Side (RHS) expression must be masked as well:

```
x = (RHS) & 0x7;
```

Furthermore, compound assignment operations need to be split into the corresponding arithmetic operation followed by the assignment operation. For both resulting operations, masking needs to be applied. Thereby, the operation `x +=` *RHS* expands to the following:

```
x = ((x & 0x7) + RHS) & 0x7;
```

Listing 1: Gaussian blur in Hipacc with annotated bit widths

```
#pragma hipacc bw(sum,12)
uint sum = 0;
#pragma hipacc bw(x,2)
uint x = 0;
#pragma hipacc bw(y,2)
uint y = 0;
for (y = 0; y < size; ++y) {
  for (x = 0; x < size; ++x) {
    sum += mask[y][x] * Input(x-1,y-1);
  }
}
sum /= 16;
output() = sum;
```

Listing 2: Generated Gaussian blur with bit width reduction

```
uint sum = 0;
uint x = 0;
uint y = 0;
for (y = ((0) & 3); ((y) & 3) < size; y = ((((y) & 3) +
    1) & 3)) {
  for (x = ((0) & 3); ((x) & 3) < size; x = ((((x) & 3)
    + 1) & 3)) {
    sum = ((((sum) & 4095) + mask[y][x] * getWindowAt(
      Input, 1 + ((x) & 3) - 1, 1 + ((y) & 3) - 1)) &
      4095);
  }
}
sum = ((((sum) & 4095) / 16) & 4095);
return sum;
```

As a matter of fact, the exact same applies to unary increment/decrement operators, such as x++ or --x.

To enable the annotation of arbitrary bit widths in Hipacc, we introduce DSL-specific pragmas that must be specified in the code line just above the variable declaration. Pragmas that are not known to the compiler are entirely ignored and are therefore a perfect fit for defining target-specific annotations. In our implementation, the desired bit width must be specified, as well as the name of the variable, to achieve a loose binding to the variable declaration in the next line. Taking the example above, the required annotation could look like the following:

```
#pragma hipacc bw(x,3)
uint x;
```

Specifying this pragma will instruct Hipacc to internally mark this variable as having a reduced bit width. This information is then put aside, until all target-specific transformations on the Abstract Syntax Tree (AST) have been applied. In the very last stage, just before pretty printing the AST to the target language's source code, the following actions are executed in this order:

1) All compound assignment and unary increment/decrement operations on a marked variable are transformed to the corresponding arithmetic operation followed by the assignment.
2) Every read operation of a marked variable is masked.
3) Every write operation of a marked variable is transformed in a way that its entire RHS expression is masked.

The bit width reduction, specified by pragmas, is valid throughout the entire kernel. Thereby, code generation addresses inserting the appropriate masking operations at all relevant locations in the source code. The resulting source codes contain plenty of operations that do not affect functional correctness and therefore become very hard to read. An example of a Gaussian kernel written in Hipacc can be found in Listing 1 and the corresponding generated code is shown in Listing 2.

Table III: Synthesis results for applying automatic bit width reduction on local operators of size 3×3 with image size 1024×1024.

Filter	Type	II	ALUTs	Registers	Logic (%)	M10K	DSP	Freq [MHz]
Sobel	normal	2	8552	12433	19.76	84	4	131.25
	reduced	2	8230	12098	19.11	84	3	152.09
Gaussian	normal	2	8593	12423	19.86	84	4	132.50
	reduced	2	8439	11843	19.26	83	3	153.11

The AOC already does a good job when it comes to bit width reduction. If inner loop trip counts are known during compile time and the unroll directive has been specified for synthesis, we were not able to achieve any better results through manually specified bit width reduction. However, if data ranges cannot be statically analyzed, major benefits can be achieved. Resource consumption decreases since fewer registers are deployed to store the data and less logic is used to implement the operations between them. More importantly, the speed is significantly higher, since the longest path between two Flip Flops (FFs) determines the achievable clock frequency of the whole circuit and the carry logic of an arithmetic operation is mostly the longest path and increases with the bit width. Synthesis results for applying bit width reduction to a Sobel operator and a Gaussian blur filter (Listing 1) can be found in Table III.

