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
atharva1@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 $9.5\times$ 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 $\mathbf{A}$ and $\mathbf{B}$, the Jaccard similarity between $\mathbf{A}$ and $\mathbf{B}$, $\mathbf{JS(A,B)}$, is defined as their intersection over their union, as shown in the Equation (1).

$$\mathbf{JS(A,B)} = \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|} \tag{1}$$

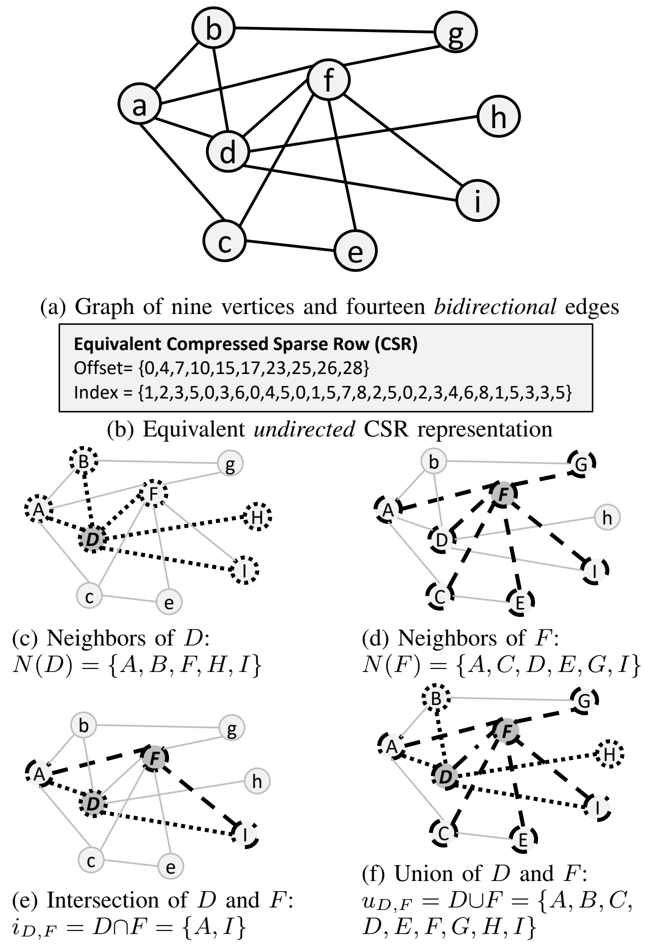
In the case of graph datasets, we consider the special case of **edge-connected** Jaccard similarity (JS): the simultaneous batch computation of $|E|$ JS calculations for all vertex pairs that are connected by an edge. So, for any given pair of vertices connected by an edge, we compute the JS of the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

$$\mathbf{JS(v_1,v_2)} = \frac{|\mathbf{N(v_1)} \cap \mathbf{N(v_2)}|}{|\mathbf{N(v_1)}| + |\mathbf{N(v_2)}| - |\mathbf{N(v_1)} \cap \mathbf{N(v_2)}|} \tag{2}$$

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]



(a) Graph of nine vertices and fourteen *bidirectional* edges

Equivalent Compressed Sparse Row (CSR)
Offset= {0,4,7,10,15,17,23,25,26,28}
Index = {1,2,3,5,0,3,6,0,4,5,0,1,5,7,8,2,5,0,2,3,4,6,8,1,5,3,3,5}

(b) Equivalent *undirected* CSR representation

(c) Neighbors of D: $N(D) = \{A, B, F, H, I\}$

(d) Neighbors of F: $N(F) = \{A, C, D, E, G, I\}$

(e) Intersection of D and F: $i_{D,F} = D \cap F = \{A, I\}$

(f) Union of D and F: $u_{D,F} = D \cup F = \{A, B, C, D, E, F, G, H, I\}$

Fig. 1: Components of the Edge-Connected Jaccard similarity. Capitalized vertices indicate contribution to each component.

experimental route to Xilinx.

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

Malicevic 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.

