# High-Level Synthesis of Irregular Applications: A Case Study on Influence Maximization


Reece Neff*
rwneff@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Marco Minutoli
marco.minutoli@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Antonino Tumeo
antonino.tumeo@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Michela Becchi
mbecchi@ncsu.edu
North Carolina State University
Raleigh, NC, USA



## ABSTRACT

FPGAs are promising platforms for accelerating irregular applications due to their ability to implement highly specialized hardware designs for each kernel. However, the design and implementation of FPGA-accelerated kernels can take several months using hardware design languages. High Level Synthesis (HLS) tools provide fast, high quality results for regular applications, but lack the support to effectively accelerate more irregular, complex workloads. This work analyzes the challenges and benefits of using a commercial state-of-the-art HLS tool and its available optimizations to accelerate graph sampling. We evaluate the resulting designs and their effectiveness when deployed in a state-of-the-art heterogeneous framework that implements the Influence Maximization with Martingales (IMM) algorithm, a complex graph analytics algorithm. We discuss future opportunities for improvement in hardware, HLS tools, and hardware/software co-design methodology to better support complex irregular applications such as IMM.



## CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; • **Mathematics of computing → Graph algorithms**.

## KEYWORDS

High-Level Synthesis, Graph Algorithms, Influence Maximization

**ACM Reference Format:**
Reece Neff, Marco Minutoli, Antonino Tumeo, and Michela Becchi. 2023. High-Level Synthesis of Irregular Applications: A Case Study on Influence Maximization. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3587135.3592196


*Also with Pacific Northwest National Laboratory.




*CF '23, May 9–11, 2023, Bologna, Italy*





ACM ISBN 979-8-4007-0140-5/23/05…$15.00
https://doi.org/10.1145/3587135.3592196


## 1 INTRODUCTION

Graph theoretic frameworks naturally capture the relationships and the behavior of highly dynamical systems by embedding their behavior in networked models. The simplicity and expressiveness of these formalisms enable the building of sophisticated analytics in application domains spanning from social sciences and marketing to biology and epidemiology. However, the algorithms and data structures that are at the core of graph analytic methods have highly irregular runtime behavior [35] caused by the poor spatial and temporal locality of their data-dependent memory access patterns.

The unique characteristics of graph algorithms and, more broadly, graph analytics have motivated the work towards their acceleration through specialized computing devices. Tera introduced the MTA [2], a large-scale multi-threaded machine optimized for irregular data accesses, while EMU Technologies [12] proposed a fine-grained multi-threaded design with hardware support for thread migration across the system. Due to the availability of products that can be easily integrated into cloud computing systems, Field Programmable Gate Arrays (FPGAs) have seen renewed interest for the acceleration of graph analytics workflows [3, 7, 30, 34, 36, 37]. Convey [9] proposed a custom multi-threaded design with a parallel memory controller optimized for irregular accesses and the programming model to build custom accelerators on their system.

The ever-increasing pace at which new data mining and machine learning based analytics on graphs are introduced makes traditional hardware design cycles extremely hard or impractical for the development of custom accelerators. High-Level Synthesis (HLS) provides methods for the automatic generation of custom accelerators from high-level programming languages, addressing the productivity gap and reducing the time-to-market of custom computing devices. However, current state-of-the-art HLS tools have been developed and optimized for the regular behavior of signal processing systems, as that has been the core market driving the HLS industry.

Our work explores optimization techniques for the automatic synthesis of analytics based on graph sampling and exposes the research gaps in the current state-of-the-art HLS approaches. We present our analysis using parallel algorithms for the Influence Maximization Problem. The lessons learned derived from our experiments can be generalized to the entire class of approaches that heavily rely on random walks and graph sampling (e.g., Link Prediction and Graph Neural Networks).

The Influence Maximization (IM) Problem aims to identify a set of k *seed* vertices that maximize the impact (number of activations) of a diffusion process over a network. The problem is central in network science and has many applications to social network analysis and disease modeling and intervention [27]. This problem is NP-hard, but has a submodular structure [19], so we can compute approximate solutions in polynomial time. However, even with these approximations, the computational costs remain high when processing medium scale graphs. We base our work on current state-of-the-art parallel implementations for homogeneous [25] and GPU-accelerated [26] high-performance computing clusters, focusing in particular on the IMM algorithm [33]. We use the IMM algorithm as our case study target because it is more complex than the simple applications typically benchmarked, modeling a real-world use case for acceleration. These implementations are also generalizable to a wider class of random graph walks and sampling, allowing the takeaways from this work to be extended to other applications not limited to influence maximization.

There are several approaches to efficiently implement graph algorithms on FPGAs [6]. These include specific algorithmic implementations, using combinatorial or sparse linear algebra formulations, but typically focusing on elementary graph kernels (breadth first search, page rank, triangle count) [3, 13, 17], graph acceleration frameworks, typically employing gather-apply-scatter approaches [20, 28], and specialized accelerator generation methods [24, 32]. While showcasing many possible approaches to accelerate graph processing with FPGAs, all these solutions either focus on studying relatively simple graph kernels or on accelerating a few computational patterns, and their performance often relies on reordering or partitioning schemes. However, Barik *et al.* [4] studied the impact of these schemes on the performance of complex graph analytics and showed that there is no improvement when graph sampling and random walks are the constraining operations.

Our analysis aims to provide insight on the performance impact of optimizations available within modern commercial HLS frameworks, their effectiveness, and shortcomings when applied to complex graph analytics incorporating graph sampling and random walks. As part of our study, we provide implementations of custom accelerators that are fully integrated within the Ripples software package and compare the HLS-accelerated algorithms on a Xilinx Alveo U250 to CPU-only runs and hybrid CPU-GPU runs of the application. We discuss the challenges with the current tools and hardware and suggest improvements to increase the quality of results.

In summary, we make the following contributions:

- we implement custom hardware accelerators for the core computational kernel of the IMM algorithm (§ 3) that relies on random walks (the LT model) and graph sampling (the IC diffusion model).
- we study the effect of several well-known HLS optimizations and discuss their limitations when applied to two different diffusion models (§ 3.3).
- we analyze the achieved performance of our proposed HLS-based accelerators (§ 4). The final design for the LT model shows up to a 34.8× speedup over the baseline FPGA model, while that of the IC model shows up to a 45.5× speedup. However, when tested in a state-of-the-art parallel heterogeneous configuration, only the LT model outperforms the CPU setup by a factor up to 3.65×, while the IC model matches the CPU setup's performance and both fall behind the GPU setup.
- we discuss opportunities for improvement on the hardware, HLS tool, and co-design methodology to make HLS-based FPGA acceleration more suitable for complex irregular workloads (§ 5).

To the best of our knowledge, this work represents the first systematic study of performance optimizations for acceleration on FPGAs using HLS of complex graph analytic algorithms using graph sampling and random walks.

