Excavator: Accelerating Graph Pattern Mining on FPGAs*

Matthew Giordano
Advised by Dr. Christopher Rossbach

May 2023

Abstract

Graph Pattern Mining (GPM) has become an increasingly useful workload in many scientific domains ranging from social to biological sciences, with example use cases such as fraud detection, protein discovery, and much more. GPM algorithms involve traversing an input graph with the goal of enumerating sub-graphs that match a given pattern, and they exhibit unique performance characteristics in addition to inheriting the traditional problems of large graphs. Graph algorithms generally require careful consideration when partitioning, loading, and making memory accesses to achieve high performance. GPM algorithms also have to balance generating massive amounts of intermediate state while exposing parallelism for good performance. Despite a large body of prior research on efficient GPM techniques, most research has focused on software solutions, and hardware-based solutions are relatively new. Additionally, most hardware-based solutions target ASIC designs.

Our system, Excavator, is a novel GPM architecture designed to: 1. leverage existing software optimization techniques, 2. accelerate key operations in hardware (set operations, data filtering, ...), and 3. run on Field Programmable Gate Arrays (FPGAs). FPGAs are increasingly seeing deployment in the cloud and are an attractive target for implementing hardware accelerators. They are a good candidate for GPM algorithms since they can be reconfigured to match the characteristics of different graphs and search patterns that exhibit a high degree of heterogeneity. With our FPGA-specific optimizations, we aim to bridge the gap with ASIC designs while retaining the attractive FPGA properties like ease of deployment and short iteration times between design changes.

1 Introduction

The ability to scan through large graphs for particular patterns is a powerful tool, with applications in social science, bioinformatics, and money laundering detection [4]. Graph pattern mining (GPM), the process of searching for particular subgraphs with a given shape, is thus an important workload to be able to run quickly. However, when compared to a traditional graph algorithm, graph mining suffers from a higher computational complexity; while depth-first search (DFS) scans through every possible *vertex*, GPM scans through every possible *subgraph* in a graph. While recent literature has discovered significant optimizations to improve the runtime of graph pattern mining, most prior work is implemented on software, and does not support patterns with over 5 vertices. Additionally, most proposed hardware designs target ASICs, which suffer from the prohibitive costs of fabricating a chip.

Our system, Excavator, targets FPGA architecture, which gives us a few key tools:

- Reconfigurability, allowing us to optimize to specific graphs and patterns
- Large blocks of local memory, allowing for FPGA-unique optimizations
- Easy deployment to the cloud, such as Amazon F1 instances

The main contributions of Excavator are:

- Piecemeal intermediate state generation, allowing intelligent control over memory bandwidth usage
- A decoupled translation-execute architecture that places translations off of the critical path

*Watch the thesis defense here: `https://youtu.be/0vwWWSxUrS0`

- A tagged embedding which allows us to cache translations inside of embeddings
- Local edge caches to minimize bandwidth for small edge reads, leveraging the hierarchical nature of GPM
- Early termination of edges guaranteed to be filtered

2 Background

2.1 Graphs

In the world of Computer Science, a graph is an abstract collection of *vertices* and *edges*.

