Extending Halide to Improve Software Development for Imaging DSPs

SANDER VOCKE, HENK CORPORAAL, and ROEL JORDANS, Eindhoven University of Technology
ROSILDE CORVINO and RICK NAS, Intel

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 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 more time-consuming and error-prone than for general-purpose processors. Halide is a domain-specific language (DSL) that enables low-effort development of portable, high-performance imaging pipelines—a combination of qualities that is hard, if not impossible, to find among DSP programming models. We propose a set of extensions and modifications to Halide to generate code for DSP C compilers, focusing specifically on diverse SIMD target instruction sets and heterogeneous scratchpad memory hierarchies. We implement said techniques for a commercial DSP found in an Intel Image Processing Unit (IPU), demonstrating that this solution can be used to achieve performance within 20% of highly tuned, manually written C code, while leading to a reduction in code complexity. By comparing performance of Halide algorithms using our solution to results on CPU and GPU targets, we confirm the value of using DSP targets with Halide.

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Domain specific languages**; • **Computing methodologies** → *Image processing*; • **Computer systems organization** → *Embedded software*; • **Hardware** → Digital signal processing;

Additional Key Words and Phrases: Halide, optimization, performance optimization, scratchpad memory, DSP, VLIW, embedded processor, ASIP, accelerator, parallel computing, image processing unit, IPU, SIMD

ACM Reference format:

Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. *ACM Trans. Archit. Code Optim.* 14, 3, Article 21 (August 2017), 25 pages.

<https://doi.org/10.1145/3106343>

Authors' addresses: S. Vocke, Wolfskuilseweg 157B, Nijmegen 6542JE, the Netherlands; email: sandervocke@gmail.com; H. Corporaal, Eindhoven University of Technology (TU/e), Faculty of Electrical Engineering, Electronic Systems (ES), FLUX building (4.134), De Rondon 70, Eindhoven 5612 AP, The Netherlands; email: h.corporaal@tue.nl; R. Jordans, Eindhoven University of Technology (TU/e), Faculty of Electrical Engineering, Electronic Systems (ES), FLUX building (4.078), De Rondon 70, Eindhoven 5612 AP, The Netherlands; email: r.jordans@tue.nl; R. Corvino, Intel Benelux BV, High Tech Campus 83, Building HGT2, Eindhoven 5656 AG, The Netherlands; email: rosilde.corvino@intel.com; R. Nas, Intel Benelux BV, High Tech Campus 83, Building HGT1, Eindhoven 5656 AG, The Netherlands; email: rick.nas@intel.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/08-ART21 \$15.00

<https://doi.org/10.1145/3106343>

1 INTRODUCTION

Image processing workloads, such as camera sensor post-processing and certain computer vision tasks, are specifically targeted by various modern Digital Signal Processors (DSP) architectures. These include the Qualcomm Hexagon and Cadence Vision families, the EVE vision accelerator found in DSP systems by Texas Instruments, CEVA XM4, Myriad M2, and the DSPs found in Intel's Image Processing Units (IPUs) (Codrescu et al. 2014; Cadence 2015; Lin et al. 2013; CEVA 2015; Ionica and Gregg 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 such as video streaming frame rates and battery life. As such, their design favors performance and efficiency 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. The typical programming models for these targets (e.g., C and OpenCL) tend to require rewriting large portions of code repeatedly to thoroughly explore optimization strategies.

The result is high implementation and optimization effort. For example, a Modified Local Binary Pattern (MLBP) feature extractor, which consists of several local pixel intensity comparisons, took in-house programmers multiple weeks to fully develop for an Intel DSP. Most of this time was spent on optimizations such as minimizing data transfer overhead and optimizing the selection of SIMD instructions. Generally, the results of such 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.

Although DSPs often offer performance and/or efficiency benefits, they are often overlooked by application developers because of the effort and expertise required for programming them. Several multi-core partitioning and data transfer strategies for DSP acceleration may need to be explored to achieve a satisfactory result.

One way to reduce the burden on developers is to lift the level of abstraction of application code and automate part of the implementation steps traditionally done manually. Halide (Ragan-Kelley et al. 2013) is a domain-specific language (DSL) and compiler for image processing that aims to do so. It radically reduces the effort spent on development and optimization by using an abstract, functional programming style, while decoupling the functionality 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, enabling straightforward exploration of workload partitioning options.

Halide is starting to expand to DSPs. For example, Qualcomm Hexagon support is under development (Halide Project 2016). However, Hexagon DSPs are relatively close to general-purpose processors and GPUs in terms of how they access data (through caches) and how SIMD code can be generated for them (through the back-end compiler's intermediate representation, which supports vector types and operations). The back-end compiler in question is based on LLVM (Lattner and Adve 2004), which Halide also relies on for generating CPU code. A Hexagon DSP can manage memory much like a CPU target would, using dynamic allocation.

However, other specialized DSPs have different properties:

- They often do not have data caches, opting instead for scratchpad memory hierarchies to ensure predictability and low-power operation.
- Multiple heterogeneous types of software-managed scratchpad memory may be available, creating a non-trivial data mapping problem.
- Compilers for such application-specific DSPs often use enriched C as their input language, requiring usage of extensions, pragmas, and/or intrinsics to specify complex instructions and low-level optimizations.

Halide is not equipped to deal with these problems in its existing model and C code generation back-end. Rather than limiting Halide to supporting only such DSPs that match well with its current code generation framework, Halide should be enabled to generate high-performance C code for a wide range of DSPs. To that end, we present:

- A Halide-based framework for DSP C code and testbench co-generation (Section 3);
- A scheduling extension to Halide that decouples data mapping to scratchpad memories from loop transformations (Section 3.1);
- An extension for automatically fusing independent Halide pipelines, which aids the VLIW compiler in exploiting instruction-level parallelism (Section 3.2).

We also apply a combination of code optimization and generation techniques to address the aforementioned challenges:

- Matching patterns of code to complex DSP instructions (Section 3.3).
- Removing duplicate memory accesses within and across loop iterations (Section 3.4).
- Maximum buffer size determination through interval analysis (Section 3.5).

We implement these proposals for the aforementioned Intel DSP, showing a significant reduction in the code complexity of image filters while maintaining 80–90% of the throughput of optimized C versions that follow the same execution strategy. We also show that by automatically fusing independent, relatively unoptimized Halide filters together, the combined filter may be more than 30% faster than its individual parts—sometimes outperforming individual C versions.

We discuss the applicability of the same individual techniques to similar DSP architectures. Finally, we evaluate the added value of the proposed solution compared to already-supported architectures, based on benchmarks of publicly available Halide applications on several CPU and GPU targets and the aforementioned DSP. Under typical operating conditions, the DSP offers performance comparable to most of the other targets investigated, while offering superior performance/power and performance/die size ratios. These results confirm that adding support for such DSPs to Halide benefits end users and merits wider adoption of the presented techniques to similar processors.

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 3×3 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 4×4 pixels. Using interval analysis, starting from outputs and moving back to inputs, Halide automatically infers the domains required of each stage (in this case: 4×6 `h` and 6×6 in pixels).

Figure 1 also shows how computation order and vectorization can be scheduled. For vectorized schedules, Halide automatically produces SIMD operations for memory accesses and arithmetic. Locality optimizations can be scheduled using `compute_root()` and `compute_at()` commands, which control computation order and granularity.

The Halide language and compiler 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. Such stages should conform to a standard calling and data passing convention. Since their behavior is defined

Fig. 1. A 3×3 blur filter in Halide, with scheduling examples. Resulting loop nests are shown as pseudocode.

externally, it cannot be influenced using Halide’s scheduling model, nor can Halide apply low-level code optimizations to it.

