EFFICIENT HLS-BASED IMPLEMENTATION OF SPARSE MATRIX-VECTOR MULTIPLICATION ON FPGA

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Mert Kara

December 2021

EFFICIENT HLS-BASED IMPLEMENTATION OF SPARSE MATRIX-VECTOR MULTIPLICATION ON FPGA
By Mert Kara
December 2021

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

M. Mustafa Özdal(Advisor)

Özcan Öztürk

İ. Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan [redacted]
Director of the Graduate School

ABSTRACT

EFFICIENT HLS-BASED IMPLEMENTATION OF SPARSE MATRIX-VECTOR MULTIPLICATION ON FPGA

Mert Kara
M.S. in Computer Engineering
Advisor: M. Mustafa Özdal
December 2021

Sparse Matrix-Vector Multiplication (SpMV) is an important core kernel used in many scientific applications. SpMV is a communication-bound algorithm that suffers poorly from spatial locality. It exhibits low computation-to-communication ratio due to its inherent irregular memory access patterns. This causes a significant waste of DRAM traffic and poor bandwidth utilization. Recently published Propagation Blocking (PB) methodology tackles this communication bottleneck by dividing the execution into binning and accumulation phases, allowing better locality in the cost of additional memory accesses. Building upon PB approach, in this study, we design two FPGA kernels for binning and accumulation phases using high-level synthesis, run together sequentially. Experimental results and projections on larger data show that our design can provide up to 7.9x speedup over the CPU baseline implementation.

Keywords: FPGA, Accelerator, SpMV, HLS.

ÖZET

FPGA ÜZERİNDE ŞEYREK MATRİS-VEKTÖR ÇARPIMININ VERİMLİ HLS-TABANLI UYGULAMASI

Mert Kara
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: M. Mustafa Özdal
Aralık 2021

Seyrek Matris-Vektör Çarpımı (SpMV), birçok bilimsel uygulamada kullanılan önemli bir çekirdek algoritmadır. SpMV, uzamsal yerellikten çok düşük oranda faydalanabilen, iletişime bağlı bir algoritmadır. Düzensiz bellek erişim kalıpları nedeniyle düşük hesaplama-iletişim oranı sergiler. Bu, DRAM trafiğinin önemli miktarda etkili kullanılamamsına ve zayıf bant genişliğine neden olur. Yakın zamanda yayınlanan Yayılma Bloklama (PB) metodolojisi, SpMV algoritmasını gruplama ve toplama şeklinde iki aşamada gerçekleştirerek bu iletişim darboğazının üstesinden gelir ve ek bellek erişimi ile daha iyi yerellik sağlar. PB yaklaşımına dayanarak, bu çalışmada, sıralı olarak birlikte çalışan üst düzey sentez kullanarak gruplama ve biriktirme aşamaları için iki FPGA çekirdeği tasarladık. Yaptığımız deneyler ve projeksiyonlar, tasarımımızın CPU temel uygulaması üzerinde 7,9 kata kadar hızlanma sağlayabileceğini gösteriyor.

Anahtar sözcükler: FPGA , Hızlandırıcı, SpMV, HLS .

Acknowledgement

I am very grateful to my advisor and the jury members for the enormous patience, tolerance, and support they have provided. I am well aware that I pushed my advisor's patience to the fullest, and I cannot thank him enough for his understanding.

I also owe a huge debt of gratitude to my friends, with a special thanks to Elmira, who did everything in her power to help me get back on my feet at times I felt burned out. I tried to test her patience as well, but she was always willing to support me.

Finally, I would like to thank my parents and extended family for all the support and love they have given me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sparse matrix-vector multiplication(SpMV) is a fundamental computational kernel for many scientific applications. It is primarily used in iterative linear solvers, where a repeated computation on the resulting vector is performed until convergence. PageRank algorithm, which calculates the importance of vertices in a graph, is a famous application where SpMV exhibits itself as the core computational kernel. SpMV and PageRank are memory communication bound algorithms that suffer from poor spatial and temporal locality.

With the increased availability and size of data, improving the performance of SpMV computation, both in terms of execution time and energy efficiency, has gained importance. In order to improve bandwidth utilization of SpMV, several two-phase algorithms have been proposed. In these algorithms, SpMV is separated into two phases as binning and accumulation, similar to scatter-gather paradigm. In the binning phase, corresponding vector and matrix inputs are multiplied and written into intermediate bin buffers as pairs of destinations and contributions where each bin buffer holds the pairs for a certain partition of the output vector. In the accumulation phase, these pairs are read and contributions are added to their destinations for each output vector partition[1, 2, 3].

Field-Programmable Gate Array (FPGA) has emerged as a promising platform for accelerator development with the growing interest in energy-efficient acceleration.[4, 5, 6, 7]. FPGAs outperform multi-core and GPU systems in performance per watt and have been integrated into data centers to accelerate computation-intensive activities [5, 6]. High-level synthesis(HLS) research has also got its share from the growing popularity of FPGAs. HLS community has seen much activity in the recent years, with a plethora of HLS tool offerings, from both industry and academia. HLS tools automatically produce a circuit specification that performs the function provided with a high-level language such as C/C++, and SystemC. This allows designers to work at a higher-level of abstraction than hardware description languages such as Verilog and VHDL, possibly shortening production time and reducing the required level of hardware expertise. [8].

Although FPGAs are primarily more suitable for applications with low random-access patterns, we observed that recently proposed two-phase SpMV algorithms can allow FPGAs to be used and perform well in communication-intensive applications like SpMV too. In this study, we adapt the two-phase SpMV algorithm to FPGA and design two separate hardware kernels for the binning and accumulation phases. We implement our design using high-level synthesis. The contributions of this work are as follows:

- We provide a hardware kernel for the binning phase of the two-phase SpMV algorithm.
- We provide a hardware kernel for the accumulation phase of the two-phase SpMV algorithm.
- We compare the performance of our design to a CPU baseline and provide theoretical and experimental evaluation results.

The rest of the document is organized in six chapters. Second chapter is devoted to problem description for SpMV and goes through the details of two-phase algorithm. Third chapter provides background information on FPGA accelerator development and high-level synthesis. Chapter four presents our proposed hardware design for SpMV computation. Fifth chapter provides theoretical and experimental evaluations of our design, and chapter five concludes the thesis.

Chapter 2

Problem Description

This chapter begins with a brief overview of sparse matrix-vector multiplication. The computational inefficiencies of SpMV are next explored in terms of various possible implementations. Finally, information on the propagation blocking, which is a two-phase SpMV algorithm that utilizes SpMV performance is provided.

2.1 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication is a fundamental computational kernel used in a variety of scientific applications. SpMV can be expressed mathematically as $y = Ax$, where A is the sparse matrix, x is the dense vector, and y is the multiplication result. SpMV can be computed either with a row-major matrix traversal or a column-major matrix traversal. In graph applications, these correspond to processing in pull direction and push direction, respectively. Pull direction traverses the incoming edges of vertices, whereas push direction traverses the out-going edges. Both implementations are explored further with pseudo-codes provided in Listings 2.1 and 2.2. Through the text, graph terminology is sometimes preferred over linear algebra terminology to ease explanation.