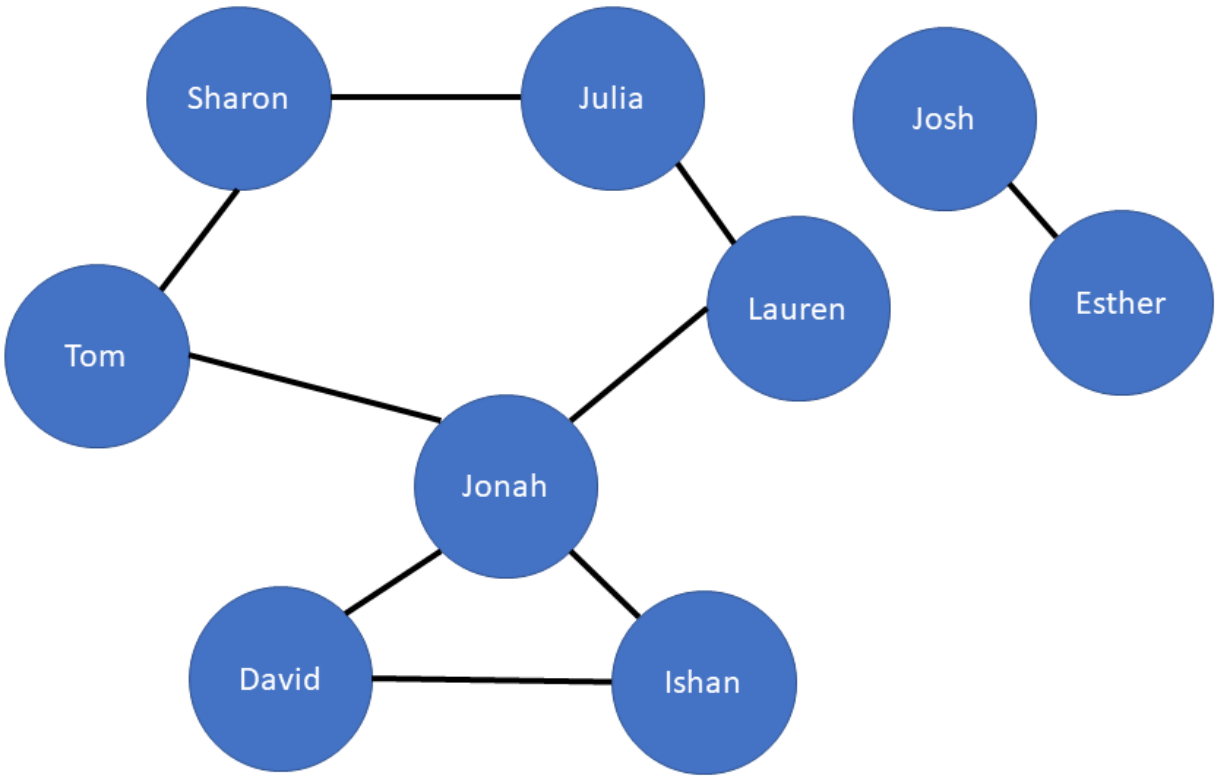


Figure 1: An example graph of people and their friends

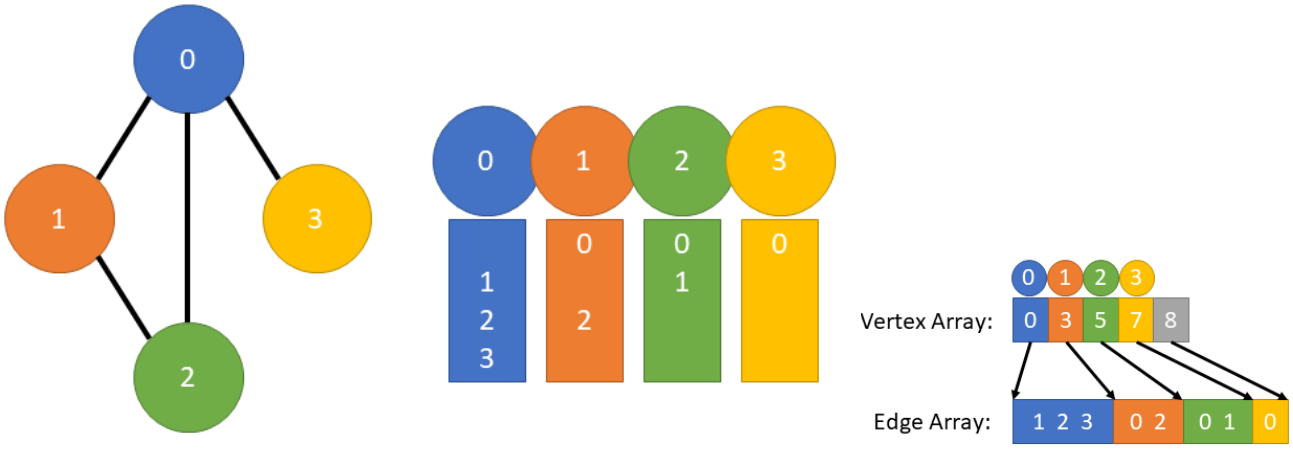
This idea of connected vertices turns out to be applicable in many domains. For example, in a social network, each vertex can represent a friend, and each edge can represent a friendship between two friends. Thus, we can build a graph of everyone's friends (see Figure 1).

