Parallel Associative Reductions In Halide

Patricia Suriana
Google, USA

Andrew Adams
Google, USA

Shoaib Kamil
Adobe, USA

Presenter : Abhinav Gupta

Introduction

- Halide is a domain specific language for fast image processing
- Halide support parallelizing and vectorizing naturally data parallel operations (like resizing an image)
- Halide separates pipelines into stages
  - Algorithm : Specifies “what” values are to be computed
  - Schedule : Specifies “how” are these values to be computed
- Changes in schedule do not change the result

Problem definition/Problem relevance

- Halide does not support the same scheduling for reductions
    - Ex: Histogram calculation
- Programmer must manipulate algorithm by manually factoring reduction into multiple stages to expose data parallelism.
    - Ex: Histogram calculation of an entire image can be broken down to computing histograms of each row and summing the partial results. (Changes in the algo!)
- Manipulation of Algorithm is bad, bug prone, hampers readability and portability, makes it impossible for automatic scheduling methods to parallelize reductions.

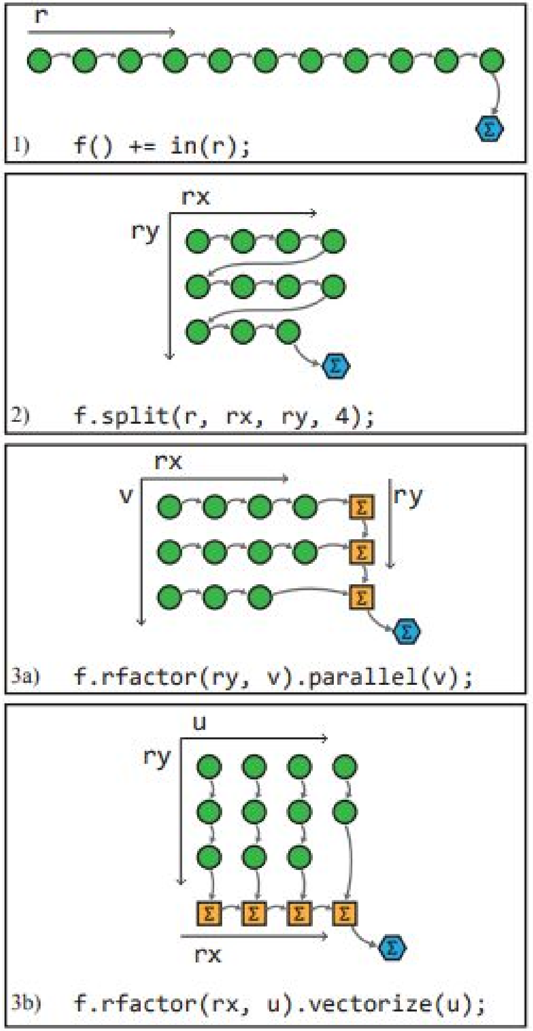
Key contributions

- Halide Scheduling primitive : Rfactor
  - Moves factoring of reduction into schedule
  - Provides a much less error-prone way of factoring a reduction as more of logic is in the schedule instead of the algorithm
- Method for automatically discovering an equivalent binary reduction operator and its identity

Approach/solution

- During the compilation process, Rfactor splits the original serial reduction into a pair of stages
  - Intermediate : Computes partial results over slices of reduction domain
  - Merge: Combines those partial results
- Implement a new stage in the Halide compiler that matches arbitrary reductions with a set of fragments that are pre-generated using the synthesis method proposed

This technique permits parallelization and vectorization of Halide Algorithm using Halides other scheduling primitives which previously required manipulating both the algorithm and schedule.



r
1) f() += in(r);
rx
ry
2) f.split(r, rx, ry, 4);
rx
v
ry
3a) f.rfactor(ry, v).parallel(v);
u
ry
rx
3b) f.rfactor(rx, u).vectorize(u);

Figure 1. `split` followed by `rfactor` used to vectorize or parallelize a one-dimensional summation. 1) A serial summation of 11 elements over a one-dimensional domain `r`. 2) The `split` scheduling directive reshapes the domain into a two-dimensional reduction. The reduction is still serial. 3a) The `rfactor` directive creates and returns a new (orange square) *intermediate* stage which computes partial results along each row. The intermediate stage is now data parallel over its outer loop, which is indexed with the new pure variable `v`. The merge stage (blue hexagon) retains only the serial loop over the reduction variable `ry`. We have *reassociated* the summation. 3b) Alternatively, one can use `rfactor` to make the *inner* loop data parallel. This can be used to vectorize reductions. The intermediate stage computes sums of whole vectors, data parallel across the vector lanes, and then the merge stage sums the lanes of the vector result. The strategies in 3a and 3b can be combined to both vectorize *and* parallelize a summation. Note that 3b requires the reduction to be commutative as it changes the order of computation.