Algorithm 1: Edge-Centric Jaccard Similarity

```
Input  : graph G(V, E) in sorted CSR+edge-list format:
         source($|E|$), dest($|E|$), sourceOffsets($|V|$)
Output: JaccardWeights($|E|$)
Data: $|E|$
foreach edge $e$ from $E$ do                    // Par. in 0th-dim
    s = source[e], d = dest[e]
    // Count neighbors
    $n_s$ = sourceOffsets[s+1]- sourceOffsets[s]
    $n_d$ = sourceOffsets[d+1]- sourceOffsets[d]
    if $n_s < n_d$ then                          // Smaller ref vertex
        ref = s, cur = d
    else
        ref = d, cur = s
    foreach dest $i$ from $ref$ do               // Intra-Thread
        refCol = dest[i]
        match = BinSearch(sourceOffsets, dest, cur)
        if match == true then
            JaccardWeights[e] += 1
    JaccardWeights[e] = JaccardWeights[e] / (($n_s$ + $n_d$) -
      JaccardWeights[e]);
```

Algorithm 2: BinSearch (inlined in practice)

```
Input  : graph G(V, E) in CSR format:
         offsets($|V|$), indices($|E|$), ‘cur’: integer vertex ID
Output: ‘match’: boolean indicating presence of ‘cur‘
left = offsets[cur], right = offsets[cur+1]-1
while $left < right$ do
    middle = (left + right) / 2
    curCol = indices[middle]
    if $curCol > refCol$ then
        right = middle -1
    else if $curCol < refCol$ then
        left = middle +1
    else
        match = middle
        break
```

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),

$$n_v = |N(v)| : \forall v \in V \tag{3}$$

and stores them in the *NeighborSum* buffer.[2] Next, the *In-*

[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.

[2] Absent vertex weights, this value could be computed *in situ* by the subsequent Intersection kernel, but we evaluate the existing solution “as is”

tersectionWeight 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)

$$i_{v_1,v_2} = |N(v_1) \cap N(v_2)| : \forall(v_1, v_2) \in E \qquad (4)$$

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).

$$u_{v_1,v_2} = n_{v_1} + n_{v_2} - i_{v_1,v_2} : \forall(v_1, v_2) \in E \qquad (5)$$

$$js_{v_1,v_2} = \frac{i_{v_1,v_2}}{u_{v_1,v_2}} : \forall(v_1, v_2) \in E \qquad (6)$$

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: $|E|$
1  foreach source vertex $s \in V$ do          // Par. in 0th-dim
2      foreach dest $d$ from $s$ do            // Par. in 1st-dim
3          $n_s$ = offsets[s+1]- offsets[s]
4          $n_d$ = offsets[d+1]- offsets[d]
5          if $n_s < n_d$ then                 // Smaller ref vertex
6              ref = s, cur = d
7          else
8              ref = d, cur = s
9          SumWeight[s,d]=NeighborSum[s]+NeighborSum[d]
10         foreach dest $i$ from $ref$ do      // Par. in 2nd-dim
11             refCol = indices[i]
12             if weighted then                // removed constexpr
13                 refVal = vWeights[refCol]
14             else
15                 refVal = 1
16             match = BinSearch(offsets, indices, cur)
17             if match == true then           // Atomic update
18                 atomicAdd(IntersectionWeights[d], refVal)
```

B. Out-of-Tree Modifications

While RAPIDS is invaluable for data science, the size and complexity of the framework would have been intractable to wholly port to FPGA. As such, we modified the core CUDA functionality to function out-of-tree and interface with our command-line mini-application. Technical details of the extraction are outside the scope of this work, but in particular, we isolated the "legacy" *GraphCSRView* storage class, which is reused by our EC implementation, and eliminated dependencies on Thrust [42] and the RAPIDS Memory Manager. The out-of-tree pipeline was used to establish the "Gold" output of all datasets to ensure numerical consistency of the *de novo* SYCL EC and ported SYCL cuGraph implementations.

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

CUDA	**SYCL**
Host (CPU) Code	
Device pointer	Buffer
`cudaMalloc`	Buffer constructor
`cudaMemcpy`	`copy` command group
Device Buffer `[]` operator	Buffer accessor
Execution Configuration (<<<...>>>) kernel launch	Queue submission lambda
Device (FPGA/GPU) Code	
`__global__` Function	*Lambda* expression **or** *Functor* Class
grid / block	nd_range / work_group
slowest-varying dimension	
2D: `Y`, 3D: `Z`	2D: `0`, 3D: `0`
fastest-varying dimension	
2D: `X`, 3D: `X`	2D: `1`, 3D: `2`

C. SYCL Translation for HLS on Intel FPGA

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 atomic add and compiled-out `if` statement mentioned in Section IV-A, the resulting SYCL kernels are *semantically identical* to their CUDA counterparts.

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.

TABLE II: Abbreviated optimizations used in Figures 2 and 3.

Abbreviation	**Description**
Baseline	Functionally correct with `-Xsclock=145MHz -Xsrounding=ieee`
+Restrict	Baseline+ `[[intel::kernel_args _restrict]]` on all kernels
+GroupSize	Baseline+ hardcoded `[[cl::reqd_work _group_size(...)]]` on all kernels
+Unroll4	Baseline+ `#pragma unroll 4` on intersection loop
+AllSimple	Baseline+Restrict+GroupSize+Unroll4
+HybridIsect	Baseline+ per-thread asymptotic cost decision between BinSearch and SortedSetMerge
+AllOpts	Baseline+AllSimple+HybridIsect

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.

$$\Theta(|N_{ref}| \times log(|N_{cur}|)) \tag{7}$$

$$\Theta(\frac{|N_{ref}|}{p} \times log(|N_{cur}|)) \tag{8}$$

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)

$$\mathcal{O}(|N_{ref}| + |N_{cur}|) \tag{9}$$

$$\Omega(min(|N_{ref}|, |N_{cur}|)) \tag{10}$$

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 for evaluation.[6] Rather than partitioning the kernels like [29], we mirror the approach of [30] and embed the cost calculation directly into the kernel, creating a *hybrid intersection*.

Algorithm 4: Sorted Set Merge (inlined in practice)

