Testing Inference Programs for Generative Scene Graphs

by

Austin J. Garrett

Department of Electrical Engineering and Computer Science
Proposal for Thesis Research in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2019

Author . . . .
Department of Electrical Engineering and Computer Science
December 11, 2019

Certified by . . . .
Vikash Mansinghka
Research Scientist
Thesis Supervisor

Accepted by . . . .
Placeholder
Chairman, Department Committee on Graduate Theses

Testing Inference Programs for Generative Scene Graphs

by

Austin J. Garrett

Submitted to the Department of Electrical Engineering and Computer Science
on December 11, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a compiler which performs optimizations that reduce the number of low-level floating point operations necessary for a specific task; this involves the optimization of chains of floating point operations as well as the implementation of a "fixed" point data type that allows some floating point operations to simulated with integer arithmetic. The source language of the compiler is a subset of C, and the destination language is assembly language for a micro-floating point CPU. An instruction-level simulator of the CPU was written to allow testing of the code. A series of test pieces of codes was compiled, both with and without optimization, to determine how effective these optimizations were.

Thesis Supervisor: Vikash Mansinghka
Title: Research Scientist

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Micro-optimization is a technique to reduce the overall operation count of floating point operations. In a standard floating point unit, floating point operations are fairly high level, such as “multiply” and “add”; in a micro floating point unit (μFPU), these have been broken down into their constituent low-level floating point operations on the mantissas and exponents of the floating point numbers.

Chapter two describes the architecture of the μFPU unit, and the motivations for the design decisions made.

Chapter three describes the design of the compiler, as well as how the optimizations discussed in section 1.2 were implemented.

Chapter four describes the purpose of test code that was compiled, and which statistics were gathered by running it through the simulator. The purpose is to measure what effect the micro-optimizations had, compared to unoptimized code. Possible future expansions to the project are also discussed.

1.1 Motivations for micro-optimization

The idea of micro-optimization is motivated by the recent trends in computer architecture towards low-level parallelism and small, pipelineable instruction sets [**?**, **?**]. By getting rid of more complex instructions and concentrating on optimizing frequently used instructions, substantial increases in performance were realized.

Another important motivation was the trend towards placing more of the burden of performance on the compiler. Many of the new architectures depend on an intelligent, optimizing compiler in order to realize anywhere near their peak performance [?, ?, ?]. In these cases, the compiler not only is responsible for faithfully generating native code to match the source language, but also must be aware of instruction latencies, delayed branches, pipeline stages, and a multitude of other factors in order to generate fast code [?].

Taking these ideas one step further, it seems that the floating point operations that are normally single, large instructions can be further broken down into smaller, simpler, faster instructions, with more control in the compiler and less in the hardware. This is the idea behind a micro-optimizing FPU; break the floating point instructions down into their basic components and use a small, fast implementation, with a large part of the burden of hardware allocation and optimization shifted towards compile-time.

Along with the hardware speedups possible by using a μFPU, there are also optimizations that the compiler can perform on the code that is generated. In a normal sequence of floating point operations, there are many hidden redundancies that can be eliminated by allowing the compiler to control the floating point operations down to their lowest level. These optimizations are described in detail in section 1.2.

1.2 Description of micro-optimization

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra “guard bits” on the standard floating point formats, to allow several unnormalized operations to be performed in a row without the loss information[1]. A discussion of the mathematics behind unnormalized arithmetic is in appendix **??**.

The optimizations that the compiler can perform fall into several categories:

1.2.1 Post Multiply Normalization

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit. If there are guard bits on the mantissas to prevent bits from "falling off" the end during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory[3].

1.2.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1,e_1) and (m_2,e_2).

1. Compare e_1 and e_2.
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2.
4. Find the first one in the resulting mantissa.

[1] A description of the floating point format used is shown in figures **??** and **??**.

[2] Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**?**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

[3] Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

5. Shift the resulting mantissa so that normalized

6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been shifted, the additions can take place one after another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

1.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. In concert with the floating point optimizations, these can provide a significant speedup.

1.3.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section **??**, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is

[4] This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μFPU, there are 3 guard bits, making up to 8 consecutive additions possible.

desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

1.3.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some combination of

$$\begin{aligned}
a_i &= a_j + a_k \\
a_i &= 2a_j + a_k \\
a_i &= 4a_j + a_k \\
a_i &= 8a_j + a_k \\
a_i &= a_j - a_k \\
a_i &= a_j \ll m\text{shift}
\end{aligned}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r, you could use:

$$\begin{aligned}
r &= 4s + s \\
r &= r + r
\end{aligned}$$

Or by 59:

$$\begin{aligned} t &= 2s + s \\ r &= 2t + s \\ r &= 8r + t \end{aligned}$$

Similar combinations can be found for almost all of the smaller integers[5]. [?]