2.1.1 SpMV in Pull Direction

Listing 2.1: SpMV Multiplication with Row-Major Traversal Algorithm

```
Inputs: Matrix(Nonzeros(Weight, Column), Offset), Vector[]
Output: Result[]

for row index r from 0 to len(Result):
  sum = 0
  for nonzero p in Nonzeros[Offset[r]:Offset[r+1]]:
    sum += p.Weight * Vector[ p.Column ]
  Result[r] = sum
```

The row-major traversal(pull direction) computation iterates through the output vector in the outer loop, and so the output vector is accessed continuously, benefiting from high spatial locality. We assume that an optimized sparse matrix format is used, which in this case can be compressed sparse row(CSR) format. In CSR format, the input matrix is stored in three vectors. ”Weight” vector stores the non-zero values, ”Column” vector stores the column indices for the corresponding non-zero values and ”Offset” vector stores the row start indices in ”Weights.” All these vectors are also accessed contiguously, so the inner loop traversal on the input matrix also benefits from high spatial locality. However, the column index ”p.Column” can refer to any column in the matrix, and thus the input vector is accessed randomly, suffering from low spatial locality. Figure 2.1 visualises the computation.

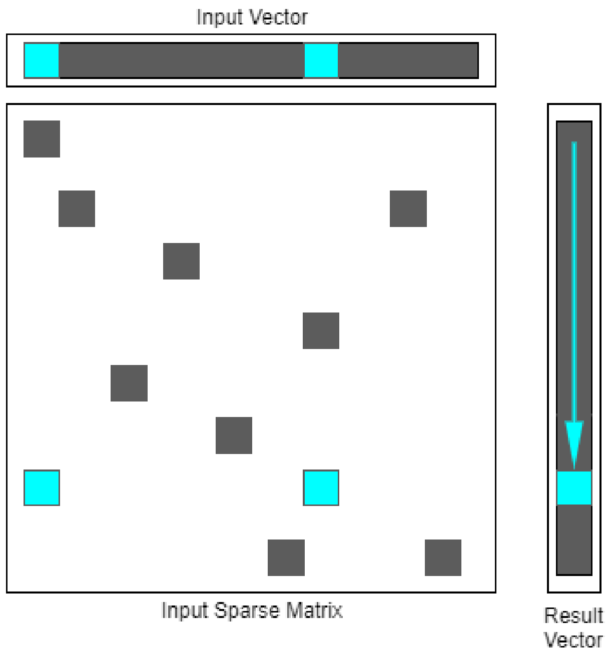

Figure 2.1: SpMV Multiplication with Row-Major Traversal (Pull Direction)

2.1.2 SpMV in Push Direction

Listing 2.2 includes the pseudo-code for SpMV computation in push direction.

Listing 2.2: SpMV with Column-Major Traversal Algorithm

```
Inputs: Matrix(Non-Zeros(Weight, Row), Offsets), V[]
Output: Result[]

Result[0..V] = 0
for column index c from 0 to V.len do:
  for non-zero p in Non-Zeros[Offset[c]:Offset[c+1]]:
    Result[p.Row] += p.Weight * Vector[c]
```

Unlike pull direction, the outer loop iterates on the input vector this time. This allows the read accesses to the input vector to occur sequentially, benefiting from high spatial locality. However, write accesses to the output vector occur in a random fashion with poor spatial locality this time, since the row index ”p.Row” can be anything. As in pull direction, an optimized matrix format that stores the data in column-major format can be used, so that the read accesses on the input matrix will refer to contiguous locations. Figure 2.2 illustrates the computation of SpMV in column-major traversal (push direction).

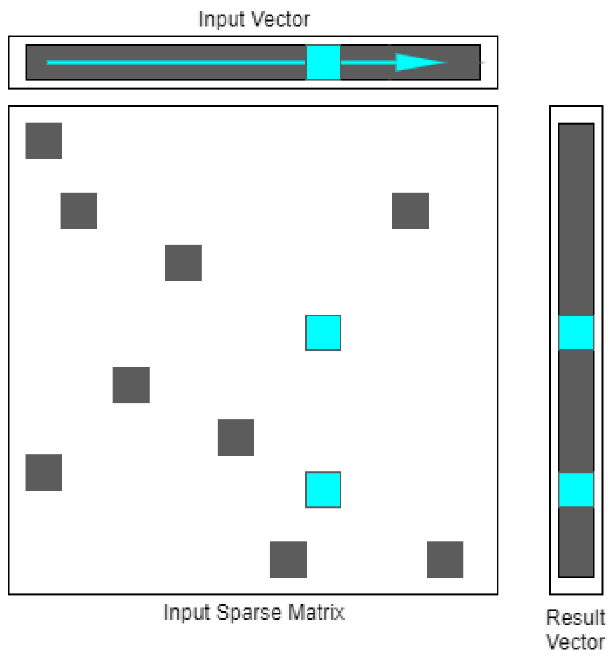

Figure 2.2: SpMV with Column-Major Traversal Illustration

In both algorithms, either input or output vector is accessed randomly. There is no way to have contiguous accesses on both vectors at the same time. This increases the amount of memory communication, causing random memory accesses to become the bottleneck.

2.2 Accelerating SpMV

Many studies have been conducted to alleviate the random-access bottleneck of SpMV, proposing various preprocessing techniques, algorithmic optimizations, and hardware accelerators. Preprocessing techniques try to provide an optimized layout with increased locality through manipulating the input data. However, it is not always easy to partition or reorder real-world graphs as they are mostly irregular and scale-free, and the cost of preprocessing itself can be high. Algorithmic optimizations for SpMV usually focus on amortizing the newly added complexity with better locality. One important example is cache blocking. In cache blocking, input or the output vector is partitioned such that each partition will ideally fit in cache, and read or write accesses for other partitions will be blocked to avoid cache misses. Cache blocking can provide increased performance, but this improvement depends on the sparsity of the graph. [2].

Many graph processing frameworks have been proposed to take advantage of different hardware.[9, 10, 11, 12, 13]. GraphChi [14], X-Stream [15], GridGraph [16], and GraphMat [17] are representative examples based on multi-core general-purpose processors, whereas Gunrock [18], nvGRAPH [19], and CuSha [20] are examples based on general-purpose graphics processing unit. General-purpose CPUs have several inefficiencies, such as ineffective on-chip memory and expensive atomic operations [21, 22, 23, 24, 25]. Specialized graph processing hardware accelerators have garnered considerable attention to overcome these inefficiencies. [26, 27, 28, 29, 22, 23, 24, 30, 25, 31, 32]. As the number of user-controllable on-chip memory resources and dense programmable logic components increases, FPGAs are becoming a promising solution in overcoming the inherent inefficiencies of general-purpose processors [33, 34, 32, 35].

2.3 Propagation Blocking

To increase locality and alleviate random memory access bottlenecks, multiple groups have suggested Two-Phase algorithms with varying optimizations in [3], [1], and [36]. Because the core premises of all three publications are similar, we will refer to Beamer et al. in this section. Listing 2.3 contains the algorithm's pseudo-code. The code is provided for the implementation of PageRank, as it was in the original paper, but it can be adapted to general SpMV easily. The algorithm's concept is similar to cache blocking; however, instead of partitioning the input graph, propagation blocking partitions the data transfers to the output vector. The flow of the propagation blocking is depicted in Figure 2.3.

