UNIVERSITY COLLEGE CORK

An Implementation of Locality Sensitive Bloom Filter

by

Mervyn Edmond Galvin

113315141

Supervised by

Marc van Dongen

Final-Year Project

BSc in Computer Science

in the

Department of Computer Science

College of Science, Engineering and Food Science

Submitted

April 2017

UNIVERSITY COLLEGE CORK

Abstract

A bloom filter is a space efficient probabilistic data structure which can quickly test for set membership.

However, since a standard bloom filter will only test for presence, it has limitations. This project shows the feasibility of Locality Sensitive Bloom Filters (LSBF) for single-dimensional data and points to further, theoretical, algorithms for distance sensitive hashes that can be used efficiently for multidimensional data. This report also contains an approximate calculation of space and time complexity.

This has applications in areas like machine learning and searching where an answer thats close enough may be satisfactory.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to anothers work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other students work, whether published or unpublished, electronically or in print.

Signed: _____

Date: _____

"We feel free because we lack the very language to articulate our unfreedom."

Slavoj Žižek

Acknowledgements

Dedicated to my mother, who instilled in me the value of education.

With thanks to Marc van Dongen for his support throughout the research process.

Contents

Chapter 1

Introduction

This report and implementation of Locality Sensitive Bloom Filter contains several distinct parts:

- A recursion based LSBF in Python
- A comparison between LSBF and other search methods
- Research that was done into implementing the project on an FPGA
- A critical analysis of this implementation of LSBF and improvements that can be made in future research

The aim of this report is to prove that an efficient implementation of LSBF is possible and to point to the research in the field of hashing algorithms that inspired this investigation. This report also contains recommended reading for future improvements that can be made to the provided implementation of LSBF.

Chapter 2

Analysis

2.1 Project Objectives

This project started as an investigation into hashing, Haskell and how Cλash can be used to program an FPGA with Haskell code. During the research phase of this project, it was noted that the student would be required to have knowledge of Hardware Definition Language (HDL) to effectively understand how to program an FPGA. Therefore the specification of this project was adapted.

The objectives of this project are to:

- demonstrate how one would set-up the provided FPGA for programming
- provide an implementation of a hashing algorithm that would be suited to the FPGA hardware

2.2 A Guide to Setting up the FPGA

The provided FPGA was the DE0 Nano[1], a compact FPGA suited to education. Resources exist on the manufacturer web-page as well as the provided software on how to set up the device. In the interest of making it clearer to future students as to how to configure and program the device a guide was written by the student, supplied at appendix B

2.3 Hashing Investigation

Due to FPGAs being suited to parallel operations and the speed of their operation, hashing was the focus of this investigation. It was decided to research Distance-Sensitive Bloom Filters, based on the work of Mitzenmacher and Kirsch. [2] This report will be using the same constants as their paper, as well as some commonly used notation with regards Bloom Filters:

Symbol	Meaning
U	A metric set (i.e. n-dimensional tuple)
S	A query set, $S \subset U$
$\varepsilon \geq 0$	The lower bound with regards distance
$\delta > \varepsilon$	The upper bound with regards distance
m	The number of bits in the array
k	The number of hash functions
n	The number of inserted elements
p	The false positive rate
d	A metric (distance function)

Chapter 3

Design

3.1 Algorithm Design

As the project was originally intended for Haskell, it was decided that a recursive implementation of LSBF would be written for ease of porting in the future.

Nested bit arrays were used as a space efficient way to store the set bits. From a high-level perspective, bits are set based on the output of the hash of the input. The number of hashes is important in determining run-time and likelihood of false positive. If a minimum false positive rate is desired while optimising the number of hashes, then k depends on m and the expected n as given below.[3]

$$k = \frac{m}{n} \ln 2$$

We can calculate the optimum number of hashes to use based on our tolerated probability of a false positive rate p using the following formulae:[4]

$$p = \left(1 - e^{-(\frac{m}{n} \ln 2)\frac{n}{m}}\right)^{\frac{m}{n} \ln 2}$$

which simplifies to

$$\ln p = -\tfrac{m}{n}(\ln 2)^2$$

meaning that

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Finally, k for our desired p can be derived as

$$k = -\frac{\ln p}{\ln 2}$$

Here, we can see that for a given p, k only depends on p and $m \propto n$.