V. Evaluation and Results

We evaluate Hipacc's code generation for AOCL by investigating implementation results for varying image filters and also by comparing Altera's example designs with those we generated Hipacc DSL descriptions of the same algorithms. Additionally, we compare our implementations in terms of performance to different hardware targets. All implementations were generated by Hipacc for each target, originating from exactly the same DSL source code.

A. Evaluation Environment

First, we will give a brief overview of the hardware targets used for evaluation.

Altera Stratix V DE-5 A High-end, 28 nm FPGA (5SGXEA7), whose clock network is designed to support up to 800 MHz fabric operations. It possesses 622 K Logic Elements (LE), 2,562 M20K on-chip memory blocks, 256 27×27 DSP blocks and 512 18×18 multipliers.

NVIDIA Tegra K1 A Multi-Processor System on Chip (MPSoC), hosting a quad-core ARM Cortex A15 CPU and an embedded GPU. The GPU contains 192 scalar compute cores, clocked at 960 MHz.

NVIDIA Tesla K20 A discrete server-grade GPU, solely targeting computing. It possesses 2,496 scalar compute cores, distributed among 13 multi-processors, each running at 705 MHz.

As a side note, all our previous findings were performed on Cyclone V. However, for the entire evaluation section, we chose to switch to Stratix V to ensure a fair comparison with server-grade GPUs.

B. Algorithms

In the following, algorithms that were chosen for evaluation throughout this section are briefly introduced.

GB The Gaussian blur, first, applies a 3×3 convolution with an unsigned integer mask, then, divides the result by 16.

Table IV: Comparison of Altera's design examples and Hipacc for on an image of size 1024 × 1024.

Filter	Source	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
LSB	Altera	1	43844	67418	19.59	347	18	305.42
	Hipacc	1	45069	68624	19.95	346	3	304.50
LK	Altera	1	53768	84746	22.59	462	47	212.03
	Hipacc	1	63595	92810	24.87	562	37	289.85

LP The Laplacian detects vertical, horizontal, and diagonal edges using signed integer arithmetic with a 5×5 local operator.
SB The Sobel, first, computes vertical and horizontal derivatives with 3×3 masks, then, calculates the Euclidean distance and clamps with a given threshold in the 3rd kernel to detect edges.
LSB An edge detection that first transforms RGB input to LUMA then applies x and y Sobel filters and finally clamps the sum of the absolutes of derivatives according to a threshold.
BL The bilateral filter [5] is a 3×3 local operator used for reducing noise while preserving edges. It consists of an exponential function and employs floating point arithmetic.
HC The Harris corner consists of 9 kernels. First, the horizontal and vertical derivate of input image are computed, then, the derivatives are squared and multiplied with each other within 3 point operators. Resulting images are blurred by 3 Gaussian kernels. Finally a point operator detect corners by clamping with a threshold after the calculating the determinant.
OF The optical flow issues a Gaussian blur and computes a bit vector signature for every pixel of two input images using the census transform [6]. Those signatures are then compared within a 15×15 window to find corresponding points, and therefore detect the optical flow. Overall 5 kernel executions are involved.
LK Lucas Kanade [7] is a dense optical flow that utilizes 9 kernels with floating point arithmetic. First difference of two input images as well as their derivatives are computed. Then their multiplications with each other are accumulated in a 7×7 window to find motion vectors. Finally, corresponding *Munsell* color indexes, including square and arctangent, are calculated.

C. Comparison to Altera Examples

Table IV presents the implementation results for the image processing design examples provided by Altera and the corresponding Hipacc implementations. The design examples consist of single work-item line buffered kernels as well as Hipacc's. Besides the functionality, all arithmetic operations in Hipacc's operators are written exactly same as the design examples for the sake of comparison. On the other hand, the device code of Hipacc's implementations are assembled from two kernels (LSB) and four kernels (LK), coupled by channels, whereas Altera's are from a single kernel only.

Hipacc's LSB uses significantly fewer DSP blocks and slightly fewer logic in exchange for more on-chip memory, M10K. Moreover, the difference in clock frequency is negligible. On the other hand, Hipacc's LK consumes 1.4× M20K blocks of the Altera design. In exchange, Hipacc's clock frequency is significantly better. We could argue that the difference in logic utilization is acceptable when employed Digital Signal Processings (DSPs) are taken into account. The reason for obtaining considerably different results from LK, on the contrary to those from LSB, are algorithm-specific manual optimizations, in which the local operators are transformed to shift registers.