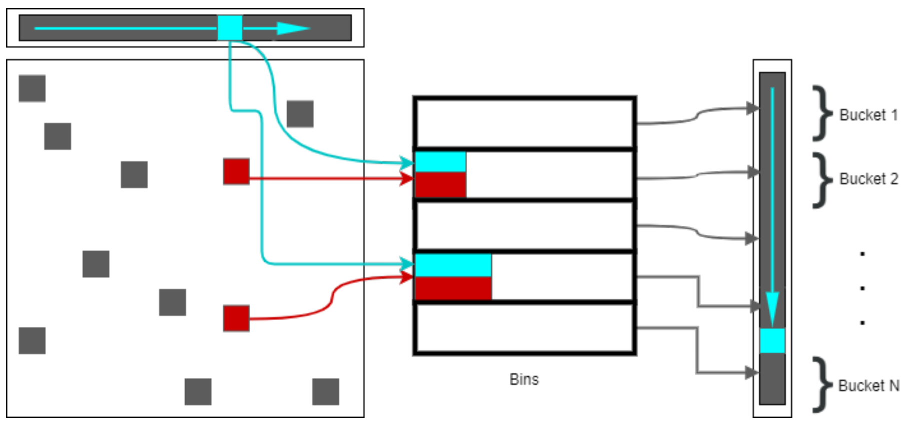

Figure 2.3: Propogation Blocking Flow

The first phase of the algorithm, called binning, iterates over the input vector and calculates the contributions in the dot-product; however instead of adding these contributions directly to the result vector, these contributions are paired with their destination addresses and written into intermediate binning buffers. The vertex set is simply divided into B groups, where B is the number of buckets. Hence, pairs are written to their corresponding buffer which is decided with the mapping function $Bid(v) = \frac{v}{B}$, where $Bid(v)$ is the index of the buffer that vertex v is assigned to. In the second phase of the algorithm, for each bucket, pairs are read from the corresponding bin buffer sequentially and contributions are added into the corresponding sums defined by the destination element of the pair. The number of buckets is picked such that the output vector partition and number of insertion points of binning buffers would fit in cache so that accesses on the input, output and the binning buffers all can benefit from high locality, in the cost of doubling the number of read/write accesses.

Listing 2.3: SpMV with Propagation Blocking

```
Input: G=(V,E), B (number of buffers)
Output: Rank[V]
sums[0..V] = 0
buffers[0..B]

# Binning Phase
for each vertex v in V:
  contribution = Rank[V] / dout(v)
  for each outgoing neighbor u of vertex v:
    append (contribution, v) to buffers[v / B]

# Accumulation Phase
for each i in 0..B :
  for each (contribution, v) in buffers[i]:
    sums[v] += contribution

for each vertex v in V:
  Rank[v] = (1 - d) / |V| + d * sums[v]
```

A significant performance gain for sparser graphs has been demonstrated by the propagation blocking technique, as reported in [36]. Although the number of memory accesses doubles due to writing into and reading from intermediate bins, this can be amortized if the matrix is sparse enough, as both read and write accesses to the input/output vectors occurs sequentially, allowing improved spatial locality.

Propagation blocking is a scatter-gather approach and is particularly suitable to adopt for an FPGA implementation. Since FPGAs do not have a transparent cache hierarchy similar to CPUs, random memory accesses can cost much more. Hence, FPGAs are usually preferred for compute-intensive applications with streaming data. As propagation blocking allows input/output accesses to be made in a streaming fashion, there is an opportunity to allocate most of the limited on-chip memory to intermediate binning buffers.

Chapter 3

Background

In this chapter preliminary information on FPGA, high-level synthesis and common optimization techniques are provided.

3.1 FPGA

A field-programmable gate array (FPGA) is a device that can be reprogrammed to realize different circuits. FPGAs are built as an array of configurable logic blocks (CLB) connected through programmable routing channels. They can also integrate other useful devices such as high-speed input and output elements, processors, and memory units.

Three key elements of a configurable logic block are lookup tables (LUTs), registers, and multiplexers. An FPGA is programmed by loading the appropriate values into LUTs and specifying the select inputs of multiplexers to manage routing. The output of a lookup table can be fed into a register to realize sequential logic, or it can be directly connected to the CLB output to realize combinational logic.

In the traditional way of FPGA programming, the desired design is provided to a synthesis tool with a hardware description language (HDL) code. The synthesis tool then generates a bitstream file for the target FPGA by compiling this code. The bitstream file includes all the information on how the LUTs, multiplexers, and routing channels should be configured. The bitstream file can then be downloaded to the FPGA to program it.

3.2 High-Level Synthesis

High-Level Synthesis (HLS) is another way of design process for FPGA programming. Unlike HDLs, where the design needs to be described and dealt with in a low-level context, high-level synthesis aims to provide an abstraction of most low-level details and provide a development process similar to sequential programming. With HLS programming, the designer provides a behavioral specification of the design. The HLS tool then tries to map and optimize this specification into a low-level RTL design. For example, most of the time, a programmer doesn't need to take care of pipelining as the HLS tool will automatically produce pipelined RTL from loops in the code.

The ultimate goal of HLS is to map sequential software to a high-performance hardware design with no additional coding or experience. However, HLS is still a developing field, and a programmer cannot simply assume that writing sequential code will result in highly optimized designs. There are inherent challenges in mapping sequential software to hardware.

3.3 Optimization Techniques

HLS tools usually provide some pragmas for programmers to guide the compiler on how hardware optimizations should be done. In this section, we provide brief information on pipelining and dataflow optimizations.

3.3.1 Pipelining

A pipeline can be defined as a series of computational stages where the output of each stage is the input for the next one. Incoming instructions are subdivided into different sub-operations which can be executed in different stages. This allows concurrent execution of multiple instructions as different sub-operations of instructions are executed in different stages at the same time.

Latency and initiation interval (II) are two important metrics for defining the performance of a pipeline.

Latency refers to the total number of computational stages of a pipeline. This is the total number of clock cycles necessary to execute an instruction, as each stage takes a single clock cycle to execute.

Initiation interval can be defined as the number of clock cycles necessary to wait until executing the next task in a pipeline. An II of 1 means that the next instruction can be fed directly into the pipeline in the next cycle, whereas in II of 2, the next instruction needs to wait for 2 cycles after the first instruction is fed into the pipeline. A lower initiation interval is desired as it directly increases throughput and provides better concurrency. Other optimization techniques such as loop flattening, loop merging and array partitioning can be used to have more efficient pipelines by allowing lower latency and initiation interval.

3.3.2 Dataflow

Dataflow optimization provides task-level pipelining. It allows a function to start running as soon as the input data is available from the previous function. This provides task-level concurrency and lowers the programmer's effort to synchronize different modules. However, in Vivado HLS, there are several limitations on when dataflow optimization can be used. If there are single-producer-consumer violations, bypassing tasks, feedback between tasks, conditional execution of tasks or loops with multiple exit conditions, Vivado HLS will not be able to perform dataflow optimization on the design.