For ease of implementation two common hashes were used to prove how LSBF would work, however suggestions on more suitable hashes will follow in chapter 5: Evaluation. Additionally the above formulae work for regular Bloom filters but would need to be adapted for LSBF.

3.2 Pseudo code

While Mitzenmacher's Distance-Sensitive Bloom Filter has both a false positive and false negative rate, this version of LSBF makes an effort to eliminate false negative by use of rounding. In the pseudo code notation, we will use α to denote the upper bound for locality and β for the rounding distance.

Input: k, m, α, β
Result: An LSBF Object
if $k > 0$ *and* $m > 0$ **then**
    initialize an array of m zeros;
    store α, β, k in the object;
end

Algorithm 1: LSBF Set Up

Adding an element to the LSBF is trivial in abstract. We assume that a function "round" exists that rounds to the nearest β, the second parameter:

Input: input
Result: None
foreach *hash, up to* k **do**
    Set a bit in the object's array to 1 based on the result of
    $hash(round(input, \beta)) \mod m$;
end

Algorithm 2: Adding an element to the LSBF

In order to make the next section of code easier to understand, we will state a function that hashes for each k and returns True when all checked bits are set to one.

Checking if a value is in the LSBF is, in this implementation, a recursive check of the space from $input - \alpha$ to $input + \alpha$ with a reduction of α by β in each successive recursion until α is less than β.

Input: input, α
Result: True when an iteration shows all checked bits set to one, False otherwise
if $\alpha < \beta$ **then**
    $i := 0$;
    while $i < k$ **do**
        if *the bit given by* $hash_i$*(input) is one* **then**
            $i := i + 1$;
        else
            return *False*
        end
    end
    return *True*
else
    $i := 0$;
    while $i < k$ **do**
        if *the bit given by* $hash_i$*(input-*β*) is one* **then**
            $i := i + 1$;
        else
        end
    end
end

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusions

Appendix A

LSBF.py

```
import hashlib
import math

from bitarray import bitarray

class LocalitySensitiveBloomFilter:
   MAX_HASHES = 2 #The number of hashes currently implimented

   #Bloom Filter Basic Operations
   def __init__( self, floatPrecision, hashes,  bloomRange, resolution ):
      if not floatPrecision >= 0:
         return None # floatPrecision needs to be positive
      self.FLOAT_PRECISION = floatPrecision
      if not hashes > 0:
         return None
      if not hashes <= self.MAX_HASHES:
         return None
      self.NUM_HASHES = hashes
      self.LOCALITY_RANGE = bloomRange
      self.LOCALITY_RESOLUTION = resolution
      self.floatString = "{0:." + str(floatPrecision) + "f}"
      emptyArray = bitarray(2**16)
      emptyArray.setall(False)
      self.bloomArray = []

      for count in range( 0, 65536 ):
         self.bloomArray.append(emptyArray)

def setArrayBit( bloomArray, bitPosition ):
   bloomArray.bloomArray[bitPosition[0]][bitPosition[1]] = True
   return

def getArrayPos( bloomArray, input ):
   #This takes the first 16 + 16 bits of the hash and turns it into
   #a tuple, the position of a bit
   return ( int(bin(int(input, 16))[2:].zfill(8)[0:16], 2),
      int(bin(int(input, 16))[2:].zfill(8)[16:32], 2) )
```

```
def addToBloom( bloomArray, input ):
   input = bloomArray.floatString.format(input)
   bloomArray.setArrayBit(bloomArray.getMD5HashPosition(input))
   if bloomArray.NUM_HASHES > 1:
      bloomArray.setArrayBit(bloomArray.getSHA1HashPosition(input))

def checkInBloom( bloomArray, input ):
   input = bloomArray.floatString.format(input)
   if bloomArray.getMD5HashPresence( input ):
      if bloomArray.NUM_HASHES > 1:
         return bloomArray.getSHA1HashPresence( input )
      else:
         return True
   return False

def localityBloomCheck( bloomArray, input, range=0, recursionDepth=0 ):
   if (range == 0) and not (recursionDepth == bloomArray.LOCALITY_RANGE /
      bloomArray.LOCALITY_RESOLUTION):
      range = bloomArray.LOCALITY_RANGE
   if (range < bloomArray.LOCALITY_RESOLUTION):
      return bloomArray.checkInBloom( input )
   else:
      if not bloomArray.checkInBloom( input - range ):
         if not bloomArray.checkInBloom( input + range ):
            if not bloomArray.localityBloomCheck( input,
               range - bloomArray.LOCALITY_RESOLUTION,
               recursionDepth+1 ):
               return False
   return True

#Hash Functions
#MD5 Based
def getMD5HashPosition( bloomArray, input ):
   m5 = hashlib.md5()
   m5.update(str(input).encode('utf-8'))
   return bloomArray.getArrayPos(m5.hexdigest())

def getMD5HashPresence( bloomArray, input ):
   inputPosition = bloomArray.getMD5HashPosition( input )
   return bloomArray.bloomArray[inputPosition[0]][inputPosition[1]]

#SHA1 Based
def getSHA1HashPosition( bloomArray, input ):
   sha1 = hashlib.sha1()
   sha1.update(str(input).encode('utf-8'))
   return bloomArray.getArrayPos(sha1.hexdigest())

def getSHA1HashPresence( bloomArray, input ):
   inputPosition = bloomArray.getSHA1HashPosition( input )
   return bloomArray.bloomArray[inputPosition[0]][inputPosition[1]]
```