Halide is an embedded DSL, meaning it is implemented as a library to a host language (C++). It is used with a generic C++ compiler to produce a compilation executable. This executable embodies the Halide compiler and code generator, as well as the specific program to be compiled. It will generate target code when executed. This workflow is shown in Figure 2(a). 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 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

To support a wide range of imaging DSPs, Halide must have back-ends that 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. Of the DSPs mentioned in Section 1, only the Qualcomm Hexagon family has a publicly available, 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. However, there is no single convention for describing SIMD operations accepted by all back-end compilers. Because of this, Halide’s C code generator

Fig. 2. Comparison of current and proposed code generation workflow. Input files are highlighted in green.

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

does not support SIMD operations at all. Also, each DSP family comes with different conventions and/or API libraries for basic multi-core programming functions, such as loading executables, controlling DSP execution flow and accessing DSP memories. These are typically required to run any task on a DSP.

We present a framework based on Halide’s C code generator, which we extend to support SIMD instructions, multi-core workload partitioning, and remote memory access. It co-generates DSP code and a host-side testbench that controls one or more DSPs. The code generation workflow using this framework is shown in Figure 2(b).

As part of the DSP code, our framework produces generic SIMD statements and data types, which adhere to fixed naming conventions. The mapping of these SIMD statements to the target ISA is implemented in *mapping libraries*. Figure 3 shows how a 16-way addition, represented as an AST subtree in Halide IR, is converted to C output and mapped onto the ISA. The mapping itself is typically one to one or, if necessary, a basic emulation using several instructions.

In a style similar to Halide’s CPU/GPU partitioning, the entire generated loop nest runs on the host processor by default. The programmer uses a scheduling command (`exec_on()`) to assign parts of the loop nest to DSPs. For such partial loop nests, a separate, loadable, and callable DSP C program is generated. The process is illustrated in Figure 5. The code to handle data transfer, and to load and start DSP programs at the appropriate time, is automatically generated as part of the host-side testbench. This testbench uses a generic API for DSP control and remote memory access. These functions are implemented in additional DSP-specific mapping libraries.

For commands such as `exec_on()`, the Halide programmer may need to use identifiers to refer to system components. For example, a list of processors available to `exec_on()` needs to be defined. Such definitions are in the *system description library*. This library also contains any

Fig. 4. Example showing the difference between current and proposed storage models.

additional system information the code generator may need, such as a listing of scratchpad memories accessible by each core.

Note that while this functionality enables testbench generation and workload partitioning over multiple cores, it cannot be considered a full multi-core optimization solution. Such a solution 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. Our framework only executes code on one processor at a time, using a blocking execution model. However, a host-controlled DSP program is automatically produced, which may include several DSPs, thereby saving lots of time otherwise spent on tedious processor interaction aspects. The individual generated DSP sub-programs can be re-used in other applications not generated by Halide.

Producing high-performance DSP code requires solutions to additional challenges, both general and target specific. In Section 3.1, we describe an extension that allows Halide programmers to optimize data storage in DSP systems containing software-managed scratchpad memories. In Section 3.2, we introduce a feature that can increase the performance of applications by fusing multiple independent loop nests together automatically. In Section 3.3, we present several optimization passes, which improve application performance by optimizing for the specific target DSP's instruction set and exposing the underlying compiler's software pipelining capability. In Section 3.4, we present a compiler pass that further improves performance by reducing the amount of redundant memory accesses.

3.1 Scratchpad Memory Management

In most modern processing systems, data locality optimizations in code directly affect performance and energy consumption due to the existence of data caches. Such caches typically operate transparently in a single global address space. Thus, code generated for such systems does not require any management of separate address spaces. If targeting a GPU, then programmers may use several address spaces, typically corresponding to different storage granularities (such as OpenCL's local and global memory). These spaces do not necessarily represent the physical memories that exist in the system. Halide offers a degree of control over these address spaces by coupling them to the organization of computation (buffers allocated at the GPU thread level are assigned to local memory, while anything else goes into global memory). However, the programmer cannot influence this mapping directly and independently of the computation schedule.

GPU buffers may be accessed by both CPU-side and GPU-side code. In such cases, they are currently allocated twice by the Halide compiler: once in CPU memory and once in GPU memory. Halide then automatically inserts buffer copy statements to ensure the buffer is valid on either side when required. This approach is illustrated in Figure 4(a). In the situation shown, there is a host and a GPU. One buffer (`buffer_1`) is accessed by both host-side and GPU-side stages of a Halide application. Thus, the Halide compiler allocates the buffer both in host and in GPU memory and inserts buffer copy statements automatically. This approach is easy for the programmer but offers limited control over the timing and granularity of data transfers between memories.

Fig. 5. Using copy stages, store_in and exec_on commands to execute a 3×3 blur line by line on a DSP, using a local scratchpad memory. By default, execution happens on a default processor (the CPU) and storage in a default memory. Since h is not explicitly scheduled, it is inlined into out.

This limited control over data transfers is acceptable in many general-purpose, cache-based processing systems, especially when large datasets are transferred and processed at a time. However, they may be too restrictive for some DSPs, in particular those with hierarchies of heterogeneous, software-managed scratchpad memories at the same level in the memory hierarchy. Reasons for incorporating such memories include saving area and power, offering direct control for specific use-cases, ensuring predictable performance, and differentiating special memories (such as those with vector or block access modes). In such systems, choosing where to store data can have a major impact on performance and energy consumption, adding new degrees of freedom to the space of optimization choices. Therefore, choosing storage locations in the compiler coupled to compute granularity will not suffice to explore the design space effectively.

We propose an alternative storage model, centered around a simple scheduling command: store_in(). It specifies in which memory a specific pipeline stage stores its data. As opposed to the pre-existing Halide approach, buffers only have a single instantiation in a single memory. Figure 4(b) illustrates this approach by showing one of the ways we can schedule storage using store_in(). A host and a DSP are shown, each with a local memory. In this case, we simply assign the buffer to host memory. The stage running on the DSP must then access the buffer remotely (which with the original approach is impossible). If duplication is desired, then the programmer could add a copy stage to copy (part of) the buffer's content to the second processor's memory on demand. This would be a Halide stage like any other, which in this case would only duplicate data. It could thus be scheduled like any other stage, effectively controlling the copy granularity, timing, and which processor performs the copy.

Figure 5 shows a simple real-world use-case, which applies store_in() together with exec_on() to process a 3×3 box blur on a DSP. Halide source code is shown in the box on the top left. The bottom right shows visually how data move from buffer to buffer and which processor performs each operation. Note that two copy stages are used to copy pixels back and forth

between CPU and DSP memory. Line-based processing and copy granularity is achieved by using the existing `compute_at()` command.

As the proposed scheduling extensions decouple memory assignment from computation in a way not possible before, this extension opens up new scheduling options for any target that has multiple address spaces shared by multiple cores. Among the architectures mentioned in Section 1, other than the Intel DSPs used in this work, some would benefit from this extension in particular:

- the Texas Instruments EVE subsystem (Lin et al. 2013), which contains several scratchpad RAMs at the same hierarchical level and sports a scalar RISC core that controls a vector co-processor;
- the Movidius Myriad 2 (Ionica and Gregg 2015), which contains multiple scalar and vector cores sporting shared memory of which subsections are considered “local” to particular cores;
- Processors from the Cadence Vision DSP family, which also sport multiple scalar and vector cores and multiple shared local memories.

