MASTER

Extending halide to improve software development for imaging DSPs

Vocke, S.

Award date:
2016

Link to publication
Extending Halide to Improve Software Development for Imaging DSPs

SANDER VOCKE, Eindhoven University of Technology

Specialized Digital Signal Processors (DSPs), which can be found in a wide range of modern devices, play an important role in power-efficient, high-performance image processing. Applications including camera sensor post-processing and computer vision can benefit from being (partially) mapped onto such DSPs. However, due to their specialized instruction sets and dependence on low-level code optimization, developing applications for DSPs is often more time-consuming and error-prone than for general-purpose processors. Halide is a domain-specific language (DSL) which enables low-effort development of portable, high-performance imaging pipelines — a combination of qualities which is currently hard, if not impossible to find among DSP programming models. I propose a set of extensions and modifications to Halide in order to support DSPs in combination with arbitrary C compilers, including a template solution to support diverse target instruction sets and heterogeneous scratchpad memories. Using a commercial Intel DSP, I demonstrate that this solution can be used to achieve performance comparable to tuned C code, while leading to a reduction in development time and code complexity. The results also show that DSPs are attractive alternatives to CPUs and GPUs for power- and area-efficient image processing using Halide.

1. INTRODUCTION

Note: this is a public version of a thesis project report. It has been edited for publication, as parts of the original work are awaiting approval from Intel to be published. Edits include removing (parts of) sentences about processor architecture, and all measurement results.

Image processing workloads, such as camera sensor post-processing and certain computer vision tasks, are specifically targeted by various modern DSP architectures. These include the Qualcomm Hexagon, Cadence Vision and Texas Instruments TMS320 families, the CEVA XM4 and the DSPs found in Intel’s Image Processing Units (IPUs) [Codrescu et al. 2014; Cadence 2015; Texas Instruments 2016; CEVA 2015], [Intel 2016a, Sec. 3.4]. They are also well-suited for many other parallel computation tasks. To be competitive, they need to achieve high throughput and efficiency, due to their direct impact on metrics like video streaming framerates and battery life. As such, their design favors results over programmability, which manifests itself in features such as Very Long Instruction Word (VLIW) execution units, heterogeneous scratchpad memories, exotic Single Instruction Multiple Data (SIMD) instructions and integrated accelerators. To reap the benefits of such features, compile-time and source code optimizations are critical. To thoroughly explore the design space of optimization strategies, the typical programming models for these targets (e.g. C and OpenCL) tend to require rewriting large portions of code repeatedly. The result is high implementation and optimization effort. Generally, the results of development efforts are
not portable, cannot be easily fused with other algorithms and are not easily achieved by developers who are not experienced with the DSP.

The effort involved in programming DSPs may discourage application developers from mapping parts of their applications to them, although doing so can lead to significantly improved performance and/or efficiency. This is especially true considering that multiple partitioning and data transfer strategies for DSP acceleration may need to be explored.

1.1. Halide and DSPs

Halide [Ragan-Kelley et al. 2013] is a domain-specific language (DSL) and compiler for image processing. It radically reduces the effort spent on development and optimization by using an abstract, functional programming style, while decoupling the description of the algorithm from that of the execution strategy (referred to as the schedule). To port Halide applications between targets or try different parallelization strategies, only the schedule needs to be changed. Halide can generate code for CPUs and GPUs, allowing straightforward exploration of workload partitioning options.

Halide is starting to expand to DSPs. For example, Qualcomm Hexagon support is currently under development [MIT CSAIL 2016]. Support for a wider range of DSPs is desirable. While their VLIW architecture poses common optimization challenges for many DSPs, some may have properties which pose additional problems, including:

— Lack of a compiler for which Halide can generate SIMD-enabled code;
— Existence of scratchpad memories, which are not handled by Halide’s schedules;
— Lack of data caches, thus requiring manual exploitation of locality;
— Lack of support for dynamic memory management.

1.2. Contributions

In this paper, I propose a Halide-based development process for DSP targets which come with arbitrary C compilers. I present:

— A framework for DSP C code and testbench generation using Halide;
— An extension to Halide’s scheduling model to explicitly manage scratchpad memories;
— An extension for automatically fusing separate Halide pipelines, which aids the VLIW compiler in detecting instruction-level parallelism (ILP).

Additionally, I decouple Halide and C compiler optimization levels by optimizing local memory access ordering and data re-use in a Halide compilation pass. I introduce maximum buffer size analysis in order for Halide to statically allocate scratchpad memory for variable-sized buffers. I also present several examples of target-specific compiler passes, which further improve performance for a commercial Intel DSP.