Appendix B

A Guide to Set-up the DE0-Nano

B.1 Software

The required software can be downloaded from the Altera website. Quarus Prime Lite is recommended, as well as ModelSim and the device drivers (Cyclone IV for the DE0-Nano) [5]

B.2 USB-Blaster Installation

The USB-Blaster is required to enable communication between the system and the FPGA device. On a Windows computer, this can be done by following these steps:

1. Connect the DE0-Nano to the PC
2. Open the Device Manager
3. Open the properties page for the USB-Blaster and click "Update Driver..."
4. Select Browse my computer for driver software and navigate to the installation folder of Quartus (we'll call it %QuartusDir%). Find the folder `%QuartusDir%\{version}\quartus\drivers\usb-blaster\x32` and select it.
5. Click next and install the driver

The USB-Blaster will now be available in Quartus Prime, allowing the device's software to be changed.

B.3 Example Code

When the software is installed, sample code can be used to test that the system is set up correctly. The examples can be found in `%QuartusDir%\{version}\quartus\qdesigns`. Note that not all of these will be compatible with the DE0-Nano.

When compiling, the target board needs to be set. For the DE0-Nano, you may open the project settings (Ctrl+Shift+E) and click the "Device/Board..." option on the top-right corner. In the Device family, select Cyclone IV E and type "E22F17C6" in the Name Filter to filter to the DE0-Nanos on-board FPGA. Accept the settings and now you can compile for the DE0-Nano.

To compile, click "Compile" (Ctrl+L). This may take some time to complete, depending on the complexity of the project being compiled. When finished compiling, one may write to the FPGA using the Programmer (Tools >Programmer). When in the programmer, open "Hardware Setup..." and select the "USB-Blaster". The FPGA software will be re-written by clicking "Start". Note that this will only program the FPGA and not the configuration device, so when the device is powered down the program will be wiped from the FPGAs memory and the program on the configuration device will be re-written onto the device when the device is rebooted. This is useful, however, during testing as writing to the FPGA is a lot quicker than writing to the configuration device.

One may think of the configuration device as a bootstrapper, writing the program to the FPGA's main memory on boot.

B.4 Integrating ModelSim-Altera

To integrate ModelSim-Altera with Quartus, open the Quartus Options (Tools >Options) and select the "EDA Tool Options"menu. Click the three dots next to ModelSim-Altera and navigate to the ModelSim executable. This will be in the "`modelsim_ase\win32aloem`"directory.

Bibliography

[1] Terasic - de main boards - cyclone - de0-nano development and education board. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=593, 2017.

[2] *Distance-Sensitive Bloom Filters*, January 2006. URL http://dl.acm.org/citation.cfm?id=2791175.

[3] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. https://books.google.ie/books?id=0bAYl6d7hvkC, ISBN 0521835402, 9780521835404.

[4] D. Starobinski, A. Trachtenberg, and S. Agarwal. Efficient pda synchronization. *IEEE Transactions on Mobile Computing*, 2(1):40–51, Jan 2003. ISSN 1536-1233. doi: 10.1109/TMC.2003.1195150.

[5] Quartus prime lite edition – download centre. http://dl.altera.com/?edition=lite, 2017.