Rapid Customization of Image Processors Using Halide

Ville Korhonen, Pekka Jääskeläinen, Matias Koskela, Timo Viitanen, and Jarmo Takala
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
Email: {firstname.lastname}@tut.fi

***Abstract*—Image processing applications typically involve data-oriented kernels with limited control divergence. In order to efficiently exploit the data level parallelism, image processors include SIMD instructions and other parallel computation resources. Generic processors that can be purchased off-the-shelf are adequate for most of the use scenarios of image processing. However, especially with embedded mobile devices, they might not be optimal for the algorithm, the environment, or the energy budget at hand. Such cases call for programmable customized architectures with just enough hardware resources to ensure the high priority applications reach their real time goals with minimal overheads.**

In order to maintain high engineer productivity, implementing image algorithms for customized processors should be as easy as with standard processors. This is emphasized at the processor co-design time; because the program is used to drive the processor design space exploration towards an optimized architecture, assembly programming is not feasible due to the required porting effort whenever the architecture is modified.

In this paper we propose an image processor customization flow that exploits the domain-specific Halide language as an input to a processor co-design environment. In addition to efficiently exploiting standard resources in the customized processors, the flow provides an easy way to invoke special instructions from Halide programs. We validate the performance benefits of custom operations using example filters described with the Halide language.

I. Introduction

Applications from the image processing domain typically involve data-oriented kernels with limited control divergence. In order to efficiently exploit the abundant data level parallelism, typical commercial image processors include SIMD instructions and other forms of parallel computation resources. Due to high demand for processors with this style of computation capabilities, generic image processors that can be bought off-the-shelf are adequate for most of the image processor use scenarios. However, especially with embedded mobile devices, they might not be optimal for the algorithm, the environment or the energy budget at hand. Such cases call for programmable customized processors. Customized processors bring in the benefits of customized hardware, but without the inflexibility of "fixed-function" hardware pipeline implementations that can execute only a fixed set of functions and do not support on the field functionality updates.

Hardware-software co-design attempts to utilize characteristics of a single interesting application or an application domain to optimize the hardware together with the software [1]. Specialization allows optimizing the hardware to match better the known more limited application set. In case of processor co-design, customization points often include the type and number of function units with basic arithmetic operations, register storage, memory hierarchies, core counts and special instructions. These enable the engineer to tune the processor to match closely the requirements of the application without excessive resources that consume power and chip area.

In order to maintain good engineer productivity, implementing image algorithms for customized processors should be as easy as with standard processors. This need is emphasized at the processor co-design time which is usually an iterative process of modifying the architecture and the software, and evaluating the result. During processor co-design, the program is used to drive the processor design space exploration towards an optimized architecture, thus assembly programming is not feasible due to the required porting effort whenever the architecture is modified. In addition, the input program language should be descriptive enough to not lose important algorithm implementation information, such as parallel constructs to utilize parallel processor resources efficiently.

In this paper we propose an image processor customization flow that exploits the image domain-specific *Halide* [2] language as an input to a processor co-design environment. Halide is a functional language implemented on top of C++ classes. It has compiler backends to utilize data parallel computation resources such as SIMD instructions and GPUs efficiently. Its separation of the algorithm from the schedule/hardware mapping makes it especially interesting for a co-design flow. The ability to tune the schedule parameters (affected by the processor variation currently at hand) without touching the algorithm part allows a clean separation of target-specific parts from generic algorithm descriptions, making the co-design process less error-prone.

The proposed flow provides, in addition to an automated way to exploit standard datapath resources in the customized processors, a straightforward mechanism to invoke special instructions from Halide programs to benefit from custom hardware embedded to the designed processors. We demonstrate this capability with example cases.

The paper is organized as follows. Section II starts with explaining the design flow, focusing on the compilation flow and the custom operation usage. Section III presents example customization cases. Section IV reviews the related work, and Section V concludes the paper.