## 2 INFLUENCE MAXIMIZATION

The Influence Maximization (IM) Problem was originally defined as the problem of identifying a small cohort of individuals that maximize the outreach over a broader population through (word-of-mouth) recommendations [11]. The seminal work of Kempe *et al.* [19] showed that its objective function ($\sigma()$) is submodular under two simple but powerful diffusion models: the Independent Cascade (IC) model and the Linear Threshold (LT) model. Under this optimization framework, IM is defined as follows:

Definition 1 (Inf-Max). *Given a graph $G = (V, E, w)$, a diffusion model $M$, and a budget $k$, the IM problem is to find a set $S \subseteq V$ of size at most $k$ such that $\sigma(S)$ is maximum. Where $\sigma(S)$ is the expected number of activated vertices over $G$ when $M$ starts from the set $S$.*

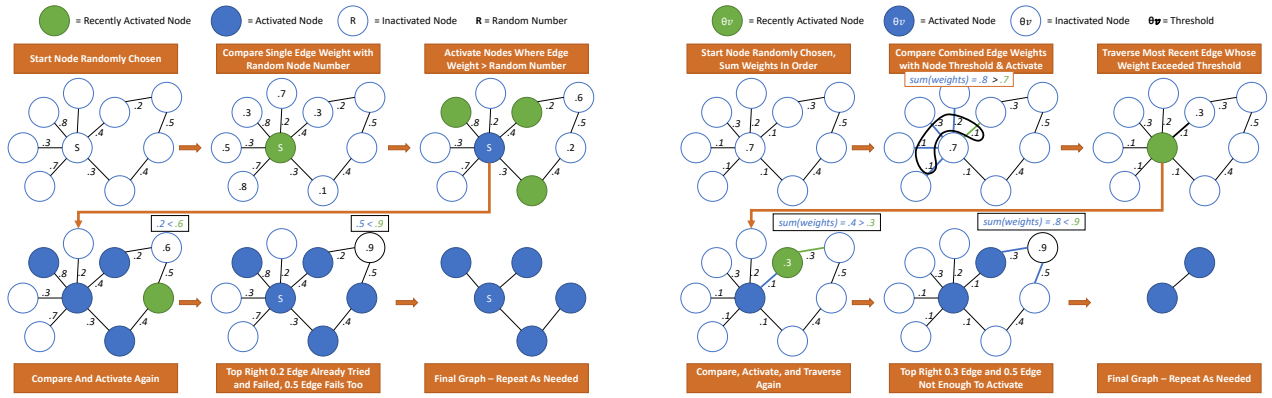
Kempe *et al.* [19] show that Inf-Max is NP-hard under the IC and LT diffusion models. However, the submodularity of the objective functions leads to efficient approximation algorithms with an approximation guarantee of $1 - 1/e - \varepsilon$. A lower $\varepsilon$ means a better quality solution.

Borgs *et al.* [8] introduced the idea of reverse influence sampling (RIS) that is foundational to the IMM algorithm from Tang *et al.* [33] and its state-of-the-art parallel version from Minutoli *et al.* [26] that we extend in our work. The reverse influence sampling scheme is better understood as a randomized experiment to identify the most likely source of activation for each vertex in the graph. The sampling process (Algorithm 1) consists of drawing uniformly at random a vertex $v \in V$ and then performing the diffusion process $M$ in reverse starting from $v$. The sets $R$ of vertices visited in the reversed diffusion process starting at $v$ are usually referred to as *random reverse reachable (RRR) sets*. The sampling process with this approach consists in building a collection of $\theta$ RRR sets ($\mathbb{R}$). The fundamental intuition is that highly influential seeds will be part of many of the RRR sets in $\mathbb{R}$. A maximum coverage algorithm over $\mathbb{R}$ [8] gives the final seed set ($S$).

**Algorithm 1:** IMM Algorithm from [33]

**Input:** $G$, $k$, $\varepsilon$
**Output:** $S$

**begin**
  $\{\mathbb{R},\theta\} \leftarrow$ `EstimateTheta` ($G$, $k$, $\varepsilon$)
  $\mathbb{R} \leftarrow$ `Sample` ($G$, $\theta$ - $|\mathbb{R}|$, $\mathbb{R}$)
  $S \leftarrow$ `SelectSeeds` ($G$, $k$, $\mathbb{R}$)
  **return** $S$




**(a) IC Model** **(b) LT Model**

**Figure 1: Representation of the Independent Cascade and Linear Threshold diffusion models on a small graph.**

Algorithm 1 shows the high level structure of the IMM algorithm [33]. Tang *et al.* proved that, through a martingale strategy, the sampling algorithm and the seed selection algorithm can be used to compute a lower bound on the sampling effort ($\theta$) required to achieve the quality of the solution requested (controlled through $\varepsilon$). The estimation of $\theta$ starts by guessing an initial value and proceeds by growing the current collection of RRR sets ($\mathbb{R}$) to $\theta$ elements and perform a seed selection step to verify that the exit condition derived from the lower bound is satisfied and doubles $\theta$ otherwise. Once we have a good estimate for $\theta$, the algorithm concludes by growing $\mathbb{R}$ to $\theta$ RRR sets, if needed, and computing the seed set $S$.

Our work builds on the IMM parallel framework proposed by Minutoli *et al.* [26] for the acceleration of the IMM algorithm on multi-GPU systems. We extend and generalize their framework by integrating custom hardware accelerators for FPGA generated through HLS.

**Independent Cascade (IC) Model:** The IC model is the simplest among the cascading models. In its settings, each newly activated vertex $v$ has single attempt at activating its neighbors $u$ and succeeds with probability $p_u(v)$, where $u$ is a neighbor of $v$. Despite its simplicity, the IC model has broad applicability in modeling (mis)information and epidemics as it is closely related to the Susceptible-Infectious-Recovered (SIR) model in the framework of networked epidemiology. Fig. 1a shows a simulation of the execution of the IC model. The process evolves in a breadth-first search (BFS) like process where only the neighbors of each active vertices enter the next horizon with some probability. Therefore, the randomized BFS-like process is not expected to go deep into the graph. The vertices visited during this process will constitute the content of an RRR set produced during sampling (Algorithm 1). Therefore, the runtime behavior of the IC diffusion model is representative of sub-graph sampling procedures commonly used in data mining and machine learning algorithms on graphs.

**Linear Threshold (LT) Model:** Threshold models capture mass behaviors. Vertices have a threshold ($\delta_v$) in the input graph that represents their inertia to activation. In the LT model, each vertex $v$ has a threshold $\delta_v$ and each edge $(u, v)$ has a weight $w_{u,v}$ and a vertex $v$ becomes active when $\sum_{u \in N_a(v)} w_{u,v} \geq \delta_v$, where $N_a(v)$ is the currently active neighborhood of $v$. Our implementation uses the equivalent LT model in [19], as it is better suited to implement algorithms based on RIS. Fig. 1b shows a simulation of this equivalent LT model. The process starts from a randomly selected node $v$ by generating a random threshold $\delta_v$ to account for the uncertainty on the real value of the threshold. Then, the diffusion process continues by considering the outgoing edges from $v$ one at a time and accumulates their weights $w_{v,u}$. When there is an edge $(v, u)$ that makes the sum of the edge weights greater than the threshold $\delta_v$, the vertex $u$ is added to the next horizon of this BFS-like traversal. Otherwise, the process stops at $v$.

The graph traversal (Figure 1b) will visit only one of the vertices $u$ that are neighbors of $v$ at every step of the process. Therefore, the LT model performs a random walk over the graph. The sets from the LT model do not contain any cycles, but the main lessons from this work (§ 5) can be applied to a broader class of random walks, including those that support cycles.

## 3 ACCELERATED ALGORITHMS

Minutoli *et al.* [25] showed that the Sampling step in Algorithm 1 is the most computationally intensive part of the algorithm. Therefore, we focus our analysis on the sampling algorithms that generate RRR sets for the LT and IC diffusion models. To develop our HLS-Based FPGA implementations, we started from the approach used for CUDA in [26] for the LT kernel, and from the approach described in [25] for the IC kernel. The Vitis HLS compiler supports C, C++, and OpenCL. We used OpenCL because of its promises of portability between accelerator systems. In our study, we target a Xilinx Alveo U250 FPGA with 4x16GB 2400MHz DDR4 RAM. The Alveo U250 is equipped with four Super Logic Region (SLRs) with combined SLR resources totaling 54MB SRAM, 1.7 million look-up tables (LUTs), 3.5 million registers/flip-flops (FFs), and 12,228 DSP slices.

### 3.1 High Level Synthesis with OpenCL

OpenCL is one of the programming models widely adopted by HLS tools to simplify porting from CPU/GPU applications; however, considerable work is required to write applications that will perform well for the targeted platform. While HLS tools try to perform automatic optimizations, the developer usually needs to restructure the code and annotate it to steer the optimization process. Hence, significant additional effort and understanding of the generated

hardware is still required during development. While OpenCL for CPU and GPU can map instructions to existing hardware present on the CPU and GPU, OpenCL for FPGA needs to generate hardware to achieve the expected functionality, and generally requires extensive modification of the code that isn't needed for GPUs and CPUs. This creates challenges for OpenCL to generate efficient hardware.

