Profiling Halide DSL with CPU Performance Events for Schedule Optimization

RAFAEL RAVEDUTTI L. MACHADO, Universidade Federal do Paraná, Brazil
ANDRÉ MURBACH MAIDL, Elastic, Brazil
DANIEL WEINGAERTNER, Universidade Federal do Paraná, Brazil

Halide is a domain-specific language (DSL) for image processing that enforces a separation of the algorithm and the execution schedule, allowing the generation of specialized code for distinct computer architectures by rewriting only the execution schedule, instead of the whole algorithm. In order to support the creation of good Halide schedules, our work extends the Halide DSL by adding a profiling API that uses the CPU Performance Events to measure events supported by the target processor during the application runtime. The proposed extension offers profiling of the application loop levels and functions’ producer and consumer relations, embedding calls to a profiling library in the loop nests of the generated code. It also supports individualized profiling by threads on parallel regions. As a case study we use the PAPI library in order to count events such as L1 cache misses, number of float operations (FLOP) and L3 data volume on an Intel Core i5-7500 CPU, and discuss how the reported results can be used to manually or automatically generate better schedules for an image processing pipeline.

CCS Concepts:
- Software and its engineering → Domain specific languages; Software performance; Source code generation;

Additional Key Words and Phrases: image processing, performance, profiler, domain specific language

ACM Reference Format:

1 INTRODUCTION

Image processing algorithms are widely used in mobile and desktop computers for image edition/enhancement, and are frequently embedded in imaging equipment and digital cameras. The majority of these algorithms require good performance, specially on portable hardware, demanding appropriate code optimization in order to run with a limited time and/or energy budget. Tuning image processing algorithms for optimal resource usage is not an easy task, because it requires a good understanding of how the code is executed in each target platform. Moreover, given the multitude of available processors and system configurations, finding the appropriate computation schedule to achieve the best performance on each of them can be a demanding and time consuming task, requiring multiple versions of the same code to be implemented and maintained.

Halide [9, 10] is a Domain Specific Language (DSL) for image processing that allows the separation of the algorithm from its schedule, where the algorithm specifies the operations to be executed and the schedule specifies how the
generated code must be organized. Schedules in Halide do not influence in the final output of the program, as they are only used for performance purposes.

Halide makes it much easier to write optimized image processing programs because only the schedule has to be adjusted, thought writing good schedules for Halide is not an easy task. To achieve good scheduling of Halide pipelines, it is still necessary to have a deep understanding of the target architectures. Besides, when the size and complexity of the pipeline grow, finding good schedules becomes much more challenging for the schedule developers.

A profiler is a good tool to help the developer on program optimization because it allows to identify bottlenecks and hot-spots in a code by providing information about resource usage during the application runtime, i.e., by measuring CPU events such as cache misses, float operations and data transfers between cache levels.

In this paper we present an extension to the Halide DSL to enhance its profiling capabilities by allowing the instrumentation of the generated code for a given pipeline schedule. We focus on counting CPU performance events during the application runtime using the PAPI Library [11]. This information can be used by experienced programmers to decide on the best schedule for a specific hardware, but can also be used to provide feedback to machine learning driven automatic schedule generation systems [6, 7].

We describe the proposed extension to the Halide language, how the profiling code snippets are inserted into the Halide generated code, its coupling to the PAPI library during runtime and demonstrate its capabilities by counting L1 cache misses, the number of floating-point operations (FLOP) and the L3 cache data volume on four different schedules for the image Blur pipeline.

The remaining of the paper is organized as follows: Section 2 summarizes related work; Section 3 introduces the basic concepts of the Halide language; Section 4 presents the PAPI Performance API; Section 5 details the proposed profiling extension to the Halide DSL and Section 6 shows the resulting measurements of profiling different schedules of an image Blurring pipeline. Finally, Section 7 concludes with some future perspectives using the Halide profiler.

## 2 RELATED WORK

Profiling an algorithm at runtime is a very effective method for finding bottlenecks. A profiler usually shows statistics about resources usage during the execution of an application. This information helps programmers to tune their applications by exposing critical resources usage. Even though profilers can help programmers find bottlenecks, they are still not much explored for high performance code generation on DSLs. To profile code generated from a DSL is not trivial since it is necessary to insert the annotations in the critical parts of the generated code, whilst the instrumentation is done in the DSL, and it is usually not clear how the DSL code will be transformed.

