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in a scholarly manner, whether from books, papers, lecture notes or any other students work, whether published or unpublished, electronically or in print.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, $hash_i(input)$. 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. However, for establishing a rough baseline, this would do fine.

3.2 Pseudocode

While Mitzenmacher's Distance-Sensitive Bloom Filter has both a false positive and false negative rate[2], this version of LSBF makes an effort to eliminate false negative by use of rounding. In the pseudocode notation, we will use α to denote the upper bound for locality and β for the rounding distance.

Algorithm 1: LSBF(k, m, α, β), LSBF Set Up

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;

    return *a reference to new LSBF object*

end

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:

Algorithm 2: add(*input*),Adding an element to the LSBF

Input: input

Result: None

$input := round(input, \beta)$;

foreach *hash, up to k* **do**

  Set a bit in the object's array to 1 based on the result of

  $hash(input) \mod m$;

end

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

Algorithm 3: checkInput(*input*), Checking bits when testing for presence

Input: input

Result: True when the bits defined by $hash_i(input)$ are all one for every i, False otherwise

$input := round(input, \beta)$;

$i := 0$;

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*

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.

Algorithm 4: recursiveInputCheck(input, α)

Input: input, α

Result: True when an iteration shows all checked bits set to one, False otherwise

$input := round(input, \beta)$

if $\alpha < \beta$ **then**
  return $checkInput(input)$
else
  if *not* $checkInput(input - \beta)$ **then**
    if *not* $checkInput(input + \beta)$ **then**
      if *not* $recursiveInputCheck(input, \alpha - \beta)$ **then**
        return *False*
      end
    end
  end
  return *True*
end

3.3 Example

To illustrate this algorithm in action, we will run through an example here. This will follow a Python-like syntax, but will not be actual Python code.

```
#Creates an LSBF object that uses 2 hashes, has a bit space of 10000,
# and will check whole numbers in a range plus-minus 5 the query value
> myLSBF = LSBF(2, 10000, 5, 1)

#Add values to the filter
> myLSBF.add(12) #now up to 2 bits are set
> myLSBF.add(42) #up to 4 bits are set

#We may use this as a normal Bloom Filter
> myLSBF.checkInput(12)
< True
> myLSBF.checkInput(14)
< False

#But it also rounds to the nearest whole number
> myLSBF.checkInput(22.1)
< Checking 12: True

#Now the recursive locality check with some verbose output
#An actual implementation would use the already defined alpha, as in Appendix A
```

```
> myLSBF.recursiveInputCheck(12, 5)
< Alpha: 5; Beta:1;
< Checking (12 - 5): False
< Checking (12 - 5): False
< Alpha: 4; Beta:1;
< Checking (12 - 4): False
< Checking (12 - 4): False
< Alpha: 3; Beta:1;
< Checking (12 - 3): False
< Checking (12 - 3): False
< Alpha: 2; Beta:1;
< Checking (12 - 2): False
< Checking (12 - 2): False
< Alpha: 1; Beta:1;
< Checking (12 - 1): False
< Checking (12 - 1): False
< Alpha: 0; Beta:1;
< Alpha < Beta; Checking 12: True
```

Chapter 4

Implementation

4.1 Language and Libraries

The programming language Python was chosen for its ease of prototyping and vast libraries covering most required functionality. Python version 2.7 was used as most libraries support it and there is extensive documentation for it online.

The libraries *math* and *hashlib* are native to Python and imported for some basic mathematical functionality and to make use of some built-in hashing algorithms. The library "*bitarray*" is an efficient object type representing an array of booleans. It is implemented in C and abstracted to Python for use in Python programs. [5] It offers much functionality but the small memory footprint is of most use to this project.

Regarding possible future work a library, MyHDL, is available for Python. The most recent release adds support for Python 3. This library allows for code written in Python to be converted to VHDL for writing to an FPGA or ASIC. [6] As previously mentioned, to program with VHDL in mind would require experience in VHDL. Nonetheless, this package demonstrates that it is possible to effectively convert a Python MyHDL design to VHDL. MyHDL could become an integrated part of an effective workflow for a student that wishes to investigate this further.

4.2 Development and Testing

A class was written in Python that implements the algorithm from Chapter 3 and is attached at Appendix A. For simplicity and to avoid clashes 65,536 bitarrays of length 65,536 are used for the Bloom filter, giving $m = 4,294,967,296$. The bit array in this case uses around 500 Megabytes. This is easy to use as we can just take the first 16 bits of a hash to find the position of the first array and the next 16 bits to select a bit in that array. In retrospect, using modulo on the result of the hash would have been an easier solution.

Two hash functions, SHA1 and MD5 were used. Chapter 5 contains the criticisms of this approach and suggested alternative.

Chapter 5

Evaluation

5.1 Hash Functions

Two hash functions were chosen to generate the hashes, SHA1 and MD5. MD5 is not considered cryptographically safe, but this doesn't matter in a Bloom filter, we just need a hash function that is uniform in its distribution and independent. MD5 is also a quick cryptographic hash so is of particular use to us in a Bloom filter. That being said, there is strong evidence that MurmurHash is a good hash to use in Bloom filters [7] despite not being considered a cryptographic hash

Looking at real world implementations of Bloom filters, it's clear that a better approach would've been to use the whole length of a 64-bit hash, splitting it up into four 16 bit indices.[8]

Considering all this, it is extremely likely that better results for LSBF would be obtained by making use of the aforementioned techniques.