I evaluate my work on this DSP, showing that using Halide with these contributions can reduce implementation times and code complexity of image filters while maintaining comparable performance to C versions which follow the same execution strategy. I also show that by automatically fusing independent, relatively unoptimized Halide filters together, the resulting filter may be more than 30% faster than sequentially executing its individual parts.

Finally, I present a value proposition for using DSPs for imaging tasks in heterogeneous compute solutions, based on benchmarks of publicly available Halide applications on the aforementioned DSP, as well as several CPU and GPU targets. Note: these benchmarks are withheld from this public version of the thesis report, as their publication is awaiting approval by Intel.
2. INTRODUCTION TO HALIDE

The Halide DSL can be used to concisely describe imaging pipelines as a set of functions from coordinates to values. For example, a 3x3 separable blur filter can be expressed as a chain of two functions, as shown in the top left corner of Figure 1.

The values of \( h \) and \( out \) are defined for unbounded integer values of \( x \) and \( y \), using pure definitions without side-effects. The \( out.realize(4,4) \) command triggers compilation and execution of the pipeline for an output range of 4x4 pixels. Using interval analysis, starting from outputs and moving back to inputs, Halide automatically infers the domains required of each stage (in this case: 4x6 \( h \) and 6x6 \( in \) pixels).

Figure 1 also shows how scheduling directives can influence computation order and vectorization. When vectorization is applied, the compiler automatically produces SIMD operations for memory accesses and arithmetic. Locality optimizations can be achieved by using \( compute_root() \) and \( compute_at() \) commands, which control the granularity and order of computation between stages.

Halide’s syntax, scheduling model and compilation process are thoroughly described by Ragan-Kelley [2014].

Not all computation problems can be expressed in Halide (i.e. it is not Turing-complete). This can be attributed to the lack of a means for arbitrary recursion. However, in cases where a subset of an algorithm’s functionality is not expressible in Halide, it may be implemented as an external stage in a different language (for example, C++) and integrated into the Halide pipeline.

Halide is an embedded DSL, meaning it is implemented as a library to a host language (C++). Compiling, then executing a C++ program which uses this library starts Halide’s compilation and code generation process (see Figure 3.1). Multiple code generation back-ends are included, including CUDA, OpenCL and C code generators, as well as LLVM-based back-ends for x86 and ARMv7 (with optional vector extensions including SSE 4.1, AVX2).
and NEON). The LLVM-based back-ends utilize LLVM to produce object files ahead of time, or to JIT-compile and execute the pipeline [Lattner and Adve 2004]. The C code generation workflow is shown in Figure 4.

The Halide compiler transforms the algorithm description to an imperative implementation: a loop nest. To describe this loop nest independently of the target architecture, Halide has an intermediate representation (IR), which takes the form of an abstract syntax tree (AST) [Jones 2003]. This IR has a small semantic gap to languages like C or LLVM’s IR. Therefore, in most cases, generating output code is a matter of translating IR nodes one by one. Each of Halide’s code generators may apply specific optimizations for its respective target instruction set.

3. DSP EXTENSIONS FOR HALIDE

Widespread DSP support for Halide poses several challenges. In this section I identify these challenges and propose a solution for each.

3.1. DSP Code and Testbench Generation

In order to support a wide range of imaging DSPs, Halide must have back-ends which can generate code for them. High-performance Halide applications have so far been synthesized primarily for CPUs and GPUs, using back-ends based on OpenCL, CUDA or LLVM-based compilers. Out of the DSPs mentioned in section 1, the Qualcomm Hexagon family is the only DSP family which is known to use a LLVM-based compiler — the others are all programmed using proprietary C and/or C++ compilers. Halide’s C code generator is a viable option in such cases, but it lacks support for SIMD operations. Also, each DSP family comes with different conventions and/or API libraries to perform the fundamental functions of multi-core programming, such as loading executables, controlling DSP execution flow and accessing DSP memories. Such functions are typically also required in order to test an application on a single DSP.

I present a framework based on Halide’s C code generator, with extensions to support SIMD instructions, multi-core workload partitioning and remote memory access. It generates both DSP code and testbench code to control execution of one or more DSPs.

For DSPs, generic SIMD statements and data types are generated, which adhere to fixed naming conventions. The mapping of these SIMD statements to the target ISA is provided in mapping libraries. Figure 3 shows how a 32-way addition, represented as an AST subtree in Halide IR, is converted to C output and mapped onto the ISA.