3.2 Automatic Pipeline Merging

DSPs rely heavily on parallel computation. For a VLIW core, the back-end compiler must schedule instructions statically, which requires finding as much Instruction-Level Parallelism (ILP) as possible from source code at compile-time. Exposing trivial parallelism in source code helps the compiler to achieve high utilization of computation units. This usually requires constructing inner loops with sufficient work, little to no branching, and few data dependencies. However, fundamental data dependencies are found in most algorithms. Combining multiple loop nests into one is a way to force more independent operations into the inner loop. The result is a single loop nest that takes multiple inputs and produces multiple independent outputs. Although such a program can in some cases be implemented in Halide by using *tuples*, this requires equal sizes for each output and a manual description of the combined program’s functionality. We propose a feature to fuse loop nests, provided they have a similar loop nest structure. This enables programmers to push the performance of multiple programs or processing stages beyond that of the individual versions. The programmer must still guide the merge by specifying which loop nest levels should be merged together at the innermost level.

The proposed implementation of this feature takes two independent pipelines to fuse, their respective arguments and the level in each pipeline’s loop nest where merge should take place. Only one level of each nest is marked—any enclosing outer loops are implicitly fused as well. Thus, loops must be equally deep in their respective nests to be fuseable. The marked loops’ bodies are sequentialized, but since their operations are independent, a typical C compiler can straightforwardly apply parallel VLIW scheduling in the combined loop body. If iteration counts of the fused loops are not equal, then an epilogue is generated. Two examples are shown in Figure 6. The same independent loop nests are fused at two different levels, leading to different results.

Merging takes place after lowering but before code generation. Each program is separately passed through the lowering process as usual, the only exception being that all variable and iterator names are suffixed to prevent naming conflicts at a later stage. At this point there are two separate loop nests as ASTs. A recursive counter checks that the depth of each nest’s merging candidate is equal. Then, merging proceeds from the inside out. First, both inner loop bodies are sequentially combined and enclosed by a loop. If iteration counts are not equal, then an epilogue is generated with only the body of the longer loop. The result is repeatedly enclosed by additional outer loops, along with the original outer loops’ sequentialized bodies, until the fused loop nest is complete.

Fig. 6. Examples of loop nest merging. The loop nests Halide generates for them individually (including dummy statements) are shown on the left. The right side shows two ways these loop nests could be fused.

This feature is a tool for increasing exposed parallelism for VLIW cores in general and would therefore benefit any VLIW architecture, including those mentioned in Section 1.

3.3 Targeting DSP Instructions and Software Pipelining

In addition to generic optimizations likely to benefit various DSP targets, the level of specialization of these processors tends to require a number of optimizations specific to the architecture and/or C compiler. This includes targeting specific instructions supported by the DSP that do not match naturally to Halide-generated (vector) operations and exposing any pragmas and/or language extensions used in its compiler that are essential for reaching high performance.

Exotic DSP operations: A DSP instruction set may include operations that are not directly targeted by Halide’s code generator. This can have different implications. If such an instruction offers performance benefits, then it may be possible to use pattern matching to detect that a section of Halide’s syntax tree maps onto it. Examples are special memory accesses and composite arithmetic instructions. The detection and mapping process is referred to as *peephole optimization*, and is also used in Halide’s existing x86 and ARM code generators. The Intel DSP used in this work supports a special element-wise shuffle that combines elements of two differently sized vectors. Using this shuffle, we accelerate unaligned vector loads by shuffling an aligned vector with several adjacent elements. Identifying unaligned vector loads is achieved using Halide’s built-in modulo remainder analysis. Any such load is then replaced by two aligned vector loads and an intrinsic for the combined shuffle operation.

In other cases, a DSP instruction may offer functionality that is highly unlikely to correspond to any Halide code in a bit-true fashion or for which pattern detection is hard to implement (e.g., saturating arithmetic or an accelerator that performs a complex function). If such functionality is desired, then a command may be added to Halide that 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 to retain portability.

Unsupported Halide operations: If a DSP does not have an instruction to effectively implement an essential Halide operation, then emulation can be built into the Halide compiler. For example, when vectorization is applied to an originally scalar access to a data-dependent address (e.g., look-up tables and histograms), this results in *gathers and scatters*, which many targets do not support. They can be emulated by splitting the operation up into per-element loads or stores, but this is expensive. On some targets, other peephole optimizations can provide faster emulation. In particular, a combination of one or more dense vector loads, followed by a (multi-)vector shuffle operation, can emulate a gather on a small buffer (up to the maximum number of elements of the shuffle operation). This optimization was applied on the Intel DSP used in this work. For example,

the *Camera* application benchmarked in this work benefits from this emulation: Its gamma curve is implemented as a 32-way, interpolated look-up-table (LUT). Limiting the LUT to 32 elements ensures that gather emulation on the LUT buffer is possible.

Software Pipelining is generally regarded as an essential tool for generating efficient code for wide VLIW processors by exposing parallelism in inner loops (Lam 1988). Since there is no support for specifying its application in standard C, and pipelining introduces an important tradeoff between performance and code size, DSP compilers may use non-standard directives such as attributes or pragmas for guiding it. Ideally, Halide programmers should be offered the same capability for offering such guidance. We add a scheduling command to Halide: `sw_pipeline()`. It is used on the `Func` object that is being produced in a particular loop. For example, `out.sw_pipeline(x)` injects a software pipelining pragma on the loop over `x` producing `out`. The implementation of this type of scheduling extensions is straightforward: A cache is kept of loop levels on which the command is called and is queried by the C code generation stage on each generated loop to determine whether a pragma should be inserted.

While detailed information about instruction sets is not readily available for the target architectures mentioned in Section 1, multiple manufacturers of said targets claim their VLIW cores support advanced imaging and data reorganization instructions. For such instructions, these techniques for peephole optimization can be applied. Said targets' compilers likely support software pipelining, although it is unclear whether and how exactly this is guided in input C code.

3.4 Memory Access Optimizations

For any DSP, minimizing the amount of memory accesses 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 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 in Halide's internal representation (IR) of the application, because it is significantly simpler to analyze than, for example, a C program. Reasons for this include:

- All data accesses in Halide's IR are made to named buffers, with the guarantee that differently named buffers will never overlap in memory. This makes it very easy to determine the sets of loads and stores that should be considered for optimizations and eases many optimizations. For example, accesses to differently named buffers may always be safely re-ordered.
- Halide's IR does not include accesses to pointers into data buffers. This removes the need for alias analysis on pointers, including the problems that tend to stem from pointer casts and pointer arithmetic.
- Because Halide synthesizes loop nests and iterators from within the compiler, developers have been able to put in place useful conventions, such as always using the same 32-bit signed integer type to represent loop iterators, buffer sizes, and buffer indices. This means that when performing comparisons of multiple accesses' buffer indices, no special attention needs to be paid to type casts.

Note also that in languages like C, programmers can make it very difficult for compilers to apply alias analysis. For example, if a non-inline function is written that takes pointer arguments, a C compiler will not be able to prove within the scope of that function that these pointers won't alias. In Halide IR, such problems do not exist, because the entire loop nest is synthesized as one function.

Fig. 7. Examples of Halide compiler optimizations to reduce the amount of memory accesses in inner loops.

Memory accesses are represented by dedicated IR node variants in Halide: Load nodes for loads and Store nodes for stores. To compare them to each other for equality of memory location, it suffices to check their name and index fields, which contain the corresponding buffer name and an expression for the index accessed within that buffer, respectively. Note that the Halide compiler internally provides readily available helper functions for performing comparison on expressions for equality of value, which is useful for performing said index comparisons.

