NTNU
Norwegian University of
Science and Technology

# Optimization of Seed Selection for Information Diffusion with High Level Synthesis

Julian Lam

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Donn Morrison, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# Abstract

Information diffusion is where a message or data is passed from vertex to vertex in a network via edges. Information diffusion is often used for simulations in network research because it estimates how information propagates through a network. A set of vertices that in initial state contains the data is known as the *seed nodes*. The diffusion starts at the seed nodes and propagates over the network. Finding the optimal seed nodes is a crucial part of information diffusion. The seed nodes are the optimal targets to start passing messages during a disaster scenario, vaccinate to prevent the spreading of a disease or even targets for viral marketing. Finding these seed nodes is an NP-hard problem and require a significant amount of computation. One common used model in information diffusion is the *independent cascade model* (ICM). ICM can be performed as a *sparse matrix-vector multiplication* (SPMV). By using a application specific hardware accelerator, we can accelerate the diffusion process and consequently, the seed selection. Recently, *high-level synthesis* (HLS) have been getting more attention. HLS transform high-level implementation and algorithms to low-level designs. With HLS we designed a application specific *intellectual property core* (IP-core) which we then implemented on the FPGA and perform seed selection.

# Sammendrag

Information diffusion er når en melding eller data blir sendt fra node til node via kantene i et nettverk. Information diffusion er ofte brukt i simulering i nettverks forskning siden den kan estimere hvordan data propagerer gjennom et nettverk. Et sett med noder som starter med dataen er kjent som *seed nodes*. Diffusjonen starter ved *seed nodes* og propagerer gjennom hele nettverket. å finne de mest optimale *seed nodes* er en essensiell del av *Information Diffusion*, siden de kan spre bedskjedet til mest mulig mennesker under en krisesituasjon, ved å vaksinere dem så kan man forhindre spredning av sykdommer, eller så er de *seed node* de optimale kandidater for å promotere produkt.

Å finne disse *seed nodes* er et NP-hard problem og krever stor mengde beregninger. En vanlig brukt modell er kjent som Independent cascade model(ICM). ICM kan bli gjort via *sparse matrix-vector multiplication* (SPMV). Ved å bruke en applikasjons spesifikk hardware akselerator, sp kan vi optimalisere diffusjons prosessen og *Seed selection*. High level synthesis har i de siste fått mye mer oppmerksomhet. HLS transformerer høy-nivå implementasjoner til lav-nivå design. Ved hjelp av HLS kan vi utvikler applikasjons spesifikk *intelectual property core* (IP core) som vi implementerte i en *field-programmable gate array* (FPGA) og utføre *seed selection*

# Foreword

This thesis concludes my study in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The supervisor for this thesis was Professor Donn Alexander Morrison.

I would like to thank Donn Alexander Morrison and Yaman Umuroglu for providing helpful advice and feedback throughout this thesis, and answering e-mails from me way past midnight. I would also like to thank my girlfriend, family and friends for helping and supporting me through these five years here in NTNU. A special thanks to my friend Magnus Halvorsen for proofreading, his insights, and critique.

Julian Ho-Yin Lam
Trondheim June 15,2016

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

*Information Diffusion* is a field in network research where a message, or data, is propagated through a *network* or a *graph*. The message originates from a chosen set of vertices, known as *seed nodes*. These seed nodes pass the message to its neighbours through the edges and thus propagates the message over the network. The effectiveness of the diffusion is measured through the spread and the speed of propagation and is dependent on the chosen seed nodes. By finding the most optimal set of seed nodes, we can potentially stop an epidemic by vaccinating influential vertices, we can find important targets for viral marketing by giving free samples, and use this information to spread messages quickly during disaster scenarios[2] [3].

There are multiple studies done regarding information diffusion, [2],[4], [5], [6]. But as far as we know, there are none that focuses on optimizing the seed selection in hardware. The current seed selection algorithm is a greedy solution[7], where every set of vertices is tested and the set with the best coverage and time is chosen. This is a time-consuming process and highly parallelizable, which makes it a good candidate for *Field-programmable gate arrays*(FPGAs).

*High-Level Syntesis* (HLS) transform high-level behaviour and constraints to lower level design.[8]. It makes it possible to implement an algorithm in a high-level language, C or C++, and generate an optimal design in *verilog* or *VHDL*. Verilog and VHDL are hardware descriptive languages designed to describe digital systems [9].

Unlike traditional hardware design, HLS allows designers with limited knowledge of hardware design to create an optimal custom *Intellectual property core*(IP-core). In HLS, programmers can test out different optimization schemes in a short

period, thus reducing development overhead.

In this thesis, we have implemented a simple IP-core that performs information diffusion using the *Independen Cascade model* (ICM) as *Breadth-First Search* (BFS) over boolean semiring. This is done by using HLS as the development tool.

## 1.2 Assignment Interpretation

From the assignment text, these task were chosen as the main focus of this thesis:

**Task 1** ***(mandatory)*** Implement Information Diffusion as Sparse matrix vector multiplication, with high level language C.

**Task 2** ***(mandatory)*** Tailor the implementation of Information Diffusion for synthesise with Vivado HLS.

**Task 3** ***(optional)*** Implement said design on a Zynq FPGA board.

**Task 4** ***(optional)*** Extend the system to be able to handle graph in the size of toy graphs(containing $2^{26}$) vertices)

## 1.3 Report Structure

We have here the basic outline for this report and a short overview of the remainder of this report:

**Chapter 2: Background** contains the theory regarding networks, information diffusion, matrix-vector multiplication and high-level synthesis.

**Chapter 3: Related Work** gives a short introduction of the state of the art of HLS implementations, information diffusion research and different optimization of BFS.

**Chapter 4: Design and Implementation** present our implementation of our IP-core and give a brief introduction regarding HLS implementation and optimization.

**Chapter 5: Result and Discussion** will compare the result our core generated compared to a C-simulation. We will also discuss some of the design choices

regarding the IP-core.

**Chapter 6: Future Work** present how our design can be further improved.

**Chapter 7: Conclusion** provides concluding remarks regarding this paper and a summary of the identified tasks.

# Chapter 2

# Background

In this chapter, we will look at the fundamental concepts and theory of the different diffusion models, seed selection algorithms and perform BFS over the boolean semiring. We will also have a look at HLS and specifications of the Zedboard. This chapter will contain notations that we will use throughout the report.

We will look at the independent cascade model, which is a special case of breadth first search [10]. By looking at how to improve BFS, we can apply such optimization to ICM and the seed selection algorithm.

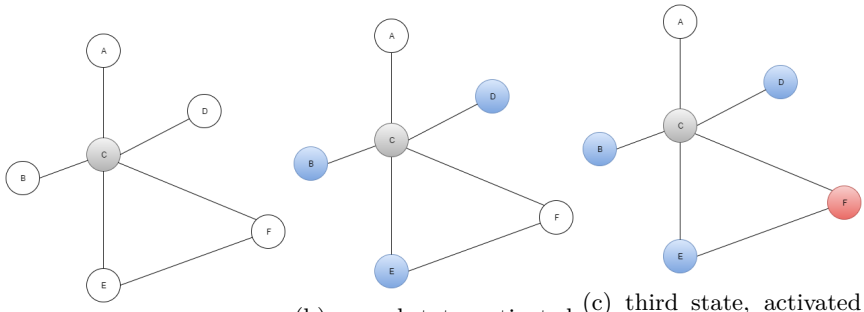
## 2.1 Information Diffusion

Information diffusion is looking at how information is propagated through a network. A vertex can be either activated(infected) or inactivated(healthy/noninfected), each vertex can spread the contagion(activation,infection) to their neighbour. Some examples would be how a meme, a trend or a disease is spread through a community. The process consists of a set of starter vertices, which we will call seed nodes, which are "infected" at initial time step. During each time step, there are a percentage $p_g$ where the "infected" vertices would "infect" its neighbours. Seed nodes is a set of $k$ vertices that in the initial time-step are infected. They will pass on the information/infection during each iteration, and the information/infection will propagate through the network.

## 2.2 Basic Diffusion Models

For information diffusion, two common models are used for simulations. Those are the *linear threshold model*(LTM) and the *independent cascade model*(ICM) [11].