```
Input  : graph G(V, E) in CSR format:
         offsets(|V|), indices(|E|), 'cur': integer vertex ID
Output: integer match count
match = 0
ref_idx = offsets[ref], ref_end = offsets[ref + 1] - 1
cur_idx = offsets[cur], cur_end = offsets[cur + 1] - 1
while (ref_idx ≤ ref_end && cur_idx ≤ cur_end do
    cur_col = csrInd[cur_idx], ref_col = csrInd[ref_idx]
    if ref_col == cur_col then
        match += 1
        cur_idx++, ref_idx++
    else if cur_col > ref_col then
        ref_idx++
    else
        cur_idx++
```

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

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

Graph Name	Vertices	Bidirectional Edges	Degree			Gini Index
			Avg.	Range	Std. Dev.	
kmer_A2a	**171M**	361M	**2.11**	39	0.56	**0.055**
europe_osm	50.9M	108M	2.12	12	**0.48**	0.085
road-road_USA	23.9M	57.7M	2.41	**8**	0.93	0.211
road-roadnet-CA	1.96M	5.52M	2.82	11	0.99	0.185
road-roadnet-PA	1.09M	**3.08M**	2.83	**8**	1.02	0.188
delaunay_n24	16.8M	101M	6.00	23	1.34	0.122
circuit5M	5.56M	54.0M	9.71	**1.29M**	**1357**	0.577
soc-LiveJournal1	4.85M	85.7M	17.7	20.3K	52.0	0.711
wikipedia-20070206	3.57M	84.8M	23.8	188K	255	**0.759**
GL7d19	1.96M	74.6M	38.2	134	6.73	0.088
dielFilterV2real	1.16M	47.4M	40.9	104	16.1	0.201
sc-ldoor	952K	41.5M	43.6	76	14.8	0.183
stokes	11.4M	**516M**	45.1	1728	61.8	0.392
sc-msdoor	**416K**	18.8M	45.1	76	13.7	0.166
ca-coauthors-dblp	540K	30.5M	56.4	3298	66.2	0.544
soc-orkut	3.00M	213M	71.0	27.5K	140	0.558
hollywood-2009	1.14M	113M	98.9	11.5K	272	0.750
HV15R	2.02M	325M	**161**	491	47.8	0.155

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 $PPS = \frac{BidirectionalEdgeCount}{2 \times totalKernelRuntime}$, 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

[6] Anecdotally 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*.

[7] Triangle counting that shares the *PA* and *CA* road networks

[8] JS on graphs of $< 100K$ edges, and k-clique on shared *soc-orkut*

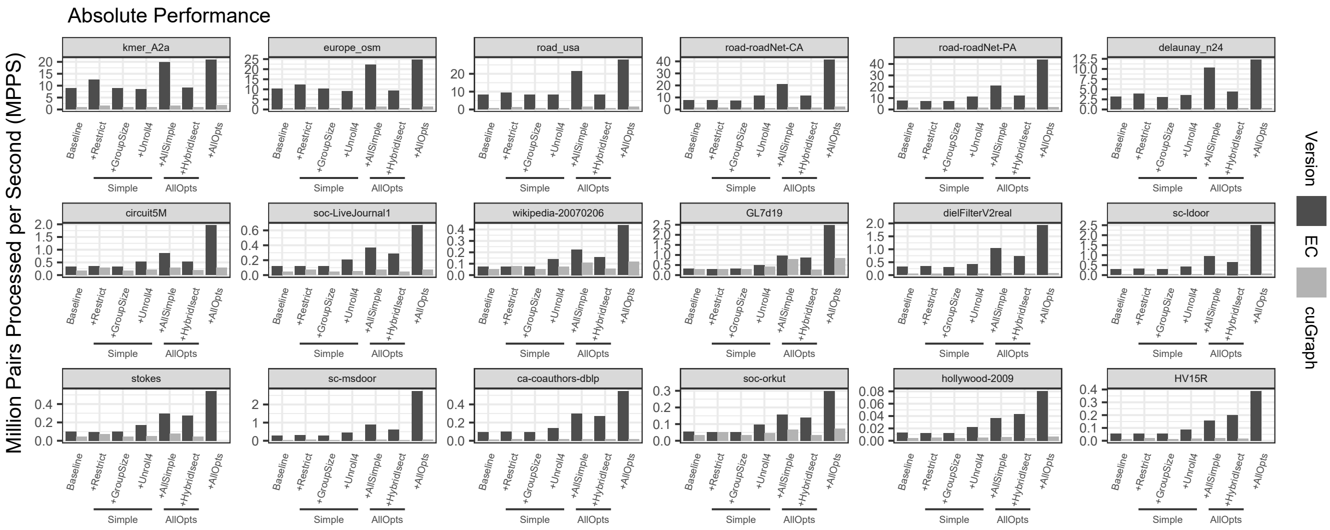


Fig. 2: Absolute performance in terms of *Millions of vertex pairs processed per second (MPPS)* of Edge-Centric (EC) and cuGraph Edge-connected JS on Intel Stratix 10 SX FPGA hardware, with optimizations applied in isolation and in concert. Sorted by increasing graph average degree.

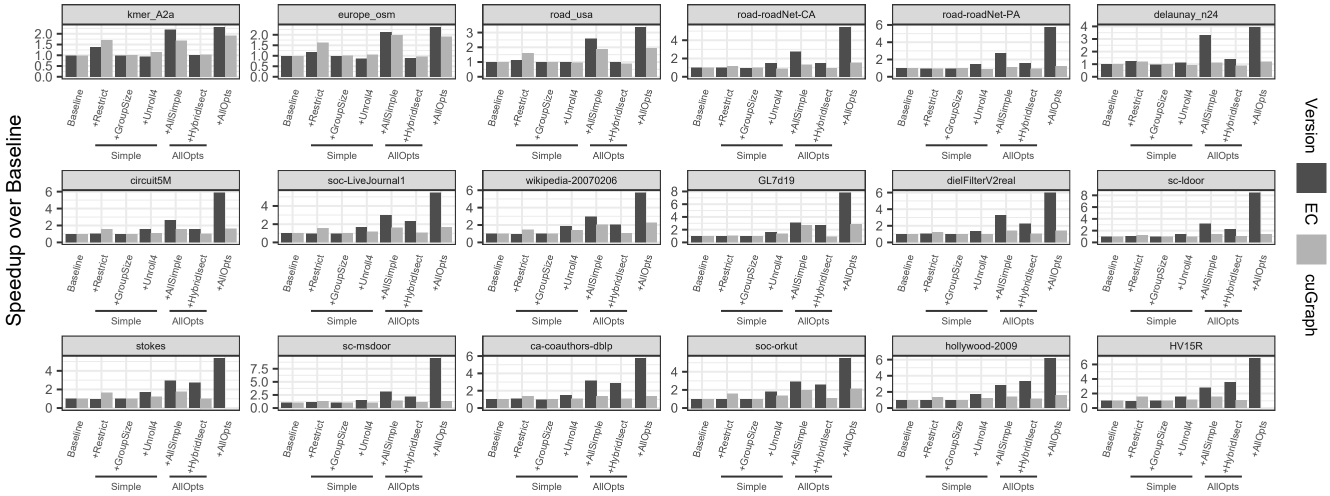


Fig. 3: Speedup relative to "Baseline" implementation of Edge-Centric (EC) and cuGraph Edge-connected JS on Intel Stratix 10 SX FPGA hardware, with various optimizations applied in isolation and in concert.

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* ($Speedup = \frac{BaselineMPPS}{OptimizedMPPS}$), 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 $\pm5\%$. 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

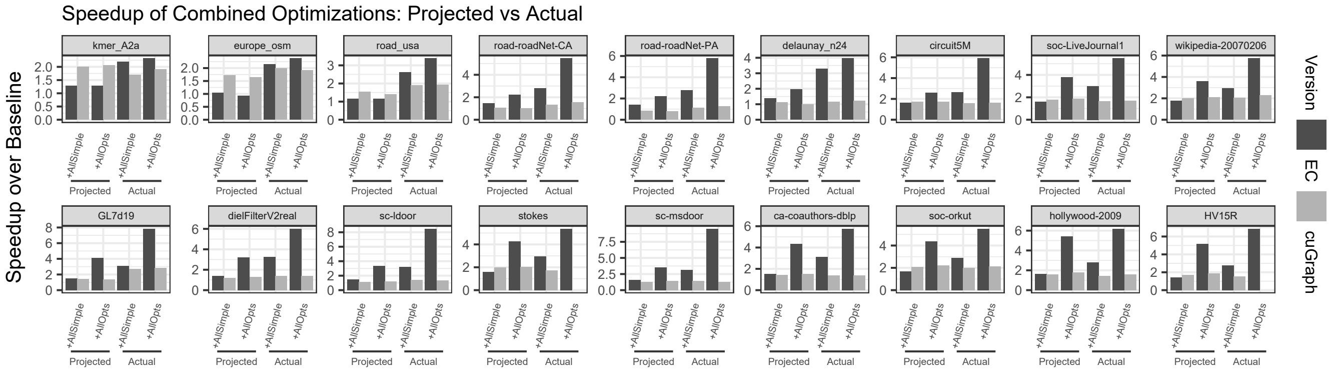


Fig. 4: Projected (multiplicative) speedup of optimizations in concert versus empirically-measured combined speedup.

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 *kmer_A2a* at $2.32\times$ and cuGraph *delaunay_n24* at $1.23\times$; the maximums are: EC *sc-msdoor* at $9.50\times$ and cuGraph *GL7d19* at $2.87\times$.