We 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 opportunities for loaded values to be passed over to the next inner loop iteration through a register, as opposed to being re-loaded.

For loop nests optimized to DSPs, the majority of execution time is usually spent in the innermost loop(s) of the loop nest. Therefore, we apply the above optimizations only to these innermost loops, thereby simplifying their implementation.

For optimizations (1) and (2), we step through pairs of Load and Store nodes in the inner loop. For each pair, the buffer name and index expressions are compared. If both are equal or equivalent, then the pair qualifies for optimization. Two loads from the same buffer and index are reduced to a single load if no store is made to the same location in-between. A store followed by a load from the same location is similarly reduced to just a store. In both cases, comparing index expressions for equivalence is done using Halide's pre-existing helper functions for math solving and expression comparison. Examples of these optimizations are shown in Figure 7(a) and Figure 7(b).

Optimization (3) is more involved. Instead of just checking index expressions for equivalence, it is necessary to determine whether the two index expressions are equivalent when a specific offset is applied in the loop iteration count. For a loop that loads elements one by one, this offset is one iteration between adjacent elements. Halide's math solving functions can be used to identify such cases. For each index pair to analyze, we modify one index. This is done by first finding the innermost loop iterator in the index expression and then modifying it with an iteration offset variable. Then we construct an equality expression of both index expressions and solve for the offset. If solving is successful and an offset of 1 is found, then we apply a simple transformation: One load is removed, while the other load's result is carried over between loop iterations. Figure 7(c) shows an example of this analysis and optimization.

Note that while Figure 7 illustrates each optimization in pseudocode, in practice the optimizations are performed on Halide IR syntax trees. They work similarly for vector loads and stores. Optimization (3) is particularly common for convolution filters, which often load multiple adjacent vectors to construct shifted versions to multiply.

The optimizations listed are generic and can benefit any VLIW compiler that does not already perform these optimizations to the same extent.

3.5 Maximum Buffer Size Analysis

Halide generates code that usually allocates memory for buffers dynamically at runtime, relying on allocation functions implemented by the underlying operating system. Although Halide will allocate memory statically in some specific cases, code generated by Halide will usually not work if dynamic allocation is not available. DSP systems may not offer dynamic allocation functions for small scratchpad memories. In fact, a static allocation scheme is often chosen because it aids predictability and ensures external actors responsible for transferring data know where to find or place it. Halide should allocate buffers in scratchpad memory statically in such cases.

However, memory for Halide buffers is typically re-allocated multiple times in a single program, often with different buffer sizes for each re-allocation. Typically, buffer sizes depend on the state of each loop iterator (e.g., if boundary conditions exist).

We propose an extension to Halide's compiler to automatically determine an upper bound on the required size for each buffer and to allocate this memory statically. Our proposed solution is simple. After Halide's lowering phase, dynamic allocation statements exist in the AST, each including an expression for amount to allocate. This expression may be a constant, in which case we can statically allocate the buffer. If the expression is data dependent (i.e., includes loads from memory), then we generate a compiler error. However, in most cases, non-constant size expressions only depend on the state of loop iterators, which all have a lower and an upper limit. These limits can be found by analyzing the corresponding For loop's *minimum* and *extent* expressions. If these expressions include other loop iterator states, we repeatedly substitute minimum and maximum expressions for each one. This way, in most cases we can determine a lower and an upper bound on each iterator's state. These bounds can then be substituted into each buffer size expression to find an upper bound on each buffer's size.

This method may produce an upper bound that is too pessimistic, because the worst-case combination of loop iterator states may not actually occur. However, the method guarantees that buffers will never exceed their statically allocated memory.

In our case, the proposed method is only applied when dealing with cache-less scratchpad memories. Therefore, the result only serves to enable the use of scratchpad memories that could otherwise not be used. The amount of memory allocated for each buffer does not directly impact performance.

If, for a given schedule, the sum of all static buffer sizes exceeds the available space in one or more of the local scratchpad memories, then it can be concluded that correct code cannot be generated. In such cases, the programmer is notified and advised to adjust the schedule. Tiled execution is common, in which case a reduction of the tile size is usually sufficient. Halide's bound syntax is still available to the programmer for explicitly setting buffer sizes in case a buffer's size is overestimated.

4 EVALUATION

The proposals made in Section 3 are implemented for a DSP found in Intel's Image Processing Units (IPUs), which are included in various commercial systems-on-chip. Effects of our modifications are evaluated by simulating this DSP. In the following sections, we describe the DSP and simulation

Fig. 8. Typical components and external interfaces of an Intel IPU. Actual quantities of resources may vary.

Fig. 9. Components and interfaces of the simulated platform. The host machine executes the simulation while also interacting with the simulated system.

environment, followed by results in terms of code complexity and performance. The results are discussed, including a comparison to benchmarks on several other architectures.

4.1 Intel IPU DSP and Simulation

A simplified overview of an Intel IPU is shown in Figure 8. It is a subsystem containing a plurality of DSPs, as well as fixed-function accelerators for common imaging tasks. Typically, it has a camera interface to operate on streaming sensor data in real time.

The specialized DSPs found in an Intel IPU are also referred to as application-specific integrated processors (ASIPs). They are part of a family of processors stemming from SiliconHive toolflow. This toolflow, the typical architectural properties of the processors produced with it, and properties of their co-generated compilers (including extensions of the C language they require) have been described in detail in Jordans (2015, Sec. 2.3). The IPU DSPs themselves are the result of multiple generations of design iteration to tailor specifically to running specific imaging workloads highly efficiently. Each DSP has a wide VLIW architecture, comprising a plurality of wide SIMD issue slots. Register files are clustered, meaning they can be plural and distributed to subsets of the processors' computational units. Each DSP has multiple explicitly managed local scratchpad memories. There are no data caches.¹

For benchmarking, we simulate one DSP and two local data memories offering scalar and vector access, respectively (see Figure 9). This configuration of DSP and local memories is similar to configurations found in commercial Intel IPUs, although the latter may have additional local memories providing different access modes.

All DSP code is compiled and scheduled using Intel's proprietary *HiveCC* compiler (rev. 20161027). The resulting VLIW schedule is static. Intel's *hivesim* cycle-accurate processor simulator is invoked to simulate each iteration of execution on the DSP. This simulation is cycle-accurate and deterministic. This is possible in part because access latency to local scratchpad memories is constant, and an arbiter on the local memories' access ports always gives the DSP priority over external accessors. Variations in runtime can only occur due to data-dependent control flow, such as data-dependent loop bounds, but none of the programs we present include such behavior.

Cycles spent in the testbench host processor are excluded from the measurement. This refers in particular to data transfers to/from the DSP's local memories, which happen before and after the execution of each DSP iteration, respectively. This execution model is not true to how applications

¹Note that protection of Intel trade secrets prevents the disclosure of the exact DSP revision used, as well as quantitative technical details (e.g., clock speeds, memory sizes, typical number of cores) in this work.

would run in a real-world situation. Data transfers would be carried out by a DMA instead of directly by the host, and double buffering would be applied to hide the latency of data transfers behind that of computation by DSPs.

If we assume that this latency is successfully hidden and sufficient space is reserved in local memories for double-buffering, then the results from our DSP simulations accurately represent the runtime of a real-world application. However, justifying this assumption is not trivial. In the Intel IPU system in which our targeted DSP resides, two-dimensional DMA transfers are typically requested dynamically from a lightweight data transfer server running on a microcontroller. This means data transfer latency depends on the server's performance, the DMA's performance, the DDR performance, and any other external actors trying to simultaneously use one or more of these resources.