Work on tuning GPGPU applications already relies on profiling measurements to find the best kernel configurations. Guerreiro et. al. [5] select the best GPU operation frequency, grid and block sizes configurations aiming performance and energy-aware optimization. The auto-tuner decisions are based on measurements of time and variation in energy consumption. However this work does not look for better algorithm alternatives and just takes into consideration the kernel configurations and GPU operation frequency.

Weber et. al. [15, 16] uses guided profiling integrated with an auto-tuner for finding the most suitable array layout on GPUs and optimize the array accesses in CUDA applications. The work uses a predictor to reduce the time required for empirical profiling, and a watch-dog that also reduces the profiling time by terminating the execution of the kernel in bad configurations.

Halide [8, 10] already has a simple profiler that can estimate memory footprint, time, and average number of threads running in parallel for each function in the pipeline. These estimates are also used for decision making on automatic
Profiling Halide with Performance Events

3. However, there are few available measurements and they do not help much when dealing with larger and more complex pipelines, because most of these measurements can be considerably influenced by others. For instance, parallelism can be influenced by memory bandwidth if there is no bandwidth available at a certain moment, as threads will be stuck waiting for transferring data into the memory. Time also does not express too much since it is usually influenced by several other factors, i.e., any resource bottleneck can significantly increase time. These influences among measurements can end up misleading the schedule developer or auto-tuner and prevent it from finding the true bottlenecks for improving the schedule.

3 HALIDE

Halide [10] is a domain-specific language for describing image processing pipelines through a high-level specification which decouples the algorithms from their execution strategy. This means that a Halide program is split into two parts: the algorithm, that specifies what to execute, and the schedule, defining the strategy or how to execute the algorithm. This separation allows for an easier development of high performance image processing applications for different architectures because only the second part has to be changed and optimized.

Listing 1 shows a $3 \times 3$ Blur box filter implemented in Halide (both algorithm and schedule) consisting of two steps (functions) of $3 \times 1$ filters. The implementation specifies a Halide pipeline with two functions: $\text{blur}_x$ and $\text{blur}_y$, where $\text{blur}_x$ can be referred as the producer function and $\text{blur}_y$ can be referred as the consumer function or the output function, i.e., $\text{blur}_y$ consumes values produced by $\text{blur}_x$ to produce the final output.

3.1 Scheduling

Schedules describe how computations are organized in Halide pipelines independently of the algorithm definition. Every Halide pipeline is modeled as a directed acyclic graph (DAG) representation, where each node is a function and each edge expresses a producer-consumer relation between two functions. The major concern one must deal when scheduling an algorithm is the order of execution of the tasks in the DAG [8]. Based on the order of execution, schedules also end up defining the amount of redundant computation and the size of the memory buffers allocation used for storing the intermediate results.

A Halide pipeline DAG provides two types of choices that must be taken by the schedule for the organization of computation [8]:

(1) Choices of organization within each stage (individual node)

(2) Choices of organization across stages (edge relationships)
For the first set of choices (1), Halide schedules can determine at which granularity to compute each of the stage inputs, at which granularity to store each for reuse, and in what order its domain must be traversed — for simplicity purposes, traversal orders in Halide are constrained to traversal orders that can be trivially expressed in form of loop nests. Parallelism also may be performed by turning one or more loops in the nest parallel.

For the second set of choices (2), Halide schedules can determine the granularity at which values are grouped and interleaved between the producer-consumer relations. Since stage relations are a producer-consumer relationship, every value must be computed and stored by the producer stage before it is used by the consumer, otherwise the schedule is not valid.

