Edge-Connected Jaccard Similarity for Graph Link Prediction on FPGA

Paul Sathre*, Atharva Gondhalekar†, Wu-chun Feng*†

* Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia, USA
{sath6220,feng}@cs.vt.edu

† Dept. of Electrical and Computer Engineering
Virginia Tech
Blacksburg, Virginia, USA
atharval@vt.edu

Abstract—Graph analysis is a critical task in many fields, such as social networking, epidemiology, bioinformatics, and fraud detection. In particular, understanding and inferring relationships between graph elements lies at the core of many graph-based workloads. Real-world graph workloads and their associated data structures create irregular computational patterns that complicate the realization of high-performance kernels. Given these complications, there does not exist a de facto “best” architecture, language, or algorithmic approach that simultaneously balances performance, energy efficiency, portability, and productivity.

In this paper, we realize different algorithms of edge-connected Jaccard similarity for graph link prediction and characterize their performance across a broad spectrum of graphs on an Intel Stratix 10 FPGA. By utilizing a high-level synthesis (HLS)-driven, high-productivity approach (via the C++-based SYCL language) we rapidly prototype two implementations – a from-scratch edge-centric version and a faithfully-ported commodity GPU implementation – which would have been intractable via a hardware description language. With these implementations, we further consider the benefit and necessity of four HLS-enabled optimizations, both in isolation and in concert — totaling seven distinct synthesized hardware pipelines. Leveraging real-world graphs of up to 516 million edges, we show empirically-measured speedups of up to 9.5× over the initial HLS implementations when all optimizations work in concert.

Index Terms—FPGA, graph, graph analysis, link prediction, Jaccard similarity, heterogeneous computing, performance, portability, productivity, programming language, high-level synthesis

I. INTRODUCTION

Graphs provide a concise and scalable representation for a diverse range of interconnected phenomena. Whether modeling protein interaction networks, financial transactions, artificial intelligence, or social interactions, the fundamental building blocks of graphs — vertices and edges — provide a context to reason about relationships between entities and what those relationships mean. We then use these building blocks to study problems ranging from thousands of entities to billions and beyond, resulting in an extreme computing challenge.

Many graph-based applications require link prediction, i.e., inferring a new edge between two entities in a network [1]. In this work, we focus on a fundamental component of link prediction — computing the intersection of two neighbor lists — which we instantiate using a metric called Jaccard similarity (JS) [2]. For any two sets A and B, the Jaccard similarity between A and B, JS(A, B), is defined as their intersection over their union, as shown in the Equation (1).

\[
JS(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]

In the case of graph datasets, we consider the special case of edge-connected Jaccard similarity (JS): the simultaneous batch computation of |E| JS calculations for all vertex pairs that are connected by an edge. So, for any given pair of vertices connected by an edge, we compute the JS of the pair’s respective neighbor vertices. That is, for a vertex pair \((v_1, v_2)\) with neighborhood sets \(N(v_1)\) and \(N(v_2)\), the Jaccard similarity JS\((v_1, v_2)\) can be computed as shown in Equation (2) below.

\[
JS(v_1, v_2) = \frac{|N(v_1) \cap N(v_2)|}{|N(v_1)| + |N(v_2)| - |N(v_1) \cap N(v_2)|}
\]

Figure 1 provides a visual example of the numerical components underlying the computation of JS for one of a trivial graph’s 14 edge-connected pairs.

Jaccard similarity underpins many application domains, including genome analysis [3], network analysis [4], and graph clustering [5]. Further, we have seen the rise of GPU-based graph processing frameworks, e.g., cuGraph [6] and Gunrock [7], and FPGA frameworks, e.g., Hitgraph [8]. FPGA-based solutions can deliver comparable performance to a GPU while achieving much better performance per watt than comparable GPU devices and frameworks [8], a desirable property for energy-aware HPC. However, most high-performance FPGA implementations are developed in register-transfer level (RTL) hardware description languages (HDL).