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 k*-truss search* to compete in the *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 *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 *greater than multiplicative* speedups of up to $9.5\times$ when multiple optimizations are combined. This effect is consistently observed for our *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.

[9] AllOpts cuGraph datapoints for *stokes* and *HV15R* are omitted due to spurious over- or under-counting of the intersection term for one direction of ≤ 10 out of the $100Ms$ of bidirectional edges. Neither different synthesis seed values nor alternate Stratix 10s on the Intel Devcloud eliminated the nondeterminism, which remains for future study.

[10] *sc-msdoor* (+AllOpt=$1.27\times$ vs +Restrict=$1.30\times$), *delaunay_n24* (+AllSimple=$1.15\times$ vs +Restrict=$1.22\times$), and *kmer_A2a* (+AllSimple=$1.69\times$ vs +Restrict=$1.70\times$)

References

[1] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 5171–5181. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/53f0d7c537d99b3824f0f99d62ea2428-Paper.pdf

[2] P. Jaccard, "The distribution of the flora in the alpine zone.1," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912. [Online]. Available: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x

[3] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 1122–1132.

[4] E. Valari and A. N. Papadopoulos, "Continuous similarity computation over streaming graphs," in *Machine Learning and Knowledge Discovery in Databases*, H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 638–653.

[5] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, pp. 27–64, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013707000020

[6] cugraph - rapids graph analytics library. GitHub. [Online]. Available: https://github.com/rapidsai/cugraph

[7] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, vol. 51, no. 8. New York, NY, USA: Association for Computing Machinery, nov 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2851141.2851145

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

[9] Cuda toolkit. Nvidia. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[10] Khronos OpenCL Working Group, *The OpenCL Specification*, Khronos Group Std., Rev. 3.0.11. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[11] ——, *The OpenCL C Specification*, Khronos Group Std., Rev. 3.0.11. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf

[12] The Khronos SYCL Working Group, *SYCL Specification*, Khronos Group Std., Rev. Version 1.2.1 Revision: 7, Apr. 2020. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf

[13] ——, *SYCL 2020 Specification (revision 5)*, Khronos Group Std. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

[14] Intel oneapi dpc++/c++ compiler. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

[15] (2022) triSYCL. [Online]. Available: https://github.com/triSYCL/triSYCL