To provide reasonable assurance that data transfer latencies can be hidden for the applications under test, we have separately characterized the total latency characteristics of the overall data transfer service in the Intel IPU. This characterization was run on silicon. A DSP in the IPU was set up to request input and output transfers from/to DDR memory to/from its local vector memory, measuring the total latency of the transaction against the requested transfer size. The resulting curve was approximated by a linear regression. Based on this model, we have estimated the total amount of cycles the combined double-buffered data transfer would take for each of the applications under test. Note that while the platform is designed to service multiple DSPs and accelerators simultaneously using a multitude of DMAs, this empirical latency estimation provides reasonable confidence only for applications running in a similar context on silicon: with only a single active DSP core contending for data transfer services and no external actors heavily contending for access to the same DDR memory.

4.2 Benchmarks

Two sets of benchmark applications were used to evaluate our contributions.

Hand-written C vs. Halide: Several DSP applications of which hand-tuned C versions are available internally at Intel were re-implemented in Halide and benchmarked. In choosing our Halide schedules, we deliberately imitated the execution strategies of the corresponding C versions. This is because Halide's advantages in quickly exploring high-level scheduling options are already well known. Choosing equal strategies instead shows us how Halide's low-level code optimization and generation capabilities compare to those of an expert embedded C programmer. We evaluate each application's performance, as well as the lines of code used to describe the application in each language.

Cross-platform benchmarks: Several publicly available Halide applications, which were previously benchmarked in Mullapudi et al. (2016), were benchmarked on the target DSP. The results help determine whether DSPs are valuable additions to the existing range of Halide targets. Note that, depending on the target DSP's instruction set, minor changes to the algorithm can lead to major performance leaps. For example, the Intel DSP used does not support floating-point arithmetic and benefits greatly from avoiding of division operations. Algorithms can be tailored to these properties by applying fixed-point conversion and dividing only by powers of two. Such tailoring is not limited to DSPs: Many existing Halide algorithms are designed to make the best use of their specific target's capabilities. Therefore, in addition to writing a new execution schedule and inserting copy stages, several minor changes were made to the benchmarked Halide applications to tailor them reasonably to the target DSP. To shed some light on the effect of these modifications on performance on general-purpose processors, both the unmodified and modified versions were benchmarked on an additional target (an AMD A8-4500M, typically used in laptop computers).

Table 1. Brief Description of Each Benchmark. Input Sizes Shown Only for Benchmarks of Which the Absolute Execution Times Are Disclosed

Benchmark	Set	Description	Source	DSP Tailoring
Sobel	Halide vs. C	Sum of horizontal and vertical 5x5 Sobel filters [Sobel 1978]. 512×512 greyscale input.	Imitation of DSP C version	n/a
CannyDet	Halide vs. C	Partial Canny edge detection algorithm [Canny 1986] (all except the recursive edge propagation stage). 512×512 greyscale input.	Imitation of DSP C version	n/a
MLBP	Halide vs. C	Modified Local Binary Pattern (also known as modified Census Transform) feature extractor [Fröba and Ernst 2004]. 512×512 greyscale input.	Imitation of DSP C version	n/a
Harris	Cross-Platform	Harris corner detector [Harris and Stephens 1988]. 1530×2554 RGB input.	Public ^a	Conv. 32-bit float to 16-bit int
MatMul	Cross-Platform	Dense matrix multiplication. 2048×2048 input matrices.	Public ^a	Conv. 32-bit float to 16-bit int
Conv	Cross-Platform	A single convolutional layer, as typically found in deep neural networks. $128 \times 128 \times 64 \times 4$ input samples.	Public ^a	Conv. 32-bit float to 16-bit int
Camera	Cross-Platform	FrankenCamera pipeline for creating a color image from raw camera data [Adams et al. 2010]. 2560×1920 RGB input.	Public ^a	Gamma curve approximated by a 32-way interpolation (resulting error < 1%).
Blur	Cross-Platform	The 3x3 blur shown in Figure 1. 6400×2400 greyscale input.	Public ^a	Approximate integer division by 9 as $(\times 7)/64$

^aMade available by Mullapudi et al. (2016).

In addition to the above, several combined applications were produced by applying automatic loop nest merging on pairs of the aforementioned applications with their original schedules. These fused applications were benchmarked against the individual versions.

All benchmarked applications, their origin, and any additional modifications made tailoring to the target DSP are briefly described in Table 1.

Scheduling: For the cross-platform benchmark set, new Halide schedules were devised. We loosely followed the same approach as the auto-scheduler presented in Mullapudi et al. (2016) but with additional steps and constraints. We did not use an exhaustive approach but instead made scheduling choices based on reasoning and intuition, trying several options if unsure which was

the best decision. We also found that the existing schedules for the handwritten C benchmark set could likely have been the result of a similar approach.

First, loop variables were re-ordered to maximize data re-use in inner loops. For example, in the separable *Blur* benchmark, re-use of partial horizontal results was achieved by computing outputs in a vertical order. Then, intermediate buffering and loop nest merging options were considered in terms of their likelihood to improve re-use of computed values and/or ILP in inner loops. It can be noted that for applications more complex than those handled in this work, this step would be significantly harder to approach intuitively, warranting a more systematic approach such as the one proposed in Mullapudi et al. (2016).

Finally, all applications were tiled. This is an absolute necessity to make buffers fit the local memories of the DSP and also improves locality. The tile size was constrained by the total local memory footprint. We scheduled such that there was always space left in local memory such that double buffering may hypothetically be applied on data transfers. The tile shape tends to pose a tradeoff. Some benchmarks can exploit most re-use if computing several lines of elements in the inner loop. However, it is also advantageous to have narrower tiles that are lengthened in a particular direction, such that more loop iterations in that direction exist, which improves the effectiveness of software pipelining. In this tradeoff, we made intuitive choices, based on the results of trying several options. Image data were assigned to vector memory, except if the inner loop after vectorization did not exhibit a vector access pattern (for example, vectorization of an access to/from a fixed index).

In cases where re-use across multiple input tiles was possible, we also applied *store_at()* commands, which can be used to explicitly enable retaining tiles in a circular buffer in local memory across DSP execution iterations. This is not a step that is described in Mullapudi et al. (2016), which can be assumed is due to data caches offering such re-use automatically.

The rest of the scheduling we applied aimed to maximize the ILP in the inner loop to reach a high density of the resulting inner loop's VLIW instruction schedule. Several techniques were used for this:

- Unrolling inner loops. We have unrolled only up to 4 times to retain a reasonable program size.
- Applying software pipelining on inner loops (see Section 3.3) if enough iterations were available for it to be fruitful. This is a step omitted by Mullapudi et al. (2016), but only because it can be assumed that the compiler used therein applies software pipelining automatically.
- Vectorizing all inner loops by the target's native vector width.

After going through this scheduling process, we estimated the data transfer latency per DSP tiled execution iteration and measured the corresponding computation time from simulation. If the results suggested that data transfer was expected to bottleneck the program's execution, then we went back and enlarged tile sizes to more effectively hide the size-invariant portion of data transfer latency.

We conclude that the process of scheduling Halide code efficiently for VLIW architectures is not fundamentally different from scheduling for general-purpose architectures. The existence of hard constraints to generate correct programs does make the process more involved for the programmer, as well as the additional task of assigning buffers to memories.