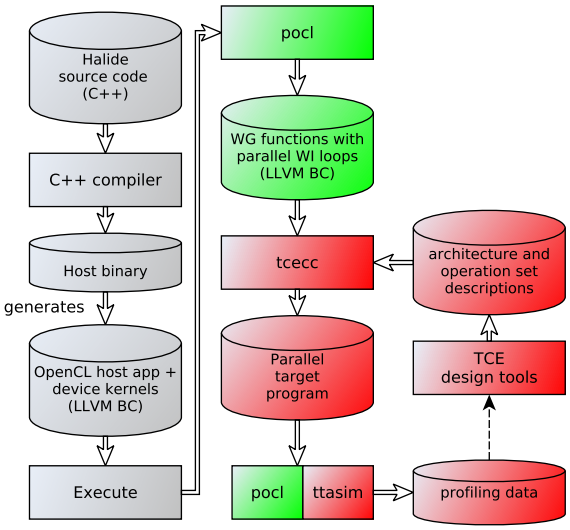

Fig. 1. The proposed customization flow utilizes three main components: Halide for generating OpenCL descriptions, pocl as an OpenCL implementation for the customized TCE cores and a retargetable compiler (tcecc) to generate efficient code for each designed TCE architecture.

II. From Halide to Customized Processors

The proposed co-design design flow (illustrated in Fig. 1) for mapping Halide programs to customized processors involves three main components: The Halide itself to support the input language, *Portable Computing Language (pocl [3])* as an OpenCL [4] implementation, and tcecc, a retargetable compiler from the *TTA-Based Co-design Environment (TCE [5], [6])* toolset. The rest of the processor generation process follows the standard flow of TCE as described in [7].

A. Compilation Flow

The compilation flow uses the (unmodified) OpenCL backend of Halide to produce parallel program input to the later phases. Another obvious option would have been to exploit the LLVM [8] bitcode backend of Halide to input kernel descriptions to the customized cores. However, LLVM IR is a sequential representation with only limited support for parallelism, which would have risked losing information about the abundant parallelism available in Halide programs. Although LLVM now has some support for describing parallel loops, it lacks a standard way for describing task level parallelism, which can be represented in OpenCL by using work-groups and launching multiple kernels simultaneously. Most importantly, heterogeneous offloading of kernels to multiple different devices is not supported when producing LLVM bitcodes directly.

Halide programs are described using Halide C++ classes. The C++11 programs are compiled to a native binary, which, when executed, produces the Halide image pipeline. In this case the image pipeline is generated as an OpenCL application which consists of the OpenCL host program (built to an LLVM bitcode) including the OpenCL C kernels embedded as global strings.

The bitcode can be executed using the LLVM JIT compiler, or built offline, and then executed. The executed OpenCL host program accessess OpenCL APIs for controlling the application, which in this case ends up calling *pocl*, an OpenCL implementation with support for customized cores designed by the integrated processor co-design suite.

When executing the OpenCL program, pocl's OpenCL C kernel compiler (as described in [3]) produces a so called *Work-Group (WG)* function for each kernel. It contains *Work-Item (WI)* loops, which are parallel loops executing all the WIs in the WG. Loops communicate the data parallel processing of multiple pixels to the compiler code generation in a scalable way. The parallel loop information is utilized by the LLVM vectorizer to produce SIMD instructions automatically, in case the target at hand supports them.

The WG functions with parallel WI code are scheduled to the fine grained parallel resources (scalar and vector function units) of the architecture at hand by a retargetable compiler from the TCE suite (tcecc). The end result is a parallel binary which can be uploaded to the customized processor for execution.

B. Custom Operations

Special instructions are typically application-specific functionality integrated to the processor datapath that accelerate the software in comparison to when only utilizing "basic operations" which are easily produced from operators in common programming languages (multiplications, divisions, additions, etc.).