[16] J. Malicevic, B. Lepers, and W. Zwaenepoel, "Everything you always wanted to know about multicore graph processing but were afraid to ask," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 631–643. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/malicevic

[17] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, oct 2015. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/2818185

[18] B. Ding and A. C. König, "Fast set intersection in memory," *Proc. VLDB Endow.*, vol. 4, no. 4, p. 255–266, jan 2011. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.14778/1938545.1938550

[19] H. Anzt and J. Dongarra, "A jaccard weights kernel leveraging independent thread scheduling on gpus," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 229–232.

[20] Y. R. Qu and V. K. Prasanna, "Fast online set intersection for network processing on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3214–3225, Nov 2016.

[21] S. Shi, Y. Qi, and Q. Wang, "Accelerating intersection computation in frequent itemset mining with fpga," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov 2013, pp. 659–665.

[22] L. Sun, H. Le, and V. K. Prasanna, "Optimizing decomposition-based packet classification implementation on fpgas," in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 170–175.

[23] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 282–297. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/3466752.3480133

[24] Heirman, Wim and Carlson, Trevor and Eeckhout, Lieven, "Sniper: scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Abstracts*. Fiuggi, Italy: High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012, pp. 91–94.

[25] X. Hu, Y. Tao, and C.-W. Chung, "I/o-efficient algorithms on triangle listing and counting," *ACM Trans. Database Syst.*, vol. 39, no. 4, dec 2015. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/2691190.2691193

[26] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–6.

[27] S. Huang, M. El-Hadedy, C. Hao, Q. Li, V. S. Mailthody, K. Date, J. Xiong, D. Chen, R. Nagi, and W.-m. Hwu, "Triangle counting and truss decomposition using fpga," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[28] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W.-M. Hwu, "Collaborative (cpu + gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[29] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader, "Logarithmic radix binning and vectorized triangle counting," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep. 2018, pp. 1–7.

[30] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on triangle counting on gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2019, pp. 1–7.

[31] V. G. Castellana, M. Minutoli, A. Morari, A. Tumeo, M. Lattuada, and F. Ferrandi, "High level synthesis of rdf queries for graph analytics," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 323–330.

[32] M. Minutoli, V. G. Castellana, N. Saporetti, S. Devecchi, M. Lattuada, P. Fezzardi, A. Tumeo, and F. Ferrandi, "Svelto: High-level synthesis of multi-threaded accelerators for graph analytics," *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 520–533, March 2022.