With this graph abstraction, we can employ *graph algorithms* to discover useful facts about the graph. For example, in a social network, finding a friend of a friend can be done by traversing two edges. If we want to find how many friend-of-friends are needed to connect myself to Keanu Reeves, we can run a *breadth first search*, which sweeps over all of my friends, then all of my friends' friends, etc. until we find Keanu Reeves.

2.1.1 Graph Representations

Graphs can be represented via many different data structures. The most common pattern involves keeping an *edgelist* for every vertex, which contains the other vertices that this vertex is connected to. This is shown in Figure 2b.

A more compact version known as Compressed Sparse Row (CSR) stores all of the edgelists in one place, and then stores the offsets into the edgelist for each vertex. These arrays are known as the *Edge Array* and the *Vertex Array*, respectively, and are shown in Figure 2c.



(a) The graph we want to represent

(b) A basic organization of vertices and their edgelists

(c) The CSR representation of a graph

Figure 2: Various representations of a graph

Excavator supports the CSR representation, which is a format used by many online graphs. Notice that there is no Vertex ID listed in Figure 2c; instead, the vertex ID is implicitly the index into the Vertex Array. This means that reading the edges of vertex N has two steps: first, we must access the Vertex Array at location N to find the edgelist offset for our vertex. Then, we must read the Edge Array at that offset. To find the end of the edgelist, we look at the Edge Array offset of vertex N+1.

This graph structure influences much of the design of Excavator, as we must constantly *translate* vertices (index the ID into the Vertex Array) and *edgeread* an Edge Array offset. These translation and edgeread processes are independent systems, with the caveat that we can only read the edges of a vertex once we translate it into an offset in the Edge Array.

2.2 Graph Pattern Mining

2.2.1 Pattern-Naive Mining

Graph Pattern Mining (GPM) is the process of searching through a graph for a particular shape. Formally, it is enumerating over every *subgraph* in the input graph and checking if it is *isomorphic* to an input pattern graph. Isomorphism means that there exists a one-to-one mapping between each vertex in the subgraph and in the pattern graph; in other words, that the subgraph is an instance of the pattern graph. An example of mining for *diamonds* is shown in Figure 3.

For instance, one could classify a "friend group" as a group of 5 people who are all friends with each other. This is known as a *5-clique*, and an example is shown in Figure 4.

In order to discover all 5-cliques in a given graph such as Figure 1, we must enumerate over every vertex in the graph. For each vertex, we must enumerate over it's *neighbors* (connected vertices). Then, for each of those vertices, we must look at its neighbors, etc. Once we have built up a shape of 5 vertices, we can check to see if they are all connected to each other, as we requested. If they are connected, we've found a 5-clique! This is our isomorphism check. Then, we continue.

I call this algorithm *pattern-naive* searching, as opposed to the term *pattern-aware* coined by [18]. This pattern-naive algorithm builds up a potential pattern match, which we call an *embedding*. Then, once an embedding is constructed to the desired length, an isomorphism check is performed against the input pattern.

The pseudocode for this algorithm looks as follows.