Writing a good schedule for a Halide algorithm is not a trivial task, because it requires understanding how the hardware architecture is designed, and how the code reorganization and transformation impacts on its execution. Ragan-Kelley [8] argues that performance on image processing programs is limited by trade offs among parallelism, locality and redundant computations. Halide schedules allow to explore different trade off configurations. For a better understanding on how these trade offs limit performance, we use the algorithm example from Listing 1 and expose four different strategies [10] as depicted in Figure 1:

1. **Breadth-first**: executes the \texttt{blur\_x} function on the entire image and then \texttt{blur\_y} using the values produced by \texttt{blur\_x};
2. **Full fusion**: computes the \texttt{blur\_x} and \texttt{blur\_y} for each pixel in the same iteration;
3. **Sliding window**: interleaves computation of \texttt{blur\_x} and \texttt{blur\_y} by storing the values of \texttt{blur\_x} in a sliding window with the size of three scan lines (three times the image width);
4. **Tiling**: interleaves the computation of \texttt{blur\_x} and \texttt{blur\_y} at the level of tiles, so that one tile is completely processed before proceding to the next.

The breadth-first strategy performs massive parallel operations as all the pixels are computed independently. However, it has poor locality since it calculates all \texttt{blur\_x} functions for the entire image and only then starts to execute the
blur_y function (again, for the entire image). It is most likely that when the breadth-first strategy is computing blur_y for a pixel, its region is not in cache anymore, which compromises performance by forcing a load from the hardware slow main memory.

In contrast to the breadth-first strategy, full fusion has full locality, that is, all blur_y functions are performed in the same iteration right after computing blur_x for a specific pixel. But full fusion requires the computation of blur_x three times for the same pixel, while breadth-first requires only one. This redundant computation also affects performance by increasing the number of float operations performed.

The sliding window strategy has full locality (as long as the computing window fits into the cache) and does not perform redundant computation, yet it does not perform parallel computations. The sliding window strategy introduces a new dependency during a blur_y computation: to execute blur_y on a pixel, all its neighbors must be computed and stored in the sliding window first. This dependency harms parallelism from the code because every blur_y computation depends on the sliding window resource. If two blur_y operations would execute in parallel, both would get incorrect results, as both would be writing and reading from the same memory region (the sliding window region).

A better balance for scheduling the mentioned pipeline can be achieved by interleaving the computation of blur_x and blur_y at the level of tiles. Redundant computations in this approach occurs in smaller scale as in full fusion, on blur_x regions that overlap between tiles, tiles can be processed in parallel because there is no resource dependency between iterations, and locality is achieved as long as the tile region fits in cache, again with memory overhead to load the tile intersection regions.

Since all of these strategies have their drawbacks, it is not straightforward defining which one is the best. Three strategies miserably fail on attending one of the three trade offs: the breadth-first fails on locality, full fusion has a lot of redundant computations, and sliding window does not allow parallelism. The best strategy could be tiling, but this is not always the case since it also depends on factors such as image size and hardware capabilities.

3.2 Compiler

The Halide compiler uses the input algorithm and schedule to generate the machine code that implements the Halide pipeline. The compiler does not take any optimization decisions during its execution, it always resorts to the schedule for it. Also, the schedule must not change the program output, it should only affect the organization of the execution. All vectorization instructions, parallelism, data management and GPU kernels are automatically generated by the compiler. The Halide compiler can be described as a sequence of well defined steps [10], which we will briefly described.

First, the compiler performs lowering and loop synthesis. The goal is to provide the loop nests and allocations for the implementation. Starting from the output function, the compiler generates its loop nests and loops for splitting dimensions when necessary – like tiles, for instance –, covering all the required region for the input. It also labels each loop as serial, parallel, unrolled or vectorized. The schedule should define the domain that loops must traverse and their labels. The compiler then keeps traversing the input functions recursively until all the algorithm functions are lowered. Each producer function must be scheduled to be stored and computed at some loop level – not necessarily the same – of its consumer function. The compiler finds the defined loop levels and injects the buffer allocation and storage right after the loop specified by the store level, and the computation for the producer function right after the loop specified by the compute level. Notice that store levels of a function cannot be inside of its production because functions cannot be stored before they are computed. After reaching the end of the pipeline, the functions are synthesized into a single set of loops.
Next, the compiler executes **bounds inference**. Starting from the output function until the input ones, the compiler uses the function definition in the algorithm part to perform interval analysis, calculating the expression bounds for each loop of the function. Dimensions are evaluated one at a time since each dimension corresponds to a different synthesized loop for the function in the generated code. At each function the interval analysis is always done using the previous computed bounds. Finally, bounds inference goes back to the output injecting the definitions for the bound variables generated at the **lowering and loop synthesis** step.