5.2 Speed Comparison

A linear search has a $\mathcal{O}(n)$ time complexity on average which is comparable to the non-LSBF search function that was implemented for comparison, attached at Appendix C.

LSBF on the other hand runs in $\mathcal{O}(\frac{k\alpha}{\beta} + k); 0 \leq \alpha < \beta; k \geq 1$ worst case with a constant space complexity of $\mathcal{O}(m)$. However, if the hash algorithm is computationally expensive (like SHA1) each k will take a long time.

Using the timing demonstration in Appendix C, we can see that for a large n LSBF would start to outperform linear search. However, it should be noted that this approach to testing execution time is influenced by system load and is a naive implementation. The tests were re-run using Python's *timeit* which only takes into account CPU time and is best practice for execution time comparison. The following results were obtained:

Algorithm	**Execution Time**
Iterative	42.044 μs
LSBF (2 Hashes)	63.502 μs
LSBF (MD5 only)	53.435 μs

Each method was run using *timeit.repeat* with a value $number = 10000$ and randomly generated numbers in the range 0 – 10000 tested as the query and 1000 values stored. It appears here that Iterative performs the best but when we increase n to 100,000 we can see something interesting.

Algorithm	**Execution Time**
Iterative	58.434 μs
LSBF (2 Hashes)	15.988 μs
LSBF (MD5 only)	9.821 μs

Here, LSBF vastly outperforms an iterative approach, likely due to the fact that n has increased beyond the range of the numbers we're picking from. This shows how effective LSBF is when a positive result is likely. With the range of numbers randomly selected increased to 0 – 10,000,000 we see this:

Algorithm	**Execution Time**
Iterative	6941.92886 μs
LSBF (2 Hashes)	29.389 μs
LSBF (MD5 only)	19.419 μs

This final result shows that for a large range and large n, LSBF still outperforms an iterative search. It should be noted, however, that false positives weren't checked in these timings.

Chapter 6

Conclusions

LSBF shows lot of promise as a method to quickly find if a similar value is in a set. Multiple dimensions will add more complexity to the execution time and algorithm overall, but is doable and may also prove a beneficial endeavour.

6.1 Future Work

Research in the field of Locality Sensitive Hashing is ongoing with promising space and time efficiency being shown of $\mathcal{O}(n^{1+p}/p_1 + dn)$ and $\mathcal{O}(d \cdot n^p/p_1)$ respectively, where p_1 is the lower bound of p. [9] Implementing a hashing algorithm based on this research or the work of Andoni and Indyk [10] would have vast applications in image recognition, databases, IP traceback and DNA sequencing. [2]

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) [11]

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.

Appendix C

Timer.py

```
import LSBF
import random
import time

current_milli_time = lambda: int(round(time.time() * 1000))

regularArray = []

def randomTest():
  myBloom = LSBF.LocalitySensitiveBloomFilter(2, 2, 5, 1)
  for i in xrange(1000):
    x = random.randrange(0, 10000)
    myBloom.addToBloom(x)
    regularArray.append(x)
    print(str(x))
  return myBloom

def iterativeCheck(array, testValue, localityRange):
  for x in array:
  if (x >= (testValue - localityRange)) and (x <= (testValue + localityRange)):
    return True
  return False

print("randomTest(): ")
myBloom = randomTest()
for i in xrange(10):
  inputFloat = float(raw_input("test a number for approximate presence: "))
  print("Timing LSBF:\r\n")
  a = current_milli_time()
  for x in xrange(10000):
    myBloom.localityBloomCheck(inputFloat)
  b = current_milli_time()
  print(str(myBloom.localityBloomCheck(inputFloat)))
  print("Took " + str( (float) (b-a)/10000) + "ms\r\n")

  print("Timing iterative approach:\r\n")
  a = current_milli_time()
```

```
for x in xrange(10000):
  iterativeCheck(regularArray, inputFloat, 5)
print(str(iterativeCheck(regularArray, inputFloat, 5)))
b = current_milli_time()
print("Took " + str( (float) (b-a)/10000) + "ms\r\n")
```

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] A. Kirsch and M. Mitzenmacher. Distance-sensitive bloom filters. *Proceedings of the Meeting on Algorithm Engineering and Experiments*, pages 117–122, 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] bitarray 0.8.1 : Python package index. https://pypi.python.org/pypi/bitarray/, 2013.

[6] Myhdl. http://www.myhdl.org/, 2015.

[7] Some performance tweaks by vmg - pull request 19 - bitly/dablooms. https://github.com/bitly/dablooms/pull/19, 2012.

[8] Squid web proxy faq - what hash functions does squid use? http://wiki.squid-cache.org/SquidFaq/CacheDigests#What_hash_functions_.28and_how_many_of_them.29_does_Squid_use.3F, 2008.

[9] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L.Schmidt. Practical and optimal lsh for angular distance. *Proceedings of the 29th Annual Conference on Neural Information Processing Systems*, September 2015. URL https://arxiv.org/abs/1509.02897.

[10] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM - 50th anniversary issue: 1958 - 2008*, 51:1236–1239, January 2008. URL http://dl.acm.org/citation.cfm?id=1327494.

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