For example, even with OpenCL, memory is managed very differently depending on the target devices. CPUs only leverage data caching. Modern GPUs have configurable on-chip memories that can act both as caches (transparently) or as scratchpad, but making use of them to prefetch and reorganize data is directly supported by the programming model. OpenCL for FPGAs, instead, needs explicit and specialized declarations (including parameters) of on-chip memory, since it needs to instantiate and reserve Block RAM (BRAM) from the available resources, and explicit data movements. Inference of BRAM caches is possible only for very simple kernels. § 3.2 and § 3.3 detail the challenges and work required to create a more efficient OpenCL kernel targeting FPGAs.

**Algorithm 2:** Baseline LT Kernel

**Input:** $G, P, S$ `// P = PRNG States, S = Num of Sets`
**Output:** $\mathbb{R}$

1 **for** $i \leftarrow 0$ **to** $S$ **do**
2 $\quad \{v_{curr}, P_i\} \leftarrow$ `GenRandomInt` $(P_i)$
3 $\quad$ **while** `NotFinished` $(\mathbb{R}_{temp})$ **do**
4 $\quad\quad \{t, P_i\} \leftarrow$ `GenRandomThreshold` $(P_i)$
5 $\quad\quad$ **for** $v_{neighbor}$ **of** $v_{curr}$ **do**
6 $\quad\quad\quad t \leftarrow t-$`Weight` $(G, v_{neighbor})$
7 $\quad\quad\quad$ **if** $t \leq 0$ **and** `NotVisited` $(v_{neighbor}, \mathbb{R}_{temp})$ **then**
8 $\quad\quad\quad\quad$ **if** $|\mathbb{R}_{temp}| < 8$ **then**
9 $\quad\quad\quad\quad\quad v_{curr} = v_{neighbor}$
10 $\quad\quad\quad\quad\quad \mathbb{R}_{temp} \leftarrow \mathbb{R}_{temp} \cup \{v_{curr}\}$
11 $\quad\quad\quad\quad$ **else**
12 $\quad\quad\quad\quad\quad \mathbb{R}_{temp} \leftarrow \zeta$ `// ζ = Invalid`
13 $\quad\quad\quad\quad$ **break**
14 $\quad \mathbb{R} \leftarrow \mathbb{R}_{temp} \cup \mathbb{R}$
15 **return** $\mathbb{R}$

We set our baseline by starting with an NDRange implementation of the LT kernel to a single work-item kernel. An NDRange kernel operates similarly to a CUDA kernel call, where the number of work-items and work-groups are specified in the call (equivalently called threads and thread-blocks in CUDA), and a single work-item implementation calls a kernel with a single work-item, and the amount of work is specified as a scalar kernel argument rather than in the NDRange call. Vitis recommends implementing kernels as a single work-item, as it allows for more fine-grained control and optimization [1]. This kernel operates similarly to the GPU version with a max walk size of eight, exploiting the observation that 99% of the RRR sets for the LT model contains fewer than ten nodes[26]. When an RRR set grows beyond eight nodes, the FPGA marks that RRR set as invalid, and the FPGA moves to the next LT walk to simulate. When the CPU is transferring the RRR sets from the FPGA into host memory, it identifies the invalid sets and replays them on the CPU, each invalid set corresponding to a single set generated by an LT random walk. This invalidation is performed

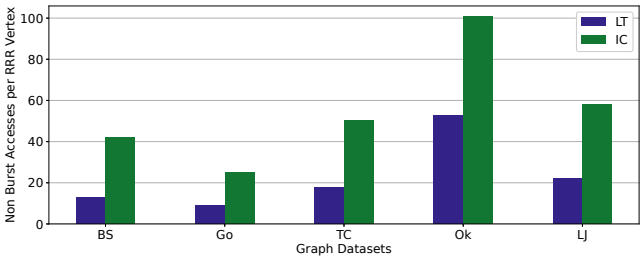



**Figure 2: LT vs IC Memory Access Characterization**

to: (1) Allow the majority of sets to be generated without memory constraints, as allocating $S \times V$ for storing results becomes quickly infeasible with 32k walks queued per call, and (2) Allows the visited nodes to be stored in on-chip memory so the FPGA can quickly determine whether a node has been previously visited. The baseline LT kernel is shown in Algorithm 2. We ported the IC kernel from the C++ implementation as a single work-item kernel, modifying the memory access pattern and the operation order as discussed in § 3.3. Figure 2 shows the difference in the number of memory access per vertex without burst transactions between LT and IC, with a burst transaction being an aggregation of multiple memory transactions into a single request. This memory access data was collected by identifying accesses that weren't synthesized as burst accesses, counting the occurrences of those memory access, and dividing by the number of visited vertices.

We dispatched workloads to accelerators, loading 32 work-items at a time for CPU, and 32,768 work-items at a time for GPU and FPGA on the LT model. The LT model has such a small execution time that the workloads are balanced between the FPGA compute units on multiple compute unit architectures. For the IC model, work-items were dispatched to CPU one at a time, GPU 32 at a time, and on FPGA between 1-16 work-items per compute unit. The workload distribution for IC is more granular due to its wider variation in execution time. Each CPU core manages a separate accelerator (denoted as "GPU/FPGA Workers") and, if there are more cores than accelerators, the extra CPU cores will also generate RRR sets ("Denoted as CPU Workers"). On the host, there is a global atomic variable that tracks the number of RRR sets that have or are in the process of being generated. For each batch, the CPU core managing either an accelerator or itself will perform an atomic fetch_add to determine the amount of work that needs to be done, finishing if the total number of RRR sets requested has been reached by the global atomic variable. Once all workers finish, the algorithm will progress to the "SelectSeeds" step (Algorithm 1). This method provides dynamic workload balancing between accelerators, allowing different architectures (such as CPU/GPU and CPU/FPGA, as tested in this work) to concurrently generate RRR sets. While it is possible to design a heterogeneous implementation that simultaneously runs on CPU, GPU, and FPGA, this is outside the focus of this work.

## 3.2 Acceleration Challenges

There are several challenges in accelerating and optimizing these kernels on FPGA, connected with the irregular memory accesses and control flow typical of graph processing. The kernels incur

many fine-grained memory reads to unpredictable locations due to the irregular nature of each graph traversal in the kernel as discussed in § 2. Graphs must be stored in high-latency off-chip memory (DRAM) since they are too large to be stored in on-chip memory (BRAM). Most recent FPGA devices have on-chip memory with sizes in the order of the tens of megabytes. For the IC kernel, the frontier queue, visited array, and output all have to be stored in off-chip DRAM as well, due to their size; consequently, they increase the pressure on the off-chip memory (Fig. 2). While storing metadata on-chip is possible, it restricts the size of the input dataset, which limits overall portability. Graph pre-processing to improve locality on FPGAs has been explored by Sabet *et al.* [29], but their approach would not be effective in the case of IMM because of unique traits of this method. In fact, a case study on the IMM Algorithm by Barik *et al.* [4] did not show significant performance benefit from the application of various vertex reordering techniques due to the randomness of its graph traversals. Moreover, during our performance analysis, we discovered that Vitis HLS is unable to pipeline the LT model due to a read-after-write (RAW) data dependency. This inability to pipeline increases the overall kernel latency, prompting the exploration of other optimizations to further improve performance.

## 3.3 Acceleration Optimizations

Below are optimizations applied to the kernels. *Algorithm-specific optimizations* are those that require a better understanding of the code and have significant code modifications to implement, and *general optimizations* are those that are more easily applied and can be done more generally regardless of the targeted code. For algorithm-specific optimizations, we describe: how the optimization is automatically applied by Vitis HLS, denoted as "With Vitis", and what are the code modifications required to trigger the optimization, denoted as "Modifications".