In a style similar to Halide’s CPU/GPU partitioning, the programmer assigns one or more parts of the generated loop nest to DSPs. For the partial loop nest to be accelerated, a separate, loadable and callable C program is generated. The process is illustrated in Figure 5. The code to handle data transfer, and to load and call this program at the appropriate time(s), is automatically generated as part of the testbench. It references generic functions for DSP control and remote memory access, which are implemented in additional mapping libraries.

Note that while this functionality enables testbench generation and workload partitioning over multiple cores, it cannot be considered a full multi-core optimization solution. The latter would require additional features such as system-level resource scheduling (e.g. to optimize DMA and bus usage) and concurrent execution of different tasks on multiple cores — the provided framework only executes code on one processor at any given time.

3.2. Scratchpad Memory Management

In code Halide usually generates for CPUs, there is a single, global address space. Data locality optimizations directly affect performance due to the existence of data caches. When targeting a GPU, Halide generates code which may use several address spaces, typically corresponding to different levels of granularity (such as OpenCL’s local and global
Extending Halide to Improve Software Development for Imaging DSPs

Compilation flow with Halide's C code generator.

Compilation flow with proposed DSP framework.

Fig. 2. Comparison of current and proposed code generation workflow. Note that the C code generator is part of the Halide compilation executable.

Halide IR subree:

Add

A (16-way, 32-bit, signed)
B (16-way, 32-bit, signed)

Halide Code Generator

Unprocessed C:

```c
h1_int32_x16 A = ... ;
h1_int32_x16 B = ... ;
H1_int32_x16 ... = SIGNED_INT VECTOR_ADD{ 32, 16, A, B };
```

Compile with Mapping Library

Preprocessed C:

```c
16vec132 A = ... ;
16vec132 B = ... ;
16vec132 ... = 16vec132_add_op{ A, B };
```

Fig. 3. A 16-way, 32-bit SIMD addition being mapped onto a fictional target ISA intrinsic call.

Halide's model for managing remote memory: local access only, automatic buffer copies.

Device 1 Stage

Automatic Buffer Copies

Device 2 Stage

Proposed model: unique buffers, remote access support.

Device 1 Stage

Remote Access

Device 2 Stage

Fig. 4. An example illustrating the difference between the current and proposed storage models.

However, these spaces do not necessarily represent the physical memories which exist in the system. In addition, Halide does not make them visible to the programmer, but manages them implicitly: buffers which are allocated at the GPU thread level are assigned to the local memory, and anything else goes into the global memory. Buffers which are referenced by both CPU-side and GPU-side code are allocated twice: once in CPU and once in GPU memory. A Halide compiler pass then analyzes the points in time at which the buffer is required to be valid in either memory, inserting buffer copies where necessary. Figure 4 illustrates this approach.

However, some DSPs have memory hierarchies which use heterogeneous, software-managed scratchpad memories. Reasons for this include saving area and power, offering more control for specific use-cases, having predictable performance and being able to differentiate memories with special properties (such as vector or block access modes). In such systems, choosing where to store data can have a major impact on performance, adding new degrees of freedom to the space of optimization choices. In that case, choosing stor-
Fig. 5. Using copy stages, store_in and exec_on commands to execute a 3x3 blur line-by-line on a DSP, using a local scratchpad memory. By default, execution happens on the CPU and storage in a default memory. Since h is not explicitly scheduled, it is inlined into out.

I propose an alternative storage model, centered around a new scheduling command: store_in(). It specifies in which memory a specific pipeline stage stores its data. To copy data between memories, extra stages can be introduced which act purely as data copiers, and can be scheduled using Halide’s existing scheduling model. Figure 4 compares this approach to the current CPU/GPU approach. Figure 5 shows how store_in() can be applied, along with exec_on(), to process a 3x3 blur line by line.

Another challenge in memory management is allocation. On CPU targets, Halide generates code which, in most cases, allocates memory for intermediate buffers dynamically at run-time. However, dynamic memory allocation is not always available for DSP scratchpad memories. I modify the C code generator to attempt static allocation of any buffer which is assigned to such a memory. A condition is that the allocation size must be known at compile-time. I add a compilation pass to Halide which allocates buffer memory statically, analyzing non-constant buffer size expressions to determine a worst-case buffer size if possible.