3.4 Memory Model

Unlike CPUs, FPGAs do not have an automated cache system. However, they may provide different kinds of memories with different read/write speeds and sizes, such as ultraRAM, dynamic RAM (DRAM), and block RAM (BRAM). The programmer is responsible for using different memory resources efficiently.

An FPGA accelerator's performance substantially depends on how the memory resources are put to use. Throughout the text, we will refer to BRAM and DRAM many times. BRAM units are on-site memory blocks on an FPGA. They are limited in size but provide fast access. A single BRAM unit can be accessed in a single cycle. DRAM refers to the global memory, which is relatively cheap and provides significant space. This is the same DDR SDRAM memory that can be found in regular CPU settings. DRAM access latency is huge, and a single access can take hundreds of cycles to access DRAM.

SDAccel Environment uses the OpenCL memory model. Definitions of different memory types in the OpenCL memory model are as follows:

- Host memory refers to the memory that is only accessible to the CPU.
- Global memory refers to the memory that is accessible both to the CPU and FPGA.
- Local memory refers to on-site memory resources on the FPGA and are only accessible to the FPGA.

In the OpenCL memory model, the CPU and FPGA can not access global memory concurrently. The CPU can only access DRAM when the FPGA kernel is not running. In the usual memory flow, the CPU first creates the input and output buffers in global memory and then copies the input data from the host memory to the buffers in global memory, if necessary. When the FPGA kernel starts, the host CPU loses access to the global memory until the kernel finishes running. Only after that can the host CPU access the global memory again to read any output data written by the kernel.

Unlike CPUs, since DRAM is not interfaced through cache on FPGAs, randomly accessing DRAM data can exhibit significant performance loss. Therefore, it is considered better practice to move as much data as possible from global memory to local BRAM blocks in burst transfers at once.

Starting with the 2019.1 release, SDAccel Environment supports direct streaming of data from CPU to FPGA. However, this functionality is not supported in the 2018.3 release, which is the latest one that our license covers.

Chapter 4

Proposed Design

We adapt the propagation blocking methodology in our design and provide two FPGA kernels for binning and accumulation phases. These kernels can run sequentially and not concurrently. It is also possible to use them in heterogeneous designs. A high-level overview of the proposed architecture is given in Figure 4.1.

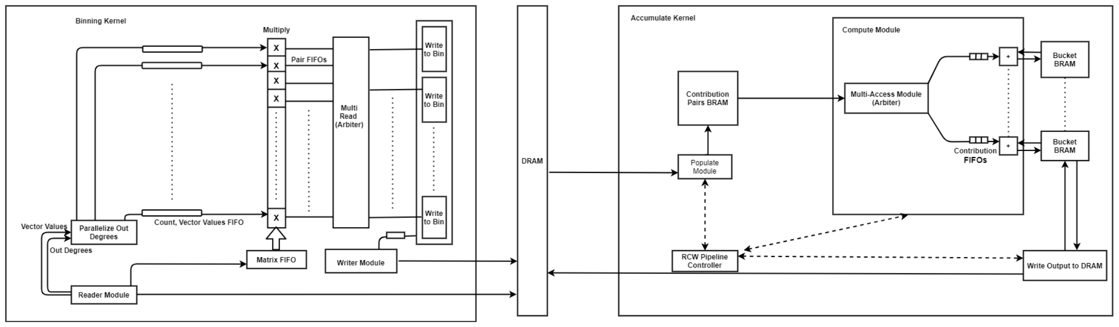

Figure 4.1: Proposed Architecture

4.1 Binning Kernel

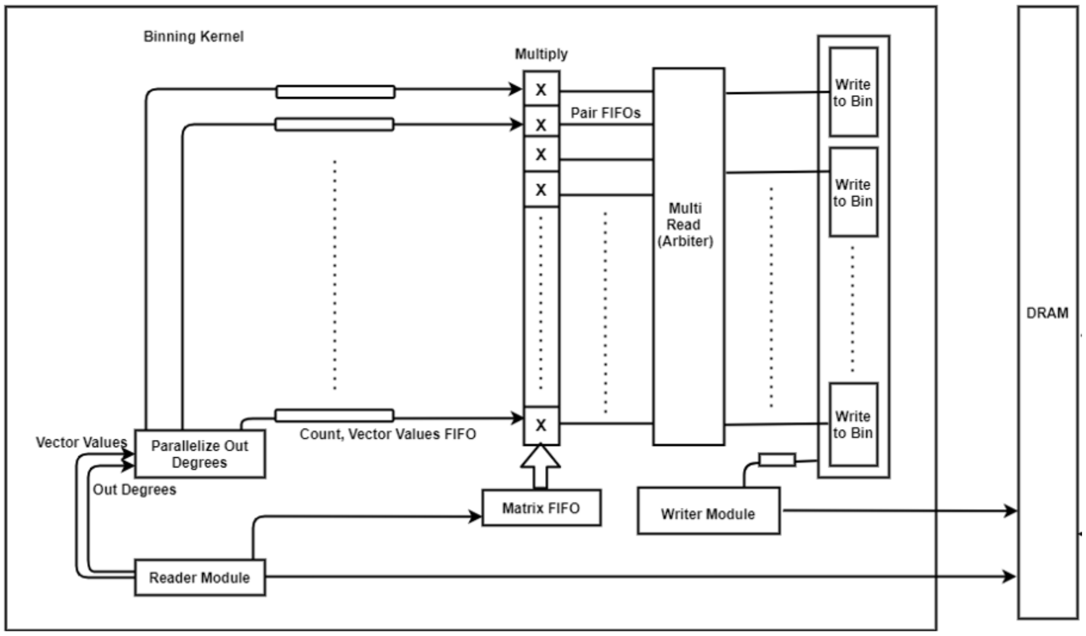

Figure 4.2: Binning Kernel

The design of the binning kernel is given in Figure 1. There are six main components of the binning kernel, and each module communicates through FIFO buffers with hls::stream interface.

The Reader Module is responsible for reading and serializing the input data from DRAM. It has 3 input ports connected to buffers in global memory. Input matrix, input vector, and out-degrees are all read from DRAM into FIFO buffers with burst transfers. All vectors are read in 512-bit lines to utilize communication bandwidth. Each FIFO connects to a serializer sub-module, which is responsible for getting the individual elements from the 512-bit lines. These elements are then sent to the Parallelize Out-Degrees and Multiply modules with FIFO buffers.

In order to allow intra-kernel parallelization, the number of times that the next vector value should be multiplied with the corresponding matrix value is calculated in the "Parallelize Out Degrees" module and fed into FIFOs. This allows processing of a single matrix cacheline per cycle. This number can be increased by replicating the multiplier units, but since the main bottleneck is the memory communication, there is no expectation of much speedup. In our implementation, we use 16 multiplication units, processing two matrix lines per cycle. Since Vivado HLS was already able to pipeline the multiplications with an initiation internal of 1 and a pipeline latency of 6, we didn't need to deal with the pipelining here.