After that, the compiler executes the **sliding window optimization** step by traversing the loop nests of the program. If a function is computed in a more internal loop than its storage, with a serial loop separating both operations, then the serial loop can benefit from reusing the computed values in the previous iterations without the risk of race conditions. The Halide compiler uses interval analysis again for shrinking the interval to be computed at each iteration since the region computed by previous iterations does not need to be recomputed. Another optimization performed in this step is storage folding. If an allocated region is only used inside a serial loop and it is monotonically traversed, then it can be folded by rewriting its access indices, which allows shrinking of the allocated region, reducing peak memory use and working set size [8].

**Flattening** multi-dimensional loads, stores, and allocations is then performed by the compiler, which causes all the buffers and indexes in the implementation to become uni-dimensional. This is done by computing a stride and a minimum for each dimension, and then transforming the accesses of a dimension into the dot product of the accessed index with the computed stride, minus the computed minimum. The flattening of multi-dimensional operations into its uni-dimensional equivalents enhances cache usage by keeping the data layout contiguous in main memory, which is usually the most appropriate layout for improving cache prediction on modern architectures.

The next step is to perform **vectorization and unrolling** of loops. Unrolling replaces a loop of size $n$ with $n$ sequential statements performing each loop iteration in turn [8], which completely unrolls the loop. Partial unrolling of loops is expressed by transforming one dimension into two and unrolling the inner dimension loop. Vectorization replaces a loop by a single statement. Every reference to the index of the loop is replaced by a special value that represents the vector with the same size as the replaced loop. Type coercion pass is then used to broadcast expressions that contain scalars combined with their respective special values. Finally, Intermediary Representation (IR) nodes are replaced by their vector type relatives, which later generates vectorized code.

At the last step, **back-end code generation** applies basic optimizations and generates code for the target backend architecture. The compiler first simplifies the IR by performing constant propagation and dead code elimination. Then, it maps the resulting IR to the LLVM IR, which at the most part is a one-to-one mapping between both IRs. After mapping to LLVM IR, the LLVM framework takes care of generating the backend machine code using its equivalent IR based on the Halide pipeline.

### 4 PAPI PERFORMANCE API

PAPI (Performance API) [11] is a framework that provides a common interface for using performance counters for a variety of microprocessors. It has C and Fortran APIs that allow preset and native events to be measured from various sources. PAPI also supports the use of components (referred as PAPI-C) [12], allowing to measure more performance events than those available in the CPU.

An usage example of PAPI-C [1] is when one wants to measure events from a CUDA GPU. To achieve this, it is necessary to compile PAPI with the CUDA component. PAPI can also be used for measuring energy and power consumed during the execution of a region of code through the use of specific components [14]. An example of this...
kind of component is the Running Average Power Limit (RAPL) component, used for measuring power and energy on Intel processors that support that feature [2].

PAPI is designed in two separate layers: the portable layer specifies the parts that are independent of the target machine, more specifically the low level and high level APIs; and the hardware specific layer specifies the interface between the hardware independent functions and the hardware dependent functions that use assembly code, operating system calls or kernel extensions for accessing the hardware counters.

Register counters can be accessed through PAPI’s high level and a low level API. The high level API is provided for simplicity purposes and provides limited PAPI measures (such as number of FLOP, cycles and instructions). The low level API can be used for developers who want more fine-grained measurement and control of the PAPI interface, as it allows the specification of event sets that define the events to be measured on the target hardware. The low level API also allows the access through all the PAPI components and native events available to specific hardware.

Advanced profiling features such as the multiplexing of events through time-sharing (used due to restrictions on the amount of registers on the target processor), measuring parallel programs and overflow handling are also supported. These features are very helpful to profile complex parallel programs and to deal with hardware specific limitations. In the proposed Halide extension we use PAPI because it can be used on a large variety of platforms, which can help to evaluate our profiler targeting other operating-systems and hardware in the future.