4.3 Reference Architectures

To provide a context for the results of IPU DSP benchmarks, the aforementioned public Halide benchmarks are also presented for several general-purpose architectures:

Table 2. Estimated Data Transfer Latency per Execution Iteration of the DSP, as a Percentage of Corresponding Simulated DSP Compute Latency

Algorithm	Transfer vs. compute latency
Blur	96%
Camera	0.9%
CannyDet	25%
ConvLayer	22%
Harris	27%
MatMul	100%
MLBP	116%
Sobel	90%

Table 3. Total Execution Cycles of Halide vs. C Benchmarks

Algorithm	Exec. cycles ($\times 10^3$)	
	C	Halide
Sobel	181	188
CannyDet	385	479
MLBP	220	230

- (1) An AMD A8-4500M PileDriver mobile CPU (four cores, 1,900MHz);
- (2) A mobile ARM A57 CPU as found in the NVIDIA Tegra X1 (four cores, 1,900MHz);
- (3) An NVIDIA K40 GPU (745MHz);
- (4) An Intel Xeon E5-2620 v3 server CPU (single-core and six-core execution, 2,400MHz).

Benchmarks for architectures (2), (3), and (4) are quoted from Mullapudi et al. (2016), always selecting the best result between the manually tuned and auto-scheduled versions therein. These benchmarks therefore do not include the DSP tailoring modifications shown in Table 1. To serve as an example of how they impact performance on a general-purpose architecture, the benchmarks of both the unmodified and DSP-tailored versions are presented for architecture (1).

4.4 Results

Table 4 shows the Lines of Code (LoC) for the *Sobel*, *MLBP*, and *CannyDet* algorithms in both C and Halide. This can be considered an indicator of code complexity. In these cases, the Halide implementation requires a dramatically lower number of lines. The code's understandability and development time are discussed in Section 4.5.

Section 4.1 mentions that we have estimated the expected data transfer performance as a linear fit to measurements made on silicon. Table 2 shows the estimated data transfer latency based on this model as a percentage of the corresponding algorithm's simulated, cycle-accurate DSP compute latency both for a single tiled execution iteration on the DSP.

Table 3 shows the execution times of the Halide benchmarks that we compared to hand-tuned C versions. Execution times for these benchmarks are listed in cycles to protect Intel trade secrets. Note that the DSP core under investigation is only a subcomponent of the full Intel IPU system, which contains a multitude of such cores along with fixed-function accelerators. Figure 10 shows the relative throughput. These results show that with our optimizations in Halide's compiler, the tested Halide applications approach the performance of the C versions they imitate. The remaining performance gap is further discussed in Section 4.5.

Table 2 shows the estimated time spent on data transfer per tiled DSP execution iteration, as a percentage of the simulated inner loop compute time. Figures under 100% suggest that since the estimated data transfer time is lower than the compute time, it is expected that data transfer latency can be fully hidden if the application is run in a double-buffered fashion. A figure over 100% suggests that we expect the application to be bottlenecked by data transfer.

Table 5 shows the final execution times on the DSP for each publicly sourced Halide program, as well as execution times for each reference architecture. A subset of this data is visualized in

Table 4. Lines of Code (LoC) for DSP Algorithms in C and Halide. Lines Were Measured Excluding Any Unused Code, Comments, and Blank Lines

Algorithm	Lines of Code	
	C	Halide
Sobel	368	74
CannyDet	432	110
MLBP	340	59

Fig. 10. Relative throughput on Intel IPU DSP, between Halide and C implementations.

Table 5. Execution Times of Halide Benchmarks (All in ms). As Shown, Some Benchmarks Include the DSP Tailoring Steps Shown in Table 1, Whereas Others Are Unmodified

Bench- mark	ARM A57 (4 cores) ^a	NVIDIA K40 ^a	Intel Xeon (1 core) ^a Original Version	Intel Xeon (6 cores) ^a	AMD A8-4500M (4 cores)	AMD A8-4500M (4 cores) DSP-tailored version	Intel IPU DSP
Harris	44.0	1.1	36.5	9.7	43.6	13.8	34.9
MatMul	1892	20.3	1364	293.6	1915	1916	375.4
Conv	395.3	35.8	212.6	48.3	292.7	117.7	254.9
Camera	26.8	2.0	41.3	7.8	35	35.5	44.7
Blur	18.1	2.0	26.4	7.6	26.9	27.1	10.8

^aQuoted from Mullapudi et al. (2016).

Fig. 11. A subset of the benchmark results in Table 5, presented as relative throughput. NVIDIA K40 and Xeon six-core results are omitted to emphasize the differences between targets that perform comparably.

Figure 11 in terms of relative throughput. We see that when compared to other architectures, the simpler algorithms (*MatMul*, *Blur*) seem to perform better on the DSP than more complex algorithms (*Camera*, *Harris*), which consist of more chained stages. We discuss this observation in Section 4.5.

Figure 12 shows the relative speedup resulting from peephole and memory access optimizations, normalized to the best achieved result. It is shown that our optimizations to adapt Halide’s code

Table 6. Performance Gain from Automatically Fusing Applications. The Execution Time Decreased by the Percentages Shown, Compared to Summing Both Individual Application's Execution Times. Sobel_U and MLBP_U Are Sobel and MLBP, with Less Optimized Schedules. They Both Have Lower ILP, Leading to a Higher Gain from Fusion. Inputs and Outputs Are Not Shared Between the Fused Programs

App. 1	App. 2	Fusion Speedup
Blur	Blur	14.3%
MatMul	Conv	13.1%
Sobel	MLBP	11.6%
Sobel_U	MLBP_U	31.8%

Fig. 12. The throughput effect of peephole and memory optimization passes in the Halide compiler. Features are cumulatively enabled from left to right. All points already include the storage assignment extension, which if absent (i.e., using only external DDR access) results in lower performance by orders of magnitude.

generator to target-specific features (in this case, gather/scatter emulation and multi-vector shuffle support) improve performance by up to a factor of 7, given the same Halide source code. We also see that in our test cases, memory access optimizations in Halide's compiler improve performance by up to 30%.

The performance results presented should be evaluated in the context of each processor's die area and power consumption. However, details regarding our target DSP's power consumption and die area cannot be disclosed in this work. For purposes of illustration, suffice it to say that on currently applicable silicon process nodes, area for such a DSP (including local memories) is in the order of magnitude of 1mm^2 , and a pessimistic bound on total power consumption is 1W for the use cases considered (including external memory access). For the general-purpose reference architectures used, exact figures for these metrics are also not available. However, it is reasonable to estimate that the area and power consumption (compensated for process node differences) of the AMD, ARM, and single-core Xeon targets are an order of magnitude higher—likely two orders of magnitude higher for the six-core Xeon and NVIDIA K40.

Table 6 shows the performance increase resulting from fusing multiple loop nests together automatically. We see that fused applications are able to outperform the two individual versions. Additionally, fusing relatively unoptimized loop nests together leads to a relatively larger improvement in performance after merging. This suggests that loop nest merging can be used as an alternative to heavy optimization.

4.5 Discussion

Prior work has shown that Halide can dramatically reduce development times and code complexity for general-purpose processors (Ragan-Kelley et al. 2013). The lines of code presented in Section 4.4 for the *Sobel*, *MLBP*, and *CannyDet* programs suggest that this may also be the case for DSP code. Although we do not present any systematic comparison of development times, it is worth noting that these programs originally took one to several weeks to implement and optimize in C, while

none of the respective Halide versions took us more than 2 days. In both cases, development started from a simple scalar version of the algorithm. This information is anecdotal, and there was already an effective overall execution strategy known when implementing the Halide versions. However, by using Halide, we avoided spending time on selecting vector instructions, refactoring loops, and adjusting the code that facilitates data transfers in C.