It is mostly possible to yield better results with enough time and expert knowledge. Yet, the results close to hand-optimized codes confirm Hipacc's success. Moreover, applying boundary conditions, which is mostly a requirement for LK, or efficiently increasing the throughput via vectorization is trivial for Hipacc, whereas Altera's designs require drastic changes.

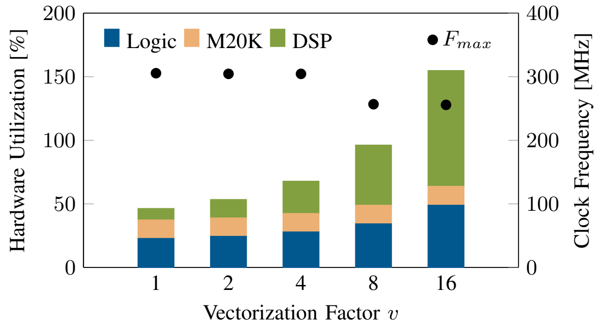


Figure 6: Hardware utilization for the vectorization of a 3×3 bilateral filter with clamping on an image of size 1024 × 1024.

Table V: Synthesis results for different vectorization factors v of a 3×3 bilateral filter with clamping on an image of size 1024 × 1024.

v	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
1	1	54490	82780	22.77	373	23	305.53
2	1	60283	89573	24.45	371	37	304.50
4	1	71772	103836	27.91	371	65	304.50
8	1	92927	118080	34.28	375	121	256.73
16	1	140368	189010	48.93	381	233	255.75

D. Kernel Vectorization

Results for the developed vectorization can be found in Figure 6 with exact numbers given in Table V. It can be observed that a speedup of roughly 13.4 can be achieved by an increase of only 2.1× in logic utilization and 10.1× in DSP blocks thanks to Hipacc's vectorization engine. As already explained in Section IV-C and shown in Figure 4, vectorization of an $n \times m$ local operator to v should increase the sliding window size and the operation by pixels to $(n + v - 1) \cdot m$. Any remaining logic, e. g., line buffer, should only facilitate supporting the memory bandwidth. Besides the sublinear increase in logic, even the increase in DSPs is close to half of the vectorization ratio. Therefore, applying vectorization to single work-item kernels with line buffering proves to be exceptionally beneficial, in particular, if vectorization can be applied through code generation, without the necessity of modifying the algorithm's source code.

E. FPGA vs. GPU

As all OpenCL codes used for evaluation are generated by a domain-specific compiler, we are able to generate GPU implementations from exactly the same DSL source code. We chose to generate CUDA implementations due to the lack of OpenCL support in recent versions of NVIDIA's programming environment. All GPU implementations were carefully tuned by considering the use of shared memory and texture memory as well as by applying optimizations, such as thread-coarsening, to ensure a fair comparison.

Our throughput comparison to Tegra K1 and Tesla K20 can be found in Figure 7. Exact numbers for the presented Stratix V implementations are given in Table VI. Algorithms are listed in order of complexity, mainly in terms of required bandwidth. In particular, the performance of GPU architectures suffers from an increasing amount of memory accesses. Therefore, the GPU throughput for the Gaussian blur, with a small 3×3 local

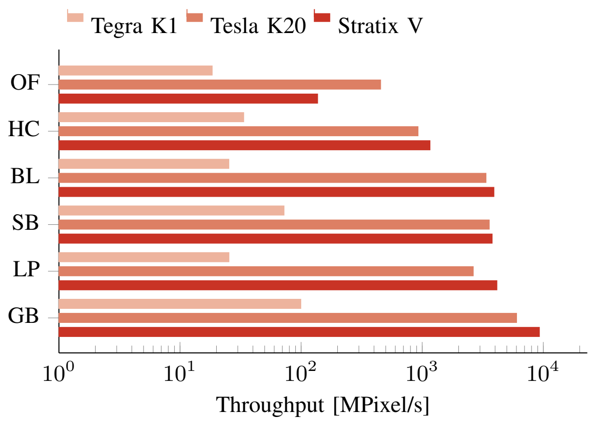


Figure 7: Comparison of throughput for the NVIDIA Tegra K1, NVIDIA Tesla K20, and Altera Stratix V.

window, is rather high compared to the optical flow, for which a 15×15 local window needs to be processed.