5 HALIDE DSL PROFILING EXTENSION

We propose an extension to Halide DSL that includes the appropriate markers for instrumenting the generated code. The extension consists of two member functions to the Func class named profile() and profile_in() and a third function named profile_at(). They can be used directly in the Halide schedule to define where the code should be profiled and how.

Listing 2 shows the C++ headers for the new functions our extension implements:

**Func::profile()**: injects the profiler markers at one of the stages (production, consumption or both) of the Func.

**Func::profile_in()**: similar to Func::profile(), but only affects the stages inside a parent function. Useful for turning the profiler off for specific parts of the code inside a region;

**profile_at()**: injects the profiler markers at the loop level of the Halide code specified at the Func parameter.

Listing 2. Headers of the profiling functions defined by the proposed Halide extension.

```c++
class Func {
  ...
  /** Profile this function. */
  Func &profile(
    int stage = PROFILE_PRODUCTION,
    bool show_threads = false,
    bool enable = true, int granularity = 1 );
  /** Profile this function stages inside
    a parent function’s production stage. */
  Func &profile_in(
    Internal::Function &parent,
    int stage = PROFILE_PRODUCTION, bool show_threads = false,
    bool enable = true, int granularity = 1 );
  ...
};
/** Profile a loop level defined by a function (Func) and a
  dimension (Var) */
void profile_at( Func f, Var var, bool show_threads = false,
    granularity = 1);`
```
These extension functions have following parameters:

**stage**: since Halide functions are converted into producer-consumer relations, this parameter defines whether to profile the function’s production stage (PROFILE_PRODUCTION), consumption stage (PROFILE_CONSUMPTION), or both (PROFILE_ALL). The default is PROFILE_PRODUCTION;

**show_threads**: allows to report the results for each individual thread in this region (true), as opposed to the default behavior of just reporting the summation of the counters for all threads (false);

**enable**: allows to enable/disable profiling, which is useful because profiling markers are enabled recursively in a loop nest and sometimes we need to disable profiling for specific regions inside this loop. Disabling profiling for specific regions can be useful if we are not interested in counting events inside them because no time is spent triggering the profiler counters;

**granularity**: defines the number of loop iterations between measurements. The performance events are only counted in a fraction of the loop iterations, thus reducing the impact of profiling in the overall runtime of the code;

**parent**: is a parameter of the profile_in() function to define in a more specific way at which stage (or Func) we want our changes (enabling or disabling the profiler) to take effect.

In order to exemplify the proposed Halide profiling extension as well as the modifications it produces in the generated code, we use the image Blur code with tiling schedule introduced in Listing 1. In this algorithm we have the blur_x and blur_y functions, and in the tiling schedule the blur_y function consumes values produced by blur_x. Production and consumption of the blur_x values occur in levels of tiles inside the production of blur_y, as can be seen in Listing 3:

Listing 3. Structure of the code generated by Halide for the Blur algorithm with a tiling schedule of 32 × 32 pixels.

```cpp
produce blur_y {
    ... // (1) Warm up code for blur_y production
    tile 32x32 { // Go through the domain in tiles of 32x32
        produce blur_x {
            ... // (2) Produce blur_x values using the input
        }
        consume blur_x {
            ... // (3) Consume blur_x values to produce blur_y
        }
    }
}
```

If we want to profile the production of blur_y in this example, our profiler must include markers separating all the regions inside the production of blur_y. In this example we have three regions:

1. the warm-up code (Listing 3, lines 2 and 11) composed of all code not inside a production or consumption region;
2. production of blur_x (Listing 3, lines 4-6); and
3. consumption of blur_x (Listing 3, lines 7-9).

With the proposed extension this can be achieved by adding the following call to the profile() function inside the Halide code, i.e. before the return command (line 14) in Listing 1:
blur_y.profile(PROFILE_PRODUCTION, false, true);

The profiler recursively includes the necessary markers for the different regions in the generated code of the blur_y loop nest, enabling profiling in all regions related to the production of blur_y. Notice the first parameter PROFILE_PRODUCTION tells the profiler we want to profile the production stage of the blur_y function, the second parameter false tells we don’t want to show each thread separately, which is only useful in parallel regions, and the last parameter true tells we want to enable the profiler.