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is "trace scheduling". This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [?, ?] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential "trace" of program execution. In this way, instructions that *might* be executed depending on a conditional branch

[5] This optimization is only an "optimization", of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

[6] This can been seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [?]

further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn't taken, correction code is inserted after the branches to undo the effects of any prematurely executed instructions.

1.4.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as seendescribed by Gibbons. [**?**]

Chapter 2

Scene Graphs

2.0.1 Mathematical Description

We model the geometric state of a scene at a single time point as a *scene graph*, which is a tuple (G, Θ, Z), where $G = (V, E)$ is a directed tree that encodes the scene graph *structure* and Θ encodes the scene graph *continuous parameters* and Z encodes the scene graph *discrete parameters*. Although the framework we present can represent articulated objects, in this paper we only consider scenes involving a set of rigid objects O. The scene graph structure includes a vertex $v_o \in V$ for each object $o \in O$ that represents the 6DoF pose of object o, as well as a vertex $r \in V$ that represents the the coordinate frame of the observer. A directed edge $e = (v_i, v_j) \in E$ between objects $i, j \in O$ encodes that the pose of object j is parametrized *relative to* the pose of object i, by parameters θ_e and z_e. The pose of object j relative to object i is denoted $\Delta x(z_e, \theta_e) \in SE(3)$. A directed edge $e = (r, v_j) \in E$ from the root to an object $j \in O$ encodes that the pose of object j is not parametrized relative to that of any object, but is instead a full 6DoF pose $\theta_e \in SE(3)$ where $SE(3)$ is the special Euclidean group consisting of all 6DoF poses. The continuous and discrete parameters of the scene graph consist of the parameters for each edge in the structure ($\Theta := \{\theta_e\}_{e \in E}$ and $Z := \{z_e\}_{e \in E}$). The set of edges E must *span* the set of vertices $V := \{r\} \cup \{v_o\}_{o \in O}$; that is the scene graph structure G is a *directed spanning tree.*

2.0.2 Computing the 6DoF poses of all objects in the scene

Given a scene graph (G, Θ, Z) the pose $x_i \in SE(3)$ of an object i relative to the observer coordinate frame can be computed by walking path in the tree G from the root vertex $r \in V$ to the vertex $v_i \in V$, and successively computing the pose of each object along the path from the pose of its parent. That is, given pose $x_u \in SE(3)$ and edge $(u, v) \in E$, the pose $x_v \in SE(3)$ is computed as $x_v := x_v \cdot \Delta x(z_e, \theta_e)$ where $\Delta x(z_e, \theta_e) \in SE(3)$ is the relative pose between vertex u and vertex v. For an edge from the root vertex to an object vertex ($e = (r, v_i)$), $x_u := \mathbf{1}$ (the identity element of $SE(3)$) and $\Delta x(z_e, \theta_e) = \theta_e \in SE(3)$ (note that there are no discrete parameters in this case, so $z_e := ()$). The functional form of $\Delta x(z_e, \theta_e)$ for an edge e between two objects is discussed below. One requirement is that $\Delta x(z_e, \theta_e)$ is a differentiable function of θ_e; this enables inference algorithm that exploit gradient information.

2.0.3 Modeling face-to-face contact between two objects

An edge $(v_i, v_j) \in E$ from object i to object j indicates that the pose of object j is represented relative to the pose of object i. Various types of relative pose parametrizations for two objects are possible; for simplicity we model objects as polyhedra, and only model face-to-face contact between objects. That is, an edge $e = (v_i, v_j)$ indicates that a face of object i is in flush contact with a face of object j. Since each object has multiple faces, the choice of which pair of faces is in contact is encoded in the discrete parameters z_e for the edge. Concretely, let F_i and F_j denote the faces of object i and object j respectively Then, $z_e \in F_i \times F_j$. The continuous parameters θ_e for an edge e between two objects is an element of $\mathbb{R}^2 \times [0, 2\pi)$ that contains two translational degrees of freedom ($s, t \in \mathbb{R}$, for the relative offset of the two faces) and one rotational degree of freedom ($\phi \in [0, 2\pi)$). For example, a cuboid object $i \in O$, the set of faces is $F_i = \{\text{Top}, \text{Bottom}, \text{Left}, \text{Right}, \text{Front}, \text{Back}\}$. For an edge $e = (i, j)$ where objects $i, j \in O$ where both objects i and j are cuboids, there are 36 possible values for z_e.

