Exploring FPGA Acceleration of Seed Selection in Influence Maximization

Reece Neff*†, Marco Minutoli*, Antonino Tumeo*, Mahantesh Halappanavar*, Michela Becchi†
*Pacific Northwest National Laboratory, Email: {reece.neff, marco.minutoli, mahantesh.halappanavar, antonino.tumeo}@pnnl.gov
†North Carolina State University, Email: {rwneff, mbecchi}@ncsu.edu

I. Introduction

Influence Maximization (IM) on a social network is the problem of identifying a small cohort of vertices that, when initially activated, results in a cascading effect that will activate the maximum expected number other vertices in the network. While the problem is NP-hard under budget constraints, it has a submodular structure that leads to efficient approximation schemes [1]. This work focuses on accelerating the Influence Maximization with Martingales (IMM) algorithm and its state-of-the-art parallel implementation [2, 3].

The IMM algorithm uses the idea of solving IM by answering the reverse question: *if a vertex is active at the end, what is the most likely cause?* The algorithmic scheme requires building a (large) collection of Random Reverse Reachable (RRR) sets that capture a realization of the diffusion process in reverse. This collection is built through the *sampling* algorithm running diffusion processes simulation in reverse, and it is similar to a randomized variation of breadth first search (BFS) [2]. After building the collection of RRR sets, the algorithm performs seed selection by solving a maximum coverage problem over it. To meet the required approximation bound, the IMM algorithm repeats *sampling* and *seed selection* in a martingale strategy until the proved bound holds. Neff et al. [4] presented a preliminary implementation and performance analysis of the sampling process on FPGAs, identifying key bottlenecks and optimizations. This poster discusses critical optimizations to improve the performance of the FPGA implementation of the seed selection algorithm.

II. Seed Selection on FPGA

Seed selection's most expensive step, counting the uncovered sets, is similar to histogram computation, but with the added step of checking a mask to see if it should be counted in the current iteration. Several previous works look at improving histogram generation on FPGAs [5, 6], but most are geared towards video processing. All consider input or output data small enough to fit on the FPGA on-chip memory, typically only dealing with small 8-bit RGB values. For IMM, however, the input data can reach gigabytes or terabytes in size, requiring storage in external (DRAM) memory.

A. Acceleration Challenges

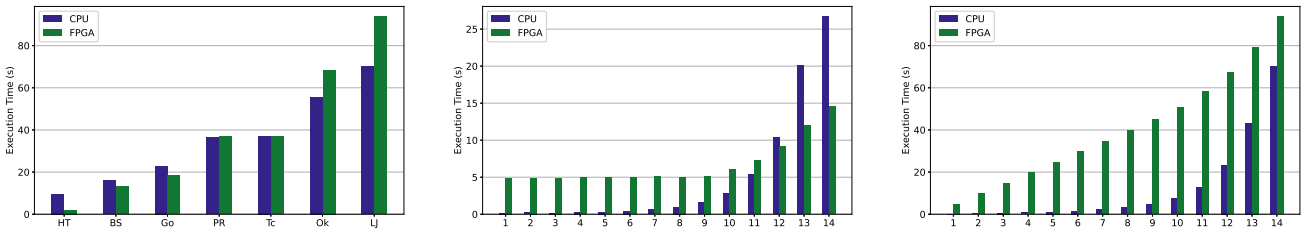
a) Set Mask: While histogram computation typically has a regular access pattern, seed selection has multiple iterations over the RRR set collection, one for each seed. Once a seed is counted, all sets containing that seed are flagged as "covered" and any seeds within those sets are not counted in any future iterations. This requires a random access to check with the set mask every single time a vertex is visited.

b) Data Size: In addition to the input data size discussed previously, the output also cannot be stored on-chip. In fact, the final count array would be too large for most FPGAs. The FPGA used in this poster, a Xilinx Alveo U250, only has 13.5MB of on-chip memory per Super Logic Region (SLR), and the soc-LiveJournal1 graph tested would require at least 16MB of memory to store the output on-chip. For this reason, writes with random patterns would have to occur on the external DRAM, significantly increasing latency.

B. Implementation

a) Data Setup: Upon receiving the collection of RRR sets from the Sampling step, the host then orders the RRR set collection by vertex, pairing each vertex with its associated set id to allow for a more regular pattern of the output and binary searching of the vertex ids. We also combine the vertex and set ids into a 2-word data structure to allow both values to be accessed in a single data transaction for lower memory port and memory bank usage.

b) Count Uncovered: We implement a multi-stage design, consisting of several steps that allow for regular burst accesses. To start, the set mask is burst read at a 512-bit port width and is cached in UltraRAM. This allows the FPGA to check one mask flag every clock cycle, eliminating the high latency cost of the random read required if the mask was located in external DRAM. The RRR set collection data (vertex id and set id) is then burst read from external memory and streamed into the compute unit to be counted. Inside the compute unit, we utilize the design from Kastner et al. [5] by caching an output counter in a register until a new vertex is encountered, and the cached output is then streamed to a write pre-processing unit. By ordering the vertices before processing, each group of vertices that are the same only need a single write to the output counter. In the write pre-processing unit, the vertices not counted by the main compute unit are zero filled and the resulting outputs are streamed to the write unit. This allows all the data to be presented in a sequential manner, giving the write unit the opportunity to burst write to memory and eliminate the latency cost of random accesses.