HDL-based FPGA development is significantly more difficult and tedious than typical GPU programming approaches such as Nvidia CUDA [9] or the OpenCL [10], [11] standard. As a consequence, high-level synthesis (HLS) approaches for FPGA programming, including C-based OpenCL, have been gaining traction due to their higher level of abstraction, which facilitates higher development productivity. More recently, SYCL [12], [13] has emerged as an even-higher abstraction, leveraging modern C++, in particular move/accessor semantics and lambda expressions. SYCL-based HLS for FPGA is supported by Intel’s DPC++ [14] compiler and triSYCL’s [15]...
In this work, we evaluate the efficacy of SYCL as an HLS language, through a case study of edge-connected Jaccard similarity. We want to consider SYCL’s utility for both (1) developing new FPGA-minded codes and (2) providing ease-of-access to FPGAs to GPU- and CPU-native codes (and their developers). By leveraging SYCL-based HLS, we are able to quickly and productively explore both angles via two different algorithmic approaches, and explore seven different optimization levels of each, which would have otherwise proven to be intractable to explore via HDL. Through this case study we provide the following contributions:

- Implementation of a de novo edge-centric, edge-connected JS kernel for FPGA via SYCL.
- Insights into the faithful translation of a commodity edge-connected JS pipeline from CUDA to SYCL and the performance effects of moving GPU-native code to FPGA.
- A characterization of the efficacy of vendor-specific and standards-driven compiler-based SYCL optimizations to improve performance, applied in isolation and in concert.
- A SYCL instantiation of a hybrid intersection approach to exploit algorithmic and device insights for increased performance and its synergy with other optimizations.

- An empirical evaluation of the above contributions on an Intel Stratix 10 SX FPGA, using real-world graphs of up to 516 million edges.

II. RELATED WORK

Malicic et al. provide a survey of edge-centric and vertex-centric approaches to graph processing on multicore CPUs, along with the impact of preprocessing, data layouts, caching, and non-uniform memory access (NUMA) [16]. Similarly, McCune et al. survey vertex-centric graph processing frameworks [17]. The rest of the section presents work related to set intersection, triangle counting and listing, and graph processing on FPGA via high-level synthesis (HLS).

A. Set Intersection

Ding et al. [18] provide a survey of serial k-way set intersection approaches. The main computational cost of our edge-connected JS implementations consist of a batch of |E| separate 2-way intersections. An alternative approach to compute JS is to express it as a sparse matrix operation [19] and leverage independent thread scheduling on a GPU.

With respect to FPGA, set intersection has been realized for (1) network processing via a sorted set merge and bitwise AND on fixed-width IDs and a parallel tree of processing elements in RTL [20], (2) itemset mining via a sorted set merge of a matrix of elements and a hash table in RTL [21], (3) packet classification via a systolic array that compares rules between two sets with a tree architecture for aggregating intersections between additional sets in RTL [22], and (4) graph processing via merge and binary search-based intersections for sorted sets [23], where performance is evaluated on a cycle-level simulator called Sniper [24], resulting in moderate speedups.

B. Triangle Counting and Listing

Triangle counting (TC) and triangle listing (TL) globally count or list, respectively, all the triangles (i.e., sets of three mutually-connected vertices) contained in a graph. Edge-connected JS is isomorphic to TL. Rather than record each triangle, instead each triangle contributes to the numerators (neighbor intersections) of the three edge-connected pairs it consists of; the third vertex is necessarily a member of both endpoints’ neighbor lists. For a summary of the distinction between TC and TL, we direct the reader to [25].

Exact TC makes up part of the Static Graph Challenge: Subgraph Isomorphism [26]. Huang et al. [27] provide an FPGA implementation of TC, which showcases superior performance-per-watt versus an existing GPU solution [28]. However, though TC can be solved using an intersection kernel — essentially computing the TL and summarizing — as detailed in [25], it is also commonly solved by cheaper techniques such as matrix formulations that are incompatible with the local context required by edge-connected JS/TL. Green et al. [29] and Pearson et al. [30] implement TC for Intel Knights Landing and GPU, respectively, both using a combination of sorted set merge- and binary search-based intersections, which inspire one of our optimizations.
C. HLS-based FPGA Graph Processing

Huang et al. [27] uses Vivado HLS for C-based synthesis. Castellana et al. [31] and Minutoli et al. [32] leverage an HLS approach called Bambu [33], showcasing support for task-level parallelism and synthesis of OpenMP annotated TC to a Xilinx Virtex-7, respectively.