ICM is a model where each spread of contagion is dependent on a "coin flip". Each person has a chance to be contaminated, and during each diffusion, dependent on the result of the coin flip, that person is either contaminated/infected or healthy. In ICM, an infected person can not reinfect the previously spared person. The probability of infection can be either globally, or locally. In Figure 2.1a, we can see a situation where $C$ have five neighbour,$A$,$B$,$D$,$E$ and $F$. In the initial state, only C is infected. Five separate "coin flips" was done and resulted in three more activations as shown in Figure 2.1b. C spread the contagion to three other people, while A and F were spared. In Figure 2.1c F is activated by E.




(a) Initial state

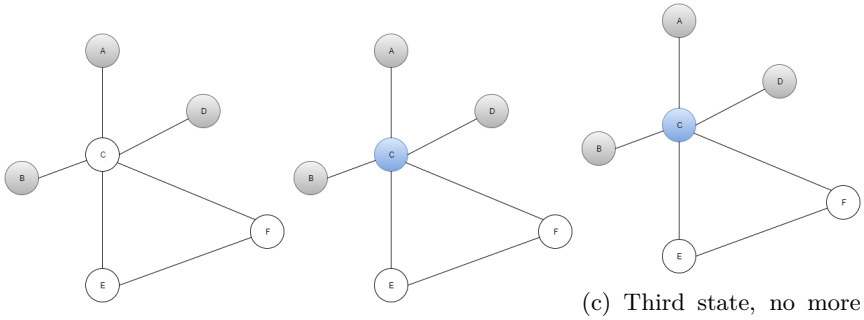
(b) second state, activated by C is marked blue

(c) third state, activated by E is marked red, no more activation from C

Figure 2.1: Independent cascade model

For the LTM activation is not dependent on a probability and a "coin flip", but an internal threshold of activated neighbours. The spread of contagion is dependent on how large of a fraction of their neighbour is activated. For our example, let say that for a person to be infected, 0.6 of their total neighbour must be infected before they can contract the contagion. In Figure **??**, we see that in the initial state $A$,$B$ and $D$ are infected. For $C$, more than 0.6 of its total neighbours are activated, this resulted in an activation of it too as seen in Figure 2.2c. We can see that both $E$ and $F$ have only 0.5 of their neighbours activated, and thus in Figure 2.2c no more activation is presence.

A real world example of the LTM would be e.g. A new product on the market. People would adopt the new product it enough of their acquaintance is




(a) Initial state, A,B,D is activated.

(b) Second state, S is activated and marked blue.

(c) Third state, no more activation, since both E and F have less then 0.6 activated neighbours.

Figure 2.2: Linear Threshold mode

using it (activated). While ICM would be more directed marketing. Where free samples are given to some users. Those users would then promote the product and potentially spread the product to other.

ICM is a special case of BFS, both algorithm traverse graphs in a level-by-level approach. The difference is that for ICM, each node is not guaranteed to be activated. If an ICM had an activation chance of a hundred percent, then the ICM would be the same as BFS.

## 2.3 Breadth First Search

BFS is a tree traversal algorithm. BFS starts at the root vertex $v_r$. The algorithm then stores all $v_r$s children vertices in a *queue*. The algorithm then takes the first vertex from the queue, $v_1$ and stores all the children vertices to $v_1$ in the back of the queue. This process continues until the queue is empty and all the vertices have been iterated over.

BFS is a common graph iteration algorithm but is often limited by the irregular memory access where the algorithm has to find the data stored in different spaces in memory.

### 2.3.1 BFS to Data Diffusion

The motivation for transforming breadth first search as matrix-vector multiplication is that displaying the graph algorithm as a matrix multiplication can display the data access pattern for the algorithm and can be readily optimized [12].

**Algorithm 1** Breadth First Search

```
1: dist[∀v ∈ V] = −1; currentQ, nextQ = ∅
2: step = 0; dist[root] = step
3: ENQUEUE(nextQ,root)
4: while nextQ ≠ ∅ do
5:     currentQ = newxtQ; nextQ = ∅
6:     step = step + 1
7:     while currentQ ≠ ∅ do
8:         u = DEQUEUE(currentQ)
9:         for v ∈ Adj[u] do
10:            if dist[v] == −1 then
11:                dist[v] = step
12:                ENQUEUE(nextQ, v)
   return dist
```

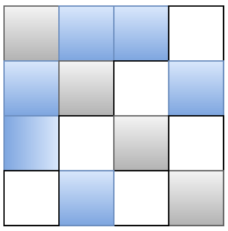
As mentioned before, ICM is a special case of the breadth first search. By modifying the algorithm proposed earlier, we can, in theory, perform ICM with matrix-vector multiplication.

## 2.4 Matrix Notations

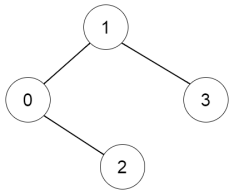
Networks can be represented as *sparse adjacency matrices* [12] [13]. By representing networks as a sparse matrix, we can often discover different ways to optimize the algorithm, or a different structure to store the data. The adjacency matrix, in particular, is an interesting way to represent the graph. A graph $G = (V, E)$, G have N vertices and M edges, and this correspond to a N×N adjacency matrix called A. If A(i,j)=1, then there is an edge from $v_i$ to $v_j$. Otherwise, it is 0. In Figure:2.3a, we can see how a undirected graph can be represented as an adjacency matrix. Each square symbolises a connection between two vertices. To generate a undirected graph as an adjacency matrix, the matrix must be mirrored diagonally, meaning if A(i,j)=1, then A(j,i)=1, if this is not true, then the matrix would be representing a directed graph.

### 2.4.1 Sparse Matrix

A sparse matrix is a matrix containing few nonzero entries. A social graph with few edges would often be represented as a sparse matrix. Since sparse matrices only have few non-zero elements, by storing only the non-zero elements, we can have savings in memory.



(a) The adjacency matrix




(b) The graph corresponding to the adjacency matrix

Figure 2.3: Sparse matrix to graph

### 2.4.2 Breadth First Search as Matrix Multiplication.

From [12], we can see that BFS can be recast as algebraic operations. BFS can be performed by applying matrix-vector multiplication over Boolean semirings [10]. The graph is represented as an adjacency matrix A, then for the root vertex, a vector x(root)=1 is multiplied with the matrix A. $A \times x_0 = y_0$. $y_0$ is the result of the first matrix-vector multiplication and in the next iteration, $x_1 = y_0$. We can see from the Figure2.4

By performing an AND operation between each element in a row in the matrix and the corresponding element in the vector, then applies the OR operation between each result from the AND operation, we can find the result, which is the row number in the. This will be further explained in Chapter 4.

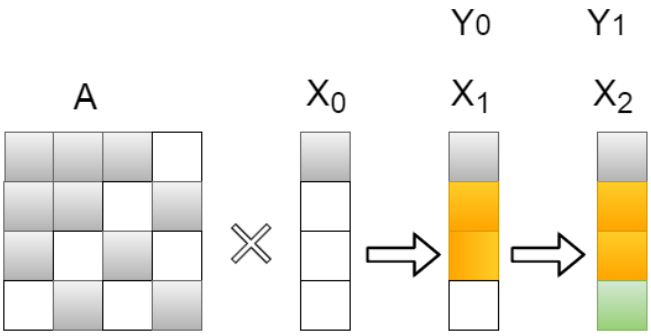



Figure 2.4: BFS on Boolean semiring

### 2.4.3 Semiring

A *semiring* is a set of elements with only two binary operations. The two operations are often known as "addition"(+) and "multiplication"(×). Semirings are often used in abstract algebra to categorize a set of elements. Boolean semirings is a set of binary elements: '1' and '0' and defined as $1 + 1 = 1$. As we shown in the previous section, the algorithm performs matrix multiplication uses the two operations, multiplications, and addition. In [10], the AND and OR operator were chosen instead of the normal addition and multiplication.