Multiplier units compute the contributions to the dot-products and output these values together with the destination values (row indices of the matrix non-zeros). These contribution and destination pairs are fed into a 16x16 arbiter network to allow parallel writes to bins. Depending on the destination values, pairs are fed into the corresponding FIFOs, which connect to "Write To Bin" units. Each pair is sent to "Write to Bin" units according to their destination bins, where their destination "Write To Bin" unit is $WriteToBin(dest) = \frac{dest}{Binwidth} \% 16$. The intermediate bin buffer is partitioned into 16 BRAM arrays, which provides two important benefits. Firstly, since BRAMs have a limited number of read and write ports (in our case, 2), concurrent accesses are limited to the number of ports per cycle. Having separate BRAM partitions increases the number of ports in total and allows us to have more concurrent accesses. As there are 16 separate BRAM partitions, 32 accesses per cycle can be made. Secondly, as BRAM size increases, multiplexers inside BRAM units also get larger. This is undesirable because it increases both complexity and latency, which makes it difficult to meet the desired timing requirements. Having separate partitions allows the control logic for address selection to get smaller. This improves latency and makes sure that single-cycle array access is possible. In each "Write to Bin" module, there is a BRAM array allocated for holding the cache lines per bin. Incoming destination contribution pairs are read from FIFO and written to the next available slot in line in address "bin index," which is calculated by dividing the destination address by the number of parallel units, which in our case is 16. Since the parallelization factor is fixed, this is a simple shifting operation. There are two issues with the "Write to Bin" unit. If there are consecutive pairs that need to be written to the same bin, the updated line may not be available if the line gets full. This needs to be checked to ensure the correctness of the module. However, the tool was only able to schedule this fullness check with an initiation interval of 2. The second issue is that when a bin buffer gets full, this may cause the feeding FIFO to block and stall the kernel until the line is written to DRAM. Our design currently relies on the available redundant memory to avoid this. Although it is not very likely that the whole kernel will stall, it is not guaranteed that this will not occur. On the other hand, it is guaranteed that for at least ”line size” cycles, each write will happen to the next index in the line without blocking the feeding FIFO.

When a cache line is full, it is fed to the write module through a FIFO. This module doesn't write the line directly to BRAM, but waits for more lines to be fed in order to write access DRAM in burst mode. One challenge here is that the bin addresses need to be known beforehand. This would require preprocessing of the input matrix to know how many lines would be needed per bin, which is not desirable. Our first proposal was to run two kernels together at the same time, passing the cache lines to the accumulation unit as soon as they are full. However, we weren't able to have such an implementation running efficiently without suffering a significant loss in achieved clock speed. In addition, the accumulation kernel needs to re-read and write the output vector partition each time it gets a new line for a different bucket, whereas in sequential running of kernels, it doesn't even need to read the output vector as it is initialized to zeros and vector partitions are written to DRAM just once. The biggest issue is that DRAM switching between read and write mode is necessary in parallel implementation, which is a significant overhead.

4.2 Accumulation Kernel

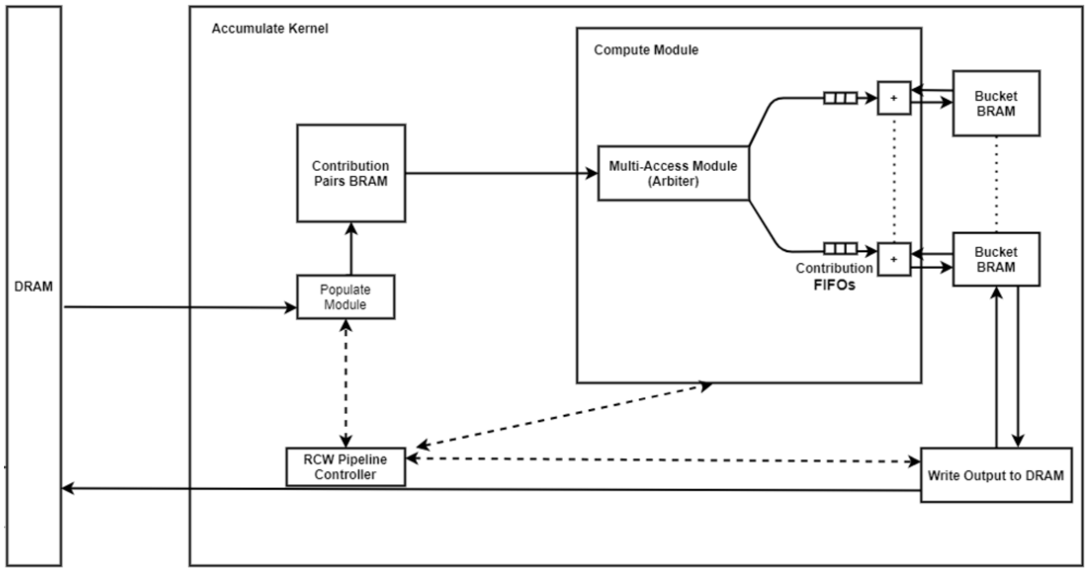

Figure 4.3: Accumulation Kernel Design

In the accumulation kernel, we design and use an RCW Pipeline Controller module, which is to allow higher-task level parallelism. The RCW Controller sends the necessary control signals to populate, compute, and write output modules by implementing input and output buffers as ping-pong block RAMs. This increases the utilization of bandwidth significantly. We again use an arbiter network in order to send pairs to their corresponding compute units. The output vector partition is further partitioned into 16 BRAM blocks, where each BRAM block is updated by a single compute unit. Processing of sixteen pairs per cycle is possible this way.

One issue in the accumulation kernel is in the compute module. II=1 can not be obtained without additional logic due to a read-after write dependency. If two consecutive pairs update to the same destination, a read-after-write hazard will occur since the destination vertex value could have been updated by the time it is read for the next pair. This is a common issue when there is a read-compute-write pipeline on the same array location where the array is accessed in a random fashion, and we solve this by forwarding the compute result to the compute stage of the next iteration, so that the invalid read data would be discarded and the forwarded value would be used. The updated value is also written to a register together with the destination index each cycle, and if for the next pair, the destination indices are the same, the read value is ignored and the register value is used. Compared to previous graph framework implementations on FPGA, we observe that keeping an invalidate flag is a common approach to overcome this dependency. Pipeline forwarding allows us to use fewer BRAM resources on FPGA as there is no need to use a flag for each index in the output vector. However, if the aim is to have a general-purpose graph framework but not just SpMV, it might not always be possible to do pipeline forwarding to achieve an initiation interval of 1, depending on the latencies of desired operations. In such a case, an invalidating flag would be needed anyway.

4.3 Implementation Details

We use the Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit with SDAccel Development Environment 2018.3 for writing the HLS code.

In the SDAccel framework, an application program consists of a host application running on a CPU and a hardware kernel running on an FPGA. The host application uses the OpenCL API to manage the hardware kernel. The CPU and FPGA communicate data using a shared DDR SDRAM. Global memory is accessible by both the host processor and hardware accelerators, whereas host memory is only accessible by the host application. The host application is responsible for allocating the necessary buffers for data transfers in global memory. In SDAccel Development Environment 2018.3, the maximum size for a single buffer is 512 MB, and direct data streaming to FPGA is not supported.