III. EDGE-CENTRIC JACCARD SIMILARITY

A typical graph processing workload requires traversal/inspection of one or more vertices. This can be implemented in two ways: edge-centric (EC) or vertex-centric (VC). Edge-connected JS requires knowledge of both source and destination vertices. A VC approach would require an irregular access to identify the neighbor(s) of the source vertex – the approach taken by the hierarchically-parallel cuGraph implementation discussed in Section IV. Prior work shows that such irregular accesses are inefficiently handled by conventional FPGA memory controllers [34]. In comparison, EC approach eliminates this access by using edge arrays to provide \(O(1)\) lookup of the vertex pairs that form the edges [35], [8]. EC methods have been explored across diverse architectures such as multicore CPUs [35], [16], GPUs [36], [37], [38], and FPGAs [8], [39]. Jia et al. [38] show that EC achieves better load balance and throughput than VC on Nvidia GPUs, but at the cost of more memory accesses. Zhou et al. [8] present an EC framework for FPGA-based graph processing, which delivers up to 5.3× and 1.8× throughput improvement, respectively, for sparse matrix-vector multiplication and PageRank versus other irregular state-of-the-art FPGA frameworks.

Given these inspirations, we sought to create an EC implementation of edge-Connected JS, resulting in Algorithm 1. Our wrapping mini-application utilizes the standard sorted compressed sparse row (CSR) format to represent the graph, which provides source-major sorting of the edge-list to support an inline binary search for intersections (Algorithm 2). Binary search bounds the worst-case performance on real-world graphs with extremely-connected vertices, such as the five used in Section VI with degree ranges from 11.5k to nearly 1.3M. The EC algorithm takes “src” and “dest” vectors as inputs; the destination vector is identical to the CSR “index” vector, and the source vector is trivially constructed offline on the FPGA.

IV. cuGraph Jaccard Similarity

To evaluate our de novo FPGA-minded EC implementation, we compare it to the standard cuGraph [6] implementation from the RAPIDS GPU data science framework [40]. For consistency, we developed a common command-line interface, and incorporated both sets of kernels in the same hardware design. We first describe the the cuGraph edge-connected JS pipeline and then briefly outline the steps necessary to use the kernels outside the RAPIDS environment and translate them faithfully\(^1\) to SYCL.

\(^1\)Faithful translation — i.e. without “de-GPU-ifying” kernels or thread configurations — was chosen to capture any stumbling points that might lurk for others who, enticed by increased programmability and availability of FPGAs, seek to port other GPU-native codes via SYCL.

A. RAPIDS cuGraph Implementation

RAPIDS contains a number of different GPU-accelerated libraries to support heterogeneous data science. We refer the reader to the RAPIDS website [40] for a detailed listing of capability and focus on the JS kernel pipeline. The JS kernels are written with a Python frontend, middle layers in templated C++ for generality, and low-level kernels in CUDA C++.

The JS implementation is based on [41] and decomposes the operation into a pipeline of three kernels that, like ours, operate on an entire graph at once. The graph is provided to the kernels as a bidirectional CSR matrix with sorted neighbor lists. Optional support for weighted vertices and “pair-list” JS is elided for brevity; we focus on the default unweighted, edge-connected JS, which matches our EC implementation.

The first RowSum kernel counts the (weighted) neighbors of each vertex, i.e., Equation (3),

\[
    n_v = |N(v)| : \forall v \in V
\]

and stores them in the NeighborSum buffer.\(^2\) Next, the In-

\(^2\)Absent vertex weights, this value could be computed in situ by the subsequent Intersection kernel, but we evaluate the existing solution “as is”
IntersectionWeight buffer is zero-filled\(^3\) to prepare for atomic accumulations in the next intersection kernel,\(^4\) which computes the intersection size of all edge-connected pairs in Equation (4)

\[
i_{v_1,v_2} = |N(v_1) \cap N(v_2) : \forall (v_1,v_2) \in E \quad (4)
\]

using a 3D thread block and is reproduced as pseudocode in Algorithm 3 for reference. The primary distinctions between Algorithm 1 and Algorithm 3 are:

- Lines 1-2: the hierarchically-parallel mapping of source and destination vertices onto the Z and Y thread IDs
- Lines 10-18: the collaborative X-dimension intersection with necessary atomic intersection increment
- Lines 9 & 12-14: accommodations for weighted vertices \(^5\)
- Line 9: storing \(n_{v_1} + n_{v_2}\) in global memory for the final union+JS kernel

The final embarrassingly-parallel 1D kernel then computes the UnionWeight denominator in Equation (5) and final JS score in Equation (6).