It is also important for code to be understandable. Writing and understanding Halide code requires an initial learning phase for getting familiar with its functional style and scheduling concepts. However, we found that the end result is code that cleanly describes the application's functionality, while also intuitively representing key optimization choices. C code tends to mingle these aspects together, while also requiring large amounts of trivial but necessary statements (also known as "plumbing code"). Although our proposed storage scheduling model slightly increases the scheduling complexity of Halide, the same advanced optimization choices also need to be made (and implemented) when using C, which tends to require more code and debugging.

Figure 12 shows that just having a framework to generate code for DSP targets is not enough to achieve acceptable performance, even with support for scratchpad memories. High performance on DSPs is likely to require target-specific optimizations in Halide's compiler. The amount of improvement achieved by our memory access optimizations depends heavily on whether the same optimizations are performed by the back-end C compiler as well. However, proprietary DSP C compilers tend to rely relatively little on such automatic optimizations and more on hand-tuning of C code. This is confirmed by our results. Thus, our generic Halide compiler optimizations may lead to performance improvements for other (future) DSP targets.

From Figure 10, it is clear that a performance gap remains between Halide and C implementations of the same algorithm, despite our compiler optimizations. However, towards the goal of positioning Halide as a competitive alternative to C, we regard these results as a success, because they show that abstracting the programming interface does not need to result in losing a lot of performance. The remaining difference is also bridgeable: Inspection of the assembly shows that it can be largely attributed to (1) a few more exotic vector instructions not being used and (2) overhead stemming from runtime computation of each buffer access index, where the reference C code often just increments a pointer from its previous value. Additional target-specific compiler passes and generic optimizations to Halide's C code generator, such as loop invariant code motion, should be added to address these issues specifically.

Based on our empirical characterization of data transfer latencies in the Intel IPU system as discussed in Section 4.1, one of the evaluated applications (*MLBP*) is expected to be bottlenecked by data transfer if run in a double-buffered fashion on silicon. Since it imitates the C version of *MLBP*, we expect the same for the C version. It must be noted that the input size of 512×512 was an arbitrary choice. *MLBP*'s schedule is line based, which means its "tile size" is dependent on the input/output size chosen. We also observe that plenty of buffer space is left in scratchpad memory for the schedule implemented, meaning the input image could be widened by several factors without exceeding scratchpad memory sizes. Doing so would decrease the relative overhead of software pipelining pro- and epilogues in the inner loop and also increase the total compute time per DSP execution. The latter has as a result that the size-invariant component of data transfer latency can be better hidden. Therefore, we expect that data transfer latency for *MLBP* could be hidden as well if a bigger data size is chosen, which may be an assumption made during Intel's scheduling choices in C as well.

Apart from *MLBP*, there are several other applications that approach the limit of being able to hide data transfer latency. This shows the importance of considering this latency while making high-level scheduling choices for platforms—an aspect where dealing with tradeoffs between memory footprint and data transfer overhead are common. Halide enables exploring this tradeoff. For

example, using existing scheduling commands, the *MLBP* application could easily be re-scheduled to buffer two input blocks and two output blocks, transferring only on every second iteration—a step that would lead to better hiding of the size-invariant component of data transfer latency. It would be helpful to also offer a model of data transfer latency as a visual aid to the programmer so instant feedback is received regarding the expected effectiveness of double buffering.

Note that latency hiding tends to be most easily achieved for algorithms with high arithmetic intensity: The applications that have an expected data transfer latency close to being “unhideable” are the least compute-intensive ones in the overall list, whereas the most compute-intensive one (*Camera*) has an easily hideable data transfer latency. This is to be expected, as compute-intensive applications have the longest inner compute loops to hide data transfer behind.

In Section 4.4, we mention that simple algorithms perform better on the DSP compared to more complex ones. Inspecting each algorithm’s schedule reveals that *MatMul* and *Blur* can be easily scheduled in a way that leads to high ILP, whereas *Camera* and *Harris* have more implicit data dependencies in their inner loops, leading to lower ILP. The difference is observable in the density of their VLIW instruction schedules.

Analysis of the generated code exposes another reason for the lower ILP of *Camera* and *Harris*: The bounds inference process, employed by Halide to compute required regions for each stage, often leads to buffer sizes that result in expensive unaligned vector loads and/or stores. In such cases, a human programmer would typically pad each stage’s production extents to the nearest multiple of the vector size, thereby computing several dummy values at the beginning and the end but alleviating the alignment issue. While it is possible in Halide to pad the upper limit of a stage’s production region through scheduling commands, one cannot yet pad the lower limit. We expect that such a feature would improve multi-stage pipeline schedules on VLIW processors.

The proposed loop nest merging feature is shown to be attractive as a way to squeeze slightly better performance out of already heavily optimized applications or as an easier alternative to heavy optimization. However, it should be noted that loop merging opportunities do not always present themselves: The individual loop nests may not contain attractive loop pairs to fuse (for example, if they have vastly different iteration counts), and the summed sizes of both programs’ buffers may exceed the available scratchpad memory size. One of the easiest and most common use-cases is to fuse an application with itself, leading to a loop nest that performs the same operations simultaneously and independently on two datasets.

5 RELATED WORK

Programmability of diverse architectures is an actively researched topic. In this section, we support the choice for a Halide-based approach for DSPs by outlining alternative solutions that also aim to decrease implementation and optimization effort. We also point out recent work on Halide that has affected our 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 that uses polyhedral techniques to automatically apply loop transformations on LLVM IR. It supports many of the same transformations that Halide can describe. However, it relies on the compiler’s capability to recognize parallelization opportunities in an algorithm implemented according to a sequential imperative paradigm, which may require the programmer to rewrite the initial loop to facilitate this recognition. On the other hand, a DSL like Halide can algorithms using an inherently parallel functional paradigm and a concise, abstract

schedule. *Polymage* (Mullapudi et al. 2015) is another DSL that offers such an abstract description style, inferring the schedule fully automatically. It generates optimized loop nests using polyhedral optimization techniques. The polyhedral framework provides a powerful means to compose the schedule of multidimensional quasi-affine functions, whereas Halide function composition relies on the inference on bounding boxes around the data accessed by kernels. As a consequence, a polyhedral framework can apply optimizations based on non-rectangular tiles achieving better schedule for quasi-affine algorithms, but Halide can describe and optimize a larger spectrum of algorithms that are non-affine, as, for instance, image processing for lens distortion corrections. Additionally, Halide is embedded in C++, and due to its rising popularity in the academic and industrial worlds, it has many back-ends, which makes it attractive to port algorithms across platforms.

SPIRAL (Püschel et al. 2004) is a project encapsulating multiple efforts to generate highly efficient code for digital signal processing algorithms autonomously. SPIRAL's major program generation tools generate implementations of a wide range of signal processing functions such as discrete Fourier transforms, filters, and basic linear algebra. Some of its tools utilize the polyhedral model for loop optimizations. SPIRAL has been targeted to distributed architectures with shared memories but not those with software-managed scratchpad memories. Furthermore, unlike Halide and Polymage, SPIRAL is not aimed specifically at imaging.

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 tradeoff between various metrics. Using such fixed optimizations also disregards possibilities for optimizing across functions, most notably fusion for data locality.

OpenCL (Howes 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.