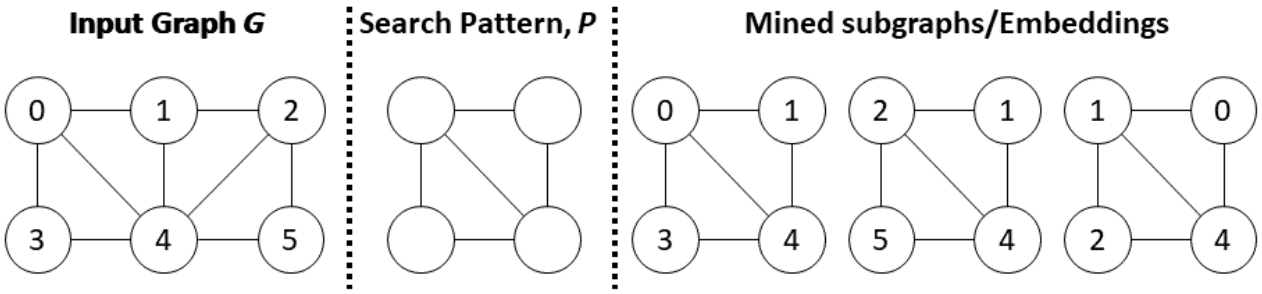


Figure 3: Mining diamonds (P) on the graph G

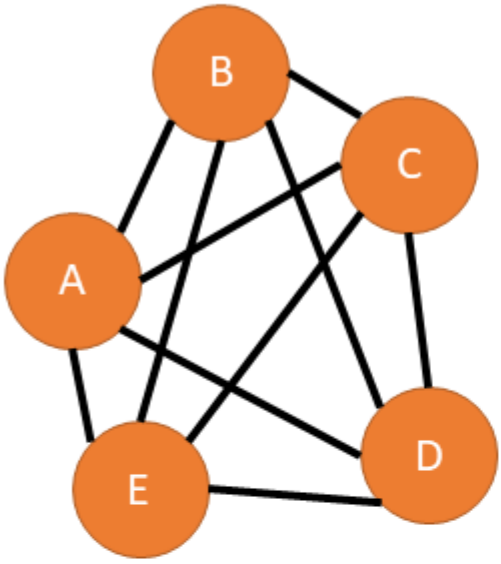


Figure 4: A 5-clique

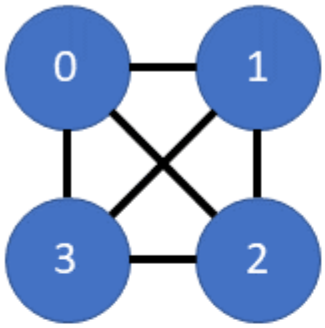


Figure 5: Does this count as a square?

```
// generate all subgraphs of size N
for vertex in graph:
    for vertex_2 in vertex.neighbors:
        for vertex_3 in vertex_2.neighbors:
            ...
                for vertex_N in vertex_(N-1).neighbors:
                    // checks that the edge connections match the desired pattern
                    embedding = [vertex, vertex_2, vertex_3, ..., vertex_N]
                    if embedding == desired_pattern:
                        found_embeddings.append(embedding)
```

To perform this recursively rather than iteratively, we can employ the following psuedocode:

```
// initialized with search([], graph.vertices);
function search(current_embedding, input_set):
    for vertex in input_set:
        new_embedding = current_embedding + [vertex]
        if len(new_embedding) == N:
            // terminate recursion at depth N
            // check that the edge connections match the desired pattern
            if new_embedding == desired_pattern:
                found_embeddings.append(new_embedding)
        else:
            search(new_embedding, vertex.neighbors)
```

In the pseudocode, I used the equality operator to perform isomorphism checks between the embedding and our desired pattern graph. This is because defining "matching" turns out to be difficult. For instance, suppose we are searching for a square. Is the graph in Figure 5 a square?

In one sense, it contains a square with some extra edges; in another, it is not an exact square. These two ideologies correspond to *vertex-induced* GPM and *edge-induced* GPM. In vertex-induced GPM, it is OK for an embedding to contain extra edges; in edge-induced GPM, it is not. These terms were coined by [18]; in general, vertex-induced GPM is slightly easier and can be faster, but is less general. Excavator focuses on edge-induced GPM.

This pattern-naive algorithm of graph pattern mining was the focus of many prior systems; however, it can be significantly sped-up via a pattern-aware algorithm.