SDaccel uses the Vivado HLS compiler to support both C/C++ and OpenCL kernels. They differ by the optimization pragmas they support. Our initial tests and implementations showed that OpenCL pragmas allow a higher level of abstraction in optimizations. Although this reduces the programmers' effort, it makes it difficult to instruct the compiler to realize the design wanted and forces the programmer to trust the compiler. We chose to implement C/C++ kernels so that we would have relatively more power to describe the design and optimizations. The implementation works on 32-bit data types, and integers and fixed-point arithmetic for floating numbers are supported.

4.4 Design Issues

Both binning and accumulation kernels are initiated by the host application once sequentially. First, the host application calculates the necessary offset addresses for intermediate bin buffers and creates input buffers in DRAM accordingly. After transferring the input data to DRAM, binning kernel is given the start signal by host. After binning kernel finishes its computation, the host application assigns the buffers in DRAM to accumulation kernel and accumulation kernel runs.

One design issue with our implementation arises from the buffer size limitations in DRAM. Although each DRAM bank has a memory size of 16 GB, the host application can allocate only 512 MBs per buffer. Even though the necessary memory to process larger graphs is available, the input data is needed to be partitioned into 512 MB buffers. This means that in order to process 16 GB graph data, 32 buffers are needed to be allocated and connected to the hardware kernel. However, Vivado limits the maximum number of AXI interfaces to 16 for kernel. It is suggested to have a design to run multiple times on the same allocated buffers in order to process large data. Later versions of SDAccel framework supports direct streaming to kernel.

Running both kernels in parallel efficiently is also challenging for several reasons. Additional logic to synchronize the two kernels significantly reduces the achievable target kernel clock. In addition, the communication overhead and communication amount increases significantly in accumulation kernel. Output vector partition for each bucket is needed to be re-read and re-written each time when two consecutive input lines destines different buckets. This not only increases communication amount, but also increases the overhead in DRAM access as read/write mode switching is expensive.

Chapter 5

Evaluation

In this chapter, we first present a theoretical analysis of the proposed design. We then provide the experimental results on the road graph and project the findings onto larger graphs. The projections are compared with the CPU baseline implementation of the GAP benchmark.

5.1 Theoretical Analysis

5.1.1 Binning Kernel

For the binning kernel, we should be able to obtain an initiation interval of 1 for every module. We have 16 partitions of the final intermediate bins. Reading input data occurs in a streaming fashion. For the cache writes to DRAM, we aim to hide the access latency in the kernel computation. In order to achieve this goal, we hold 16 cache lines per bin in BRAM and do not write to DRAM before these 16 cache lines are full. These lines are written into contiguous locations in DRAM and allow us to achieve burst access. Assuming the latency to write access is 100 clock cycles on average, we get 1.28 pairs written to DRAM per cycle, which is the bottleneck in our design. If 32 cache lines per bin are held in the BRAM per bin, 2.56 pairs can be written per cycle. Assuming a 200 MHz clock speed can be obtained, a total of 512M edges can be processed per a single DDR bank. In this case, 4 Mb of BRAM is needed, but 3.81 GB/s of memory write speed is obtained.

5.1.2 Accumulation Kernel

Assuming the output vector partition fits in BRAM, all the reads and writes can be done in a streaming fashion, which allows us to utilize DRAM memory bandwidth fully. In our case, the maximum theoretical memory bandwidth is 16 GB/s per bank. Although this number is a theoretical maximum and not really possible to obtain, it is not expected to decrease much and therefore would be more than enough compared to the write speed of the binning kernel, which is the main bottleneck. With an initiation interval of 1 in all modules, processing 16 edges per bucket is possible. With a clock speed of 200 MHz, 3200M edges can be processed, so again the bottleneck becomes the DRAM accesses.

5.2 GAP Benchmark Suite

For experimental evaluation, we conducted our experiments using the graph data provided in the GAP benchmark suite. Our experiments follow the rules and guidelines given in the benchmark.

GAP is a graph processing benchmark suite that aims to help standardize graph processing evaluations. It provides graph kernels, input graphs, and evaluation methodologies, together with optimized baseline implementations that represent state-of-the-art performance[37].

Information on the graphs provided with the GAP benchmark is shown in Table 5.1.

Graph	Vertices (M)	Edges (M)	Directed	Ref.
Twitter	61.6	1,468.4	Directed	[37, 38]
Web	50.6	1,949.4	Directed	[37, 39]
Road	23.9	58.3	Directed	[37, 40]
Kron	134.2	2,111.6	Undirected	[37, 41, 42]
Urand	134.2	2,147.4	Undirected	[37, 43]

Table 5.1: GAP Benchmark

5.3 Experimental Evaluation

Due to the buffer size limitations of the OpenCL API on SDAccel Development Environment 2018.3, we were only able to conduct experiments on ”Road Graph”, which has 23.9 million vertices and 58.3 million edges. The problem with the buffer size is further explained in the discussion.

We compare our design with a single iteration of the CPU PageRank implementation of the GAP benchmark. Execution time in seconds and million traversed edges per second (MTEPS) are used as evaluation metrics. The term ”optimized layout” in the table refers to the use of a graph directly from the GAP benchmark API, without any shuffling or preprocessing done from our side. We shuffle the input graph randomly in order to obtain a graph that is not optimized for increased locality. We ran the PageRank implementation of the GAP benchmark for a single iteration on a single core of an Intel(R) Core(TM) i5-7500 CPU at 3.40GHz for the baseline. Table 5.2 shows the experimental results on Road Graph.

	Execution Time (s)	MTEPS
Single CPU / Optimized Layout	0.26786	221.9
Single CPU / Shuffled Layout	5.50293	10.5
FPGA Total	0.57886	99.6
Binning (FPGA)	0.55026	105.9
Accumulation (FPGA)	0.0286	2037.4

Table 5.2: Experimental Results on Road Graph

The data shows that there is a dramatic difference between the performance of binning and accumulation kernel. It is expected for the execution time of binning kernel to be longer than the execution time of accumulation kernel from the theoretical analysis. However, experimental results on the road graph demonstrates that the difference is much bigger than expected. There are two assumptions in the theoretical analysis that fail. First one is the assumption that a clock speed of 200 MHz will be obtained. However, due to the timing path failures, the clock speed is changed to 111.4 MHz. Second failing assumption is that all of the pipelined modules will have an initiation interval of 1. Although this should be achievable with further work on the implementation, the initiation interval of "Write To Bin" module in the binning kernel is currently 3. This shows that there is a large room for improvement in the binning kernel. On the other hand, it can be seen that accumulation kernel performs closely to theoretical limit.

It can be observed that a single-core CPU with an optimized layout significantly outperforms our current design. However, when the input graph is shuffled randomly, the execution time of the CPU drops dramatically. This demonstrates that the road graph is small enough for the cache hierarchy in the CPU to provide substantial performance improvements. With larger graphs with less locality, our design is expected to outperform the CPU as the benefits of the cache hierarchy of the CPU will degrade. To support this, we also run the baseline implementation with the rest of the graphs provided in the GAP benchmark. Table 5.3 shows the evaluation of CPU baseline on all graphs.

Input Graph	Execution Time (s)	MTEPS
Road	0.26	221.9
Web	2.37	822.39
Twitter	3.66	400.68
Kron	130.08	16.23
Urand	169.10	12.69

Table 5.3: CPU Baseline Evaluation on GAP Benchmark

On Twitter and Web graphs, we see a significant improvement in the performance of the baseline, compared to the road graph. In these graphs, the average degree increases more than the number of vertices, so we can say these graphs are likely to provide better locality. However, on Kron and Urand graphs, the number of traversed edges per second drops significantly.

