Exploring Halide

Mingsheng Xu

Explaining the DSL

Halide is a domain specific language for image processing. Halide proposes to separate the algorithms from detailed implementation, trying to facilitate the coding for image processing pipeline [1]. This separation, however, also enables Halide to autotune the execution [2].

A Halide pipeline consists of functions and a schedule. Each function represents a stage of the pipeline, describing how to derive the output from the input at the granularity of a single pixel. Then, the schedule dictates on how to realize the pipeline, using highly abstracted operations not only to manipulate each of but also to interleave all of the stages.

The paper describes loop constructions as a trade-off between locality, reuse, and parallelism. These characristics form the space where all kinds of implementations reside. By making use of the operations Halide provides, programmers now can probe the space with different schedules and find one with the best performance. One thing worth mentioning is that, Halide’s abstraction of loops is not as expressive as hand-written ones. It means the operations may not lead to every spot in the space, as the paper says, “they (the resulting min/max bounds of interval analysis) can only describe iteration over axis-aligned boxes, rather than arbitrary polytopes”. But it is hardly a defect. We are more tolerate for a domain specific language because “for many domains, including image processing, this is an acceptable simplification: most functions are applied over rectilinear regions.”

The autotuner of Halide is a bonus thanks to the separation and abstraction, which simplifies different implementations into different schedules that enumerate various operations with various parameters. Halide utilizes the genetic algorithm to generate schedules and locates the best among. With prior knowledge, Halide biases the search towards certain design patterns but also pertains the ability to escape from local minima. Halide achieves fresh things with no precedents, otherwise there should be a comparision between different tools in the paper. Other studies do inspire Halide, but they either are not designed for image processing or only optimize in a more rigid space.

Experience Report

I think Halide’s abstraction and separation resembles object-oriented programming (and indeed Halide functions are implemented as C++ objects). The properties of a stage include the domain, traversal order, dimensions order, etc. A stage also has methods that manipulate the properties, like split and reorder, and a method that defines the loop calculations. This encapsulation of typical data and behaviors provides a template to reuse for all stages in image processing. At this point, [how about schedule, the granualarity, is it also a method, therefore store and computation spot? wait for me to finish the tutorials… -> another obeject]. Programmers now simply instanciate loops by feeding in specifications and enumerate strategies by combining object methods with different parameters, therefore saving detailed implementations.

At the right level of abstraction?

What are some trade-offs that the language made in terms of specification vs. general computation?

Was it easy to install and run the tutorial programs?

What does it do well?

How do you thing it could improve?

One improvement of the language I could think of is to make the order of parameters consistent. When I declare a function foo(x, y), x represent column index while y represent row index.

Are programs easy to reason about?

Are common bugs avoided by the language constraints?

Experiments