Evaluation

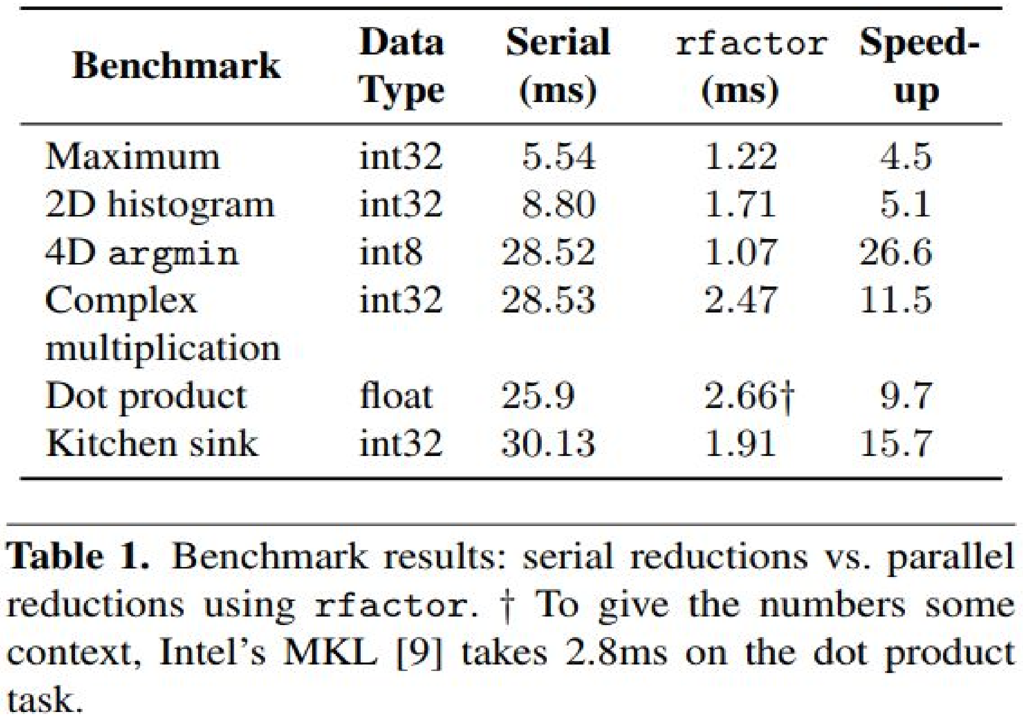
- Speedup observed was equivalent to manually factoring Halide reductions
- Benchmarked the feature using a suite of reductions of varying complexity
- Programs which are easily vectorizable or parallelizable such as matrix multiplication and convolution were not used for benchmarking

Evaluation

- Benchmarks used :
  - Maximum: The maximum integer in a list.
  - Dot product: The dot product of two vectors.
  - 4D argmin: The coordinates of the minimum value in a four-dimensional volume.
  - 2D histogram: A histogram of values present in an 8-bit image.
  - Complex multiplication: The product of a list of complex numbers.
  - Kitchen sink: An 8-tuple-element reduction that simultaneously computes the result of several operations in a list of integers.
- All benchmarks were run on inputs of size 2^{24}, which is typical number of pixels for an input image to a Halide program.
- Benchmarks were run on 8 cores of a Xeon E5-2690 processor

Evaluation

- In each case the same Halide algorithm was used
- Have taken the minimum of 10 trials, each trial is the average of 10 iterations
- Execution time of the pipeline, not including the compilation time were taken