*3.3.1 Algorithm-Specific Optimizations.* **Pipelining.** Loop pipelining is one of the key features exploited in FPGA acceleration to improve throughput. With Vitis: Vitis HLS automatically tries to pipeline loops with variable bounds or loops greater than 64 iterations, so the baseline version already includes this optimization. We were able to perform pipelining only on loops with small instruction counts such as the burst access loops described in the next section. Vitis HLS is unable to pipeline the LT model due to loop-carried dependencies. In fact, there is a RAW dependency on the threshold variable in the loop from Line 6 through Line 13 in Algorithm 2. Vitis HLS could, however, pipeline the corresponding IC kernel loop since there is no dependency on its threshold variable. Modifications: To expose more pipelining opportunities in the IC model, we were able to reorder (and eliminate) a loop-carried dependency by allowing duplicate entries in the frontier queue and eliminating them in later loop iterations. This reordering reduced the initiation interval (II), which is the number of cycles that pass before issuing each new loop iteration, of the IC model from 142 to 5 cycles (out of total pipeline depths of 291 and 290 cycles, respectively). Further II reduction was not possible as the linear congruential generator takes 5 cycles to generate a new pseudorandom number. Using a faster method such as a linear feedback

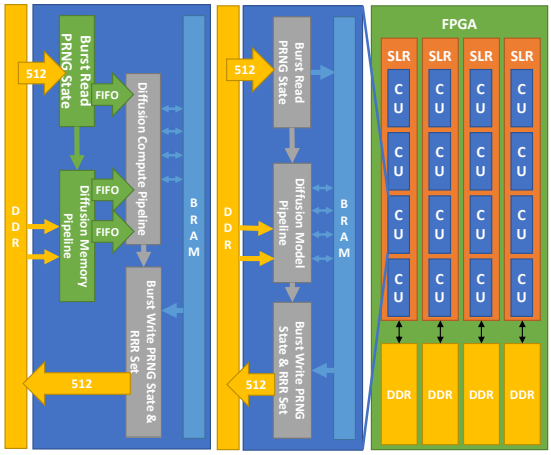



**Figure 3: High level architecture and data transfers of implemented optimizations. The left compute unit is with pipes, and the middle compute unit is without pipes.**

shift register would change the random behavior of the walk compared to the CPU and GPU, so the pseudorandom number generator (PRNG) method was kept the same. For the LT model, it is possible to perform efficient pipelining, but requires major code restructuring and a detailed knowledge of what the hardware is capable of. Because the resulting code is barely distinguishable from the baseline, this is out of the scope of this work, where we focus on optimizations more easily available to the higher level user.

In addition to refactoring for data dependencies, we performed array partitioning to prepare for pipelining. To achieve a lower II, this must also occur in pipelines where the same BRAM is accessed more than twice, as the on-chip BRAM only has two ports. Partitioning increases the number of BRAMs and, therefore, memory ports. This allows more pipeline stages to concurrently access on-chip memory without incurring resource contention. Each CU shown in Fig. 3 is a set of three main components: (1) a burst read unit to access the PRNG states, (2) a diffusion model unit to perform the random walk, and (3) a burst write unit to write back PRNG states and the resulting set(s). Further II details are reported in § 4.

**Burst Memory Accesses.** Burst memory access can reduce the number of clock cycles required to retrieve data from external DRAM, increasing throughput. With Vitis: To achieve burst memory access in Vitis, arrays must be declared statically to act as buffers for the memory being accessed externally. The loop performing the burst access must also have an II of 1, and can only be one read or write in a contiguous sequence for Vitis HLS to infer a burst memory access from the loop. In the baseline implementation, a single pseudorandom number generator (PRNG) state used in a linear congruential generator for threshold values [5] is read from external memory once per RRR set, so the burst accesses cannot be inferred. Modifications: The Alveo U250 has a maximum burst length of 1024. To take advantage of this burst length, we implemented burst memory accesses to move a batch of data to local memory in the form of on-chip BRAM that can be accessed at a lower latency than data stored in off-chip DRAM. Both PRNG data reads/writes and LT result writes are accessed contiguously, so they can be burst accessed.

```
// Declare pipe, set depth to 1024
pipe float weight_pipe
    __attribute__((xcl_reqd_pipe_depth(1024)));

// Memory Kernel Section
for(int i = start_edge; i < end_edge; i++){
    float weight = global_weights[i];
    // Write weight to buffer
    write_pipe_block(weight_pipe, &weight);
    [...] // Mirror control flow of compute kernel
}

// Compute Kernel Section
for(int i = start_edge; i < end_edge; i++){
    float weight;
    // Read weight from buffer
    read_pipe_block(weight_pipe, &weight);
    [...] // Continue with computation
}
```

**Listing 1: Pipes sample code**

To promote more contiguous accesses as opposed to a single access per iteration, the kernel was refactored to run in batch cycles, loading 1024 data values at a time in the case of LT and running that batch of 1024, burst offloading the buffer to external memory, and obtaining the next batch to be loaded into the buffer. In Algorithm 2, a for loop loading all the PRNG states into a buffer and writing the states from the buffer to global memory was added at the beginning and at the end of the algorithm outside the outer for loop. The burst access and data movement is visualized in Fig. 3. Burst accesses support accessing up to 1024 values at a time, reducing the memory overhead for accessing each item individually between loop iterations.

**Pipes.** Pipes split memory access and compute operations into separate kernels, introducing a buffer between the two kernels. We experimented using pipes to pipeline DRAM accesses and computation. With Vitis: To insert pipes via kernel splitting, the control flow must be replicated on the kernel handling external memory accesses. Modifications: We divided the OpenCL kernels into 2 kernels connected through pipes: a *DRAM-read*, and a *compute/write* kernel. Breaking the last kernel into a compute and write kernel was unnecessary since the *DRAM-write* kernel would only operate at the end, resulting in a similar run time with higher resource utilization for an extra kernel. The first kernel reads the required data from DRAM and feeds them to the compute kernel, and the compute kernel performs the computations and then writes the final result to DRAM at the end. Figure 3 shows how the two kernels run and transfer data concurrently, and Listing 1 shows a code sample of the pipes optimization with communicating edge weights across the FIFO buffer instantiated with OpenCL pipes. We used blocking pipes with a capacity of 1024 elements each to avoid any pipeline starvation as this was the batch size used for burst accesses. We observed performance improvements over the baseline OpenCL kernels, but higher resource utilization from replicating the control flow and no performance advantage over the burst optimization (which does not require additional resources).

*3.3.2 General Optimizations.* **Loop Unrolling.** Loop unrolling allows increasing throughput by exposing higher instruction level parallelism deriving from multiple loop iterations at the cost of increased resource utilization. When Vitis is instructed to perform loop unrolling by a factor of $x$ in conjunction with array partitioning by a factor of $x$ or $x/2$, the compiler attempts to partition the buffers created for the burst memory transfers into multiple BRAMs. This reduces the structural conflicts that may arise from concurrent reads from a single BRAM. Because the LT model cannot achieve pipeline parallelism due to data dependencies, we turn to loop unrolling to achieve operator level parallelism. In an effort to improve the accelerator's throughput, we requested the compiler to unroll it by a factor of 16 since the external memory transaction width is 512 bits, which would allow processing 16 32-bit values at a time. We remind that a complete unroll is not possible because the number of iterations is unknown at synthesis time. This unrolling was performed on Line 1 in Algorithm 2 as this loop does not contain any loop-carried dependencies.

**Compute Unit Replication.** We use spatial replication to increase parallelism. We replicated the kernel's compute unit (CU - the generated kernel in OpenCL terminology) by a factor of four or eight. These CUs reside in one Super Logic Region (SLR), so they all share a single DRAM module, reducing the need for duplicate information; however, this causes DRAM contention among the CUs, which can be detrimental to performance.