## 2.5 Seed Selection Algorithm

The seed selection algorithm is the algorithm used to select the initial $k$ seed nodes to be chosen at the start of the information diffusion. Each selected vertices is in the initial timestep activated. During each time step, the seed nodes will propagate the activation along the network depend on what diffusion model is used. We can compare it to a new gadget or a cosmetic company trying to promote a new product. By selecting a few influential persons to give a free sample, the new trend would most likely spread through *viral marketing* [14]. The seed selection algorithm would be the algorithm to select the few influential individuals to receive this free sample. There are multiple different schemes to choose from, in this section, we will focus on four different algorithms, greedy algorithm, degree algorithm, random algorithm and the independent greedy algorithm.

### 2.5.1 The Greedy Algorithm

The greedy algorithm [7] [15] proposed by Kempe et al, is known to be the optimal algorithm in seed selection according to the result from [7].

The greedy algorithm starts by choosing one vertex and perform one ICM over the entire network and stores its spread. This process repeats for every vertex available. The vertex with the best coverage would then be stored in a set $S$. The algorithm continues and chooses a new vertex in combination with the vertices in set S. This process continues until there are $k$ different vertices in set S, which will be returned as the most optimal set of k vertices for this matrix.

### 2.5.2 The Degree Algorithm

Another popular algorithm is the degree algorithm [7]. Unlike the greedy algorithm, does not compute the coverage of vertex, the algorithm picks the top $k$ vertices according to the degree distribution instead. The vertex chooses the top $k$ vertices with the highest degree and stores them as the seed nodes. This

**Algorithm 2** Greedy Algorithm

1: Start with $A = \emptyset$
2: **while** $|A| \leq l$ **do**
3: For each vertex $x$, use repeated sampling to approximate $\sigma(A \cup x)$ to within $(1 \pm \varepsilon)$ with probability 1âĹŠ$\delta$
4: Add the vertex with largest estimate for $\sigma(A \cup x)$ to A.
5: Output the set $A$ of vertices.

approach benefits over the greedy algorithm by not having as much computation time as the greedy algorithm since only one iteration is needed to compute the degree to the vertex. The disadvantage is that this algorithm does not take the degree correlation into account. High degree vertices would often have common vertex as neighbours[16]. This would result in multiple overlapping activated vertices chosen.

**Algorithm 3** Degree Algorithm

1: Start with $A = \emptyset$
2: **while** $|A| \leq l$ **do**
3: For each vertex $x$, use repeated sampling to compute DegreeMax($x$).
4: Add the vertex with largest degree to A.
5: Output the set $A$ of vertices.

### 2.5.3 Independent Algorithm

Another algorithm is the independent greedy algorithm. The algorithm iterates through the network, computing the spread of each vertex. The algorithm then chooses the vertex with the largest coverage independent of the other previous chosen vertices. This algorithm is a special case of the greedy algorithm mentioned above.

**Algorithm 4** Independent Algorithm

1: Start with $A = \emptyset$
2: **while** $|A| \leq l$ **do**
3: For each vertex $x$, use repeated sampling to approximate $\sigma(A \cup x)$ to within $(1 \pm \varepsilon)$ with probability 1âĹŠ$\delta$
4: Add the vertex with largest estimate for $\sigma(x)$ to A.
5: Output the set $A$ of vertices.

### 2.5.4 Random Algorithm

The last one is the random algorithm. The random algorithm just picks a random seed node. This approach is the simplest to implement and easiest. The downside is that this is random, and there are no strategic choosing of seed node.

## 2.6 High-Level Synthesis

High-Level Synthesis convert algorithms implemented on the higher level down to *Register Transfer Level* (RTL)[17]. RTL is models of digital circuits displaying the flow of data between register, logical operations and such. It is commonly used to describe low-level digital systems. HLS is known to be able to reduce development effort and cost of creating specialized hardware compared to traditional hand-drawn RTL designs[18][19][17]. By taking high-level languages such as C, C++, and SystemC implementations and generate the optimal architecture. HLS allows the user to generate custom *Intellectual Property*(IP) core. An IP-core is a custom created data core that has an output and an input port.

## 2.7 ZedBoard

The Zedboard that we used for this project, is *Xilinx Zynq-7000 All programmable System-on-chip(SoC)* Z-7020. Consist of a dual core *ARM cortex-A9 MPCore* based processing system(PS) and an *Artix-7 XC7Z020* FPGA. The FPGA is the *programmable logic*(PL). The Zedboard have 512 MB DDR3 RAM, 256MB Quad-SPI Flash, and 4GB SD card.[20]. The system offers the flexibility and scalability of an FPGA[21].

THe FPGA use *Advance eXtensible Interface*(AXI4)bus protocol. There are three types if AXI4 interfaces:

- **AXI4 Lite**- Simple, memory mapped communication. Useful for small single read.
- **AXI4-Stream** - for continues streaming of data.
- **AXI4** - For memory mapped applications.

A component with PL implemented would be able to connect to the PS through an AXI4 bus port. The close coupling between t

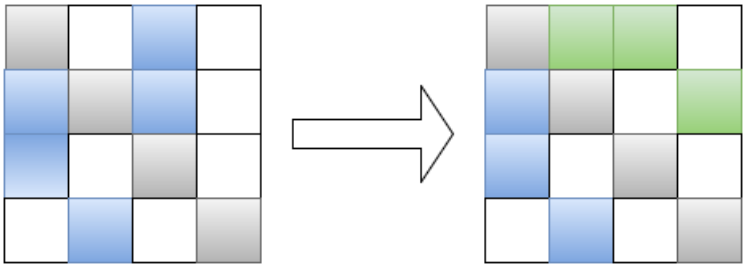


Figure 2.5: How the adjacency matrix is flipped on the diagonal

## 2.8 RMat

One problem during graph analyzation and calculation is finding suitable graphs to analyses. Generating large reproducible networks is one of the strengths of R-mat. One solution proposed by Chakrabartiy et al. is to use the ”recursive matrix” or R-mat model. The R-mat model generates networks with only a few parameters, the generated graph will naturally have the small world properties and follows the laws of normal vertices, and have a quick generation speed [1]. The R-mat models goal is to generate graphs that match the degree distribution, exhibits a ” community ” structure and have a small diameter and matches other criteria. [1]. The algorithm to generate such a recursive matrix is as follows: The idea is to partition the adjacency matrix into four equally sized part branded A, B, C, D, as shown in Figure2.5. The adjacency matrix starts by having all element set to 0. Each new edge is ”dropped” onto the adjacency matrix. Which section the edge would be placed in, is chosen randomly. Each section have a probability of $a$, $b$, $c$, $d$, and $a + b + c + d = 1$. After a section is chosen, the partition that was chosen is partitioned again. This continues until the chosen section is a 1x1 square and the edge is dropped there. From the algorithm, we can see that the R-mat generator is capable of generating graphs with total numbers of vertex $V = 2^x$. Since the algorithm partitioned the matrix into four part. This is approach would only generate a directed graph. To generate undirected graph, $b = c$ and the adjacency matrix must make a ”copy flip” on the diagonal elements, like Figure 2.5. The Rmat is reproducible if the random placement of the vertices is deterministic.

# Chapter 3

# Related Work

In this chapter, we will look at the state of research regarding High-Level Synthesis, information diffusion, and optimization of independent cascade model and breadth-first search.

## 3.1 Information Diffusion

There are multiple studies done regarding information diffusion. Studies show how information diffusion can be applied during an disease outbreak[2], viral marketing[14], coordinat during crisis situation[22].

Different models of information diffusion have been done on blogs[23][24], and Twitter[25]. We can see that in an age of social media, the studies of information diffusion is more relevant than ever.

While [6] have argued that the emerging of social networks and media have changed the traditional model. The activation is no longer only relying on neighbour vertices, but also an external influence. They found that a large amount of information volume in Twitter is the result of network diffusion, while a small amount is due to external events and factors outside the network[6]. Another study shows that during the 2011 Egyptian Uprising, how a large amount of rebel movement were "tweeted"[22] during the uprising.