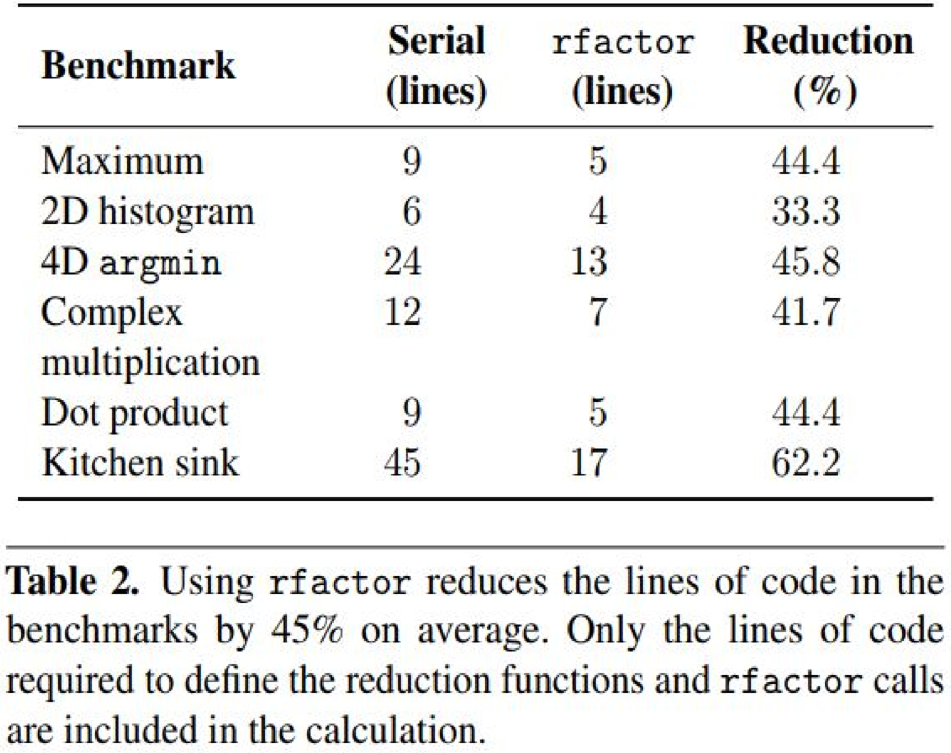


Benchmark	Data Type	Serial (ms)	`rfactor` (ms)	Speed-up
Maximum	int32	5.54	1.22	4.5
2D histogram	int32	8.80	1.71	5.1
4D `argmin`	int8	28.52	1.07	26.6
Complex multiplication	int32	28.53	2.47	11.5
Dot product	float	25.9	2.66†	9.7
Kitchen sink	int32	30.13	1.91	15.7

Table 1. Benchmark results: serial reductions vs. parallel reductions using `rfactor`. † To give the numbers some context, Intel's MKL [9] takes 2.8ms on the dot product task.

Evaluation

- Without Rfactor, algorithm required twice the times of code to reach the same performance

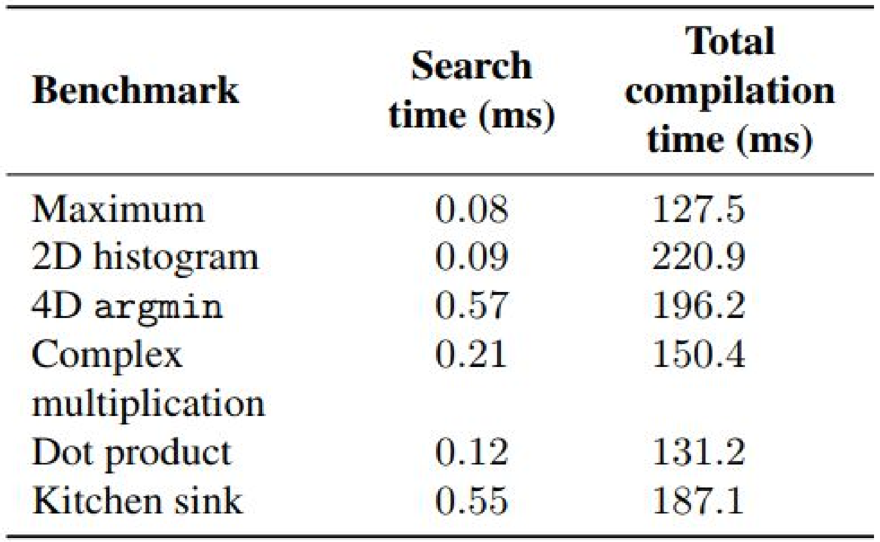


Benchmark	Serial (lines)	`rfactor` (lines)	Reduction (%)
Maximum	9	5	44.4
2D histogram	6	4	33.3
4D `argmin`	24	13	45.8
Complex multiplication	12	7	41.7
Dot product	9	5	44.4
Kitchen sink	45	17	62.2

Table 2. Using `rfactor` reduces the lines of code in the benchmarks by 45% on average. Only the lines of code required to define the reduction functions and `rfactor` calls are included in the calculation.