Complex special instructions that cannot be exploited automatically by the instruction selector of the compiler backend are commonly invoked from high-level language sources using some sort of *intrinsics*. The intrinsics are usually implemented either as compiler-specific builtin functions or inline assembly snippets. In order to integrate complex custom operation calls to Halide programs with as little execution and engineering time overhead as possible, we developed a mechanism which instantiates a *Halide::Internal::Call* object that contains only a call to a specially named *external function*. These external function calls are converted to OpenCL C standard conformant *vendor extension* calls by the Halide OpenCL backend, which are finally expanded to inline assembly that invoke the target-specific custom operation in question.

```
// Return value type, operation name with
// a _tce_ prefix, function input types
HalideExtern_3 (uint8_t, _tce_wavg3,
                uint8_t, uint8_t, uint8_t);
```

Fig. 2. Halide statement declaration for introducing a custom operation called *wavg3* with a uint8_t result and three uint8_t inputs to Halide.

In order to call TCE custom operations in Halide programs, the programmer has to first declare the custom operation statement in the Halide source file. It can be done as shown in the snippet of Fig. 2. Only this call interface is specific to the Halide-based design flow; the operation simulation behavior and other properties are defined using the standard *OSEd* tool of TCE as described in [7].

After introducing the Halide statement declaration for the TCE custom operation of interest, the programmer can integrate the call to the Halide program as shown in Fig. 3.

```
Image<uint8_t> input = load<uint8_t>("rgb.png");

// Blur horizontally
Func blur_x (x, y, c) =
   _tce_wavg3 (input (x-1, y, c),
               input (x, y, c),
               input (x+1, y, c));

// Blur vertically
Func blur_y (x, y, c) =
   _tce_wavg3 (blur_x (x, y-1, c),
               blur_x (x, y, c),
               blur_x (x, y+1, c));
```

Fig. 3. Calling the example custom operation in a Halide program.

The compilation flow treats the custom operation calls as follows. The custom operation call is generated by Halide to OpenCL C vendor extension function calls. Pocl's OpenCL C kernel compiler defines OpenCL C vendor extensions for the custom operations as macros. The macros expand to TCE specific inline assembly snippets which are finally converted to machine code operation calls for the processor at hand by tcecc.

C. Iterative Processor Co-Design Using TCE

TCE is a design and programming toolset for statically scheduled customized processors which is used for the processor customization phase. The processor co-design process is iterative in nature; variations to the architecture are experimented with based on the feedback from the compiler and instruction set simulator. In order to make this process feasible, the input programming language description must require minimal modifications for each tested alternative. This is enabled by the clean custom operation interface and the schedule separation in the Halide-based design flow.

In the design flow, after the parallel program has been compiled for the architecture variation at hand, the TCE driver of pocl calls the instruction set simulator to evaluate the performance in terms of instruction cycle count. This phase produces profiling data which directs the designer's focus to the Halide functions that consume the most instruction cycles.

The hot spots can be accelerated by adding more resources to the architecture at hand. In addition to enabling customization of basic resources such as parallel function units (vector or scalar), register files and connectivity, TCE's support for defining custom operations [9] can be used for adding specialized hardware. Often a good choice, especially with smaller processor designs, is to accelerate the software by adding one or more custom operations which perform a chain of custom wide arithmetic basic operations as a single processor instruction, which often results in a reduced execution latency in terms of clock cycles.

The standard TCE customization flow produces VHDL or Verilog register transfer level descriptions of the designed architecture. The descriptions can be implemented as ASICs or FPGA soft cores.

III. Evaluation

In order to evaluate and validate the flow, we used it to design customized processors for two application cases, *Blur* and *Bilateral grid*.

A. Blur

For a baseline Blur processor design we created a small scalar architecture with only basic operations. The architecture design consists of an integer ALU, a load-store-unit, scalar and boolean register files and a *Real Time Clock (RTC)* unit for execution time measurement. We specifically avoided adding parallel resources to the machine to isolate the benefits from custom operations.

This architecture was extended with a special instruction that calculates a weighted average of three input values and is used in basic blurring of the images. The formula of the algorithm implemented by the special instruction can be seen in Equation 1.

$$r = \frac{p_0 + 2p_1 + p_2}{4} \tag{1}$$

where p_0, p_1 and p_2 are the parameter values and r is the result of the special instruction. All the values are 8-bit integers, which are commonly used to store image data. This is because the result of the average cannot overflow. It is sufficient to handle intermediate overflows correctly in the hardware implementing the special instruction. The instruction may seem to be limited to blur window of just 3×3, but it can be applied multiple times to achieve greater blur window sizes.