\[
\begin{align*}
    n_{v_1,v_2} &= n_{v_1} + n_{v_2} - i_{v_1,v_2} : \forall (v_1,v_2) \in E \quad (5) \\
    j s_{v_1,v_2} &= \frac{i_{v_1,v_2}}{n_{v_1,v_2}} : \forall (v_1,v_2) \in E \quad (6)
\end{align*}
\]

Algorithm 3: Vertex-Centric Intersection

**Input:** graph \(G(V,E)\) in CSR format:
- offsets\([V]\), indices\([E]\),
- \(vWeights\([E]\) // Optional vertexWeights

**Output:** IntersectionWeights\([E]\), SumWeight\([E]\)

**Data:** \([E]\)

```
foreach source vertex \(s \in V\) do // Par. in 0th-dim
    foreach dest \(d\) from \(s\) do // Par. in 1st-dim
        \(n_s = \) offsets\([s+1]\)-offsets\([s]\)
        \(n_d = \) offsets\([d+1]\)-offsets\([d]\)
        if \(n_s < n_d\) then // Smaller ref vertex
            ref = s, cur = d
        else
            ref = d, cur = s
        SumWeight\([s,d] = NeighborSum\([s]\)+NeighborSum\([d]\)
        foreach dest \(i\) from \(ref\) do // Par. in 2nd-dim
            refCol = indices\([i]\)
            if weighted then // removed constexpr
                refVal = vWeights\([refCol]\]
            else
                refVal = 1
            match = BinSearch\(\) offsets, indices, cur
            if match == true then // Atomic update
                atomicAdd\(\) IntersectionWeights\([d], refVal\)
```

**Table I:** Analogous API elements from CUDA to SYCL encountered in the Jaccard Similarity host and device code.

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device pointer</td>
<td>Buffer</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>Buffer accessor</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>Queue submission lambda</td>
</tr>
<tr>
<td>Device Buffer</td>
<td>Device (GPU) Code</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td><strong>global</strong> Function</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>Lambda expression or Functor Class</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>grid / block</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>nd_range / work_group</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>slowest-varying dimension</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>fastest-varying dimension</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>2D: Y, 3D: Z</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>2D: 0, 3D: 0</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>2D: X, 3D: X</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>2D: 1, 3D: 2</td>
</tr>
</tbody>
</table>

Fundamentally, SYCL’s model is asynchronous heterogeneous offload, much like CUDA or OpenCL, in which device kernels are explicitly enqueued on one or more offload devices (FPGA, GPU, CPU, etc.) and coordinated by the host CPU. The original SYCL [12] standard shared substantial conceptual overlap with the lower-level OpenCL [10], [11] standard (which in turn overlapped significantly with CUDA [9]); however, the SYCL 2020 specification [13] has diverged from the more restrictive OpenCL model. Further, its basis in modern C++ standards makes for an easier transition from the “C with objects” style of typical CUDA C++ applications compared to OpenCL’s C99-based device language.

Beginning from the out-of-tree cuGraph pipeline, the kernels, data structures, and mini-app were manually ported to the semantically-equivalent SYCL. Further, as the FPGA has limited silicon area and must be offline-compiled, templates were explicitly-specialized to only utilize 32-bit data types to avoid combinatorial template explosion.

In this work, we target the Intel Stratix 10 Programmable Acceleration Card [44] and thus utilize Intel’s Data Parallel C++ [14] SYCL compiler from the oneAPI [45] environment. As both CUDA and SYCL are single-source C++, much of the host (CPU) and device (GPU/FPGA) code overlapped; we needed only to replace the unique CUDA syntax, data types, and runtime functions, the most important of which are summarized in Table I.

1) Host (CPU) Code: We translate CUDA’s device pointers and explicit transfers to SYCL “buffers” and allow the SYCL runtime’s implicit dependency graph to manage data migration to/from the device. We consider only kernel runtime, which

\(^3\)using a trivial Thrust [42] kernel

\(^4\)As of the 2022.1 oneAPI Add-on for FPGA [43] used in this work, Intel supports 32-bit atomic add via SYCL only for integers, so to preserve the floating point weights used in the original CUDA, floating point atomic add was emulated via a typical “type cast with compare-and-exchange” loop

\(^5\)we use a C++17 constexpr, to compile 12-14 out, whereas the original CUDA does not
is queryable from the SYCL profiling API without the need to manually synchronize and separate transfer overheads from kernel execution. Kernel launches are converted to SYCL's lambda-based queue submissions, replacing CUDA grid/block specification with the corresponding SYCL "nd_range" using the same number and organization of threads, and using buffer *accessors* to provide arguments to the kernel and to inform the SYCL runtime what data must be resident on the device and quiescent before launch.