3.3. Automatic Pipeline Fusion

DSPs rely heavily on parallel computation. For a VLIW core, the compiler must schedule instructions statically, which requires it to find the available parallelism in source code at compile-time. Exposing as much trivial parallelism to the compiler as possible often helps it to achieve high utilization of computation units. This usually involves constructing inner loops with sufficient computation work, little to no branching and as few data dependencies as possible. This can be difficult due to the fundamental data dependencies found in many algorithms. Combining multiple programs into one is a way to force more independent
Fig. 6. The process of loop merging. $P_1$ and $P_2$ are independent Halide functions. The loop nests Halide generates for them individually (including some dummy statements) are shown on the left. On the right, two ways these loop nests could be merged are shown. Any loops enclosing the ones marked for merging will be merged too as a result. The marked loops must be equally deep in their respective nests. If iteration counts are not equal, an epilogue is generated, as shown on the right.

operations into the inner loop. The result is a single program which takes multiple inputs and produces multiple independent outputs. While such a program can be implemented in Halide using tuples, this requires equal sizes for each output and manual description of the combined pipeline. I propose a feature to fuse distinct programs automatically, provided they have a similar loop nest structure. This enables programmers (and potentially auto-schedulers) to push the performance of multiple programs beyond that of the individual versions.

The proposed implementation of this feature requires specification of two independent pipelines to co-compile, their respective arguments, and the point in each pipeline’s loop nest where fusion should take place. These points must be equally deep in each pipeline’s respective nest. The process is illustrated in Figure 6.

3.4. Peephole and Memory Optimizations

I propose several additional features to improve Halide’s usability and/or performance for DSPs. Some are universally applicable, while others tailor Halide to a specific DSP architecture.

DSPs’ instruction sets do not always match well with Halide output code, leading to sub-optimal performance. Such mismatches generally fall into one of two categories:

1. Exotic DSP operations which are not directly targeted by Halide’s code generator.
2. Operations required by Halide, for which no corresponding DSP instruction exists.

The first category can have different implications. It may be possible to use pattern matching to detect that a section of Halide’s AST maps onto an exotic DSP instruction — for example, a special memory access or a composite arithmetic instruction. This process is referred to as peephole optimization, and is also used in Halide’s x86 and ARM code generators.

In other cases, the DSP instruction set offers functionality which is highly unlikely to correspond to any Halide code in a bit-true fashion, or for which pattern detection is hard to implement: for example, saturating arithmetic or an accelerator which performs a complex function. In such cases, a command may be added to Halide which directly represents said functionality, to be used explicitly by the programmer. On the DSP, the command is mapped directly onto the hardware, while for other platforms, emulation of the functionality can be built into Halide, retaining portability.

If a DSP does not have an instruction to effectively implement an essential Halide operation, emulation can be built into the Halide compiler.

For any DSP, minimizing the amount of memory accesses, and ordering them to best hide latencies, is an important aspect of maximizing performance. In general, opportunities
to pass data through registers instead of memories should be utilized. While the Halide compiler performs common subexpression elimination (CSE), it does not perform all possible optimizations to reduce the total amount or ordering of memory accesses — for that, it mostly relies on the back-end compiler, which must perform thorough address expression analysis to apply such optimizations. However, they are more straightforward to implement at Halide’s abstract AST level, where accesses are made to element indices rather than raw addresses. Doing so also decouples Halide’s performance from the back-end compiler’s optimization level. I add an optimization pass to Halide, which:

1. Identifies and optimizes away duplicate loads of the same element(s).
2. Identifies read-after-write dependencies, replacing such reads by register passing.
3. Identifies if two adjacent loads are performed in a loop, in a way that steps monotonically through a buffer. Reduces this to a single load, passing one value to the next iteration using a register.

4. EVALUATION

Note: results have been removed from the public version of this thesis, as they are awaiting approval for publication by Intel.

5. RELATED WORK

Programmability of diverse architectures is an actively researched topic. In this section, I support the choice for a Halide-based approach for DSPs by outlining alternative solutions which also aim to decrease implementation and optimization effort, in an image processing context or otherwise.

I also point out recent work on Halide which has affected this work directly, or may improve Halide’s value proposition for DSPs in the foreseeable future.