The special instruction demonstrates TCE's ability to handle instructions with more than two inputs. This is handy in image processing where multiple low precision values are often used for the computations. Thanks to the customized arithmetics and the ability to use hard wired shifting to implement the division by two, the operation can be performed in one clock cycle in hardware.

The benchmark computes a 3×3 pixel blur for a 512×512 resolution image. The first application uses a clampless algorithm and produces an image of size 510×510 because border pixels are not computed. The other benchmark uses Halide's inbuilt clamp-to-edge feature which allows an uniform method to be used for border and inner pixels, thus resulting in an output image that is the same size as the input image.

In both benchmarks Halide divides the blur computation by serially launching the same kernel for each color channel in the image (Halide stores images in color-planes for easier vectorization). The Halide application's schedule was organized in a way that the produced an OpenCL kernel launch consists of one workgroup with an OpenCL local size of $510 \times 510 \times 1$ for clampless, and $512 \times 512 \times 1$ for clamped. Kernel source code produced by Halide describes the procedure for a single channel component of a pixel.

The performances of the applications were determined by measuring the execution time of the kernels in instruction cycles by utilizing the RTC operation in the processor instruction

set. The system clock frequency assumed for the machine was 100 MHz.

Results from blur test are shown in Fig. 4. The clampless variation without custom operation took 80 ms per kernel (color channel). The custom operation accelerated version took 50 ms per kernel yielding 30 ms reduced execution time and $1.63\times$ speedup. The clamped variation without the custom operation took 173 ms and the custom operation accelerated version 147 ms. The custom operation version was 26 ms faster with $1.18\times$ speedup.

Although the used blur application is computationally very light, which emphasizes the memory operations and address computation overhead in the execution time, the measurement shows that the simple weighted average custom operation brings a considerable speedup. Introducing the clamp-to-edge to the computation diminishes the relative gains achieved with the custom operation by adding roughly a 100ms overhead to both variations.

B. Bilateral Grid

The second evaluated application is *Bilateral Grid* algorithm that can be found in the example codes of the Halide project. The idea of the Bilateral Grid is to apply edge preserving blur to the input image. This means that smooth areas of the input image are blurred, while edges stay intact. [10]

The test application was executed with similar machine to the *blur* example, but enhanced with a floating point unit, since the application uses floating point arithmetics. This baseline machine was compared to a machine with custom operations. The first custom operation is for speeding up the blurring. The operation is a floating point, five data point "semi" weighted average operation, which multiplies the inputs with weights and sums them, but does not divide the result. The latency of this operation is five cycles. The second custom operation is 3D lerp, or 3D linear interpolation between 8 voxels. The lerp operation takes 8 intensity and 3 weight input arguments and produces a single output value. The latency for this operation is 15 cycles. The input image used for evaluation was a gray scale image of resolution 256×256.

The results are in Fig. 4. Total execution time of all the kernels without custom operations was 221 ms, and with only the blur acceleration, the execution time was 207 ms. With added 3D lerp accelerated version execution time dropped to 181 ms. A total of 40 ms faster execution ($1.22\times$ speedup) was achieved.

The speedups of the individual kernels varied from $1x$ to $1.62x$. The first two kernels did not use any custom operations, thus no speedup. The kernels that were using the semi weighted average gained $1.33\times$, $1.63\times$ and $1.69\times$ speedups. The one kernel using the 3D lerp gained a $1.37\times$ speedup.