2) **Device Code:** SYCL kernels may be written as either C++ lambda *expressions* or *functor classes*, which function similarly. We utilized functors to retain the existing code structure, modularity, and clarity. The majority of both CUDA and SYCL kernels are standard C elements which remain unmodified, but thread identifiers and device API functions are mapped to analogous functions. The CUDA kernels use a mix of 1D, 2D, and 3D invocations, a potential performance tripping-point. When converting to SYCL, the order of these dimensions (fastest- vs slowest-varying) is intentionally *inverted* from CUDA/OpenCL norms, as noted in Table I for consistency with modern C++'s multi-dimensional array syntax. To emphasize, other than the emulated floating-point calculation, enabled by the HLS approach.

### V. Optimizations

Optimizing algorithms for the FPGA is a notoriously tricky process, and existing literature largely performs it at the RTL level. RTL provides the finest-grained control over synthesized hardware, but requires a monumental development investment for even a single iteration of a complex algorithm. Fortunately, with HLS approaches frequently come “simple” optimization techniques in the form of attributes, pragmas, language extensions, and compiler flags with which to better steer synthesis. Further, HLS’s ease of use returns development time to be spent on “complex” algorithmic refactoring and innovation.

In this work we evaluate the mapping of JS to the Intel Stratix 10 FPGA, and thus follow the Intel FPGA programming Guide’s [46] recommended standard and vendor-extension “simple” optimizations. To our baseline implementation we add variants with three such optimizations in isolation and a fourth variant with “complex” refactored intersection calculation, enabled by the HLS approach.

Specifically, we utilize the following optimizations in both EC and cuGraph implementations (summarized in Table II):

- **Restrict:** Intel extension to inform the synthesis tools that all buffers used by a kernel occupy disjoint memory
- **GroupSize:** Standard SYCL kernel attribute to inform the synthesis tools the guaranteed number of threads in each dimension of a collaborative thread *work group.*
- **Unroll4:** The standard C *unroll* pragma, to transform loop iterations into chunks of sequential code. We set it to 4, as a middle ground between the sparsest graphs, which rarely need more iterations, and the densest ones which might benefit from more unrolling.

### A. Hybrid Intersection

The $|E|$ batched intersections of vertex pairs is the dominant cost to computing edge-connected JS. Each pair’s cost scales relative to the size of the neighbor lists ($n_{ref}$ and $n_{cur}$), which cannot be known a priori. We address a broad range of graph complexities (see Sec. VI-A), and thus must balance between highly-connected worst-case performance and the FPGA device’s preferred algorithmic strategy.

Initially, both EC and cuGraph implementations utilized a looped binary search (Algorithm 2) which performs a series of $\log(|N_{cur}|)$ searches for the less-connected (ref) vertex’s neighbors among the more-connected (cur) vertex’s. The logarithmic term provides a tight upper bound when $|N_{cur}|$ is large relative to $|N_{ref}|$. Our EC implementation evaluates one vertex pair per thread, and cuGraph parallelizes the search over $p = 8$ threads, with costs in Equations (7) and (8), respectively.

$$\Theta(|N_{ref}| \times \log(|N_{cur}|))$$

(7)

$$\Theta\left(\frac{|N_{ref}| \times \log(|N_{cur}|)}{p}\right)$$

(8)

This approach provides decent algorithmic complexity but is effectively random access. Both FPGA and GPU rely on wide memory buses and overlapped communication to hide load/store latency, and random access inhibits effective automatic prefetching, inducing stalls.