On the other hand, the Stratix V is less severely affected by enlarged local window sizes. As a matter of fact, line buffering structures increase, leading to almost double the M20K usage when comparing the Gaussian blur with the optical flow. However, other bottlenecks are more severe, such as relying on signed integer arithmetic (LP), as well as the overhead introduced by channels when deploying a filter chain involving multiple data-dependent kernels (SB, HC, OF). Consequently, we were unable to increase the vectorization factor v to more than 4 without sacrificing an Initiation Interval (II) of 1. Unfortunately, even without vectorization, synthesis was not able to maintain an II of one 1 for the optical flow. Nevertheless, for the Gaussian blur, bilateral filter, and Harris corner, the Stratix V was even able to outperform the server-grade Tesla K20 GPU.

VI. Related Work

Early work that proposed the compilation of a data-parallel programming model to hardware implementations was presented with FCUDA by Papakonstantinou *et al.* [8]. Here, CUDA, heavily driven by GPU industry and the main inspiration for the OpenCL standard, was adapted to serve as a source language. Employing a source-to-source compiler, CUDA thread blocks are transformed into parallel C code for Autopilot [9], which is the predecessor of Xilinx' Vivado HLS. The Throughput Oriented Performance Porting (TOPP) framework [10] optimizes memory access by decreasing the lifetime of variables and removing synchronization points using FCUDA annotations. Shagrithaya *et al.* [11] and Kępa *et al.* [12] use Vivado HLS for kernel code generation and focus on designing application-specific kernel interfaces for fine-grained parallelism. Altera, on the other hand, targets royalty-free OpenCL to extract the thread-level parallelism that is specified by the developer. The compiler uses the LLVM infrastructure to transform kernels into dedicated hardware pipeline circuits that can be replicated many times [13]. Altera OpenCL SDK implements a complete heterogeneous system by facilitating all required memory and host interfaces for the targeted device via the integration tools QSYS and Quartus. Silicon-OpenCL (SOpenCL) [14] is another LLVM-based compiler that automatically adjusts the parallelism level of a kernel for a specified FPGA using hardware templates. Gurumani *et al.* [15] uses application-specific knowledge as

Table VI: Synthesis results of multiple algorithms with different vectorization factors v.

Filter	v	#Kernels	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
GB	32	1	1	47045	73584	20.64	363	0	303.58
LP	16	1	1	107310	142069	39.69	419	64	270.62
SB	16	3	1	58308	96673	24.59	497	96	247.34
BL	16	1	1	140368	189010	48.93	381	233	255.75
HC	4	9	1	135808	192397	45.86	493	36	303.39
OF	1	5	2	81551	128816	32.61	646	18	286.36

well as analytical models for design space exploration, which is too complex in the case of multiple communicating kernels.

Despite the significant improvements in HLS methodologies and tools, hardware design expertise is still a game changer in terms of good results. A recently proposed approach to overcoming this obstacle is to employ a DSL to leverage the algorithm description even at a higher abstraction level. Whereas Darkroom [16] uses an image processing DSL to directly generate HDL, Hipacc [3] utilizes Vivado HLS for the same domain. George *et al.* [17], [18] use the same approach to target machine learning applications. Both have been presented roughly at the same time and demonstrated impressive results. Furthermore, additional improvements that benefit from domain-specific restrictions even lead to better implementation results. Among other contributions, our work is unique for being the first DSL approach that implements a completely heterogeneous system based on OpenCL.

VII. Conclusion

In this work, we presented a thorough investigation into the Altera SDK for OpenCL. As such, to the best of our knowledge, we are the first to provide programming style guidelines that go far beyond the recommendations given by Altera. Moreover, we provide detailed analyses regarding kernel fusion and the impact of channels in Altera OpenCL, revealing that both have their strengths and weaknesses, depending on whether the design goal is less area utilization or a higher clock frequency. Furthermore, we developed a novel code generation for Altera OpenCL that uses the image processing DSL Hipacc to leverage algorithm descriptions to a higher level. Hipacc's streaming pipeline generation was enhanced to use fewer channels, which significantly improves area utilization as our findings show channels to be rather costly. In addition, a language extension was introduced to automatically apply bit width reduction to the generated code.