As we mentioned in Chapter 2, we mainly focus on two common information diffusion models, ICM, and LTM. But there are different models too. [5] proposed several different problems with traditional models where each vertex is either *activated*(infected, influenced, '1') or *inactive*(healthy, not reached, '0'), and passes the *contagion*(information, data, infection, influence) to neighbouring vertices through the edges. The report mentioned different assumptions that such models make. Among them is that a complete graph is provided, the spread

of contagion is from a known source, and that the structure of the network is sufficient to explain the behaviour[5]. The report proposes an alternative model, *Linear Influence Model*(LIM), where the focus is on the global influence that an infected vertex has on the rate of diffusion through the implicit network. This model makes the assumption that newly activated vertices are dependent on previously activated vertices. The LIM does not need explicit knowledge of the entire network. Instead, the model takes the newly activated vertices and models them as a *influence function*, which is used to find the global influence.

## 3.2 High-Level Synthesis

High-Level Synthesis as a concept has been around since the mid-1980s and early-1990s[17][26]. Carnegie-Mellon University design automation (CMU-DA)[27][28] was a pioneering early version of HLS tools. The tool gathers quickly considerable interest. Many HLS tools were built in later years mostly for prototyping and research[29][30][31]. Some of these were able to produce real chips, but the reason for the lack of further development and adaptation was that RTL synthesis was not a widely accepted and an immature field. This often lead to suboptimal solutions.

Around the year 2000, new HLS tools were developed in academia and the industry. These tools, used high-level language, C and C++. Vivado HLS, designed by Xilinx [32], is one such HLS tool. The Vivado HLS became free during their 2015.4 update[33]. This resulted in a revived interest in HLS. The community around HLS is also evolving, on the Xilinx-forum, there are multiple answers and active members. We can see that the solution designed by HLS tools is close to traditional hand-crafted designs[34].

Cong et al,[17] discusses different problems and reason for failed commercialization of early HLS tools. concluded that the problem with the early HLS tools can be summarized by the following reasons: "Lack of comprehensive design language support", "Lack of reusable and portable design specification", "Narrow focus on datapath synthesis", "lack satisfactory QoR", and "Lack of a compelling Reason/event to adopt new design methodology".

Early versions of HLS tools was not a C-to-RTL transformation. Most of them needed a custom *Hardware Descriptive Language*(HDL). Lacking reusable and portable design specification resulted in that HLS tools required users to include detailed information regarding timing and interface information into source codes. This resulted in a target dependent solution and can't easily port to other devices. The narrow focus on datapath synthesis resulted in a lack of focus on an interface to other hardware modules and platform integration. Those aspects were left to the users to solve system integration problem. The lack of foundation to accurately measure HLS result and often failed to meet timing and power

requirement with early HLS tools were another limiting factor. The last reason was that there was no real driving force to turn developers over to such a young and early development format. HLS tools showed exciting capabilities, but most developers did not want to move from the safe and tested RTL design methodology. The paper concludes with that current(2011)HLS tools showed tremendous potentials in becoming standard in selected deployment.

### 3.2.1 Applications using HLS

In [35], HLS was used to design an accelerator for database analytic and SQL operation. The design was implemented on a Virtex-7 xc7vx690t-g1761-2 FPGA with a focus on accelerating operations; join, data filter, sort, merge and string matching. The accelerator was implemented in C++ in Vivado HLS and optimized with UNROLL directive, PIPELINE directive, and ARRAY_PARTITION. The UNROLL directive unroll all of the specified loops, while the PIPELINE directive allows multiple accelerators to process data at each clock cycle. The ARRAY_PARTITION directive partition data into registers. The accelerator showed promises, giving a 15-140× speedup compared to Postgres software DBMS running selected TPCH queries.

[36] explored the advantages and disadvantages of HLS implementation of image processing. They argued that custom algorithms on FPGA platforms will most likely result in an improvement, but the algorithms must be tailored to the platforms. The author conducted different case studies to show both the strengths and the weaknesses of HLS. The report goes through image filtering, connected component analysis and two-dimensional fast-Fourier transformation(FFT). One example that the author brings up is during image filtering where HLS was not able to identify the standard accessing pattern during special cases. This resulted in that the HLS built additional hardware to counter such an exception. The report concluded that while HLS can significantly reduce development time and improve utilization of the design space, it is still important to focus on careful design. The report concludes that HLS can offer many benefits, and is an improvement over conventional RTL-designs, but is not a replacement for hardware designers or clever designs.

[37] implemented a fast Fourier transform (FFT) algorithm for different digital system processing application in HLS. There the authors used Simulink for verification of their design, and implemented it in HLS.

[19] discuss improvements to the current HLS tools with polyhedral transformation. Here they present a problem with HLS, which is that unless the code is inherently compatible, HLS can not apply most of the optimizations. Zuo et al. proposed the polyhedral model, the model takes data dependent multi-block program as input and performs three steps: Classification of array access

patterns, performance Metric, and implementation. During the classification of array access patterns, a set of data access pattern is defined and classified. Then the appropriate loop transformation is applied. The next step, the performance of each loop transformation with data-dependency is estimated, and the best improvement is chosen. In the final step, the chosen solution, loop transformation, and inserting HLS directive are applied. Then an interface block for the data-dependent blocks is generated. The generated communication block is then optimized depending on how it behaves. The paper concludes with that the polyhedral model can model can find important loop transformation, thus enable optimization such as pipeline and parallelization.

[34] is a case-study where HLS is used to implement two compute heavy machine learning techniques with different computational properties. The two algorithms that were tested was *Lloyd's Algorithm* and a *Filtering Algorithm*. The result was that for the first case, a similar performance between the HLS solution and a hand-written solution, while the second algorithm was severely worse with HLS if the developer did not customize for HLS.

## 3.3 Different optimization scheme

There are a large amount of different optimization research on graph traversal, especially on Breadth First Search.

[10] proposed a hybrid FPGA-CPU heterogeneous platform for BFS. The idea is to run the first couple of steps on the CPU core, then switch over to the FPGA accelerator to explore the rest of the graph. The CPU is better suited for calculating while there is a smaller frontier, while the FPGA core is better suited for a larger frontier. By exploiting the characteristics of small-world networks where the frontier is much larger after two-three iterations, one can significantly improve computation time. The report proposed an alternative method of performing the breadth-first search. By performing it as a sparse matrix-vector multiplication over a boolean semiring, more parallelization option was discovered. The result was a speedup of 7.8 compared to a pure software implementation ,and 2× better compared to an accelerator-only implementation.

[38] propose a hybrid solution, combining a conventional top-down algorithm and a novel bottom-up algorithm. The optimization in this paper focuses on examining fewer edges, thus reduce computation time and trying to circumvent one major drawback with BFS; memory-bound on shared memory. The top-down approach is the traditional algorithm, where a frontier expands and visits all vertices on that level, before each vertex checks its neighbour for unvisited vertices. Unvisited vertices are placed in the frontier vector and marked as visited. The bottom-up algorithm, contrary to the top -down the algorithm, is where each children vertex tries to find a potential parent. A neighbour vertex can be the

parent if the neighbour is also in the frontier vector. This results in that after a vertex finds its parent, there is no need to traverse the rest of the frontier. By using different approach during different time, the report was able to achieve a speedup of 3.3 - 7.8× on synthetic graphs and 2.4 âĂŞ 4.6× on real social network graphs.

# Chapter 4

# Design and Implementation

In this chapter, we will present our implementation of the sparse matrix-vector multiplication (SpMV) over boolean semiring, how we implemented with HLS and how we connected the IP-cores in the FPGA.

### 4.0.1 High Level Implementation

Our implementation of the SpMV is done in HLS. The algorithm takes in an adjacency matrix and a set of seed nodes as input and outputs a result vector showing which vertices were activated. The algorithm stops when all the vertices have been activated, or it is unable to find any new candidates. Unlike normal SpMV, where each iteration will activate all their neighbours, an ICM is dependent on random number generator (RNG) and a global or local probability. For this project a global probability of 5% was used, i.e. each vertex had a 5% chance of being activated. If $v_1$ was not activated by $V_r$ on the first iteration, $V_r$ can not reactivate $v_1$ on the next iteration. To prevent a reactivation, as mentioned above, a frontier vector will be sent in instead of the result from the previous iteration.