As such, we implement Algorithm 4, an alternative with a guaranteed sequential access pattern, based on sorted set merge, similar to [27]. Two pointers *traverse* ref and cur’s neighbor lists, advancing the lesser on mismatch, and advancing both on a match. If either list runs out, the algorithm terminates, as nothing matches the empty set. This algorithm has upper and lower bounds, noted in Equations (9) and (10)

$$O(|N_{ref}| + |N_{cur}|)$$

(9)

$$\Omega\left(\min\{|N_{ref}|, |N_{cur}|\}\right)$$

(10)

When $n_{ref}$ and $n_{cur}$ are similar in size, clearly sorted set merge is cheaper than binary search. Further, strict sequential access can support prefetch and more efficient use of the wide memory bus. However, without the strongly-bounded worst-case performance of binary search, the Sorted Set Merge could not cope with the high-degree-range real-world data we use.
VI. ANALYSIS

HLS provides a more productive route to programming FPGAs, but current environments rely heavily on annotations to bridge the RTL performance gap. Though SYCL's modern syntax and advanced runtime model have promise, current toolchains require similar annotation, for which we sought to quantify the importance. Further, SYCL-based HLS allowed us to rapidly generate seven different hardware designs to empirically measure the effect of the optimizations from Sec. V, both in isolation and in concert.

A. Datasets

We evaluate the baseline and six optimized EC and cuGraph implementations with real world graphs from the SuiteSparse matrix collection [47], [48]. Summary statistics are provided in Table III, but briefly they span:

- Vertices: 416K to 171M
- Bidirectional Edges: 3.08M to 516M
- Average Degree: 2.11 to 161
- Degree Range: 8 to 1.29M
- Standard Deviation of Degree: 0.48 to 1357
- Gini Inequality Index: 0.055 to 0.759

To the best of our knowledge, these are some of the largest and most-varied graphs to date for which edge-connected JS has been empirically evaluated on a single FPGA hardware device. Unfortunately we cannot fairly compare our data center FPGA to the nearest related efforts, as [27]7 uses a low power system-on-a-chip and [23]8 utilizes a simulated architecture.

To prepare data, the MatrixMarket-formatted inputs are converted to unweighted sorted CSR offline as follows:

- Drop any edge weights
- Remove any unconnected vertices, without ID relabeling
- Remove any self-edges
- Generate reverse edges to create bidirectional edge pairs, deduplicating any repeats from directed inputs
- Primarily sort edges by increasing source vertex ID
- Secondarily, for each source vertex, sort adjacency list by increasing destination vertex ID

As noted in Section III, the additional source edge list for the EC implementation is constructed in situ on the FPGA from the sorted CSR’s destination edge list. Neither format conversion is included in performance measurements.

B. System Configuration

We utilize an Intel Stratix 10 SX FPGA mounted on their consumer PCIe Programmable Acceleration Card [44] with 32GB DDR4. Hardware synthesis is performed using DPC++ [14] from the 2022.1.0 oneAPI FPGA Toolkit add-on [43], which also provides the board support package to support SYCL and OpenCL workflows. The host machine has dual Intel Xeon Gold 5217 CPUs and 384GB RAM.

C. Performance

Our absolute performance metric is vertex pairs processed per second (PPS): the measured throughput of the entire batch of |E| pair-wise JS computations for a given graph input. This is computed as $PPS = \frac{\text{BidirectionalEdgeCount}}{2 \times \text{totalKernelRuntime}}$. using edge counts from Table III and the sum of kernel execution times for all pipeline stages (1 for EC, 4 for cuGraph)—data transfers and graph preprocessing are not included. Absolute performance for all 18×7 graph/optimization-variant combinations discussed in Sec. V are provided in Figure 2. Clearly our EC implementation drastically outperforms the ported cuGraph in all but a few circumstances, which is

---

6Anecdotally edge-centric wikipedia-20070206 sustained a roughly 5× performance loss vs. Baseline, and the cuGraph port which typically takes minutes was terminated after > 24 hours.

7Triangle counting that shares the PA and CA road networks

8JS on graphs of < 100K edges, and k-clique on shared soc-orcut

---

TABLE III: Graph datasets used to evaluate our JS approaches, sorted by average degree with minima and maxima in bold.