Now, suppose we still want to profile the production of blur_y values, but we don’t want to profile the production of blur_x values inside of it. We can achieve this by adding following line of code:

blur_x.profile_in( blur_y, PROFILE_PRODUCTION, false, false );

The first parameter tells the profiler that this call should just affect child relations of the blur_y production loop nest, so that if there are other blur_x consumption regions inside other functions they would not be affected. The remaining parameters are similar to the profile() call.

We list the relevant markers created by the proposed profiling extension and provide a brief explanation of their functionality:

- **pipeline_start/stop():** define the start and end of the pipeline execution, initializing all performance counters and the profiler API (start), and reporting the results (stop);
- **enter/leave_warmup_region(f):** start and end markers for counting the performance events in the warm-up region of a function f;
- **enter/leave_func(f, prod):** start and end markers for a production or consumption region of the function f. The prod parameter is a zero or one multiplexer that indicates if the counters should be stored in the production or in the consumption counters;
- **enter/leave_loop(l):** start and end markers for a loop level region. l defines the region for counting the events and store the values in the loop level counters;
- **enter/leave_parallel_region():** defines a parallel region of the code, used to manage thread structures that trace the threads individually; and
- **incr/decr_active_threads():** defines the region where a thread is active, used to manage the event set needed to count events for the thread.

For the Blur algorithm profiled in this paper the generated code of Listing 3 is modified by the proposed Halide profiling extension by injecting the region markers, producing the code shown in Listing 4. Notice the markers that define the start and end of the warm-up region (lines 4,7 and 18,21) as well as the production (lines 9-11) and consumption (lines 14-16) regions of the Halide functions. Since this schedule does not use parallelism, there’s just one active thread during the entire execution of the pipeline, and only two markers at the start and end of the code are necessary to start and stop the event sets (lines 1-2 and 23-24).
In order to inject these markers, we traverse the Abstract Syntax Tree (AST) of the pipeline and perform modifications when finding producer-consumer relations or loop levels that should be profiled with our functions (profile(), profile_in() and profile_at()). When finding a producer-consumer relation node that must be instrumented, we look deeper in the AST to check if this node contains child producer-consumer relations. If that is the case, we inject the enter_warmup_region() marker as the first part of the current node body, and the leave_warmup_region() as the last part of the current node body. We then traverse all internal producer-consumer nodes inserting the leave_warmup_region() before entering them (Listing 4, line 7) and the enter_warmup_region() after leaving (Listing 4, line 18). This procedure is executed recursively.

When producer-consumer relation nodes that have no producer-consumer child nodes are found, we insert the enter_func() and leave_func() markers in the first and last parts of the body, respectively. For loops, the procedure is analogous to the nodes without children, but using the loop markers instead.

We simplify the tree by removing unnecessary enter_warmup_region() calls immediately followed by leave_warmup_region() calls (in Listing 4, markers between the production and consumption of blur_x on lines 12-13 were removed).

Markers pipeline_start and pipeline_end are inserted at the start and end of the AST, and markers for the activation and deactivation of a thread expressing the execution of the serial region (Listing 4, lines 2 and 23). If a parallel loop is found, we insert the parallel region markers and the markers to control the activation and deactivation of threads.
5.1 Profiler Runtime Code

So far we explained how to include calls to the profiler markers into the code generated by the Halide compiler. Figure 2 illustrates how the profiling code (white box) is called during the Halide application run. To control the profiling for all the different regions in the Halide code and also access the hardware performance counters, we need to implement these markers in a Halide runtime module (because all functions executed during runtime must be implemented in a runtime module). The Halide runtime module Profiler Module (depicted in Figure 2) contains counters for each of the profiled functions’ production, consumption and warm-up regions and/or standalone profiled loops.

However, in order for the runtime code to access hardware performance counters it needs platform-specific code and this cannot be implemented in the Profiler Module (Halide modules must be platform-agnostic). A separate library, the libprofhalide, is used to overcome this restriction, and is responsible for calling the platform-specific functions. In our case, libprofhalide uses the performance counters provided by the PAPI library [11], but any profiling tool such as perf_events [4] or Likwid [13] could be used instead. The available performance events for profiling are defined by these platform-specific libraries and the processor capabilities, not being limited in any way by the proposed Halide extension. The events to be measured (i.e. L1_CACHE, FLOP) are defined in a configuration file loaded by libprofhalide (Events box in Fig. 2).