Slack variables for face-to-face contact We extend the parametrization of face-to-face contact between objects with three additional degrees of freedom of *slack variables*: (i) one degree of freedom that encodes the perpendicular distance ($d \in [0, \infty)$) between the two contact faces, and (ii) two degrees of freedom for relative orientation of the two faces, encoded the surface normal unit vector $\mathbf{n}$ of the child object's face, relative to the parent object's face, which takes values on the sphere S^2. Therefore, $\theta_e \in \mathbb{R}^2 \times [0, 2\pi) \times [0, \infty) \times S^2$ for an edge $e = (v_i, v_j)$ between two objects i and j. Note that although this edge parametrization uses six degrees of freedom for object-to-object edges like the edge parametrization for edges from the root ($e = (r, v_j)$), the prior distribution on these parameters (described below) encourages object j to be *almost* in face-to-face contact with object i; whereas the prior distribution on the pose of object j for an edge of the form (r, v_j) is very different, and is typically a uniform distribution over positions within the scene bounding volume, and a uniform distribution on orientations.

2.0.4 A Prior Distribution on Scene Graphs

Various prior distributions on scene graphs are possible. In our experiments, we use a generic prior distribution on scene graphs over a collection of objects O that factors into two components: (i) a prior distribution on scene graph structures G, denoted $p(G)$, and (ii) a prior distribution on scene graph parameters (Z, Θ) given structure, denoted $p(Z, \Theta | G)$. While uncertainty about the number of objects is possible to represent in our framework and implementation, for simplicity assume that the set of objects is known a-priori, and that there is one vertex for each object, and one vertex representing the observer coordinate frame, so V is fixed a-priori to $\{r\} \cup \{v_o\}_{o \in O}$. Therefore, $p(G)$ reduces to a prior distribution on edges in the scene graph. For the prior distribution on structure, we use a uniform distribution on directed trees that are rooted at vertex r and span all $|O| + 1$ vertices in the graph. This set of directed trees is isomorphic to the set of undirected spanning trees over $|O| + 1$ vertices. Therefore, the prior probability of a graph $G = (V, E)$ is obtained by using Cayley's formula to count the number of undirected spanning trees on $|O| + 1$

vertices:

$$p(G) := p_{\text{unif}(|O|)}(G) := \begin{cases} (|O|+1)^{1-|O|} & \text{if } G \text{ is a directed spanning tree over vertices } V \text{ rooted at } r \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The prior distribution on scene graph parameters factors over the edges in the scene graph:

$$p(Z, \Theta | G) := \prod_{e \in E} p_e(z_e, \theta_e) \tag{2.2}$$

where $p_e(z_e, \theta_e)$ is a probability distribution on z_e and θ_e that depends on the two vertices in the edge $e = (u, v)$. For edges $e = (r, v_j)$ where j is an object, $z_e = ()$ and $p_e(z_e, \theta_e)$ is the uniform distribution on elements of $B \times SO(3)$ where B is the bounding volume of the scene and $SO(3)$ is 3D rotation group (the uniform distribution on $SO(3)$ is given by the Haar measure). For edges $e = (v_i, v_j)$ where i and j are objects, recall that $z_e \in F_i \times F_j$ and (using the edge parametrization including slack variables) $\theta_e = (s, t, \phi, d, \alpha_1, \alpha_2) \in \mathbb{R}^2 \times [0, 2\pi) \times [0, \infty) \times [0, 2\pi)^2$. The prior distribution on edge parameters is:

$$p_e(z_e, \theta_e) := \frac{1}{|F_i||F_j|} \cdot p_{\text{norm}(0,\sigma)}(s) \cdot p_{\text{norm}(0,\sigma)}(t) \cdot \frac{1}{2\pi} \cdot p_{\exp(\beta)}(d) \cdot \frac{1}{2\pi} \cdot \frac{1}{2\pi} \tag{2.3}$$

(where p_{norm} and p_{exp} are the normal and exponential distribution density functions, respectively).

