Research Statement

Yuting Yang

Programs are ubiquitous in creative design processes even for non-programmers: artists may utilize graphical programming tools to design programs that procedurally render textures, and musicians can tune and connect different modules in their synth, which ultimately composes a program that synthesizes electronic sound. While the content being generated are fascinating, the programs themselves and the vast potential hidden under the computation structures are usually ignored. For example, the program parameters can be automatically optimized to generate content closer to a specific style or target by approximating the distributional derivative of a program; the generated content can also be perfected if the program is involved in a deep learning model. Unfortunately, these techniques are usually less accessible to amateur programmers because the required domain expertise in advanced math or machine learning doesn’t always align with the expertise of these content creators.

My research goal is to allow programs written by amateur programmers easily benefit from the advanced techniques in computer graphics and machine learning. I aim to achieve this by developing compiler frameworks that automatically transform the program as if a domain expert would manually design it. In the rest of this document, I will use my past projects to give examples of how this methodology is applied to shader programs (programs that generate images) as well as visual computing programs (programs that process visual data). This includes approximating the distributional derivative to arbitrary discontinuous programs to allow gradient-based optimization on program parameters; discovering which program traces are beneficial for deep learning tasks; approximating the convolution of arbitrary programs with a smoothing kernel to analytically antialias the generated imagery; and improving program runtime performance on easy-to-prototype programming languages.

Automatically Differentiate Discontinuous Programs.

Automatic Differentiation (AD) has profoundly impacted graphics, vision, and machine learning applications and can be easily applied to general programs via libraries like TensorFlow and PyTorch. These AD methods ignore gradients at discontinuities and instead treat them as continuous. However, in many domains, the program result intrinsically relies on discontinuities, such as in rendering they are crucial at object silhouettes, and in general for any branching operation.
In computer graphics, differentiating discontinuities is a fundamental part of inverse rendering, and analytic solutions have been manually designed for specific applications such as triangle meshes and vector graphics. However, generalizing the solutions to arbitrary programs is limited to a smaller scope because of the inconsistencies between the traditional calculus rules and the properties of the Dirac delta, a concept in distribution theory that mathematically express the derivative of discontinuities.

Our project Aδ [SIGGRAPH 2022] enables differentiating discontinuities on a much larger set of programs by designing a novel set of derivative rules that correctly accounts for the existence of the Dirac delta distribution. Our key insight is that we could sample along any axis to measure the scale of the discontinuities. This is in contrast from traditional derivative rules, where they simply assume the function is continuous around the point of differentiation. With our compiler framework that extends reverse mode AD with novel derivative rules, we can differentiate pixel shader programs that render images. These programs represent similar complexities as those found on shadertoy.com, an artist community for sharing their shaders. The generated gradient programs further enable novel applications that easily find program parameters to best match some target illustrations. For example, given a shader program that simply draws 3D boxes, our framework easily finds the box parameters to render an output image identical to the TensorFlow logo. Furthermore, the program representation allows programmers to creatively modify and animate the optimized programs, such as the camera in the TensorFlow logo scene can be easily moved to generate the animation that makes the letters T and F visible.

Aδ exemplifies my research: motivated by extending research advanced or complicated math (inverse rendering and distribution theory) to a larger set of programs written by amateur programmers, I design new set of rules that generalizes the manually designed solutions (Aδ novel derivative rules) and build compiler framework so that arbitrary programs could benefit from the new technique and explore new territories of applications (optimize and match target illustrations).

Learning from Program Traces.

Deep learning for image processing typically treats input imagery as pixels in some color space. For images with 3D scenes, researchers have also explored augmenting the RGB data with hand-picked features like depth or surface normal. These auxiliary features are manually identified by an expert. Moreover, the extent to which these auxiliary features help learning depends on the choice of features, the particular program, and the learning goal. However, we believe that other program-specific information useful to the learner remains hidden within the program execution and that a learning process could automatically identify and leverage that information.
To validate this theory, our Eurographics 2022 best paper awarded project explores a learning-based approach that utilizes all of the information produced during the execution of a program, which we refer to as *program trace*. Specifically, we investigate the learning task for pixel shader programs. Without extra manual tuning, our compiler framework automatically translates shader programs written in our domain specific languages (DSL) to TensorFlow, collects program trace from shader programs, and prune and concatenate them into a large tensor as input to a deep learning model. We empirically verified that models learned with extra program information indeed achieve better performance both quantitatively and qualitatively for every application we explored: removing sampling noise; extrapolating from a simplified program and synthesizing more complicated texture; adding postprocessing filters; extrapolating non-imagery simulation of flocks of Boids. To take a step further towards understanding why extra program information helps to learn, we also conduct a series of analyses that show certain features are important within the trace: these coincide with intuitively important aspects of the program.

**Automatic Program Smoothing.**

In many contexts, functions that have aliasing or noise could be viewed as undesirable. For example, in procedural shader programs, one common visual error is *aliasing*, which occurs when the sampling rate is below the Nyquist limit. Mathematically, the high-frequency component can be removed by convolving the program with a low-pass filter. While exact analytic band-limited formulas are known for some specialized functions such as the Gabor noise functions, in most cases, however, the shader developer must manually calculate the convolution integral. Unlike differentiation which can be easily automated through AD, automating the symbolic convolution to an arbitrary program is much more difficult because of the integration involved.

Our Eurographics 2018 paper proposes to approximate the convolution of arbitrary programs with a Gaussian smoothing kernel through a compiler approach combined with a genetic search. The key insight is to model each intermediate state in the program as a Gaussian distribution with specific mean and variance, and propagate these values from the program input (e.g. pixel coordinates) all the way to program output (e.g. pixel color). We designed a set of mean/variance propagation rules with various accuracy/performance trade-offs, and use genetic search to generate the Pareto frontier of smoothed program variants that all optimally trades-off accuracy and performance. We evaluate this framework for the application of bandlimiting procedural shader programs on a variety of geometries and complex shaders, including shaders with parallax mapping, animation, and spatially varying statistics. Our smoothing variants always outperform previous approaches both numerically and aesthetically.
Automatic Program Translation.

Visual datasets are rapidly growing in size, and this trend will continue to strengthen, due to forces such as increasing photograph resolutions and increased worldwide Internet adoption. As a result, many amateur programmers start to write programs for visual computing using graphics and vision. However, this introduces a tension between the run-time efficiency of the final programs and easy exploratory prototyping. At one extreme, highly efficient programs can be crafted by hiring an engineer who is a domain expert, or learning to code in some domain specific languages such as Halide with some nontrivial learning curve. At the other extreme, a developer may choose to work in a language that facilitates rapid prototyping, such as Python or MATLAB. These “dynamic” languages are beneficial for quick iterative design, but the resulting programs run slowly because such languages typically incur large overheads in run-time. This prevents amateur developers to make the best use of the advances in graphics and vision algorithms.

Our VizGen project [SIGGRAPH Asia 2016] helps alleviate the tension by taking steps towards the proposed goal: it should be possible to automatically translate visual computing prototypes in dynamic languages into highly-performant code. We propose a compiler framework based on several key properties of visual computing programs, such as many arrays have small, constant, or bounded sizes; many operations are supported in hardware or are embarrassingly parallel; small numerical error is acceptable in human perception. Our compiler directly translates python programs into highly efficient C. It adopts various program transformations that may benefit program runtime, and automatically decides where in the program to apply these transformations. This allows users to still enjoy the easy prototyping in Python, while gaining performance comparable to or even faster than moderately optimized C equivalence.