A vector is a list of vertices. The frontier vector is generated by comparing the result from current iteration with a list of activated vertices. The vertices that are in the frontier vector are the vertices that were not activated in the previous iteration but were activated in the current iteration.

Our algorithm applies SpMV over the matrix and the frontier. For each iteration, each element in a row of the matrix is applied an AND (&&) operation with the corresponding element in the frontier vector. The results from these operations will be ORed (||) together, which gives the result for the row.

Unlike the breadth-first search on a boolean semiring, each vertex will have a chance not to be activated (set as '1'), even if matrix_row[x] and frontier[x] = 1.

```
(martix_row[0] && frontier[0])||(matrix_row[1] && frontier[1])|| ...
(matrix_row[n] && frontier[n])=result[row]
```

This resulted in that for each &&- operation, we need to && another *coin toss*, which determined if the activation takes place. The coin-toss is determined by the RNG and the global probability.

The algorithm will continue until either all of the vertices are activated, or no more vertices can be activated. This is solved with a function dist_gen, a function that stops the algorithm when either no more vertices are activated in frontier vector, or all the vertices in the result vector are activated.

In the high-level implementation, the implementation of the algorithm was done on two levels. The top level was the *TopLevelWrapper*. The TopLevelWrapper was set as the main function in our HLS implementation. The function takes in the address of the location of the matrix, the address of the result, and the address to the frontier. Since we are working with ICM, a global probability, a random seed as the initial state for the LFSR is also set as input. The TopLevelWrapper stores the useful data in a local buffer, where it is sent to the datapath function. The datapath function is a sparse matrix-vector multiplication.

We can see the overview of our implementation in Figure 4.1. Where the processing system sends the adjacency matrix and the frontier vector to out IP-core; called TopLevelWrapperd

### 4.0.2 Seed Selection

Seed selection that was implemented was the greedy seed selection. Our IP-core performed ICM with an input of a set of vertices as seed nodes. The greedy algorithm was implemented on PS, and continuously sends new sets of seed nodes to the IP-core much like the 4.1.

The address to the matrix, result, frontier, and the random seed and global probability is all mapped as AXI4-Lite while the matrix, frontier, and the result are memory mapped. This allows us to send in the address of the memory location where the variables are stored and apply our algorithm.

Since ICM is dependent on a random function, we ran each set of seed nodes 50 times to find the average runtime and coverage to find the most optimal set of vertices.

## 4.1 IP-core

For this project, we have implemented an IP-core with Vivado HLS. The IP-core applies a modified version of SpMV over an adjacency matrix and a set of vertices

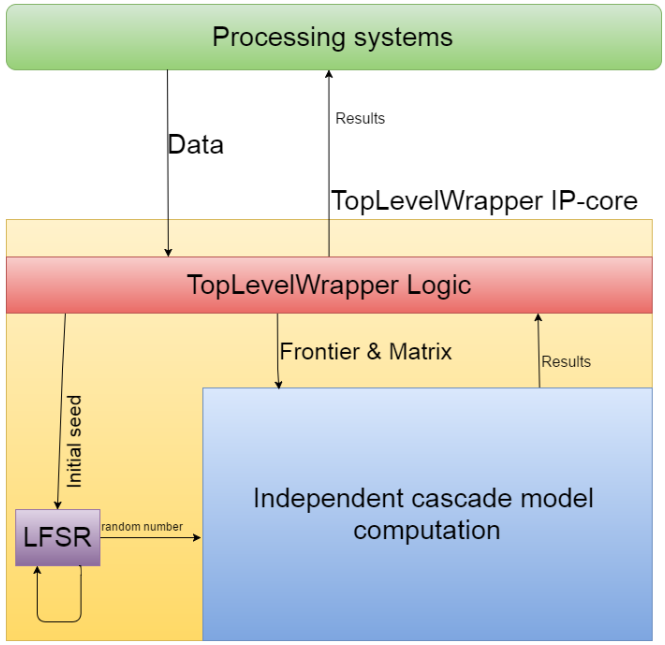



Figure 4.1: Overview of out implementation

known as seed nodes. We will give a brief introduction to HLS programming and how to customize the algorithm for HLS. For our implementation, our IP-core would perform the diffusion and return the activated node. The PS would then calculate the spread of this iteration and supply the IP-core with the next set of seed nodes. This process continues until we have found the best set of seed node with $k$ vertices

## 4.2 Design flow of HLS

For this project, Xilinx Vivado HLS 2016.1 was used. The usual workflow in designing with Vivado HLS is as follow.

1. Define your function/algorithm.
2. Simulate as compile code.
3. Synthesise.
4. Co-simulate.

There are some steps that Vivado HLS requires before the project can start. In the beginning, Vivado HLS would require the designer to specify which function is the top level of the implementation. That top function would determine which port the IP core would have and what type of AXI4 protocol to implement. Vivado HLS also enables the user to specify to which platform this implementation is for.

The first step in designing with Vivado HLS is to define the algorithm that will be synthesised. E.g. for this report, it is the matrix-vector multiplication. After identifying different requirements and dependencies, the algorithm is implemented in C with Vivado HLS. Vivado HLS has some limitation regarding the high-level implementations:

- No dynamic memory (Need to be static), Vivado HLS does not support malloc, free, new or delete.
- NO STD, FILE-IO, etc., (no system calls).
- avoid recursive functions.

The next step is to *run C simulation*. This will verify that the C implementation is correct, by running the test in the testbench. The test in the testbench is created by the designer. After verifying that the implementation is correct, next step is to synthesise the implementation. Vivado HLS will then generate the appropriate Verilog or VHDL, depending on the designer choice. The finished generated solution is then reviewed by reading the Vivado synthesis report. The report containing crucial information regarding the generated solution. There we can find the performance estimates of the generated core, the utilization estimates, and the interface to the generated core. After all, this is done, Vivado offers a *run C/RTL Cosimulation*. Vivado will then run both the C simulation and testbench and the same testbench on a simulated version of the implemented core. This function allows the designer to verify that the generated core have the same behaviour as the simulated C implementation.

After Cosimulation is done, the IP-core is ready for export. *Export RTL* generates the necessary RTL files and exports the IP core. The exported IP-core can then be found in the project folder and is ready to be uploaded to the FPGA.

The input and output port of the IP core is determined by the variables that the top function requires. Variables that are read from will automatically be set as input, while variable that are only written to will be set as output. In order for the core to understand what is mutable, the output is often set as a pointer (For C code). VIVADO HLS generates control signals cl automatically.

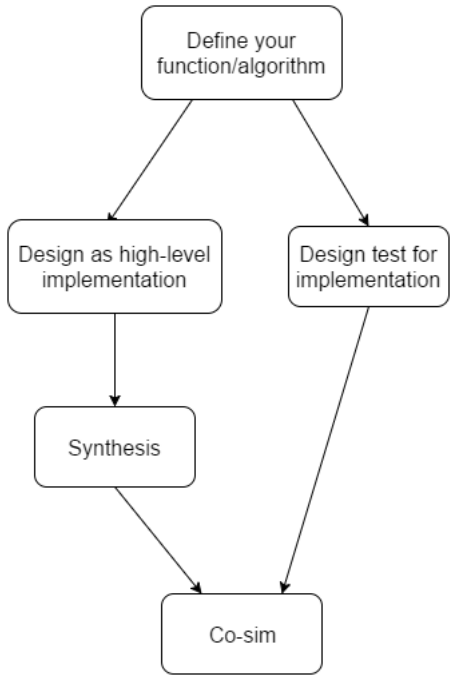



Figure 4.2: Diagram of HLS wokflow

## 4.3 Linear-Feedback Shift Register

The LFSR is a commonly used pseudorandom number generator(pRNG)[39]. Different sized LFRS can generate a wider range of pseudorandom numbers. LFSR generates a pseudorandom number based on the previous pseudorandom number it generated. The LFSR implemented for this project is a 16-bit shift register, that can generate a pseudorandom number in the range from 0-$2^{16}$(65536). To generate the new number, the bits from positions 16,14,13 and 11, is XORed. I.e. (((16 XOR 14) XOR 13 ) XOR 11) = new bit. The new bit is pushed into bit position 1 and the entire registers shifts towards right 1 bit. This allows the IP-core to generate a pseudorandom number based on an initial seed input. From Figure: 4.3 we can see that to generate continuously pseudorandom numbers the output from the LFSR is used as the next iterations seed.