IV. Related Work

HIPAcc is a flow from an image processing DSL to GPU-style processors [11]. Its OpenCL/CUDA application generation component resembles the one our flow uses from the upstream Halide project. HIPAcc is extended to target customized FPGA-based hardware designs in [12]. It relates

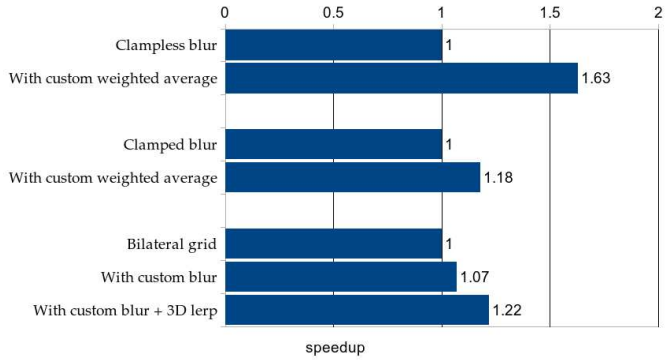

Fig. 4. Results from Blur and Bilateral tests (higher is better). The impact of the custom operations presented as a speedup against the unaccelerated version.

to ours in its goal of utilizing customized hardware from a DSL. The actual hardware generation component utilizes a commercial *High-Level Synthesis (HLS)* tool from Xilinx to perform the actual hardware synthesis after generating Vivado HSL optimized C++ for the kernels.

Darkroom is another DSL for image processing [13]. The authors mention that their language is different to Halide in the way it restricts the image processing to *stencils*, static fixed sized windows, which eases the implementation of hardware based on the language. Like HIPAcc, Darkroom aims at generating fixed function hardware pipelines. Our focus is on programmable application-specific processors. The custom processors designed using the proposed flow can be implemented as softcores in FPGAs, but preferably as ASICs for better performance and programmed on the field.

V. Conclusion

In this paper we described an image processor customization flow utilizing Halide as a parallel program input. A key point in programming customized processors is to present a fluent way to invoke user defined custom operations in the processor hardware from the higher level language. For this we proposed an easy to use mechanism embedded in the Halide language. The evaluation results show that Halide can serve as an efficient input to drive customization of image processors with custom instructions having significant potential impact to the execution performance.

This paper concentrated on the design flow of individual customized processors, while Halide is an interesting candidate for programming entire customized parallel image processing pipelines built of multiple, heterogeneous programmable devices. In the future we look into extending the design flow to produce platforms of multiple different customized processors implementing image pipelines.

Acknowledgment

The authors would like to thank their funding sources: Academy of Finland (funding decision 253087), Finnish Funding Agency for Technology and Innovation (project "Parallel Acceleration 2", funding decision 40081/14), and ARTEMIS JU under grant agreement no 621439 (ALMARVI).

REFERENCES

[1] A. Sampson, J. Bornholt, and L. Ceze, "Hardware-Software Co-Design: Not Just a Cliché," in *Summit on Advances in Programming Languages*, Asilomar, CA, May 3–6 2015, pp. 262–273.

[2] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, Jul. 2012.

[3] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *Int. J. Parallel Programming*, vol. 43, no. 5, pp. 1–34, 2014.

[4] *The OpenCL Specification*, v1.2r19 ed., Khronos Group, Beaverton, OR, Nov. 14 2011.

[5] TCE, "TTA-based co-design environment," 2015. [Online]. Available: http://tce.cs.tut.fi

[6] P. Jääskeläinen, C. de La Lama, P. Huerta, and J. Takala, "OpenCL-based design methodology for application-specific processors," *Trans. HiPEAC*, vol. 5, no. 4, 2011.

[7] O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proc. Int. Conf. Field Programmable Logic Appl.*, Milan, Italy, Aug. 31 – Sept. 2 2010, pp. 217–222.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. IEEE/ACM Int. Symp. Code Generation Optimization*, San Jose, CA, Mar. 20–24 2004, pp. 75–87.

[9] H. Kultala, P. Jääskeläinen, and J. Takala, "Operation set customization in retargetable compilers," in *Conf. Record Asilomar Conf. Signals Syst. Comput.*, Pacific Grove, CA, Nov. 6–9 2011, pp. 761–765.

[10] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Trans. Graph.*, vol. 26, no. 3, Jul. 2007.

[11] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Korner, and W. Eckert, "HIPAcc: A domain-specific language and compiler for image processing," *IEEE Trans. Par. Distr. Syst.*, 2015, to appear.

[12] 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 *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, New Delhi, India, Oct. 12–17 2014, pp. 17:1–17:10.

[13] 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 Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014.