Overall, the presented results of our extensive evaluation are exceptional. We were able to show that our generic approach can lead to results close to those of hand-optimized Altera examples. Despite comparable results, the generative approach remains superior as core fragments of the implementation, such as boundary conditions or the vectorization factor, can be changed without severe modifications. Furthermore, we are able to outperform a server-grade GPU in terms of throughput for a wide variety of image filter algorithms. In conclusion, our proposed code generation from a DSL represents an additional advancement to Altera OpenCL as it eases the burden on the developer of writing target-specific code while simultaneously being able to attain outstanding results.

Acknowledgment

This work is supported by the German Research Foundation (DFG), as part of the Research Training Group 1773 "Heterogeneous Image Systems". The Tesla K20 used for this research was donated by the NVIDIA Corporation.

References

[1] *The OpenCL specification, version: 2.0 document revision: 22*, http://www.khronos.org/registry/cl/specs/opencl-1.0, Khronos OpenCL Working Group, 2014.

[2] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, Jan. 1, 2016. DOI: 10.1109/TPDS.2015.2394802.

[3] 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 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, (New Dehli, India), Oct. 12–17, 2014. DOI: 10.1145/2656075.2656081.

[4] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in C-based high-level synthesis", in *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, (Toronto, Canada), IEEE, Jul. 27–29, 2015, pp. 166–173. DOI: 10.1109/ASAP.2015.7245730.

[5] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images", in *Proceedings of the Sixth International Conference on Computer Vision (ICCV)*, IEEE, 1998, pp. 839–846.

[6] F. Stein, "Efficient computation of optical flow using the census transform", in *Pattern Recognition*, ser. Lecture Notes in Computer Science, vol. 3175, Springer, 2004, pp. 79–86. DOI: 10.1007/978-3-540-28649-3_10.

[7] J.-Y. Bouguet, "Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm", *Intel Corporation*, vol. 5, no. 1-10, p. 4, 2001.

[8] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs", in *Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP)*, (San Francisco, CA, USA), Jul. 27–28, 2009, pp. 35–42. DOI: 10.1109/SASP.2009.5226333.

[9] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based ESL synthesis system", in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds., Springer, 2008, pp. 99–112. DOI: 10.1007/978-1-4020-8588-8_6.

[10] A. Papakonstantinou, D. Chen, W.-M. Hwu, J. Cong, and Y. Liang, "Throughput-oriented kernel porting onto FPGAs", in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, (Austin, TX, USA), May 29–Jun. 7, 2013, 11:1–11:10. DOI: 10.1145/2463209.2488747.

[11] K. Shagrithaya, K. Kępa, and P. M. Athanas, "Enabling development of OpenCL applications on FPGA platforms", in *Proceedings of the IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, (Washington, DC, USA), IEEE, Jun. 5–7, 2013, pp. 26–30. DOI: 10.1109/ASAP.2013.6567546.

[12] K. Kępa, R. Soni, and P. M. Athanas, "Inferring custom architectures from OpenCL", in *Proceedings of the 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 1–4, 2015, pp. 9–16. DOI: 10.1109/PATMOS.2015.7347581.

[13] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs", in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, (Oslo, Norway), pp. 531–534. DOI: 10.1109/FPL.2012.6339272.

[14] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs", in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, IEEE, 2011, pp. 186–193.

[15] S. T. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen, "High-level synthesis of multiple dependent CUDA kernels on FPGA", in *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 22–25, 2013, pp. 305–312. DOI: 10.1109/ASPDAC.2013.6509613.

[16] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines", *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, 144:1–144:11, Jul. 2014. DOI: 10.1145/2601097.2601174.

[17] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware system synthesis from domain-specific languages", in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–8.

[18] N. George, H. Lee, D. Novo, M. Owaida, D. Andrews, K. Olukotun, and P. Ienne, "Automatic support for multi-module parallelism from computational patterns", in *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2015, pp. 1–8.