The performance improvement provided by spatial replication comes with a price in resource utilization. With the LT kernels, external memory stalling increased with CU replication, but not at the levels observed in IC. The reason is that the LT kernel allows allocating more data structures into the on-chip memory, while the IC kernel depends mainly on external memory, as detailed in Fig. 2. This provided moderate performance gains with further CU replication. When compared with the optimizations previously described, the spatial replication of CUs incurs the highest cost in terms of resource utilization on the FPGA. This process requires synthesizing multiple CUs and enqueuing multiple tasks. To achieve CU replication, we use an out-of-order command queue to queue separate tasks which are then assigned by the Xilinx Runtime (the runtime for facilitating the host ↔ FPGA interactions) to each available CU. Work is partitioned in each kernel launch in contiguous blocks to ensure memory is accessed contiguously for bursting. For LT, the host array must be broken up into sub-buffers and assigned to each SLR - this ensures each SLR writes to its own sub-buffer, and the host only has to read one array at the end. Due to the execution time variance of the IC model, each CU was processed independently of one another in its own buffers and managed with our asynchronous handler on the host side. We use OpenCL events to operate asynchronously, allowing memory reads, writes, and kernel enqueues to overlap as long as they are independent.

**Multiple Super Logic Regions.** Xilinx FPGAs can have multiple SLRs, an SLR being a single FPGA die. Since each SLR has its own DRAM module, DRAM contention does not increase with respect to replicating CUs in other SLRs. Up until this point, all CUs were synthesized in a single SLR. We replicated the CUs across all four

**Table 1: SNAP Graphs**

| Graph | # Nodes | # Edges | Avg. Degree |
|---|---|---|---|
| web-BerkStan (BS) | 685,230 | 7,600,595 | 22.18 |
| web-Google (Go) | 875,713 | 5,105,039 | 11.66 |
| soc-pokec-relationships (PR) | 1,632,803 | 30,622,564 | 37.51 |
| wiki-topcats (TC) | 1,791,489 | 28,511,807 | 31.83 |
| com-Orkut (Ok) | 3,072,441 | 117,185,083 | 76.28 |
| soc-LiveJournal1 (LJ) | 4,847,571 | 68,993,773 | 28.47 |

SLRs of the Alveo U250, creating a total of 16 and 32 CUs. Each SLR accesses a different memory bank, so all the data structures need to be replicated across the four off-chip memories, increasing the setup overhead. This optimization further increases parallelism and, ideally, throughput. In Vitis, each global array for each CU must be declared for its respective SLR and DRAM modules to ensure the CU is synthesized on the appropriate SLR. We utilize OpenCL events to ensure the asynchronous execution and memory transfers discussed with CU replication. Figure 3 shows the layout of the CUs, SLRs, and external memory within the FPGA fabric.

**NDRange Kernel.** The Xilinx Vitis platform also supports OpenCL NDRange kernels, which are the ones typically used on GPU platforms. The underlying kernel is the same as shown in Algorithm 2, but with the inner for-loop on Line 1 removed, as $S$, the number of RRR sets to generate, is defined in the NDRange kernel call instead in the form of work-items and work-groups (the CUDA equivalent of threads and thread-blocks). Enabling the burst optimization on NDRange kernels required making the kernels "flexible" (i.e., adding an outer grid-stride loop allowing kernel invocations with flexible thread configurations [15]). We performed this optimization within a flexible NDRange kernel for the LT model, as it contains up to 32k work-items that can be assigned, compared to the 1 to 16 assignable in the IC model. Assigning more work-items to the FPGA led to performance degradation due to the FPGA causing starvation on the CPU cores that were concurrently generating RRR sets.

# 4 EXPERIMENTAL EVALUATION

Our prototypes have been implemented using the OpenCL synthesis flow in the Vitis Unified Software Platform version 2020.2. We targeted a frequency of 300MHz for all the designs. The system used in our experiments is equipped with two Intel Xeon E5-2637 v4 CPUs with 8x32GB 2400MHz DDR4 RAM and the Xilinx Alveo U250 described previously (§ 3). The designs that were synthesized for a single SLR use only the DRAM module (16 GB) directly connected to it. For our real-world heterogeneous system comparison, we have three configurations, all based on the same state-of-the-art parallel framework described in [25, 26]. The base configuration is 7 CPU cores working together along with: (i) another CPU core for a full parallel state-of-the-art CPU configuration in OpenMP [25] using all eight cores, (ii) one CPU core managing the Alveo U250 FPGA Data Center Accelerator Card (using our HLS-accelerated design), and (iii) one CPU core managing an NVIDIA Tesla P100 (12 GB GDDR5) written in CUDA [26]. Energy consumption is measured using RAPL w/perf for CPU, nvidia-smi for GPU, and the Vitis power profiler for FPGA. We use the same input graphs (Table 1) of [25, 26]. These networks are from the SNAP data set collection and contain real-world social networks [22]. We report performance as the average of the results over three consecutive runs. The standard deviation across the three runs for total RRR set generation of the CPU, GPU, and FPGA version remained under 1%, 24%, and 6% of the average for LT, respectively. The GPU observed larger deviation on its runs that took a short amount of time (under 8 seconds on LT), with the remaining runs staying under 7% of the average runtime. For the IC runs, all versions had standard deviations less than 1% of the average runtime for IC for all versions.

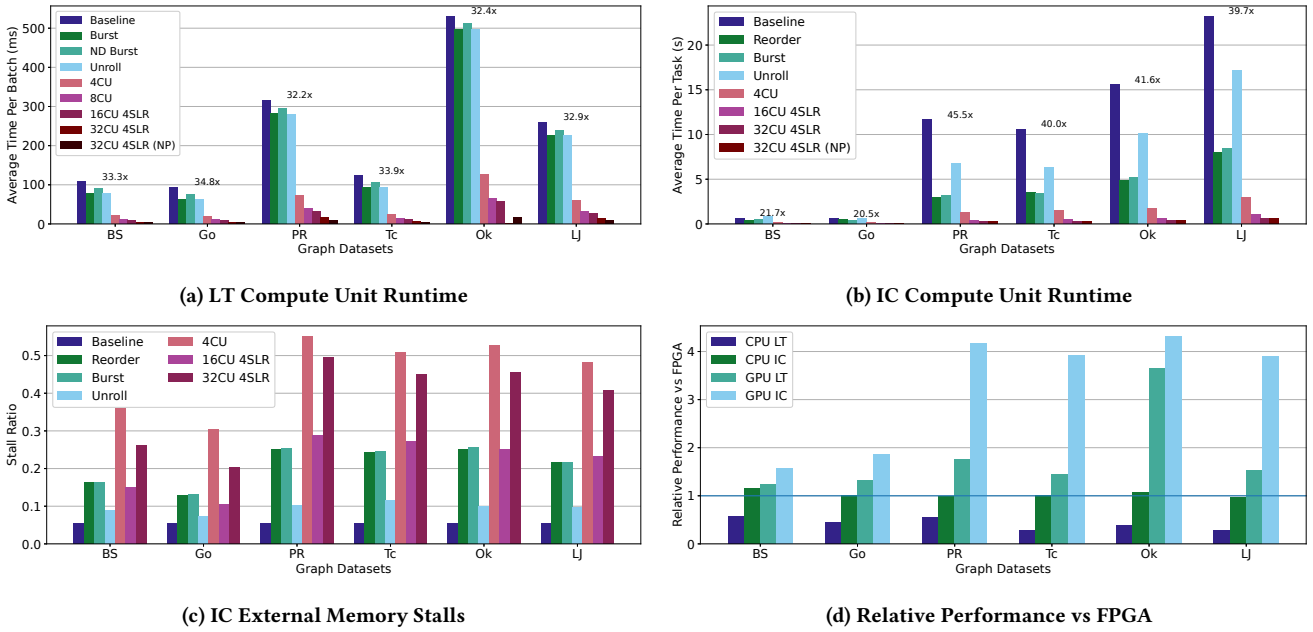
**Table 2: LT (top) and IC (bottom) Kernel Synthesis Results**