In order to avoid introducing a new dependency to Halide, the Halide runtime module does not generate error messages during the Halide build process when the profiler library is not available, and simply ignores the profiler steps, proceeding as usual.

6 EXPERIMENTAL RESULTS

In order to demonstrate the use of the proposed Halide profiling extension we chose to use the four schedules for the Blur algorithm [8] depicted in Figure 1: breadth first, full fusion, sliding window and tile 32x32. The aim is to compare their theoretical strengths and weaknesses (trade-off between data locality, redundant computation and parallelism) with the real executions of these schedules on a general purpose processor.

6.1 Protocol

We measure time, number of L1 cache misses, number of float operations and L3 cache data transfer volume for each schedule, with single thread and four threads, on 8 Megapixel (3840x2160) and 44 Megapixel (10240x4320) images. These image sizes were chosen for their common use on high-definition devices (4K and 10K resolution) and therefore their relevance for real-world image processing applications.

Measurements were made separately for blur_x and blur_y whenever there were separate production/consumption regions. The following code was inserted after the breadth first schedule:
blur_x.profile( PROFILE_PRODUCTION );
blur_y.profile( PROFILE_PRODUCTION );

For full fusion and tile 32x32 schedules only blur_y has to be profiled because blur_x is produced inside it:
blur_y.profile( PROFILE_PRODUCTION );

On the sliding window schedule both productions of blur_x and blur_y are performed inside the most inner loop, and profiling these stages individually greatly increases the time overhead for calling the profiling functions. Therefore this schedule was profiled differently, at blur_y’s outer loop, not allowing to separate the results from blur_x and blur_y counters:
profile_at( blur_y, c );

Parallel versions of the schedules were profiled by individual threads. No parallel version for the sliding window schedule was tested due to a potential race condition in its Halide implementation.

The following PAPI events were measured:

- **PAPI_L1_DCM**: PAPI preset for counting L1 data cache misses. Poor locality schedules tend to increase these events;
- **PAPI_SP_OPS**: PAPI preset for counting the number of single precision floating-point operations. Schedules that perform redundant computation tend to increase these events;
- **L3_DATA_VOLUME**: a composed event to count Bytes transferred to/from L3 cache, which is the first level of cache memory and is shared among the CPU cores. This event allows to evaluate schedules’ utilization of the (slow) main memory bus. Increase in data volume indicates redundant load/store operations. It is computed as:

\[ L3_{DATA\_VOLUME} = \text{CACHE\_LINE\_SIZE} \times (L2_{LINES\_OUT:NON\_SILENT} + L2_{RQSTS:MISS}) \]

where:

- **L2_LINES_OUT:NON_SILENT**: CPU native event that counts the number of L2 cache lines evicted;
- **L2_RQSTS:MISS**: CPU native event that counts the number of requests that miss the L2 cache.

Experiments were executed on an Intel Core i5-7500 CPU at 3.40GHz, with 4 cores per socket and 1 thread per core (no hyper-threading), L1, L2 and L3 caches of size 32kB, 256kB and 6MB respectively, and 8GB of RAM. For the parallel schedules we used 4 threads, pinning them to the same core so as to enforce the use of the same L1 and L2 caches, improving data locality. We performed 30 executions of each schedule for each PAPI event on each image size, for one and four threads (except sliding_window), presenting the averaged results on Figure 3. Standard deviation on all experiments was below 2%.

### 6.2 Results

On Figures 3a and 3b we can observe that full fusion has a slightly lower execution time than the tile schedule for both image sizes, and they are the best performing schedules. The worst performing schedule is the sliding window. Parallel efficiency is not very good on any schedule (less than 50%), indicating that this is a memory bound problem and simply increasing the number of processors might not be the best strategy.

