Efficient Incremental Computation for Halide

TYLER HOU[*], UC Berkeley, USA

Advisor: Mae Milano **Category:** Undergraduate

Interactive programs that display or manipulate graphics often need to update those graphics quickly in response to external input. Upon external input, such programs would ideally incrementally update only the parts of the output graphics that actually changed. However, in practice, programmers must write and maintain complex, hand-written procedures to support incremental updates for graphics programs. We propose a technique to create graphics pipelines that are easy to write and maintain, but have incremental performance comparable to efficient hand-written implementations. We target graphics pipelines written in Halide, a domain-specific language and compiler for building image and array processing code. From a Halide pipeline, we synthesize an incremental pipeline that efficiently updates the result of the original computation. We plan to evaluate the performance of our synthesized pipeline against both hand-written incremental pipelines and existing techniques for incremental computation on an image compositor benchmark.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: incremental computation, self-adjusting computation, Halide

1 INTRODUCTION

Many programs render graphics that need to be updated in response to external input—for example, window managers, web browsers, image editors, and graphical user interface frameworks. However, it can be challenging in practice to write efficient programs that support incremental updates.

For a concrete example, consider how a stacking window manager might render a desktop. Given an input buffer for each window, the stacking window manager composites these buffers in accordance with their stacking order. Suppose the window manager has already rendered the desktop. When the buffer for some window changes, how might the window manager update the rendered desktop in response? One strategy might be to re-render the desktop from scratch. This is simple to implement, but may result in unnecessary work. For example, suppose the window that changed was completely occluded by another window; then no re-render was necessary.

Another approach might be to hand-write incremental versions of the original pipeline. But this adds maintenance burden, as the incremental pipeline has is generally complex and also must be kept in sync with the original pipeline. Furthermore, stacking window managers often support additional visual layers such as transparent windows and blur effects, which add additional complication to the update logic. The incremental update logic for Mutter, the window management and compositor library for GNOME, is (roughly estimating) about 1000 lines long [6].

Ideally, an efficient update pipeline could be synthesized from the original rendering pipeline. Such a pipeline would be easy to write, maintain, as well as support efficient updates. There are existing techniques that can automatically incrementalize general purpose programs. However, applying these techniques to graphics-like pipelines creates programs that are up to an order of magnitude slower than the original non-incremental program [2].

Instead of targeting general purpose programs, we instead target pipelines written in Halide. Halide is a domain specific language embedded in C++ that makes it easy to write efficient graphics processing pipelines [4]. Halide pipelines are computation graphs that consist of stencil and reduction stages over buffers of data. Given a computation graph and a processing schedule, Halide generates an efficient parallel and vectorized program that executes the graph to compute the desired output. Because pipeline stages written in Halide explicitly declare their dependencies,

[*]ACM#: 1252366

Halide programs are easier to analyse and optimize than general purpose programs. Furthermore, Halide already supports a variety of analyses we need to generate efficient incremental pipelines.

We plan to develop a tool that accepts a Halide pipeline H and synthesizes two Halide pipelines H^+ and H'. H^+ computes the same output as H with low additional overhead, while storing data for H'. H' incrementally updates the output of H^+ in much less time than it would take to recompute H. Our end goal is to enable programs that use Halide for graphics processing to easily add support for efficient incremental updates.

2 PRIOR WORK

Self-adjusting computation is a trace-based technique to incrementalize general purpose programs. Programs written in the self-adjusting style wrap values that might change in a "modifiable" type (`mod a`). These modifiables are not readable and writable directly; instead, reads and writes are proxied via `read` and `write` functions [1].

When the program initially executes, the self-adjusting runtime observes calls to `read` and `write` to construct an acyclic dependency graph on all accessed modifiables. When an input modifiable changes, the runtime marks that modifiable as tainted. To recompute the new output, the runtime recomputes all modifiables that are a descendant of some tainted modifiable in topological order.

There are two major limitations to the self-adjusting style for graphics programming. First, self-adjusting programs must materialize an explicit dependency graph of the computation [3]. However, in graphics processing pipelines, buffers may have tens of millions of elements. A self-adjusting program thus has to explicitly store and traverse a graph with tens of millions of vertices and (possibly) hundreds of millions of edges. This results in significant overhead; in prior work, wrapping each integer in a modifiable resulted in a 8.5x slowdown for a 400x400 matrix multiplication [2]. Our approach is more efficient because we use Halide stencils to (conservatively) approximate the dependency graph instead of materializing the graph explicitly.