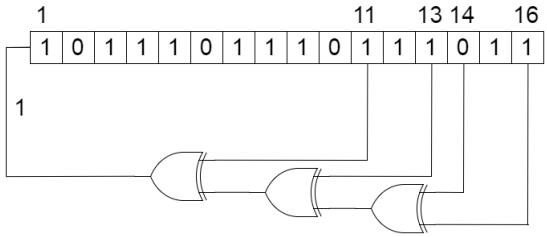



Figure 4.3: 16-bit Linear-Feedback Shift Register

## 4.4 Network and graph generator

The graphs that were used in this report was generated by an R-mat generator. For this project, we choose to implement an R-mat generator as mentioned in Chapter: 2.8 and [1]. The generator is implemented in *Python* [40] with *numpy*. The generator generates adjacency matrix with $2^k$ vertices, where $k$ is known as the scale of the sparse matrix. The total amount of edges the graph contains is set as $total\_amount\_of\_edge = k \times \ edge\_factor$. The edge factor is the ratio between the graph's edge count and ins vertex count[41].

## 4.5 How is it connected

As shown in Figure 4.4, out IP-core is the *TopLevelWrapper*, and the PS is the Zynq processing system. The AXI interconnect serves as a interconnect between our IP-core and the PS.

Our core is memory mapped and we use the bus interface AXI4-Lite for communication. The core includes a DMA-component, which the HLS would initiate for us. This allows us to utilize the High-performance slave interface on our PS.

## 4.6 Xilinx SDK

In Xilinx vivado, a Xilinx Software Development Kit was provided. In the SDK, the IP-core was initiated, and control signals were set. For our experiment, the seed selection was implemented in Xilinx SDK. in this project; a greedy seed selection was implemented. In our PS, we generate a vertex

## 4.7 Optimization

The function matrix_vector_multiplication() performs a single matrix-vector multiplication. From the pseudocode, we can see that there is room for parallelization of the SpMV. The outer for loop from the pseudocode can be parallelized since the for loop is not dependent on the variables from the inner for-loop.

Another possible parallelization is during the simulation, after the SpMV, the frontier vector needs to be calculated. And a converged() function is called in the end to determine if the simulation is finished. The frontier calculation and converged can be run in parallel.

Include the different directives, PIPELINE, LOOP UNROLL and DEPENDENCE. Directives are a predefined optimization that HLS provides, by incorporate them in the high-level implementation, HLS would apply specific optimization. For our implementation, the following directive was added.

- **PIPELINE** Allows the implementation to utilize concurrent execution of operations [42]. This allows loops and functions to execute in parallel.
- **UNROLL** generates multiple independent operations instead of sequential operation such as for-loops.[42]
- **DEPENDENCE** Provides additional information in regarding loop-carry dependency. Allows loops too be pipelined.[42]

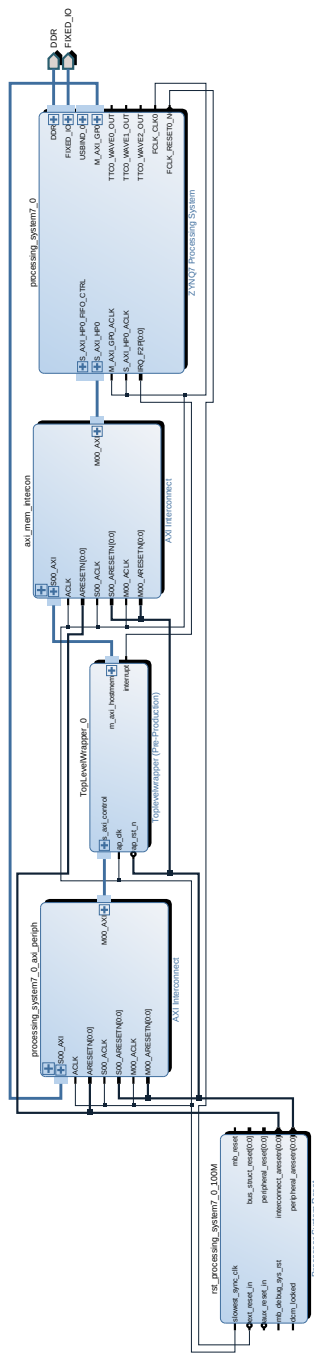



Figure 4.4: Our IP-core and how it is connected.

# Chapter 5

# Result and Discussion

Here we will present the results that we received from the experiments. The algorithm was able to finish an extreme scaled down version of the original sparse matrix-vector multiplication. The result was a 16 $\times$ 16 adjacency matrix, which is unfortunate since our goal was to iterate over large graphs. But these results are still interesting and proves that HLS can be an improvement over traditional hardware design.

This resulted in that most of the numbers here are taken from Vivado HLS simulation and Cosimulation.

## 5.1 Experimental Setup

For this experiment, we used a 16×16 adjacency matrix generated with a R_mat generator. A adjacency matrix with the following variable was created for this experiment.

- **A** = 0.57
- **B** = 0.19
- **C** = 0.19
- **D** = 1-0.57-0.19-0.19 = 0.05
- **Edge factor** = 16
- **k (Scale)** = 4

For this experiment, we set a global probability for activation at 5% and an initial seed of 42 for the linear feedback shift register. We ran each information

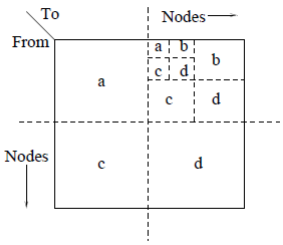



Figure 5.1: The R-mat model [1]

diffusion iteration 50 times and found the average coverage and time usage for each run; each run used a new initial seed, thus each run is different.

Our implementation was a greedy solution, where the PS generated a new set of seeds.

The processing system (PS) sends the matrix and frontier to the Processing logic (PL). The

The implementation was done on a Zedboard [43], with a ZYNQ7 Processing system. The adjacency matrix was stored in the 4GB SD card. The Zedboard was using a 15000000 JTAG Clock Frequency.

## 5.2 Results

In Table 5.3, we can see that by varying the buffer size, we get variable results. We can see that in C simulation, by using a larger buffer so that the IP core can take the entire row of the matrix, the result is somewhat better than the other. While for the RTL implementation, there does not seem to be the case. For the RTL simulation, by choosing buffer size equal to eight, get a better time.

For table 5.1, the first column shows the how many vertices were in the seed nodes, the second column displays the coverage. Unsurprisingly, the more seed nodes that were chosen, the larger was the coverage. For table 5.1, the measurements are in microseconds. The time usage(diffusion) column shows us how much time was spent on an average ICM iteration, while the last column shows how much time each seed selection took.

For Table 5.2, it's the same experiments in software C simulation. The results are measured in seconds.

from these results, we can clearly see that the RTL simulation is faster then the C simulation.

Table 5.1: Results from Zedboard, different sized seed nodes.

| k | coverage | Time usage(diffusion)in microseconds | Seed selection time usage in micro sec |
|---|---|---|---|
| 1 | 0.198750 | 32.74 us | 0.45 us |
| 2 | 0.247500 | 936.88 us | 0.87 us |
| 3 | 0.306250 | 1237.62 us | 1.26 us |
| 4 | 0.391250 | 1501.96 us | 1.63 us |
| 5 | 0.460000 | 1936.74 us | 1.97 us |

Table 5.2: Results from C simulation, different sized seed nodes.

| k | coverage | Time usage(diffusion) | Time usage(seed) |
|---|---|---|---|
| 1 | 0.222500 | 0.000680 s | 0.472000 s |
| 2 | 0.270000 | 0.000900 s | 1.209000 s |
| 3 | 0.307500 | 0.004400 s | 10.443000s |
| 4 | 0.368750 | 0.005720 s | 13.901000 s |
| 5 | 0.482500 | 0.004560 s | 16.497999 s |

## 5.3 Performance

As we can see, the hardware implementation is better than the C-simulated implementation, even at this low scale, we can see that the

We have here included a snapshot of our synthesis report. As we can see, we have a rather large usage of *look-up table* (LUT) and *flip-flop*(FF) for our design. This is the result of the PIPELINE and Loop unrolling. Our high-level implementation does contain several Loops and dependencies; this results in a significant amount of resource used.