<table>
<thead>
<tr>
<th>Graph Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Avg. Degree</th>
<th>Range</th>
<th>Std. Dev.</th>
<th>Gini Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmer_A2a</td>
<td>171M</td>
<td>361M</td>
<td>2.11</td>
<td>39</td>
<td>0.56</td>
<td>0.055</td>
</tr>
<tr>
<td>europe_owsm</td>
<td>50.9M</td>
<td>108M</td>
<td>2.12</td>
<td>12</td>
<td>0.48</td>
<td>0.085</td>
</tr>
<tr>
<td>road-road_USA</td>
<td>23.9M</td>
<td>57.7M</td>
<td>2.41</td>
<td>8</td>
<td>0.93</td>
<td>0.211</td>
</tr>
<tr>
<td>road-roadnet-CA</td>
<td>1.96M</td>
<td>5.52M</td>
<td>2.82</td>
<td>11</td>
<td>0.99</td>
<td>0.185</td>
</tr>
<tr>
<td>road-roadnet-PA</td>
<td>1.09M</td>
<td>3.08M</td>
<td>2.83</td>
<td>8</td>
<td>1.02</td>
<td>0.188</td>
</tr>
<tr>
<td>delanany_n24</td>
<td>16.8M</td>
<td>101M</td>
<td>6.00</td>
<td>23</td>
<td>1.34</td>
<td>0.122</td>
</tr>
<tr>
<td>circuit5M</td>
<td>5.56M</td>
<td>54.0M</td>
<td>9.71</td>
<td>1.29M</td>
<td>1357</td>
<td>0.577</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>4.85M</td>
<td>85.7M</td>
<td>17.7</td>
<td>20.3K</td>
<td>52.0</td>
<td>0.711</td>
</tr>
<tr>
<td>wikipedia-20070206</td>
<td>3.57M</td>
<td>84.8M</td>
<td>23.8</td>
<td>188K</td>
<td>255</td>
<td>0.759</td>
</tr>
<tr>
<td>GL7d19</td>
<td>1.96M</td>
<td>74.6M</td>
<td>38.2</td>
<td>134</td>
<td>6.73</td>
<td>0.088</td>
</tr>
<tr>
<td>dieFilterV2real</td>
<td>1.16M</td>
<td>47.4M</td>
<td>40.9</td>
<td>104</td>
<td>16.1</td>
<td>0.201</td>
</tr>
<tr>
<td>sc-idoor</td>
<td>952K</td>
<td>41.5M</td>
<td>43.6</td>
<td>76</td>
<td>14.8</td>
<td>0.183</td>
</tr>
<tr>
<td>stokes</td>
<td>11.4M</td>
<td>516M</td>
<td>45.1</td>
<td>1728</td>
<td>61.8</td>
<td>0.392</td>
</tr>
<tr>
<td>sc-mdsoor</td>
<td>416K</td>
<td>18.8M</td>
<td>45.1</td>
<td>76</td>
<td>13.7</td>
<td>0.166</td>
</tr>
<tr>
<td>ca-countours-dblp</td>
<td>540K</td>
<td>30.5M</td>
<td>56.4</td>
<td>3298</td>
<td>66.2</td>
<td>0.544</td>
</tr>
<tr>
<td>soc-orcut</td>
<td>3.00M</td>
<td>213M</td>
<td>71.0</td>
<td>27.5K</td>
<td>140</td>
<td>0.558</td>
</tr>
<tr>
<td>hollywood-2009</td>
<td>1.14M</td>
<td>113M</td>
<td>98.9</td>
<td>11.5K</td>
<td>272</td>
<td>0.750</td>
</tr>
<tr>
<td>HV15R</td>
<td>2.02M</td>
<td>325M</td>
<td>161</td>
<td>491</td>
<td>47.8</td>
<td>0.155</td>
</tr>
</tbody>
</table>
attributable to the underwhelming mapping of the single-instruction, multiple-thread (SIMT) GPU paradigm onto the heavily-pipelined FPGA architecture. This represents a performance pitfall that may, despite improved programmability, still discourage FPGA adoption by GPU-native developers, if insufficient care is taken to both translate and refactor existing GPU-native codebases. However, more important is the clear effect of the optimizations across all sizes, sparsities, and inequalities, confirming their importance regardless of workload.

However, the optimizations are clearly not equally beneficial, neither relative to each other nor across the varied inputs. To explore further, we examine their speedup over baseline \((\text{Speedup} = \frac{\text{Baseline MPPS}}{\text{Optimized MPPS}})\), presented in Figure 3. Not all optimizations were helpful in isolation, a mix of ten EC and cuGraph trials from the sparse end of the input spectrum had a more than 5% slowdown, mostly due to excessive unrolling or the added complexity of the Hybrid Intersection. Fifty-nine trials, covering the full range of densities from kmer_A2a to HV15R had marginal change between ±5%. Restrict, Unroll, and Hybrid Intersection make up the remaining seventy-five isolated trials with greater than 5% speedup, with greater effects as both density and size of the input graph increase. Hybrid Intersection on EC was the only optimization to contribute more than \(2 \times \) speedup in isolation, and did so on eleven trials, maxing out at \(3.55 \times \) on HV15R.