2.0.5 Probabilistic Dynamics on Scene Graphs

We build temporal models of scene geometry based on Markov chains of scene graphs $\{(G_t, \Theta_t, Z_t)\}_{t=1}^{T}$, where t indexes time, with transitions $p(G_t, \Theta_t, Z_t | G_{t-1}, \Theta_{t-1}, Z_{t-1})$. We factor the dynamics model on scene graphs decomposes into a dynamics model on scene graph structure, and a dynamics on parameters:

$$p(G_t, \Theta_t, Z_t | G_{t-1}, \Theta_{t-1}, Z_{t-1}) := p(G_t | G_{t-1}) \cdot p(Z_t, \Theta_t, | Z_{t-1}, \Theta_{t-1}, G_{t-1}, G_t) \tag{2.4}$$

The dynamics on graph structure is based on a mixture of a random walk on structures (to capture incremental changes to the structure of a scene that often occur) and a uniform distribution on structures (to model sudden unexpected changes in structure):

$$p(G_t|G_{t-1}) := 0.9 \cdot p_{\text{walk}}(G_t|G_{t-1}) + 0.1 \cdot p_{\text{unif}(|O|)}(G_t) \tag{2.5}$$

where $p_{\text{walk}}(G'|G)$ is the distribution on graph structures induced by a sampling process in which one vertex in the undirected spanning tree is selected at random, severed from the tree, and grafted back onto the graph at a uniformly chosen vertex, subject to the condition that the resulting graph is a spanning tree. Recall that since the tree contains the root node r, this process can change the edge type (u, v_j) for an object i from (r, v_j) (representing independent 6DoF pose) to (v_i, v_j) (representing flush contact pose with another object v_i. The dynamics on parameters factor according to edges in the graph:

$$p(Z_t, \Theta_t|Z_{t-1}, \Theta_{t-1}, G_{t-1}, G_t) := \prod_{e \in E} p_e(z_e, \theta_e|G_t, G_{t-1}, z_{t-1_e}, \theta_{t-1_e}) \tag{2.6}$$

If an edge e is present in both G_t and G_{t-1}, then this distribution is a mixture of a random walk and a uniform distribution. If an edge e in G_t was not present in G_{t-1}, then this distribution is a uniform distribution.

Algorithm 1 Probabilistic pseudocode for generative model

```
@gen function step_forward(
    t::Int,
    prev_scene_graph::Union{MetaDiGraph,Nothing},
    hypers::Hypers,
)
    # structure and parameter dynamics
    if t == 1
        structure ~ UniformDiForest(N)
        params ~ params_init(structure)
    else
        prev_structure = get_structure(prev_scene_graph)
        structure ~ Mixture(...)
        params ~ params_dynamics(structure, prev_scene_graph, hypers)

    end

    # observation
    scene_graph = SceneGraph(structure, params)
    obs ~ noise_model(scene_graph)

    return scene_graph
end
```

2.1 Example Applications of Scene Graph Models

TODO: replace me

2.1.1 YCB Objects on a synthetic tabletop

TODO: replace me

2.1.2 Real YCB objects on a physical tabletop

TODO: replace me

2.1.3 Simulated objects in AI2Thor

TODO: replace me

Chapter 3

Visualizing Inference

TODO: replace me

Chapter 4

Tests of Inference by Enumeration

TODO: replace me

Chapter 5

Real-world Tests

TODO: replace me

Chapter 6

Conclusion

TODO: replace me

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V. Gehler. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. *CoRR*, abs/1402.0859, 2014.

[2] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

[3] Abhijit Kundu, Yin Li, and James M Rehg. 3d-rcnn: Instance-level 3d object reconstruction via render-and-compare. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3559–3568, 2018.

[4] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. Deep object pose estimation for semantic robotic grasping of household objects. *CoRR*, abs/1809.10790, 2018.

[5] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes. *arXiv preprint arXiv:1711.00199*, 2017.

[6] Ilker Yildirim, Tejas D Kulkarni, Winrich A Freiwald, and Joshua B Tenenbaum. Efficient analysis-by-synthesis in vision: A computational framework, behavioral tests, and comparison with neural representations. In *Proceedings of the thirty-seventh annual conference of the cognitive science society*, 2015.

[7] Ben Zinberg, Marco Cusumano-Towner, and Vikash K. Mansingkha. Structured differentiable models of 3d scenes via generative scene graphs. *Workshop on Perception as Generative Reasoning, NeurIPS 2019, Vancouver, Canada.*, 2019.