An Optimizing Compiler for Low-Level Floating Point Operations

by

Lucien William Van Elsen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1990

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 1990

Certified by . . . .
William J. Dally
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Optimizing Compiler for Low-Level Floating Point Operations

by

Lucien William Van Elsen

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1990, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

The recent increase of power of computation combined to the decrease of power consumption of devices led to an explosion of smart devices for all the possible purposes, and it's called the ***Internet of Things***.
The IoT includes all devices that has an interface to the real world and are connected to the internet, such as: *Smart Fridges, Smart TVs, Thermostats, Alarms, Cameras and etc.* These devices were born to simplify normal's people life, let's take the smart fridge as example: it will alert you when the food you have in the fridge is going to expire, the thermostat will try to save money applying smart strategies for heat consumption. One of the most important application of IoT to people's life is *House Security*: exploiting the IoT concept to improve the house security with smart sensors like cameras, leap motion sensors and microphones. House Security has to be of concern, because house invasion is an always trending crime. However these systems are not always perfect and the most common problem are false positives, for the which there is an active branch of research focused on.
The raise of devices connected introduced another problem: heterogeneity between IoT devices. As the popularity of smart devices increased more and more companies started producing their own different ecosystems. Communication between different ecosystems is another research topic which will analyzed in this document.

Thesis Supervisor: William J. Dally
Title: Associate Professor

Acknowledgments

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

IoT surveillance systems has been proved to be effective during the time, catching the burglar committing the crime and helping the authorities to catch him[1]. Nonetheless these systems are not yet perfect, and during this paper we will investigate on some optimizations with respect to the *integration* with other ecosystems and the *reduction* of false positives. Here follows the main structure of the document:

Chapter two introduces the technologies used in our scenario, with a brief introduction on the techniques adopted to face the various problems.

Chapter three illustrates the main architecture of the optimizations proposed during the document and the motivation behind them.

Chapter four shows the implementation of our use case with the optimizations we have described formerly.

Chapter five is an extensive analysis of the results obtained during the whole analysis of the scenario.

1.1 Motivations

The idea behind our research in this field is to improve the existing solution with an innovative and different approach. Nowdays there are many standalone solutions for IoT surveillance, which acts separately from all the other components in the house. We address this isolation, trying to create a system capable of interconnecting with other existing components in the house. There are currently 4 billion of connected devices [2] as 2016, and the

forecasts says they will be 13.5b in 2020. This means a growth in the heterogenity of companies and products, which makes a must the interopability between devices from different producers.
Most of the common Smart devices are built to work appositely with their own application, without any external support which leads to a loose coupling between devices.

1.2 Description of scenario

During this document we will refer to a use case scenario related to house security. In this scenario we will have a small house with different floors, each floor having a leap motion sensor to detect any movement and a camera recording.

1.2.1 M2M: Machine to Machine

Most of common smart sensors, in order to be *smart* they need to provide a form of connectivity: let it be BLE (Bluetooth Low Energy), Wi-Fi or R-FID. *M2M* is treanding with the raising of IoT, requiring a higher number of devices to be interconnected without any human interaction. Different devices means different protocols, which introduces difficulties in the communications between each other. The typical approach is to define a set of translators which are able to deal with both sides, also called *Gateways*. Gateways are high-level objects that knows the various protocols needed to communicate and provide these knowledge as a service to whom has the necessity to access the functionalities provided by the device.
In our architecture each gateway is a device exposing a service using a **RESTful** architecture, for a better simplicity of use.

1.2.2 Microservices and IoT

The Microservice architecture is an innovative modelling pattern that aims to solve a well defined class of problems: scaling. The idea behind microservices is to split the architecture on different machines usually communicating through a RESTful interface. This pattern is similar to the *microkernel* architecture for Operative Systems, where the kernel containes the most vital functions and each functionality is a Plug-In (or Driver Module) that can be added externally. Each service holds a functionality isolated in it's context, which can be deployed at runtime without any interruption of the service. In microservices the equivalent of the kernel is the API interface that exposes all the functionalities to the external world. With the raising of the need of interopability, *microservices* seems to hold the key for this problem. Vendors have published their protocols, and have exposed APIâĂŹs to their various hubs. A MicroService can serve as an adapter between various protocols. It can be lightweight and disposable, both desirable traits in a rapidly evolving environment.[3]

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[1].

1.2.3 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.
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[2]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

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

[2]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.

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 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

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \end{aligned}$$

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

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[3]. [?]

1.4 Other optimizations

1.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[4]

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

[3]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.

[4]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. [?]

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 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. [?]

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.