Second, the self-adjusting computation runtime can only observe when `reads` and `writes` occur; the runtime cannot observe how values are actually combined. That is, the actual computation done on modifiable values is opaque to the self-adjusting runtime. This results in missed optimization opportunities. For example, self-adjusting computations cannot reason about associativity, commutativity, and invertibility to optimize a histogram reduction (see Section 3.2)[5].

3 OUR APPROACH

Our approach relies on static analyses of the program to generate more efficient incremental programs. For example, while self-adjusting computations rely on tracing to materialize a dependency graph, we instead statically approximate the dependency graph. We use these static analyses to generate parameters for the pipeline that specify which regions need to be recomputed. The actual execution of the pipeline does not create or traverse a dependency graph and is thus highly efficient.

From a Halide pipeline H, we synthesize two Halide pipelines H^+ and H'. H^+ computes the original output of H, while additionally storing some intermediate buffers for H'. H^+ stores these intermediate buffers because incremental computations may further depend on previous values that are not part of the regions directly invalidated by the change in input. See Figure 2 in the Appendix for an example pipeline. When H' depends on such values, it may be more efficient to have H^+ materialize intermediate buffers than have H' recompute those values. Therefore, we allow intermediate stages to be marked as materialized. If a stage is marked as materialized, H^+ stores its result buffer. Then, during an incremental update, H' may the values in the stored buffer during an incremental update. H' is also responsible for maintaining the intermediate buffers.

H' accepts as parameters coordinates that represent rectangular regions that must be recomputed for each layer. When given the appropriate coordinates, H' incrementally updates the output of H^+, producing the same result as what H would have produced while also maintaining the contents of materialized layers. To create H' from H, we run an static analysis over H: for each intermediate layer in the pipeline, we pick an execution strategy to recompute the changed region in that layer, taking into account whether that layer is computed using associative, commutative, and invertible operations (Section 3.2), and which prior layers have been materialized. We construct the Halide pipeline H' from these layers, and then use Halide to compile and optimize the pipeline.

On update, after receiving the changed input region(s), we first find the coordinates for the to-be-updated output region by "pushing" the changed input region(s)'s coordinates forward through the pipeline using the reverse stencil dependency graph (Section 3.1). We then "pull" the to-be-updated output region back through the graph to identify which regions need to be recomputed, stopping at materialized buffers. Finally, we pass these coordinates into H' to compute the incremental update.

3.1 Reverse stencil dependency graph

Halide stencils are written from the perspective of the output. From a Halide stencil, we can infer a dependency graph. For example, from the Halide stencil `Out(x,y) = In(x-1,y) + In(x,y) + In(x+1,y)` we can infer the dependency graph with edges `Out(x,y) <- In(x-1,y)`, `Out(x,y) <- In(x,y)`, and `Out(x,y) <- In(x+1,y)` for all `Out(x,y)`. See Figure 1 in the Appendix for a visual example.

We can interpret such a representation as `Out` "pulling" data from `In`. On the other hand, when an input changes, we need to "push" the new value from `In` to `Out`. That is, to find which outputs depend on an input efficiently, we need to reverse the dependency graph. From the stencil, we can synthesize a *reverse stencil dependency graph*. When the arguments to `In` are an affine transformation, we can do this easily using computer algebra. For example, using a CAS, we can rewrite `Out(x,y) <- In(x-1,y)` as `Out(x+1,y) <- In(x,y)`, which we interpret as: "when `In(x,y)` is modified, we need to update `Out(x+1,y)`."

3.2 Associative, commutative and invertible operations

We would also like to support more efficient incremental updates for operations that are associative, commutative, and invertible. For example, consider a Halide program that computes a histogram over an integer buffer: for each value in the input, we increment the counter for that value. The output of the program is the count of input elements that have value 0, value 1, value 2, and so on.