(a) Total Function Time (b) soc-LiveJournal1 Breakdown (c) Running Sum soc-LiveJournal1

Fig. 1: (a) shows total time spent on functions within CPU vs FPGA, while (b) and (c) show a breakdown of each seed selection iteration on soc-LiveJournal1. The higher the iteration, the higher the number of RRR sets to be processed.

c) SLR Reduction: Here, we take the two FPGA and the single CPU count arrays and burst access them, each with a port width of 512 bits, giving the FPGA 3x 16-word vectors per clock cycle. As we are only concerned with finding the vertex with the maximum number of occurrences, we do not need to write the resulting sum. Instead, we send each of the 16 values to be compared with the maximal vector, storing the largest of each 16 values. At the end, the 16 values are then reduced to a single maximally occurring vertex id, which is returned to the host along with its count.

d) Update Mask: After the SLR reduction has determined the most influential seed (vertex), it is sent to the Update Mask unit. The RRR set collection is sorted by vertex id, so the Update Mask unit can identify the vertex range through a lower and upper bound binary search. Following this operation, the set mask bits corresponding to the seed are updated to a "1", marking all vertices belonging to those sets as covered.

e) Topology: The Xilinx Alveo U250 contains four Super Logic Regions (SLRs). An SLR corresponds to a "slice" of the die of programmable logic with its own memory channel connecting to an external DRAM module of 16GB. There are two Count Uncovered kernels that write into memory banks located on adjacent SLRs. The CPU count is written into a third adjacent SLR. The Reduce SLR kernel is located between all three counters, minimizing the limited and costly SLR crossing logic needed to access DRAM of other SLRs. Finally, the Update Mask kernel is located on the outside SLRs as it only needs to access the RRR set collection and set mask, both located in the outer SLR's DRAM.

III. Experimental Setup

For the FPGA implementation, we wrote the kernels in C++ and synthesized them with the Vitis Unified Software Platform 2020.2 [7]. Our host system provides an Intel Xeon E5-2637 v4 processor with 8x32GB 2400MHz DDR4 RAM modules. The FPGA is a Xilinx Alveo U250 with 4x16GB 2400MHz DDR4 RAM modules. We measured the time spent for the FPGA-accelerated kernels vs the CPU. For the FPGA setups, we utilized a heterogeneous configuration composed of 3 CPU cores and 1 FPGA running in parallel (1 CPU core managed data transfers and task enqueues to the FPGA), and for the CPU only setup, we used all 4 cores in a homogeneous system with OpenMP from the state-of-the art implementation of influence maximization [2, 3]. We used the following input graphs from the SNAP data set [8]: cit-HepTh (HT), web-BerkStan (BS), web-Google (Go), soc-pokec-relationships (PR), wiki-topcats (TC), com-Orkut (Ok), and soc-LiveJournal1 (LJ), ordered from smallest to largest number of vertices.

IV. Preliminary Results

We compared the execution time of the functions accelerated on CPU (with 4 cores) versus the execution time of functions accelerated on FPGA (with 1/4 of the workload, equivalent to the work distributed to a single CPU core). The FPGA implementation begins to fall behind due to its static write overhead from the count uncovered kernel. Because the write pre-processor inserts 0 values to ensure a smooth burst, this can cause a large imbalance when vertices in the graph outnumbers the RRR set collection. In Fig. 1a, we can see that the FPGA shows a speedup of up to $4.78\times$ over CPU on cit-HepTh, but also that it falls behind the CPU on larger graphs with only $0.75\times$ the relative performance of the CPU on soc-LiveJournal1. Upon further inspection into each seed selection iteration for soc-LiveJournal1 in Fig. 1b, we realized that the static write overhead has a large impact on earlier iterations where the ratio of the workload to the number of graph vertices is lower. Figure 1c shows a running sum of the iterations detailing the impact of the overhead on earlier iterations for the accelerated function execution time.

V. Conclusion and Future Work

We developed an FPGA architecture utilizing all memory banks and SLRs, maximizing port widths and burst reads while utilizing on-chip pre-processing to enable burst memory writes. While this comes with overheads for large graphs, we demonstrated that the FPGA can outperform the CPU on smaller graphs. This is due to a lower overhead on later iterations of the seed selection step, when the workload to vertex ratio is higher. For the future, we anticipate: 1) Parallelizing the count uncovered compute units via port width widening on the burst reads and adding a reduction tree for the final write 2) Only running the FPGA on iterations where it outperforms CPU, and 3) Partitioning the data to reduce the total range of countable vertices per SLR, lowering the static write overhead. We hope this work can provide insights into accelerating other "almost-regular" applications to create architectures that can utilize regular burst accesses on external memory banks.

REFERENCES

[1] D. Kempe, J. M. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, 2003.

[2] M. Minutoli, M. Halappanavar, A. Kalyanaraman, A. Sathanur, R. Mcclure, and J. McDermott, "Fast and scalable implementations of influence maximization algorithms," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–12.

[3] M. Minutoli, M. Drocco, M. Halappanavar, A. Tumeo, and A. Kalyanaraman, "Curipples: Influence maximization on multi-gpu systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. ACM, 2020.

[4] R. Neff, M. Minutoli, A. Tumeo, and M. Becchi, "Fpga-accelerated ripples," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, 2021.

[5] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," *ArXiv e-prints*, May 2018.

[6] A. Shahbahrami, J. Y. Hur, B. Juurlink, and S. Wong, "Fpga implementation of parallel histogram computation," in *2nd HiPEAC Workshop on Reconfigurable Computing*. Citeseer, 2008, pp. 63–72.

[7] V. Kathail, "Xilinx vitis unified software platform," in *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, S. Neuendorffer and L. Shannon, Eds. ACM, 2020, pp. 173–174.

[8] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.