Qualcomm's *Symphony SDK* (Qualcomm 2017; Kumar 2016) provides a flexible SDK that allows efficient mapping of an application onto a heterogeneous multi-core platform through C++ APIs. As such, a unified interface is provided for programming the overall application and high-performance kernel code. This allows for an efficient mapping of parts of code onto accelerator devices in mobile systems. Symphony relies on OpenCL for facilitating the offloading to GPUs. Halide offers a similarly scoped overall package but extends this with the ability to perform high-level application optimizations such as tiling and loop fusion. These optimizations help improve final application performance further, as they can impact data locality and memory access patterns. Halide also differentiates itself by opting for a functional language syntax. The Symphony SDK does offer dynamic scheduling of parallel compute tasks over the system's CPU, GPU, and DSP resources and manipulation of power management settings to fine-tune for performance or power-efficiency. These features are not offered by Halide. It remains to be seen whether and how such capabilities could be best added to the Halide framework. We discuss this in Section 6.

5.2 Related Halide Work

Halide support for *Qualcomm Hexagon DSPs* is currently under development (Halide Project 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 2 do not apply to Hexagon (which has an LLVM compiler, a single address space, and data caching).

Ansel et al. (2014) have proposed a tuner that can determine and improve Halide schedules through a heuristic search. Polymage (Mullapudi et al. 2015) has provided automatic determination of execution schedules from an abstract DSL. Mullapudi et al. (2016) have presented an algorithm to automatically develop Halide schedules without requiring iterative execution-based tuning.

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 that 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

In this article, we have presented a Halide-based approach to DSP C code and testbench generation, which can be applied to a wide range of DSP targets. Moreover, we have demonstrated that with an extended storage model, an automatic pipeline merging feature, and target-specific compiler optimizations, Halide code can reach performance comparable to hand-optimized C code on a DSP at a fraction of the development effort. Halide is well suited to describe imaging tasks that imaging DSPs excel at.

Halide is likely suitable for supporting all major processor types used in SoC platforms. Automatic scheduling enables programmers to write efficient imaging code for any CPU or GPU target with Halide support, without detailed knowledge about these targets. Extending auto-scheduling techniques such as those used in Mullapudi et al. (2016) for DSPs would further reduce programming effort. To realize this, automatic scheduling algorithms need to be extended to take into account optimization choices specific to VLIW cores and heterogeneous scratchpad memories.

The extent to which the use of Halide contributes to a reduction in development effort on DSPs could be evaluated by letting multiple skilled developers produce an image processing application to a specification in multiple languages, including Halide. Recording the performance of the end product and the time spent in each language would provide insight into the (dis-)advantages of Halide compared to other languages.

While the core capabilities of Intel's IPU DSP are targeted by the extensions presented in this work, there are ample opportunities for further optimization: many exotic instructions may be targeted using additional pattern matching in Halide's compiler. Making use of fixed-function accelerators in the IPU system would enable Halide-generated applications to increase performance and power efficiency further but poses a challenge as such components are non-programmable. A possible approach could be to handle them as extern function calls, which are currently supported in Halide, which would reduce the challenge to mostly the implementation of interfacing with said accelerators correctly.

The evaluation of Halide compared to hand-tuned C for programming imaging DSPs can be improved by porting and examining a larger set of imaging applications, especially ones containing more complexity. This would shed light on whether execution strategies of said tasks could be equally well imitated or improved through high-level transformations.

Extending Halide's scheduling model with 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 using such concepts could statically fuse the loops of subsequent stages, in a way similar to the inter-program loop merging presented in this article.

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 (including DMAs), 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

- Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. 2010. The Frankencamera: An experimental platform for computational photography. *ACM Trans. Graph.* 29, 4, Article 29 (July 2010), 12 pages. DOI : <http://dx.doi.org/10.1145/1778765.1778766>
- Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 303–316. DOI : <http://dx.doi.org/10.1145/2628071.2628092>
- Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc.
- Cadence. 2015. Tensilica Vision P5 DSP Datasheet. (2015). Retrieved May 27, 2016 from <http://ip.cadence.com/vision>.
- John Canny. 1986. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 6 (1986), 679–698.
- CEVA. 2015. CEVA-XM4 Intelligent Vision Processor White Paper. (Feb. 2015).
- Lucian Codrescu, Willie Anderson, Suresh Venkumhananti, Mao Zeng, Erich Plondke, Chris Koob, Atul Ingle, Charles Tabony, and Rick Maule. 2014. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, IEEE 34, 2 (2014), 34–43.
- Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 5.
- Bernhard Fröba and Andreas Ernst. 2004. Face detection with the modified census transform. In *Proceedings of the 6th IEEE International Conference on Automatic Face and Gesture Recognition, 2004*. IEEE, 91–96.
- Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly: Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4 (2012), 1250010.
- Halide Project. 2016. Halide GitHub Repository. Retrieved July 2, 2016 from <http://github.com/halide/Halide>.
- Chris Harris and Mike Stephens. 1988. A combined corner and edge detector. In *Alvey Vision Conference*, Vol. 15. Citeseer, 50.
- Lee Howes. 2015. The OpenCL Specification v2.1 Rev 23. Specification. Khronos OpenCL Working Group.
- Intel. 2016a. 6th Generation Intel Processor Datasheet for UY-Platforms. (March 2016).
- Intel. 2016b. Integrated Performance Primitives. Retrieved February 11, 2016 from <http://software.intel.com/en-us/intel-ipp>.
- Mircea Horea Ionica and David Gregg. 2015. The movidius myriad architecture's potential for scientific computing. *IEEE Micro* 35, 1 (2015), 6–14.
- Joel Jones. 2003. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLOP'03)*. 1–10.
- Roel Jordans. 2015. *Instruction-set Architecture Synthesis for VLIW Processors*. Ph.D. Dissertation. Eindhoven University of Technology.
- Tushar Kumar. 2016. Heterogeneous Computing Made Simpler with Symphony SDK. Retrieved April 2, 2017 from <https://developer.qualcomm.com/blog/heterogeneous-computing-made-simpler-symphony-sdk>.
- Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Not.* 23, 7 (June 1988), 318–328. DOI : <http://dx.doi.org/10.1145/960116.54022>

- Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, USA, 75. <http://dl.acm.org/citation.cfm?id=977395.977673>.
- Zhihong Lin, Jagadeesh Sankaran, and Tom Flanagan. 2013. Empowering automotive vision with TIs vision accelerationpac. White Paper SPRY251. Texas Instruments.
- Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. DOI:<http://dx.doi.org/10.1145/2897824.2925952>
- Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443. DOI:<http://dx.doi.org/10.1145/2786763.2694364>
- Cedric Nugteren and Henk Corporaal. 2012. Introducing “bones”: A parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units (GPGPU-5)*. ACM, New York, NY, 1–10. DOI:<http://dx.doi.org/10.1145/2159430.2159431>
- Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. 2004. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perf. Comput. Appl.* 18, 1 (2004), 21–45.
- Qualcomm. 2017. Symphony System Manager SDK. Retrieved April 2, 2017 from <http://developer.qualcomm.com/software/symphony-system-manager-sdk>.
- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Not.* 48, 6 (2013), 519–530.
- Jonathan Millard Ragan-Kelley. 2014. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. Ph.D. Dissertation. Massachusetts Institute of Technology. Cambridge, MA.
- Erik Rainey, Jesse Villareal, Goksel Dedeoglu, Kari Pulli, Thierry Lepley, and Frank Brill. 2014. Addressing System-Level Optimization with OpenVX Graphs. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'14)*. IEEE Computer Society, Washington, DC, USA, 658–663. DOI:<http://dx.doi.org/10.1109/CVPRW.2014.100>
- Irvin Sobel. 1978. Neighborhood coding of binary images for fast contour following and general binary array processing. *Comput. Graph. Image Process.* 8, 1 (1978), 127–135.

Received October 2016; revised April 2017; accepted May 2017