| Config | FFs | LUTs | BRAMs | II/Depth | Freq |
|---|---|---|---|---|---|
| Baseline | 11771 (0.37%) | 7843 (0.51%) | 8 (0.35%) | 1/1 | 300.0 MHz |
| Burst | 10158 (0.32%) | 6997 (0.45%) | 18 (0.78%) | 2/2 | 300.0 MHz |
| ND Burst | 10039 (0.32%) | 7676 (0.50%) | 8 (0.35%) | 1/1 | 300.0 MHz |
| Unroll | 40918 (1.30%) | 33980 (2.20%) | 23 (1.00%) | 2/2 | 300.0 MHz |
| 4 CU | 44118 (1.40%) | 32599 (2.11%) | 96 (4.18%) | 3/3 | 300.0 MHz |
| 8 CU | 87985 (2.79%) | 65337 (4.23%) | 192 (8.36%) | 3/3 | 300.0 MHz |
| 16C4S | 173960 (5.52%) | 130412 (8.45%) | 384 (16.72%) | 3/3 | 164.2 MHz |
| 32C4S | 347920 (11.04%) | 260824 (16.91%) | 768 (33.43%) | 3/3 | 161.3 MHz[1] |
| Baseline | 6033 (0.19%) | 4848 (0.31%) | 2 (0.09%) | 142/291 | 300.0 MHz |
| Reorder | 6476 (0.21%) | 5319 (0.34%) | 2 (0.09%) | 5/290 | 300.0 MHz |
| Burst | 25708 (0.82%) | 16724 (1.08%) | 15 (0.65%) | 6/291 | 300.0 MHz |
| Unroll | 141383 (4.49%) | 114304 (7.41%) | 15 (0.65%) | 5/293 | 215.3 MHz |
| 4 CU | 57451 (1.82%) | 39515 (2.56%) | 8 (0.35%) | 6/291 | 300.0 MHz |
| 16C4S | 229080 (7.27%) | 158024 (10.24%) | 32 (1.39%) | 6/291 | 161.3 MHz |
| 32C4S | 454866 (15.54%) | 314347 (22.45%) | 64 (3.84%) | 6/291 | 162.6 MHz[1] |

## 4.1 Linear Threshold Evaluation

Figure 4a reports each LT model optimization's average runtime per batch of RRR Sets to generate (typically 32k). The numbers above the bars denote the speedup of the best performing optimization over the baseline. Burst optimization marginally helped, but spatial parallelism techniques such as **CU Replication** and **Multi SLR** showed the best speedup by increasing external memory bandwidth utilization and increasing the number of available memory banks, respectively. Speedup for 32CU 16SLR (NP) specifically ranged from 32.2× to 34.8× over the baseline, showing good scalability. External memory stalling showed a similar trend as IC's stalling (Figure 4c), but had much lower stall ratios, peaking around 25%. External memory stalling increased with CU replication as DRAM pressure went up. Surprisingly, Multi SLR configurations show a reduction in external memory pressure. This unexpected behavior is due to the 1.88× reduction in clock frequency induced by the intellectual property (IP) block that the synthesis flow includes to collect profiling information (Table 2). The profiling IP performs expensive inter-SLR communication, causing the resulting design to incur timing violations at the 300MHz target and lower the frequency around 160MHz to meet constraints. The consequential frequency drop reduces the rate of external memory transactions and reduces memory pressure. Moreover, we suspect that the profiling IP sends all memory and stall trace data to a single memory bank rather than the originating one. We disabled the profiling IP and noted that the frequency of the synthesized accelerator went up to the 300MHz target in the Multi SLR configuration (denoted as "NP"), but this prevents the collection of memory stall data as a result. Our experiments suggest that the stalling ratio in this configuration should follow the common trend and increase. However, we

[1] Frequency is 300MHz and 291.3MHz when synthesized without profiling, respectively.




**(a) LT Compute Unit Runtime**

**(b) IC Compute Unit Runtime**

**(c) IC External Memory Stalls**

**(d) Relative Performance vs FPGA**

**Figure 4: LT and IC performance data of various optimizations ($k = 100$, $\varepsilon = 0.5$).**

did not observe any performance degradation in this configuration. We predict that, for LT, the stalling ratio was low enough to not cause performance degradation. **ND Burst** showed a slight decrease in performance and increased resource utilization, while **Piping/Channeling** performed the same as Burst but required 66.98% more LUTs, 79.15% more FFs, and 94% more BRAM. This is due to both the memory and compute kernels having to replicate the control flow of the original kernel, as the control flow has memory dependencies relating to values retrieved and computed within the kernel run. **Loop Unrolling** showed no benefit, as Vitis HLS scheduled the loop iterations serially instead of in parallel. This is due to Vitis HLS utilizing the same memory port per unrolled iteration rather than inferring the creation of one port per iteration, causing resource contention. Vitis HLS does not warn the user about reuse during synthesis, but the issue was evident after inspecting the schedule viewer in the GUI and correlating this with the Vitis HLS report showing an unexpectedly low number of instantiated ports in the design. Despite the low resource utilization footprint shown in Table 2, further scaling is limited by hardware constraints (mainly the number of ports assignable to DRAM). Increasing the number of ports can allow for better scalability.

## 4.2 Independent Cascade Evaluation

Figure 4b reports the runtime of IC kernel optimizations per task. **Reorder** and **Burst** optimizations show a speedup from 1.2× to 3.3× over the baseline, with the majority of speedup also from spatial replication. We note that external memory stalling (Fig. 4c) shows a similar trend as LT with respect to spatial parallelism and frequency, but at higher overall stalling. The workload variation and nature of the IC diffusion model also require the FPGA to perform more non-burst DRAM accesses per node (Fig. 2) since the frontier queue and visited array are too large for on-chip memory and

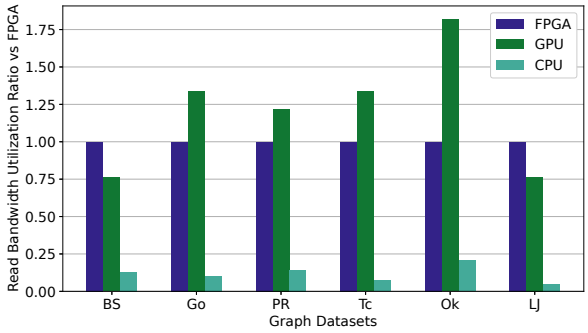



**Figure 5: LT read bandwidth utilization compared to FPGA's 160MHz 32CU 4SLR version, 16CU 4SLR on com-Orkut (Achieved bandwidth / Max theoretical bandwidth)**

must be stored in DRAM, resulting in more costly off-chip accesses. Performance improvements between the ≈160MHz 32CU version and the ≈290MHz 32CU (NP) version are limited, as the stalling ratio for the ≈160MHz version is already high (≈20-50%). **Loop Unrolling** encounters the same issue experienced in the LT results with an added frequency reduction, and shows higher stalling due to more irregular read patterns on the same port compared to a single traversal. Table 2 shows minimal resource utilization, but the memory stalling prohibits further performance improvements, even by utilizing replication.

## 4.3 CPU and GPU Comparison

Figure 4d compares the best FPGA implementation obtained with HLS tools to state-of-the-art CPU and GPU implementations. On LT, the FPGA setup outperforms the CPU setup on all input datasets,

ranging from 1.73× to 3.7× speedup. This is due to the use of spatial parallelism by FPGA to keep DRAM occupied; however, the GPU setup outperforms FPGA's 32CU 4SLR (NP) version between 1.24× and 3.65×. A frequency 8× faster, 10× the peak memory bandwidth, GPU's thousands of threads compared to FPGA's 32 CUs, and the inability of the complex algorithm to be easily pipelined due to a RAW dependency on threshold calculations do not allow the FPGA to reach the same performance. While the GPU also encounters the same RAW dependency, all LT walks occur for each thread on the GPU in parallel while the FPGA, in the 32 CU setup, has to perform up to 1024 LT walks per compute unit. The GPU can also take advantage of its memory subsystem with automatically utilizing 32 to 128-byte cache lines, depending on coalescing behavior. This allows the GPU to easily refer to up to 8-32 cached neighbors while traversing them, depending on memory alignment and number of neighbors, reducing the latency cost between loop iterations of cached neighbors. An implementation similar to this would have to be manually added in the FPGA and isn't inherently supported by the compiler (either automatically or through a templated cache system supplied by the HLS tools). The lack of pipelining forces the FPGA to process each LT walk serially rather than in a pipelined manner, limiting its effectiveness. While there are methods to efficiently perform pipelining on the LT model, it requires major restructuring of the code and a detailed knowledge of the hardware and compiler for the compiler to perform the intended analysis and optimization. Improvement opportunities are discussed in § 5.