Evaluation

- Increase in the compile time due to the call to Rfactor was consistently under 3 milliseconds



Benchmark	**Search time (ms)**	**Total compilation time (ms)**
Maximum	0.08	127.5
2D histogram	0.09	220.9
4D `argmin`	0.57	196.2
Complex multiplication	0.21	150.4
Dot product	0.12	131.2
Kitchen sink	0.55	187.1

Table 3. The time taken to search the table to find a matching operator is relatively small with respect to the total compilation time.

Evaluation

- For non-associative operation Rfactor must search to the end of the table. This took around 1.2 ms

Benchmark fall in 2 categories from which we benefit

- Vectorization and Multi-core parallelism
    - Dot product, 4D argmin, complex multiplication and kitchen sink test
- Multi-core parallelism alone
    - Histogram, maximum (Halide LLVM vectorizes the reference code)

Impact

- By moving this factoring into schedule alone, Rfactor makes it possible for automatic schedule generation tools to parallelize and vectorize reductions

Comment on Pros/Cons of the paper.

Cons

- The precomputed table only recognize reductions decomposable into elementary reductions of a fixed maximum size
- Can only recognize operations that Z3 can prove to be associative, a fairly large set
  - Summing a sequence of 128 bit integers expressed as two 64 bit tuples where addition is element wise plus some logic to handle the carry bit
- Only recognize reductions where the first stage in the factorization -the intermediate- has the same form as the original update information
  - Repeated subtraction of a list of values from some initial value
  - This can be manually factored into an intermediate that sums slices of the list and then a merge stage that does repeated subtraction

Comment on Pros/Cons of the paper.

Pros

- Rfactor provide a much less error-prone way of factoring a reduction as more of the logic is in the schedule instead of the algorithm
- Rfactor lifts the burden of factoring a reduction from the programmer
- This improves readability, which is entirely responsible for “what” values are computed, is shorter
- Also improves portability, as reductions can be factored in different ways on different platforms, without risking producing different results on different platforms

Comments on how current work differs from previous works

- Prior work explored automatic generation of parallel associative reductions from a serial reduction

➢ But most work requires that an explicit associative binary reduction operator be given
  - OpenMP has first class parallel reduction construct that takes one of some number of primitive reduction operators.
  - Cilk additionally supports user specified reduction operators.This approach is not applicable to Halides though

★ Current work describes a method for automatically discovering an equivalent binary reduction operator and its identity

Comments on how current work differs from previous works

- LLVM can automatically recognize a small set of associative reductions for the purpose of auto vectorization
  - ➢ Faces usual problems associated with auto-vectorization, small changes to the program can cause auto-vectorization to fail for unknown reasons, only simple loops auto vectorize
    - ★ Combined with other Halide scheduling primitives , such as split, Rfactor allows Halide to represent a broad class of schedules for parallel and vectorized reductions

Comments on how current work differs from previous works

- Morita et al. introduced automatic generation of divide-and-conquer parallel programs using a framework based on the third homomorphism theorem and derivation of weak-right inverse

➢ It requires that programmer specify the leftward and rightward forms of the sequential function

Comments on how current work differs from previous works

- Teo et al. proposed a method to synthesize parallel divide-and-conquer program from a recurrence function (which is similar in form to a Halide serial reduction) through induction

➢ The technique is slow

➢ Unable to deal with reductions like argmin

Comments on how current work differs from previous works

Recent work has applied program synthesis

- SKETCH and ROSETTE are two solver aided programming languages with support for program synthesis

➢ Too slow to apply directly at compile time

Comments on how current work differs from previous works

- More recent work has used stochastic search and program synthesis to find replacements for larger sequences of instructions, which use superoptimization
- ★ In the current work a combination of enumeration and synthesis is used, also larger replacements are searched than what most super optimizers do

Comments on how current work differs from previous works

- CHILL and URUK allow users to apply a series of high level transformations to FORTRAN and C code freeing users from hand rewriting the code to implement complicated optimizations

➢ These code transformations do not support reductions

★ The current work support reductions

Thank You