5.4 Projection on Larger Graphs

In order to have a better comparison, we project the results we obtained on road data onto larger graphs with certain assumptions.

For the accumulation kernel, input data is read sequentially in bursts to a ping-pong BRAM buffer in a streaming fashion. As long as the output vector partitions fit in BRAM, there is no random DRAM access and whole read/write accesses to global memory occur sequentially. Therefore, with the assumption that the output vector partition fits in BRAM, we expect to have the same performance in terms of MTEPS with larger graphs too. This assumption actually holds for the largest graphs in the GAP benchmark with our current implementation. Our design currently provides a maximum of 2048 buckets, where each can hold 73728 elements. Kron and Urand graphs have 134.2 million vertices, and only 1821 buckets are needed.

For the binning kernel, there are random writes of bin lines to global memory, and the latency of such a write access is not guaranteed to be hidden. If we have sequential edge data coming that needs to be written into the same bin for a long time, the performance of the binning kernel will degrade significantly as the buffers for that bin will saturate and the kernel will need to wait idle until that particular bin buffer is written to global memory. However, if the incoming edges were coming in a more random fashion, the latency of DRAM access would be more likely to be hidden. Therefore, in order to project the performance of the binning kernel, we assume that the incoming edge sequence will arrive as distributed as in the road graph. The input graph data on FPGA runs is already shuffled, so it is not expected to have a higher density of such sequences. In addition, it is again assumed that bin lines would fit in BRAM. In the current implementation, we have BRAM allocated to support 2048 bins, and therefore, this assumption also holds as the maximum number of buckets needed for the largest graph in the benchmark is 1821.

With these assumptions, we expect to have the same MTEPS performance independent of graph size, whereas on the CPU, since the performance relies on the success of caching, we expect the performance to degrade with larger graphs.

In terms of million traversed edges per second, Figure 5.1 compares the performance of CPU implementation to the projected performance of binning and accumulation kernels.

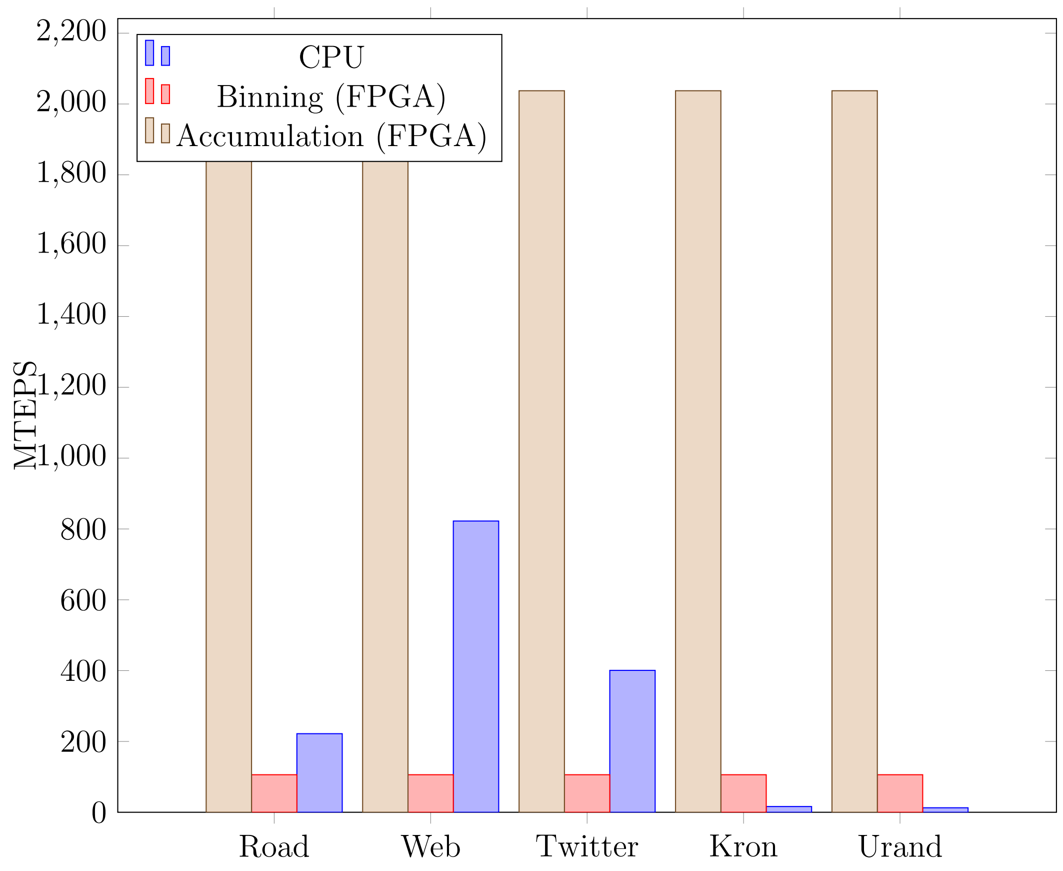

Figure 5.1: MTEPS Comparison

Table 5.4 shows projected execution times of FPGA kernels together with measured CPU execution times per graph.

Input Graph	CPU (s)	Binning (s)	Accumulation (s)	Total FPGA Time (s)
Road	0.26	0.55	0.02	0.57
Web	2.37	18.39	0.95	19.34
Twitter	3.66	13.85	0.72	14.57
Kron	130.08	19.92	1.03	20.95
Urand	169.10	20.26	1.05	21.31

Table 5.4: Projected FPGA Execution Times

Figure 5.2 puts the data in Table 5.4 into a visual context.