By using the random() function provided by Python, we placed '1' in its des-

Table 5.3: Results from simulation with vary buffer size

| Buffersize | Time in milliseconds (with C simulation) | Time in milliseconds (with RTL simulation) |
|---|---|---|
| 4 | 7.200 ms | 0.920 ms |
| 8 | 6.260 ms | 0.680 ms |
| 16 | 5.180 ms | 0.760 ms |

ignated position. After the matrix was generated, we further applied a diagonal copying as shown in Figure:2.5. This is obligatory to create an undirected graph.

## 5.4 Discussion

### 5.4.1 Results

From our results its hard to say if HLS was an improvement or not. Since we worked on a network of such small scale, it is hard to say anything concrete regarding the effectiveness to HLS and our implementation. From the simulated data, we can see that C simulation is much slower than the hardware implementation. One reason that we got odd numbers might be that the testbench program that the C simulation used were slightly different than the FPGA.

But the result from Table 5.2 is interesting. We can see that the seed selection algorithm was severely worse then the HLS implementation on FPGA.

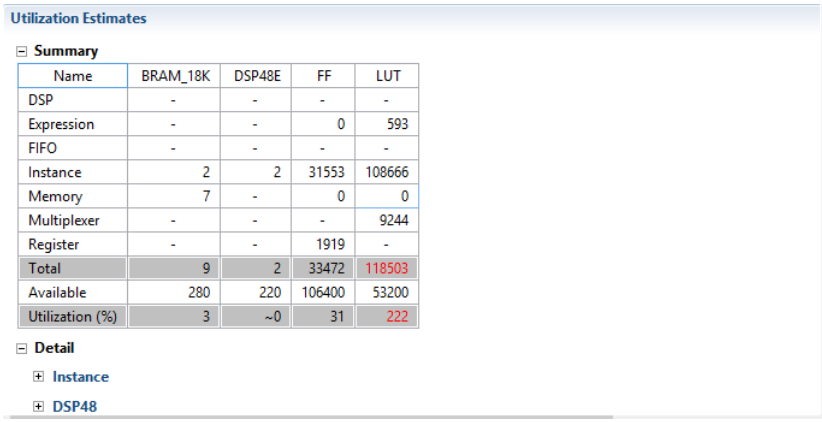
### Analysis of the performance

Our IP core was originally designed for networks with 1024 vertices, but after implementing with the optimization directives, the resource usage was massive. This resulted in a severe scale down for this project. The resulted network was reduced down to the size of 16 vertices. In network analysis, this is such a small scale that the results from this experiment would not be reliable. Since in such a small graph there might be multiple different factors can affect the results.

The original idea with table 5.3 was to observe how the IP-core behaved when it would receive part of the matrix row. By finding an optimal buffer size that would satiate the bandwidth and still receive enough elements to compute. The result we got was not very representative. Since our implementation and the matrix we used was so small compared to other networks, we would not have been close to satiate the bandwidth.

In Figure 5.2, we can see that by synthesis an IP-core with several 1024 buffers, the resource estimate would be 222% of the available LUT. This is due to our implementation have several large for and while loops. By including PIPELINE directive and UNROLL directive, results in significantly larger usage of resources then a core with a smaller buffer.

Figure 5.3 is the synthesis report from our implementation with buffer size 16. The resource usage is much more realistic to what we can implement compared to Figure 5.2.



Utilization Estimates

Summary

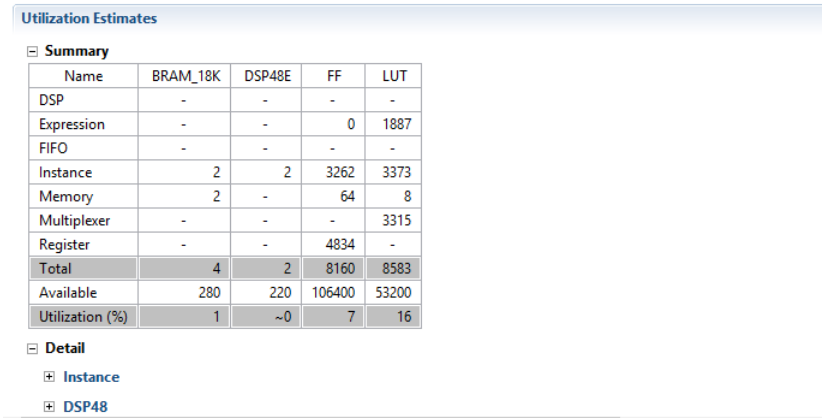
| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 0 | 593 |
| FIFO | - | - | - | - |
| Instance | 2 | 2 | 31553 | 108666 |
| Memory | 7 | - | 0 | 0 |
| Multiplexer | - | - | - | 9244 |
| Register | - | - | 1919 | - |
| Total | 9 | 2 | 33472 | 118503 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 3 | ~0 | 31 | 222 |

Detail

Instance

DSP48

Figure 5.2: Synthesis report for IP-core with buffersize 1024



Utilization Estimates

Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | 0 | 1887 |
| FIFO | - | - | - | - |
| Instance | 2 | 2 | 3262 | 3373 |
| Memory | 2 | - | 64 | 8 |
| Multiplexer | - | - | - | 3315 |
| Register | - | - | 4834 | - |
| Total | 4 | 2 | 8160 | 8583 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 1 | ~0 | 7 | 16 |

Detail

Instance

DSP48

Figure 5.3: Synthesis report for IP-core with buffer size 16

## problem that was encountered

One problem that was encountered during this project was that the output signal from the synthesiser was not the correct direction. The output signal was often set as an input signal. The HLS would automatically set the values as output signal or input signal.The return value from a function would be set as an output signal, while the variable that the function takes would be set as the input signal. Another way to specify that something is the output signal would be to explicitly set them as pointer arguments. This will in set the signal to be the output signal.

Another problem that I often encountered was that the Vivado HLS often stop working. The problem was fixed by creating a new project and include the previous files.

Sometimes Vivado HLS was not able to Cosimulate the implementation the first time. I was not able to find a solution to this issue except resynthesis the project, which usually works.

There was an incidence where the implementation on the Zedboard did not behave as the implementation. This was solved by reprogramming the device and sometimes, restarting the Zedboard.

# Chapter 6

# Future work

Information diffusion and seed selection, in general, computes large graphs, and thus is very time-consuming. There is, therefore, several improvements that have yet to be explored.

1. **Different architecture** for this implementation, we used a core including the LFSR; it would be interesting to explore different design. One solution that we did not have the chance to explore is implemented a large buffer connected to a single LFSR that continuously generates a random number. The implemented cores would be then each pop one random number for each coin toss. This solution requires a large buffer and would potentially generate large overhead with reading from the buffer. A large enough buffer would also be required since there are in worst case scenario for a single SpMV run, we would need $n^2$ coin toss. The potential benefits of such a design would be better space utilization. A smaller core would use less resource of the Zedboard. This can result in more parallelization.

2. **Use larger graph** In graph theory, graph used is often at scale(26-42). The smallest mentioned graph from Graph500, is $2^{26}$, a toy graph. Our graph is not even close to such a large graph. Testing this architecture up against a larger graph would be beneficial.

3. **Customize algorithm** As mentioned in Chapter 3, HLS can generate a close to hand written design if the algorithm is customized for HLS. An interesting potential improvement would be to analyse algorithm and explore different solutions and implementation.

4. **Compare different solutions** In this report, we have only shown the result from one architecture, it would be interesting to compare different

schemes and other solutions.

5. **Memory Optimization**. For this algorithm, we store the entire adjacency matrix. This is inefficient for a sparse matrix. A potential improvement would be to explore a different storage format for the adjacency matrix.

6. **Try different seed selection algorithm** In this report, we implemented the greedy seed selection. It would have been interesting to compare different seed selection algorithm and compare the results. There are papers [15] that proposes an alternative greedy solution.

