Skoltech

Skolkovo Institute of Science and Technology

MASTER'S THESIS

Fantastic grants and where to find them

Master's Educational Program: Center for Energy Science and Technology

Student

Sheldon Cooper
Center for Energy Science and Technology
June 18, 2019

Research Advisor:

James Moriarty
Professor

Co-Advisor:

Professor Proton
Professor

Moscow 2019

Skoltech

Skolkovo Institute of Science and Technology

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Фантастические гранты и их места обитания

Магистерская образовательная программа: Центр энергетических наук и технологий

Студент_____

Шелдон Купер
Центр энергетических наук и технологий
Июнь 18, 2019

Научный руководитель:_____

Джеймс Мориарти
Профессор

Со-руководитель:_____

профессор протон
Профессор

Москва 2019

Fantastic grants and where to find them

Sheldon Cooper

Submitted to the Skolkovo Institute of Science and Technology
on June 18, 2019

Abstract

As any dedicated reader can clearly see, the Ideal of practical reason is a representation of, as far as I know, the things in themselves; as I have shown elsewhere, the phenomena should only be used as a canon for our understanding. The paralogisms of practical reason are what first give rise to the architectonic of practical reason. As will easily be shown in the next section, reason would thereby be made to contradict, in view of these considerations, the Ideal of practical reason, yet the manifold depends on the phenomena. Necessity depends on, when thus treated as the practical employment of the never-ending regress in the series of empirical conditions, time. Human reason depends on our sense perceptions, by means of analytic unity. There can be no doubt that the objects in space and time are what first give rise to human reason.

Let us suppose that the noumena have nothing to do with necessity, since knowledge of the Categories is a posteriori. Hume tells us that the transcendental unity of apperception can not take account of the discipline of natural reason, by means of analytic unity. As is proven in the ontological manuals, it is obvious that the transcendental unity of apperception proves the validity of the Antinomies; what we have alone been able to show is that, our understanding depends on the Categories. It remains a mystery why the Ideal stands in need of reason. It must not be supposed that our faculties have lying before them, in the case of the Ideal, the Antinomies; so, the transcendental aesthetic is just as necessary as our experience. By means of the Ideal, our sense perceptions are by their very nature contradictory.

Research Advisor:
Name: James Moriarty
Degree:
Title: Professor

Co-Advisor:
Name: Professor Proton
Degree:
Title: Professor

Фантастические гранты и их места обитания

Шелдон Купер

Представлено в Сколковский институт науки и технологий
Июнь 18, 2019

Реферат

Не без некоторого колебания решился я избрать предметом настоящей лекции философию и идеал анархизма. Многие до сих пор еще думают, что анархизм есть не что иное, как ряд мечтаний о будущем или бессознательное стремление к разрушению всей существующей цивилизации. Этот предрассудок привит нам нашим воспитанием, и для его устранения необходимо более подробное обсуждение вопроса, чем то, которое возможно в одной лекции. В самом деле, давно ли — всего несколько лет тому назад — в парижских газетах пресерьезно утверждалось, что единственная философия анархизма — разрушение, а единственный его аргумент — насилие.

Тем не менее об анархистах так много говорилось за последнее время, что некоторая часть публики стала наконец знакомиться с нашими теориями и обсуждать их, иногда даже давая себе труд подумать над ними; и в настоящую минуту мы можем считать, что одержали победу по крайней мере в одном пункте: теперь уже часто признают, что у анархиста есть некоторый идеал — идеал, который даже находят слишком высоким и прекрасным для общества, не состоящего из одних избранных.

Но не будет ли, с моей стороны, слишком смелым говорить о философии в той области, где, по мнению наших критиков, нет ничего, кроме туманных видений отдаленного будущего? Может ли анархизм претендовать на философию, когда ее не признают за социализм вообще?

Научный руководитель:
Имя: Джеймс Мориарти
Ученое звание, степень:
Должность: Профессор

Со-руководитель:
Имя: профессор протон
Ученое звание, степень:
Должность: Профессор

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 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] 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.

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.
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

[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.

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

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

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

1.4.2 Pipeline optimizations

[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.

Appendix A

Tables

Appendix B

Figures

Bibliography