Figures 3c and 3d show that the full fusion schedule presents the smallest number of cache misses, since it has better data locality (each pixel is accessed only once). The breadth first schedule shows the biggest number of L1 cache misses and, as expected, they are higher in the blur_y function because there the computation traverses the image in a strided memory access pattern, resulting in bad data locality.
Redundant computation can be analyzed through the number of floating-point operations (FLOP) in Figures 3e and 3f. The number of FLOP is similar for all schedules except for full fusion, which is reasonable since full-fusion does not use a buffer to store intermediate results of the blur\_x computation, and computes the blurring in the x-direction three times for each pixel.

Data volume measured at the L3 cache level can be seen in Figures 3g and 3h. We notice that the parallel versions of the breadth first and full fusion schedules transfer more data than their single threaded counterparts. Full fusion transfers more than twice as much bytes in the 44MP image because threads split the image in y-direction, one line per thread, and thus each thread has to access the same memory as its neighbours. Breadth first accesses almost three times as much memory on the blur\_y function for the same reason. This also explains their bad parallel performance: although they realize the same amount of FLOP in total, threads make redundant memory access competing for the memory bus.

None of the measured events is capable of explaining the sliding window performance. It has average cache misses and float operations, and low data volume, but is much slower than all other schedules. In order to explain this it is necessary to hypothesize on the reasons and measure events that confirm the hypothesis. One possibility is that sliding window gets stalled because of excessive function calls in the inner-most loop. This could be verified by counting number of instructions executed per second.

For the presented schedules we can conclude that optimizing locality is much more effective than redundant computations for this specific algorithm, which demonstrates a very instructive example of the use of redundancy. On the other hand, using more computational intensive stages in a pipeline may show the opposite behavior, which is something that can be identified using a profiler.

Profiling can significantly increase the runtime (up to ten times in our experiments using the tile schedule), specially when instrumenting inner loops, because the markers must be inserted deeper in the loop nest, incurring in many calls to the profiler API. In order to mitigate this problem, the programmer can adjust the granularity parameter of the profiling functions and/or avoid inner loop profiling by disabling instrumentation on internal regions.

7 CONCLUSION

In this paper we presented a Halide DSL extension for profiling Halide code through instrumentation of the generated code with performance event counting functions. Profiling hardware events can help developers and auto-tuners to check whether specific Halide functions in the pipeline must be scheduled to optimize memory accesses or to avoid redundant computation. In deeper and more complex pipelines, finding the most appropriate optimization for each stage becomes much more challenging without the appropriate feedback.

The proposed Halide profiling extension offers three simple but powerful functions that allow the obtainment of important performance data from different regions of a Halide pipeline. To demonstrate the use of the proposed extension we profiled the Blur algorithm with four different schedules. We used the PAPI library for exposing CPU events and measured time, L1 cache misses, FLOP, and L3 data volume on serial and parallel versions.

Experimental results show that our extension provides very important performance information to Halide developers, making it easier to spot performance issues on their schedulers, considering the “right questions are asked”. We demonstrate that the number of cache misses in the breadth first vastly increases due to its poor data locality, and the number of FLOP in the full fusion schedule is also higher than the other schedules since it does not store intermediate values and performs more redundant computations. We also show that the parallel versions of breadth first and full fusion schedules perform redundant memory access.
Our approach is very flexible by allowing the integration of different profiling tools. Even though we have used PAPI in this paper, it is possible to change the profiling library by another that better suits the user requirements or hardware platforms not supported by PAPI. The choice of events to be profiled is not limited by our proposal, and should be directed by the optimization target being pursued, which can be runtime, or runtime and energy consumption.

Finally, the profiler’s output can be used by an auto-tuner to automatically generate better schedules. As future work, we intend to integrate our profiler to a machine learning based auto-scheduler in order to achieve a faster convergence by feeding the auto-scheduler with more sensitive performance data. Integrating the profiler to an auto-scheduler requires a strategy to profile each application, because the auto-scheduler must decide where to insert the markers in the generated code. When manually scheduling Halide applications, the developer is responsible for the profiling strategy, which is not necessarily a simple task.

REFERENCES

Fig. 3. Execution time in ms (a-b), number of L1 cache misses (c-d), number of float operations (e-f), and L3 data volume (in MB) (g-h) for each schedule, with single thread (1t) and four threads (4t), on 8 and 44 Megapixel images.