7. **Better optimization with HLS** would result in potentially better utilization of the resources on the Zedboard. Our design was not the most optimal and could only compute small graphs. One future aspect would be to optimize for much larger networks.

8. **A general IP core** was implemented, but due to time constraints, were not used for this project. The general IP core had a fixed buffer size, but could accept matrix and vector of a larger size. The IP-core would store nodes in the buffer, compute the SPMV, and then send the result back. This solution resulted in massive overhead with transporting data back and forth between PL and PS. This core was not used due to time constraints.

# Chapter 7

# Conclusion

In this report, we have explored the possibility to use HLS as an implementation tool to accelerate seed selection for information diffusion. We used a modified version of sparse matrix-vector multiplication to perform ICM iterations. We have implemented a simple IP-core that applies ICM with an adjacency matrix and a set of seed nodes. The core returns the percentage of infected nodes in the network. The design was loaded onto an FPGA and multiple tests were conducted both on the chip and in the simulation. From the result we saw that the implementation done on the FPGA was much faster then the C simulation, but due to the small size of our adjacency matrix, it is hard to conclude that HLS generates better solutions. Even though results were not optimal, the development time was greatly reduced by Vivado HLS.

In this project, we were not able to synthesise an IP-core that could compute large matrices. By synthesising an IP-core with s larger buffer, the resource usage was much higher than what was available on our Zedboard. This is likely the result of lack of customization of the high-level implementation.

**Task 1** ***(mandatory)*** Implement information diffusion as sparse matrix-vector multiplication, with high level language C. Completed

W have in this project implemented a modified sparse matrix-vector multiplication that perform ICM. The implementation detail can be found in Chapter 4, and the theory can be found at 2

**Task 2** ***(mandatory)*** Tailor the implementation of Information Diffusion for synthesise with Vivado HLS. Complete

The design was able to synthesise in Xilinx Vivado HLS and gave back some-

what interesting results. The detail about the HLS and different optimization used can be found at chapter 4.

**Task 3** ***(optional)*** Implement said design on a Zynq FPGA board.

Out IP-core was able to upload on the FPGA and gave back the result. Chapter 5 presents the result from the FPGA and other

**Task 4** ***(optional)*** Extend the system to be able to handle graph in the size of toy graphs(containing $2^{26}$) vertices)

Due to unfamiliarity with HLS and time constraints. We were not able to extend the system to compute larger graph.

# Bibliography


[1] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. 4:442–446, 2004.

[2] Daniel Gruhl, R. Guha, David Liben-Nowell, and Andrew Tomkins. Information diffusion through blogspace. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 491–501, New York, NY, USA, 2004. ACM.

[3] Daniel M. Romero, Brendan Meeder, and Jon Kleinberg. Differences in the mechanics of information diffusion across topics: Idioms, political hashtags, and complex contagion on twitter. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 695–704, New York, NY, USA, 2011. ACM.

[4] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and P Krishna Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 10(10-17):30, 2010.

[5] J. Yang and J. Leskovec. Modeling information diffusion in implicit networks. In *2010 IEEE International Conference on Data Mining*, pages 599–608, Dec 2010.

[6] Seth A. Myers, Chenguang Zhu, and Jure Leskovec. Information diffusion and external influence in networks. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 33–41, New York, NY, USA, 2012. ACM.

[7] David Kempe, Jon Kleinberg, and ÃĽva Tardos. Influential nodes in a diffusion model for social networks. 3580:1127–1138, 2005.

[8] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, Feb 1990.

[9] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[10] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. pages 1–8, Sept 2015.

[11] Éva Tardos David Kampe, Jon Klein. Maximizing the spread of influence through a social network. pages 137–146, 2003.

[12] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[13] MH McAndrew. On the product of directed graphs. *Proceedings of the American Mathematical Society*, 14(4):600–606, 1963.

[14] Pedro Domingos and Matt Richardson. Mining the network value of customers. pages 57–66, 2001.

[15] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 199–208, New York, NY, USA, 2009. ACM.

[16] M. E. J. Newman. *The Structure and Function of Complex Networks*, volume 45. 2003.

[17] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[18] Ritchie Zhao, Shreesha Srinath Gai Liu, Christopher Batten, and Zhiru Zhang. Improving high-level synthesis with decoupled data structure optimization. In *Proceedings of the 53rd Annual Design Automation Conference*, page 137. ACM, 2016.

[19] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.

[20] XIlinx. Zynq-7000 All Programmable SoC release notes, installation,and licensing,ug585 (v1.10), 2015.

[21] Xilinx. Zynq-7000 All Programmable SoC Overview, 2014.

[22] Kate Starbird and Leysia Palen. (how) will the revolution be retweeted?: Information diffusion and the 2011 egyptian uprising. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 7–16, New York, NY, USA, 2012. ACM.

[23] Eytan Adar and Lada A. Adamic. Tracking information epidemics in blogspace. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '05, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Manuel Gomez Rodriguez, Jure Leskovec, and Andreas Krause. Inferring networks of diffusion and influence. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[25] Eytan Bakshy, Jake M. Hofman, Winter A. Mason, and Duncan J. Watts. Everyone's an influencer: Quantifying influence on twitter. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 65–74, New York, NY, USA, 2011. ACM.

[26] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design &amp Test of Computers*, 26(4):18–25, 2009.

[27] S. Director, A. Parker, D. Siewiorek, and D. Thomas. A design methodology and computer aids for digital vlsi systems. *IEEE Transactions on Circuits and Systems*, 28(7):634–645, Jul 1981.

[28] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The cmu design automation system: An example of automated data path design. In *Proceedings of the 16th Design Automation Conference*, DAC '79, pages 73–80, Piscataway, NJ, USA, 1979. IEEE Press.

[29] John Granacki, David Knapp, and Alice Parker. The adam advanced design automation system: Overview, planner and natural language interface. In *Proceedings of the 22Nd ACM/IEEE Design Automation Conference*, DAC '85, pages 727–730, Piscataway, NJ, USA, 1985. IEEE Press.

[30] P. G. Paulin, J. P. Knight, and E. F. Girczyc. Hal: A multi-paradigm approach to automatic data path synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, DAC '86, pages 263–270, Piscataway, NJ, USA, 1986. IEEE Press.

[31] H. D. Man, J. Rabaey, P. Six, and L. Claesen. Cathedral-ii: A silicon compiler for digital signal processing. *IEEE Design Test of Computers*, 3(6):13–25, Dec 1986.

[32] D. Navarro, ÃŞ LucÄśÂťa, L. A. BarragÃąn, I. Urriza, and ÃŞ JimÃĺnez. High-level synthesis for accelerating the fpga implementation of computationally demanding control algorithms for power converters. *IEEE Transactions on Industrial Informatics*, 9(3):1371–1379, Aug 2013.

[33] XIlinx. Vivado Design Suite User Guide release notes, installation, and licensing,ug973 (v2015.4), 2015.

[34] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using vivado hls. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 362–365, Dec 2013.

[35] Gorker Alp Malazgirt, Nehir Sonmez, Arda Yurdakul, Adrian Cristal, and Osman Unsal. High level synthesis based hardware accelerator design for processing sql queries. In *Proceedings of the 12th FPGAworld Conference 2015*, FPGAworld '15, pages 27–32, New York, NY, USA, 2015. ACM.

[36] Donald G. Bailey. The advantages and limitations of high level synthesis for fpga based image processing. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, ICDSC '15, pages 134–139, New York, NY, USA, 2015. ACM.

[37] Shahzad Ahmad Butt, Mehdi Roozmeh, and Luciano Lavagno. Designing parameterizable hardware ips in a model-based design environment for high-level synthesis. *ACM Trans. Embed. Comput. Syst.*, 15(2):32:1–32:28, February 2016.

[38] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[39] M. Murase. Linear feedback shift register, February 18 1992. US Patent 5,090,035.

[40] Anaconda. Anaconda python 2.7, 2015.

[41] Graph 500 Steering Committee. Graph 500 Benchmark 1 ("search"), 2010.

[42] Xilinx. Vivado design suite user guide high-level synthesis.

[43] Zedboard.org. Zedboard technical specifications.