The energy consumption savings over the CPU setup ranges from 1.1× to 3.3×. The average power consumption of the entire FPGA setup (7 CPU Cores + 1 FPGA performing sampling + 1 CPU core managing the FPGA) is higher than the CPU-only setup (8 CPU cores performing sampling). Due to this, more power is used overall, but the speedup is enough to garner energy savings. The GPU setup shows the best energy savings, but only because of the large performance increase, as it uses the most power out of the three setups.

For IC, the FPGA's added power consumption vs a single CPU core and its similar performance to the CPU setup keeps the FPGA from outperforming either device in energy savings. The GPU outperforms both FPGA and CPU on all graphs as it utilizes a modified version of the NVIDIA library's BFS, performing a parallel BFS across the GPU with atomic operations (unsupported on Xilinx FPGAs) rather than one BFS per CU/thread. Even the pipelining on the FPGA is not fast enough compared to the GPU's superior memory bandwidth, clock speed, and massive amount of concurrent threads to process each frontier compared to the FPGA's 32 compute units, even when pipelined. Figure 5 shows the utilization of available bandwidth between the CPU and GPU relative to the FPGA implementation on the LT model (with the FPGA's 16 CU 4 SLR version on com-Orkut, and the 32CU 4 SLR 160MHz version for the others, as the 300MHz version was unable to perform profiling). Most notably, the FPGA and GPU trade on utilization, showing the memory bandwidth as a limiting factor for performance. In addition to improved hardware, easy-to-use HLS tool support for improved memory subsystems such as those described in § 5 can help the FPGA compete with the GPU.

# 5 DISCUSSION

We showed that exploiting spatial parallelism and maximizing memory utilization are key performance optimization techniques when dealing with highly irregular workloads, such as our IMM case study. When optimizing the sparser LT model, we found that usage of on-chip memory greatly improved performance, as it halved the number of nonburst memory accesses when compared to the denser IC model. The relatively small execution time of each work-item on LT also enabled better static workload balancing for spatial parallelism. From our experiments, it clearly emerges that the architectural decisions of the current state-of-the-art tools and their optimization pipelines target regular workloads or very small workloads that can fit in on-chip memory. Such design decisions compromise the performance of sparse and irregular applications.

While previous works claim current HLS tools can be used to accelerate irregular applications, many existing approaches impose stringent limitations such as a maximum possible node out-degree, walk size, or graph size to be used (§ 6). In real-world applications, these constraints are impractical. Our IMM use-case shows that graph analytics on real-world inputs require most, if not all, data structures to be located in off-chip memory. In contrast to the current literature, we have shown that traditional HLS optimization methods targeted at improving operation-level parallelism and spatial replication alone are ineffective.

The roofline model from Calore and Schifano [10] shows that better memory technologies are needed to compete with the GPU implementation. In fact, the LT model shows an estimated 1.1 FLOPS per 4-byte word for each memory transaction, and the IC model presents an even lower ratio. While high bandwidth memory (HBM) provides an immediate mitigation (top of the line FPGA boards already support it), the HBM protocol is optimized for large (regular) memory transactions. Therefore, technology geared towards efficiently supporting small memory transactions in HLS flows can become a fundamental break-through in the acceleration of irregular applications. Su *et al.* [31] implemented a random walk for GNN sampling in Verilog HDL using an Alveo U280 equipped with HBM, but was still outperformed by a Tesla V100 by 4.79×-5.07×.

Although HLS tools have greatly improved over time, their user experience is still far from what software compilers offer. As previously mentioned, the current state-of-the-art tools significantly lack details in reporting crucial information to their users. The HLS compiler fails to report when optimizations requested on specific loops via pragma annotations fail or cannot be applied. In fact, to detect that loop unrolling was not successfully applied, we had to carefully inspect the interconnect and the schedule of operations. The compiler and documentation also failed to report that multi-SLR profiling added inter-SLR connections, resulting in a drop in frequency to ≈160MHz. Close inspection of the timing report was required to identify the connections belonging to the profiling IPs.

Commercial HLS tools provides optimizations mainly targeting regular computation and code patterns typical of digital signal processing. Their key optimizations (such as loop unrolling and loop pipelining) focus on extracting instruction level parallelism. Introducing compiler-level optimizations for irregular applications, which traverse pointer-based data structures, is much more complicated due to the lack of effective compiler analysis passes. As

demonstrated by our paper, OpenCL implementations of graph algorithms for GPUs (or their equivalent in CUDA) do not directly translate to good FPGA designs (if they even work, since not all the functionalities of the OpenCL standard are supported, such as atomic operations). The main optimization possible with commercial tools is replicating the compute unit, which is a simple way to extract more task-level parallelism and, consequently, memory level parallelism, which are abundant in these applications. However, there are no optimizations for task-level parallelism, nor to support fine-grained memory accesses. An alternative solution would be to resort to Xilinx's HLS libraries in C++ that can define specialized hardware components (arbitrary precision, stream buffers, and split/merge units [18]) which can then be integrated in the overall architecture, but require explicit use of the functions in the libraries and thus may need significant change to the algorithms. A more effective approach, however, requires employing a different programming model than OpenCL. An example of this would be Svelto [24], which combines support for task level parallelism with specialized hardware templates that are instantiated with pragma annotations in the code typical of parallel programming. The hardware templates connect replicated accelerators to a dynamic task scheduler and a multi-ported memory interface that can support multiple memory transactions in parallel. The approach generates accelerators with multiple contexts that allow tolerating memory access latency while keeping the system utilized.

## 6 RELATED WORK

**Parallel Influence Maximization.** Minutoli *et al.* [25] proposed the first parallel and scalable algorithm for shared memory and distributed memory systems based on the IMM algorithm of Tang *et al.* [33]. The same authors later extended their framework to support multi-GPU systems by introducing a custom dispatching engine that dynamically distributes work units among all the available CPU cores and GPUs. We extend their framework by introducing FPGA workers into the same engine that offloads work to custom accelerators. Our objective is studying the impact of HLS tools and optimizations on a real-world application.

Göktürk and Kaya [14] proposed INFUSER a parallel IM algorithm based on CELF [21]. INFUSER uses fusion and vectorization techniques to increase the parallel efficiency of the algorithm. The approach leverages direction-oblivious pseudo-random number generators to fuse the sampling step and the computation of the influence score. It also attempts at reducing the memory pressure by reusing each edge access for multiple simulations through batching and instruction-level parallelism. Finally, the approach suggests the use of memorization to speed up the computation of the marginal gains in the CELF based algorithm. However, their approach is limited to undirected graphs.

**FPGA Optimizations of OpenCL codes.** Liu *et al.* [23] perform BFS optimizations for OpenCL to FPGA by implementing a five-stage pipeline, memory coalescing, on-board buffering, and level update shifting. While each diffusion model in the IMM algorithm's sampling stage is based on BFS, the proposed local buffer optimizations other than pipelining are incompatible with the diffusion model's implementations. The proposed memory coalescing does not work due to unique graph traversal of each thread (see § 3.2 for more details). Local buffering and level update shifting both rely on large on-chip BRAM memory to store data for the entire graph, but the U250's ~13.5MB per SLR does not support larger graph sizes. Hassan *et al.* [16] explore FPGA specific optimizations for irregular OpenCL applications running on Intel FPGA. Their analysis includes breadth-first search, compares the single work-item and NDRange implementations of this algorithm, and evaluates CU replication, loop unrolling, and the use of BRAM to store the set of active states. They observed noticeable performance improvements only when using a single work-item implementation and storing the active set in BRAM which is infeasible in larger graphs with active sets larger than available BRAM.