Suppose exactly one input element x updates from 0 to 1. Under the self-adjusting framework, such an update would invalidate all histogram counters (because every counter's `write` is downstream of every value `read`) and the incremental computation would have to iterate over the entire input buffer to recompute all histogram entries. However, we can clearly do better: since addition associates, commutes, and has an inverse, we can remove x's contribution from the 0 counter ($c_0 := c_0 - 1$) and add its new contribution to the 1 counter ($c_1 := c_1 + 1$). Halide already supports analyzing whether a reduction is commutative and/or associative via the rfactor scheduling primitive [5]; we aim to extend that analysis to include invertibility.

4 PLANNED EVALUATION

To evaluate the performance of our incremental Halide pipeline, we plan to write three implementations of an incremental compositor: 1) a hand-written C++ implementation, vectorized using the Highway SIMD library [7]; 2) an implementation using Halide and our incrementalization approach; and 3) an implementation using `CEAL`, a self-adjusting runtime for C [3]. We aim to show that the Halide implementation is much faster at initially rendering a scene and incremental updating the scene than both the `CEAL` and the hand-written vectorized C++ implementations, as well as being simpler to write.

5 APPENDIX

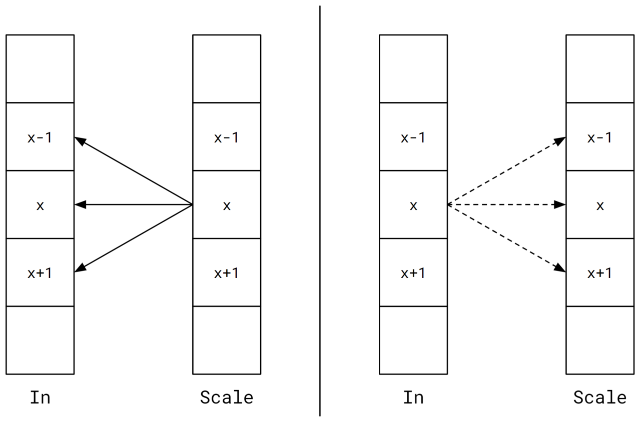

Fig. 1. **Left:** the stencil dependency graph, showing how an element in Scale depends on the elements in In, with solid edges denoting a dependency. **Right:** the reverse stencil dependency graph, showing which elements in Scale need to be updated when an element in In changes. Dotted-edges show the reverse dependency.

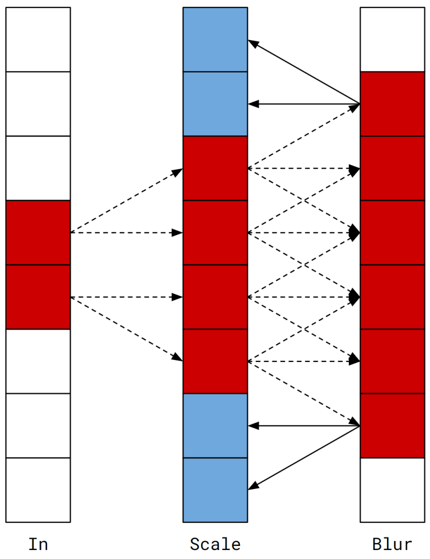

Fig. 2. The pipeline In -> Scaled -> Blur. Solid edges are part of the stencil dependency graph, while dotted edges are part of the reverse stencil dependency graph. A region of two elements in In are updated (shown in red). The red elements in Scale and Blur need to be recomputed. To recompute Blur, the values of the blue elements in Scale are also needed (if they were not materialized by the original computation, they need to be recomputed by H').

REFERENCES

[1] Umut A. Acar. 2009. Self-Adjusting Computation: (An Overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Savannah, GA, USA) *(PEPM '09)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1480945.1480946

[2] Yan Chen, Jana Dunfield, and Umut A. Acar. 2012. Type-Directed Automatic Incrementalization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2254064.2254100

[3] Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: A C-Based Language for Self-Adjusting Computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/1542476.1542480

[4] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[5] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 281–291.

[6] The Mutter Development Team. 2023. *Mutter - A Wayland display server and X11 window manager and compositor library.* https://gitlab.gnome.org/GNOME/mutter

[7] Jan Wassenberg. 2023. *Highway - A C++ library that provides portable SIMD/vector intrinsics.* https://github.com/google/highway

Received 10 November 2023