[33] C. Pilato and F. Ferrandi. Bambu: A free framework for thehigh-level synthesis of complex applications. [Online]. Available: https://panda.dei.polimi.it/

[34] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-fpga systems," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 264–273. [Online]. Available: https://doi.org/10.1145/2847263.2847269

[35] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 472–488. [Online]. Available: https://doi.org/10.1145/2517349.2522740

[36] K. Meng, J. Li, G. Tan, and N. Sun, “A pattern based algorithmic autotuner for graph processing on gpus,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 201–213. [Online]. Available: https://doi.org/10.1145/3293883.3295716

[37] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, “Gunrock: Gpu graph analytics,” *ACM Trans. Parallel Comput.*, vol. 4, no. 1, Aug. 2017. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/3108140

[38] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, “Chapter 2 - edge v. node parallelism for graph centrality metrics,” in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 15–28. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000022

[39] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W. Wong, and D. Chen, “On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-Based FPGAs,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 67–73.

[40] Rapids open gpu data science. Nvidia. [Online]. Available: https://rapids.ai/

[41] A. Fender, N. Emad, S. Petiton, J. Eaton, and M. Naumov, “Parallel jaccard and related graph clustering techniques,” in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi-org.ezproxy.lib.vt.edu/10.1145/3148226.3148231

[42] N. Bell and J. Hoberock, “Chapter 26 - thrust: A productivity-oriented library for cuda,” in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 359–371. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000265

[43] Intel fpga add-on for oneapi base toolkit. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html

[44] Intel programmable acceleration card (pac) with intel stratix 10 sx fpga. Intel. [Online]. Available: https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/po/product-brief-pac-with-stratix-10-sx.pdf

[45] oneapi: A new era of heterogeneous computing. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[46] (2022, Apr.) Fpga optimization guide for intel oneapi toolkits. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html

[47] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[48] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, “The suitesparse matrix collection website interface,” *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019. [Online]. Available: https://doi.org/10.21105/joss.01244

[49] M. Daga, T. Scogland, and W.-c. Feng, “Architecture-aware mapping and optimization on a 1600-core gpu,” in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, Dec 2011, pp. 316–323.

[50] C. del Mundo and W.-c. Feng, “Towards a performance-portable fft library for heterogeneous computing,” in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2597917.2597943