**Custom FPGA Implementations of Graph Algorithms.** Several efforts [3, 7, 30, 34, 36, 37] proposed custom FPGA implementations of graph algorithms, often written in Verilog and VHDL. These works focus on simple conventional kernels such as breadth-first search, single source shortest path, weekly connected components, and page rank. We target a larger application leveraging randomization that exacerbates the irregularity of memory accesses.

## 7 CONCLUSION

This paper discusses the challenges and opportunities of using HLS tools to accelerate complex irregular applications. Through Vitis HLS, we designed a custom accelerator for influence maximization algorithms and deployed it on a system that integrates a Xilinx Alveo U250 FPGA board. We explored a wide set of optimizations allowed by the HLS tool. We discussed the challenges of accelerating the emerging class of graph analytics, represented by the IMM algorithm, which use (sub)graph sampling and random walks. To the best of our knowledge, this is the first systematic study of the performance optimizations available within HLS tools on FPGA for this class of graph analytics. We discuss current limitations and future opportunities for improvement and expect our analysis to help guide future HLS development in expanding support for irregular applications.

## ACKNOWLEDGEMENT

The research is supported in part by the U.S. DOE Exascale Computing Project's (ECP) (17-SC-20-SC) ExaGraph codesign center and Laboratory Directed Research and Development Program at Pacific Northwest National Laboratory (PNNL), and by NSF award CNS-1812727 at North Carolina State University.

## REFERENCES

[1] 2021. Vitis Unified Software Platform Documentation: Application Acceleration Development. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1393-vitis-application-acceleration.pdf
[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*. 1–6.
[3] Osama G. Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. 2014. CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search. In *IPDPS '14*. 228–235. https://doi.org/10.1109/IPDPSW.2014.30
[4] Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R Tallent, and Ananth Kalyanaraman. 2020. Vertex reordering for real-world graphs and applications: An empirical evaluation. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 240–251.
[5] Heiko Bauke. 2021. *Tina's Random Number Generator Library.* https://www.numbercrunch.de/trng/trng.pdf

[6] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. arXiv:1903.06697 [cs.DC]

[7] Brahim Betkaoui, Yu Wang, David B. Thomas, and Wayne Luk. 2012. A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration. In *ASAP '12*. 8–15. https://doi.org/10.1109/ASAP.2012.30

[8] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing Social Influence in Nearly Optimal Time. In *Proc. of SODA '14*. SIAM, 946–957. https://doi.org/Portland,Oregon

[9] Tony M. Brewer. 2010. Instruction Set Innovations for the Convey HC-1 Computer. *IEEE Micro* 30, 2 (2010), 70–79. https://doi.org/10.1109/MM.2010.36

[10] Enrico Calore and Sebastiano Fabio Schifano. 2021. Performance assessment of FPGAs as HPC accelerators using the FPGA Empirical Roofline. In *Proc. of FPL '21*. 83–90. https://doi.org/10.1109/FPL53798.2021.00022

[11] Pedro M. Domingos and Matthew Richardson. 2001. Mining the network value of customers. In *Proc. of KDD '01*. ACM, 57–66.

[12] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2–9. https://doi.org/10.1109/IA3.2016.007

[13] Iman Firmansyah, Du Changdao, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. 2019. FPGA-Based Implementation of Memory-Intensive Application Using OpenCL *(HEART 2019)*. ACM, New York, NY, USA, Article 16, 4 pages.

[14] Gökhan Göktürk and Kamer Kaya. 2020. Boosting Parallel Influence-Maximization Kernels for Undirected Networks with Fusing and Vectorization. *CoRR* abs/2008.03095 (2020). arXiv:2008.03095 https://arxiv.org/abs/2008.03095

[15] Mark Harris. 2013. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[16] Mohamed W. Hassan, Ahmed E. Helal, Peter M. Athanas, Wu-Chun Feng, and Yasser Y. Hanafy. 2018. Exploring FPGA-specific Optimizations for Irregular OpenCL Applications. In *ReConFig '18*. 1–8. https://doi.org/10.1109/RECONFIG.2018.8641699

[17] Sitao Huang, Mohamed El-Hadedy, Cong Hao, Qin Li, Vikram S. Mailthody, Ketan Date, Jinjun Xiong, Deming Chen, Rakesh Nagi, and Wen-mei Hwu. 2018. Triangle Counting and Truss Decomposition using FPGA. In *HPEC '18*. 1–7.

[18] Vinod Kathail. 2020. Xilinx Vitis Unified Software Platform. In *Proc. of FPGA '20*, Stephen Neuendorffer and Lesley Shannon (Eds.). ACM, 173–174. https://doi.org/10.1145/3373087.3375887

[19] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence through a Social Network. In *Proc. of KDD '03*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/956750.956769

[20] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2020. GPOP: A Scalable Cache- and Memory-Efficient Framework for Graph Processing over Parts. *TOPC '20* 7, 1, Article 7 (March 2020), 24 pages.

[21] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *KDD*. ACM, 420–429.

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[23] Cheng Liu, Xinyu Chen, Bingsheng He, Xiaofei Liao, Ying Wang, and Lei Zhang. 2019. OBFS: OpenCL Based BFS Optimizations on Software Programmable FPGAs. In *ICFPT '19*. 315–318. https://doi.org/10.1109/ICFPT47387.2019.00056

[24] Marco Minutoli, Vito Giovanni Castellana, Nicola Saporetti, Stefano Devecchi, Marco Lattuada, Pietro Fezzardi, Antonino Tumeo, and Fabrizio Ferrandi. 2022. Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics. *IEEE Trans. Comput.* 71, 3 (2022), 520–533. https://doi.org/10.1109/TC.2021.3057860

[25] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. CuRipples: Influence Maximization on Multi-GPU Systems. In *Proc. of ICS '20*. ACM. https://doi.org/10.1145/3392717.3392750

[26] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. 2019. Fast and Scalable Implementations of Influence Maximization Algorithms. In *CLUSTER '19*. 1–12. https://doi.org/10.1109/CLUSTER.2019.8890991

[27] Marco Minutoli, Prathyush Sambaturu, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyananaraman, and Anil Vullikanti. 2020. PREEMPT: Scalable Epidemic Interventions Using Submodular Optimization on Multi-GPU Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00059

[28] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proc. of FPGA '16*. 111–117.

[29] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proc. of ASPLOS '18*. ACM. https://doi.org/10.1145/3173162.3173180

[30] K. Sridharan, T. K. Priya, and P. Rajesh Kumar. 2009. Hardware architecture for finding shortest paths. In *TENCON '09*. 1–5. https://doi.org/10.1109/TENCON.2009.5396155

[31] Chunyou Su, Hao Liang, Wei Zhang, Kun Zhao, Baole Ai, Wenting Shen, and Zeke Wang. 2021. Graph Sampling with Fast Random Walker on HBM-enabled FPGA Accelerators. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 211–218. https://doi.org/10.1109/FPL53798.2021.00042

[32] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. 2015. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *ICCAD '15*. 78–85.

[33] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence Maximization in Near-Linear Time: A Martingale Approach. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1539–1554.

[34] Matti Tommiska and Jorma Skyttä. 2001. Dijkstra's Shortest Path Routing Algorithm in Reconfigurable Hardware. In *Proc. of FPL '01*. Springer-Verlag, Berlin, Heidelberg, 653–657.

[35] Antonino Tumeo and John Feo. 2015. Irregular applications: From architectures to algorithms [guest editors' introduction]. *Computer* 48, 8 (2015), 14–16.

[36] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Optimizing memory performance for FPGA implementation of pagerank. In *ReConFig '15*. 1–6. https://doi.org/10.1109/ReConFig.2015.7393332

[37] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *FCCM '16*. 103–110. https://doi.org/10.1109/FCCM.2016.35