D. Synergistic Effects of Optimization

In search of further speedup, we sought to understand how the optimizations performed in concert, similar to earlier GPU studies undertaken by [49], [50]. These results are also present...

Fig. 2: Absolute performance in terms of Millions of vertex pairs processed per second (MPPS) of Edge-Centric (EC) and cuGraph Edge-connected JS on Intel Stratix 10 SX FPGA hardware, with optimizations applied in isolation and in concert. Sorted by increasing graph average degree.

Fig. 3: Speedup relative to “Baseline” implementation of Edge-Centric (EC) and cuGraph Edge-connected JS on Intel Stratix 10 SX FPGA hardware, with various optimizations applied in isolation and in concert.
in Figures 2 and 3, as the +AllSimple and +AllOpts bars. Conceptually, +Restrict and +GroupSize should simplify hardware, by obviating some memory- and thread-safety circuitry, respectively. Conversely, +Unroll4 and +HybridIsect utilize increased area and complexity to improve pipeline depth and reduce asymptotic costs. When combined they could either:

- (worst case) undermine each other’s performance gains, resulting in reduced speedup or slowdown
- (acceptable) improve performance, but by less than the product of their individual speedups
- (best case) synergize to exceed their product

To differentiate outcomes, we began by projecting the expected maximum “acceptable” speedup by multiplying the speedups from each of the optimizations that compose the two combined variants. These are plotted against the empirically-measured results for the 18 input graphs in Figure 4. Our EC code was able to strictly outperform the projected speedup in all cases, encouraging an “all of the above” approach to HLS optimization. The observed speedups of the cuGraph port were muddier\(^9\); some outperformed and some underperformed, but only three trials actually lost a marginal amount of speedup.\(^{10}\) For all four optimizations in concert, the minimum combined speedups are: EC \texttt{kmer\_A2a} at 2.32× and cuGraph \texttt{delaunay\_n24} at 1.23×; the maximums are: EC \texttt{sc-msdoor} at 9.50× and cuGraph \texttt{GL7d19} at 2.87×.

VII. FUTURE WORK

The increasing use of FPGAs as general-purpose accelerator devices, combined with the growth of vendor and research HLS environments present an exciting opportunity to apply modern software development techniques to the design of flexible and power-efficient hardware platforms. Specific to edge-connected JS, opportunities for extension include (1) manual vectorization and/or compute unit replication to improve parallelism, (2) task-parallel approaches similar to [32], (3) “binning” JS pairs to separate concurrent hardware pipelines, inspired by [29], (4) caching techniques to improve pipeline performance, (5) fixed-precision/integer-based refactoring, and (6) incorporating alternative 2-way intersection algorithms. To contribute to the HLS-based graph processing space, we can implement TC and \textit{k-truss search} to compete in the \textit{Static Graph Challenge} [26], including possibly translating and contrasting implementations from cuGraph. Finally, SYCL provides software portability to GPU and CPU, promoting multi-device performance portability studies.

VIII. CONCLUSION

In this work, we characterize the effectiveness of a SYCL-based HLS workflow for FPGA computing via the study of two implementations of edge-connected Jaccard similarity: a \textit{de novo} edge-centric parallelization and a hand-translated GPU-centric pipeline. HLS-based approaches complement traditional RTL-based workflows by accelerating software design. Single-source SYCL leverages modern C++ to simplify the HLS experience versus prior OpenCL-based solutions.

The development efficiency of SYCL promoted rapid exploration of the optimization space. Seven variants of the two implementations are synthesized and evaluated on Intel Stratix 10 FPGA hardware, to explore the effect of four optimizations, both in isolation and in concert. We have demonstrated \textit{greater than multiplicative} speedups of up to 9.5× when multiple optimizations are combined. This effect is consistently observed for our \textit{de novo} edge-centric implementation on eighteen graphs of up to 516 million edges, across an extreme range of graph densities and irregularities.

ACKNOWLEDGEMENTS

The work detailed herein has been supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC). The authors are grateful for access to the Intel Devcloud’s Stratix 10s for cross-examination of some results.