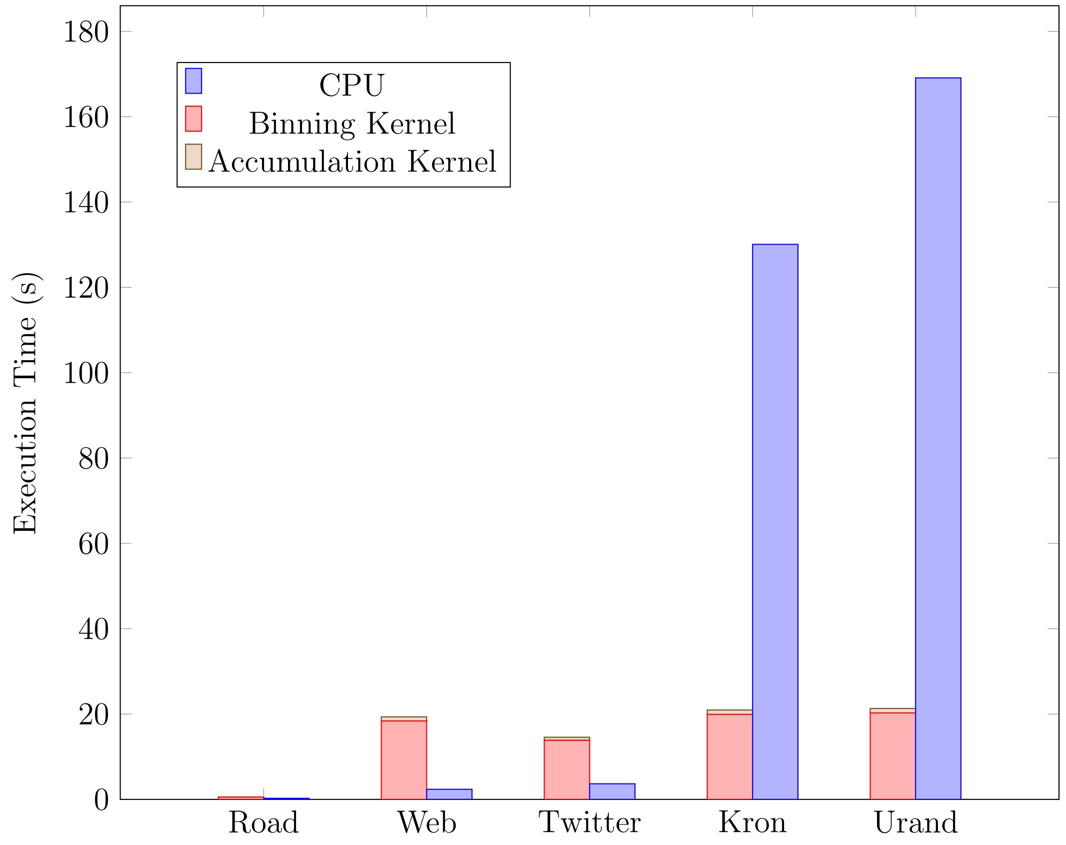

Figure 5.2: CPU and Projected FPGA Execution Time Comparison

A comparison of the projected performance of our design and the CPU baseline shows that our design can provide up to 6.2x and 7.9x speedups on Kron and Urand graphs, respectively. From the evaluation results, it can be said that although the CPU baseline performs well in smaller graphs, it cannot scale up well and is projected to be outperformed by our design as the graphs get larger.

The results also demonstrate that our design is not suitable for small graphs with already high locality. There is no improvement over the baseline in smaller graphs. Considering the performance of the binning kernel and CPU baseline on Road, Twitter and Web graphs, having a heterogeneous system where the binning phase is performed on the CPU and the accumulation phase is performed on the FPGA, might be worth looking at for small and mid-sized graphs. Figure 5.2 shows that, as the execution time for the accumulation kernel is already too short, a concurrent implementation of binning and accumulation kernels may not be as marginally beneficial as expected, considering the necessary design effort.

Chapter 6

Conclusion

SpMV is an algorithm that suffers poorly from poor spatial and temporal locality and is not inherently a suitable application to accelerate on FPGA. This is because FPGAs are more powerful in streaming applications with low random-access patterns, but they are not an ideal choice to accelerate random-access applications. Recently proposed two-phase algorithms such as propagation blocking help to utilize the memory communication of SpMV. They aim to provide high locality at the cost of an additional number of memory accesses and can allow a significant increase in SpMV performance as graphs get sparser. They also allow the opportunity for efficient FPGA implementations. In this study, we adapted the propagation blocking algorithm to an FPGA implementation and provided two kernels for the binning and accumulation phases. Through experiments on the road dataset, we show that when the input matrix is shuffled and the locality is lower, our implementation can provide comparable performance to the CPU. We project our findings on the road dataset to larger graphs and show that our implementation can provide up to a 7.9x speedup over the CPU baseline.

Bibliography

[1] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. T. Pileggi, and F. Franchetti, "Pagerank acceleration for large graphs with scalable hardware and two-step spmv," *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, 2018.

[2] S. Beamer, K. Asanovic, and D. A. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," *2015 IEEE International Symposium on Workload Characterization*, pp. 56–65, 2015.

[3] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *ICS '16*, 2016.

[4] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. Santambrogio, "Heterogeneous exascale supercomputing: The role of cad in the exafpga project," pp. 410–415, 03 2017.

[5] L. Di Tucci, K. O'Brien, M. Blott, and M. Santambrogio, "Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl," pp. 716–721, 03 2017.

[6] "Gpu vs fpga performance comparison white paper 2," 2016.

[7] Graph 500, "Brief introduction."

[8] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1591–1604, 2016.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI*, 2012.

[10] J. E. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI*, 2014.

[11] R. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," *ArXiv*, vol. abs/1402.2394, 2014.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD Conference*, 2010.

[13] T. R. Rao, P. Mitra, and A. Goswami, "The taxonomy of distributed graph analytics," *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pp. 315–322, 2018.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," pp. 31–46, 10 2012.

[15] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," *SOSP 2013 - Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 11 2013.

[16] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (USA), p. 375–386, USENIX Association, 2015.

[17] N. Sundaram, N. Satish, M. M. A. Patwary, S. Dulloor, S. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," 03 2015.

[18] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. Owens, "Gunrock: a high-performance graph processing library on the gpu," pp. 1–12, 02 2016.

[19] "nvgraph."

[20] F. Khorasani, K. Vora, R. Gupta, and L. Bhuyan, "Cusha: vertex-centric graph processing on gpus," 06 2014.

[21] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[22] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," pp. 1–13, 10 2016.

[23] S. Zhou, C. Chelmis, and V. Prasanna, "High-throughput and energy-efficient graph processing on fpga," pp. 103–110, 05 2016.

[24] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," pp. 217–226, 02 2017.

[25] J. Zhou, L. Shaoli, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing," pp. 731–734, 05 2017.

[26] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 166–177, 06 2016.

[27] L. Song, Y. Zhuo, X. Qian, and Y. Chen, "Graphr: Accelerating graph processing using reram," 08 2017.

[28] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "Hyve: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing," *IEEE Transactions on Computers*, vol. PP, pp. 1–1, 01 2019.

[29] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. Hoe, J. Martinez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," pp. 25–28, 05 2014.

[30] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," pp. 111–117, 02 2016.

[31] J. Zhang, S. Khoram, and J. Li, "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," pp. 207–216, 02 2017.

[32] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on fpga-hmc platform," pp. 229–238, 02 2018.

[33] S. Zhou, C. Chelmis, and V. Prasanna, "Optimizing memory performance for fpga implementation of pagerank," pp. 1–6, 12 2015.

[34] G. Lei, Y. Dou, R. Li, and F. Xia, "An fpga implementation for solving large single source shortest path problem," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 46, pp. 1–1, 01 2015.

[35] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," pp. 36–43, 05 2014.

[36] S. Beamer, K. Asanovic, and D. A. Patterson, "Reducing pagerank communication via propagation blocking," *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 820–831, 2017.

[37] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[38] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, (New York, NY, USA), p. 591–600, Association for Computing Machinery, 2010.

[39] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.

[40] "9th dimacs implementation challenge: Shortest paths."

[41] Graph 500, "Brief introduction."

[42] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Proceedings of the 9th European Conference on European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, ECMLPKDD'05, (Berlin, Heidelberg), p. 133–145, Springer-Verlag, 2005.

[43] P. Erdös and A. Rényi, "On random graphs i," *Publicationes Mathematicae Debrecen*, vol. 6, p. 290, 1959.