5.1. Heterogeneous Platform Programmability

Various optimizing compilers and languages exist for automating loop transformation and analysis. Polyhedral representations provide a mathematical basis for such solutions. For example, Polly [Grosser et al. 2012] is a plug-in for LLVM, which uses polyhedral techniques to automatically apply loop transformations on LLVM IR. It supports many of the same transformations which Halide can describe. However, it typically requires an initial loop nest to be provided by the programmer, whereas a DSL like Halide can describe the functionality at an abstract level. PolyMage [Mullapudi et al. 2015] is another DSL which offers such an abstract description style. It automatically generates optimized loop nests using polyhedral optimization techniques. Although PolyMage has been shown to be competitive with Halide, it is not publicly available, which makes it currently unsuitable for wide adoption. Generally speaking, polyhedral optimization techniques are more complex than Halide’s interval analysis. On the one hand, this makes more advanced optimizations possible (for example, generating loop nests over non-rectangular iteration spaces), but on the other hand it makes it difficult to intuitively describe schedules.

Solutions such as Intel’s Integrated Performance Primitives [Intel 2016b], OpenCV [Bradski and Kaehler 2008] and OpenVX [Rainey et al. 2014] abstract away from target-specific optimization problem by providing pre-implemented functions to the user. However, it may not be acceptable for a user to be dependent on a slow-changing vendor library. Algorithmic skeletons [Nugteren and Corporaal 2012] instead provide optimization templates for entire classes of algorithms, which may be automatically detected from source code by a tool. However, fixed optimized functions or optimization templates disregard the fact that in practice, there is rarely a single “best point” in the design space, but more likely a trade-off between various metrics.
OpenCL [Group 2015] addresses many of C’s shortcomings pertaining to parallel programming and multi-core aspects by using a system-level API and a C-like programming language for accelerators. However, it requires low-level tuning of programs to the target architecture, and most loop transformations must be applied manually.

5.2. Related Halide Work

Halide support for Qualcomm Hexagon DSPs is currently under development [MIT CSAIL 2016]. Some similarities exist with the Intel IPU DSP used in this work, including a VLIW architecture and wide vector access memory. As such, some of the improvements made to Halide as part of the Hexagon work have also contributed to results on the Intel IPU DSP, including improved automatic loop nest splitting and storage alignment options. However, the challenges outlined in section 1.1 do not apply to the Hexagon family (which has an LLVM compiler, a single address space and data caching).

Ansel et al. [2014] and Mullapudi et al. [2016] have demonstrated that automatic scheduling is possible for Halide through heuristic searches or model-based analysis, respectively. These approaches can likely be applied to DSPs as well, which would further reduce development times and increase portability.

Helium [Mendis et al. 2015] can automatically extract Halide code from x86 kernel binaries, effectively opening up a large library of pre-existing applications which can be automatically ported to Halide, and thus, to any Halide-enabled DSP. Halide has also been extended to support data-parallel distributed processing [Denniston et al. 2016], which may enable data-parallel mapping of applications onto a homogeneous set of multiple Halide-enabled DSPs — for example in an Intel IPU subsystem.

6. CONCLUSIONS AND FUTURE WORK

The framework presented in this paper can be applied to a wide range of DSP targets. Moreover, I have shown that with an extended storage model, an automatic pipeline fusion feature and target-specific compiler optimizations, Halide code can reach performance comparable to optimized C code on a DSP. It is well-suited to describe the imaging tasks which imaging DSPs excel at, making for a natural combination.

Clearly, Halide is suitable for supporting all major processor types typically used for image processing in SoC platforms. Recent advances in automatic scheduling may enable programmers to write efficient imaging code for any CPU, GPU or DSP target with Halide support, without detailed knowledge about any of these architectures. To realize this, automatic scheduling algorithms need to be extended to take into account optimization choices specific to VLIW cores and heterogeneous scratchpad memories.

Extending Halide’s scheduling model to describe task-based parallelism and buffer queues would allow for additional optimizations, which could target both VLIW and multithreading targets. Queues can decouple subsequent stages, allowing them to execute in parallel in a pipelined fashion. Such parallelism is task-based. For VLIW targets, a Halide schedule incorporating such concepts could be used to statically fuse the loops of subsequent stages, in a way similar to the inter-program loop fusion presented in this paper.

In a broader context, Halide’s usability for imaging on heterogeneous platforms could be improved by advancing its system-level capabilities. This includes system resource management, concurrent execution on a heterogeneous set of targets and support for fixed-function hardware and other accelerators. The aforementioned extensions for task-based parallelism and buffer queues would allow Halide to conceptually describe many system-level optimizations, such as hiding data transfer latencies through double-buffering. Alternatively, a framework using a separate system-level task scheduler (for example, an OpenVX-like graph compiler) could handle system resources, task concurrency and accelerators, with Halide as one of the options for implementing individual computation tasks.
REFERENCES