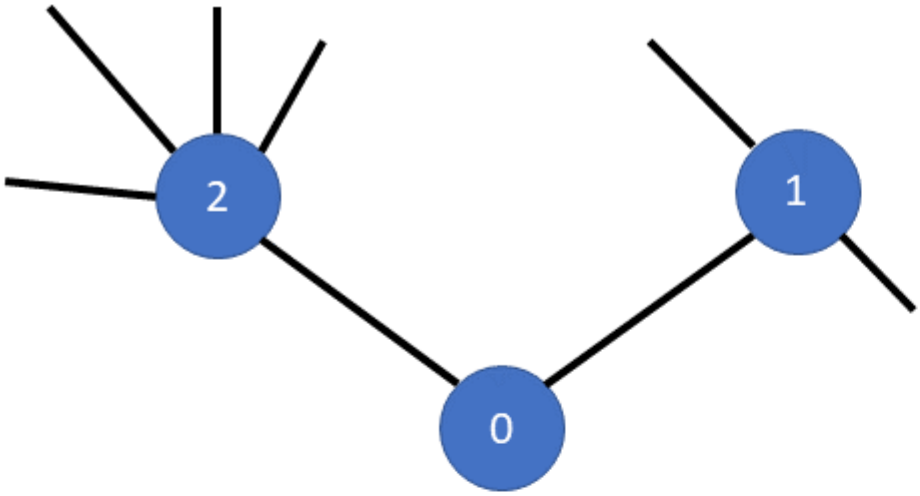


Figure 6: If we start at vertex 0, there is no way we are in a 5-clique.

2.2.2 Pattern-Aware Mining

Pattern-aware GPM takes advantage of the fact that we know the pattern *before* constructing the final embedding. For instance: if we are searching for a 5-clique, and we start out at a node with only 2 neighbors, we know that we do not have to recurse any further. Figure 6 shows visual proof of this.

There are many such optimizations that we can employ while considering whether or not to add a vertex V1 which is a neighbor of V0, such as:

- Does V1 have a high enough *degree* (number of neighbors)? If not, reject it.
- Is V1 already in the embedding? If so, reject it.
- Does V1 exist as a neighbor of V0 in the pattern? If not, reject it.

In fact, we can generalize this last bullet point to early terminate most search paths: every vertex that we add should "look the same" as a vertex in the pattern. That is, each new vertex must have the correct connections to other vertices already in the embedding.

In other words, we must build up a mapping from the pattern's A, B, C... labels to graph vertex IDs. This is precisely what an embedding is: a one-to-one mapping between a vertex in the graph and the pattern vertex. In pattern-aware mining, we focus on building our embeddings based on knowledge of the pattern's connections. Algorithmically, this changes our focus from looping over every vertex to looping over every embedding, and checking if we should *extend* it by adding another vertex or if we should throw it away. This is analogous to searching through the *embedding tree* shown in Figure 7, a tree made up of embeddings that can be extended to one another. To extend this analogy, pattern-aware mining lets us terminate search paths early, while pattern-naive mining does not.

The advantage of an embedding-centric (or pattern-centric) approach is that we have full access to the information of *how* a new vertex should be added, as opposed to iterating over all possibilities. For instance, when searching for triangles ABC when we know a vertex A and vertex B assignment, we can perform a set intersection on the neighbors of vertex A and vertex B rather than looping through B's neighbors and then checking to make sure they share an edge with A. This is shown in the following pseudocode:

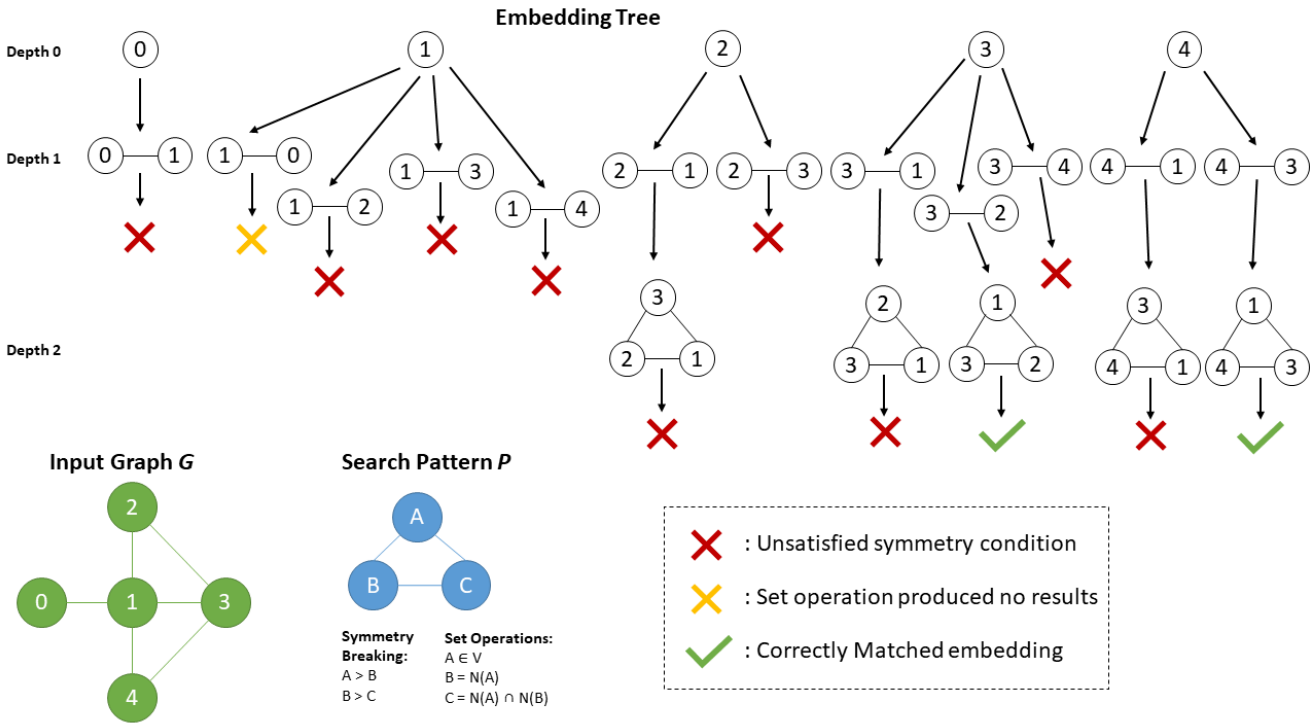


Figure 7: Embedding tree (more accurately a forest)

```
// Search for triangles in a graph
n1_embeddings = []
// initialize V embeddings that map every vertex to pattern vertex ID 0
for vertex in graph:
    if vertex.degree < pattern['A'].degree: continue // reject it
    n1_embeddings.append({'A': vertex})
// Now, we want to find all vertices B that are connected to the pattern vertex A
n2_embeddings = []
for embedding in n1_embeddings:
    vertex_A = graph[embedding['A']]
    // because A is connected to B in the pattern graph,
    //  add all of A's neighbors as potential embeddings
    for vertex_B in vertex_A.neighbors:
        n2_embeddings.append({'A': vertex_A, 'B': vertex_B})
// Now we have a vertex A and B assigned. Let's look for vertex C:
n3_embeddings = []
for embedding in n2_embeddings:
    vertex_A = graph[embedding['A']]
    vertex_B = graph[embedding['B']]
    for vertex_C in (vertex_A.neighbors ∩ vertex_B.neighbors):
        n3_embeddings.append({'A': vertex_A, 'B': vertex_B, 'C': vertex_C})
// Now, n3_embeddings contains all of our triangle instances!
```

We can generalize this idea by defining the pattern itself as a list of set operations that should occur at each phase of execution. For instance, a square is defined by:

```
A ∈ graph.vertices
B ∈ A.neighbors
C ∈ B.neighbors
D ∈ A.neighbors ∩ C.neighbors
```

```
A --- B
|     |
|     |
D --- C
```

However, you may notice that this is slightly incorrect: it allows for an edge to exist between A and C. This goes back to the vertex-induced vs edge-induced debate. If we are ok with vertex-induced GPM, then set intersection between vertices is the only operator we need to define a pattern. However, if we want edge-induced GPM, we will need to add the set difference operator. An edge-induced square is defined as follows:

```
A ∈ graph.vertices
B ∈ A.neighbors
C ∈ B.neighbors - A.neighbors
D ∈ (A.neighbors ∩ C.neighbors) - B.neighbors
```

In other words, Cs cannot be neighbors of A, and Ds cannot be neighbors of B. With the tools of set intersect and difference, we can define any pattern [18].

2.2.3 The Doubling Counting Problem

In the previous example, a simple graph with one square would return 8 squares instead of 1. This is because we have introduced a double counting problem; B and D are interchangeable (what is known as an *automorphism*), and A can be any starting vertex. To correct this, we must apply some constraints to our graph vertex IDs. To fix the first problem, we can enforce that an embedding's vertex ID of $B < D$. This ensures that only one vertex can become B, and one can become D. To fix the fact that A can be any vertex, we require $A < B$ and $A < C$. Transitively, this implies that $A < D$. This means that A must be the lowest vertex ID in the group, ensuring that only one square is defined from each set of 4 connected vertices in this fashion.

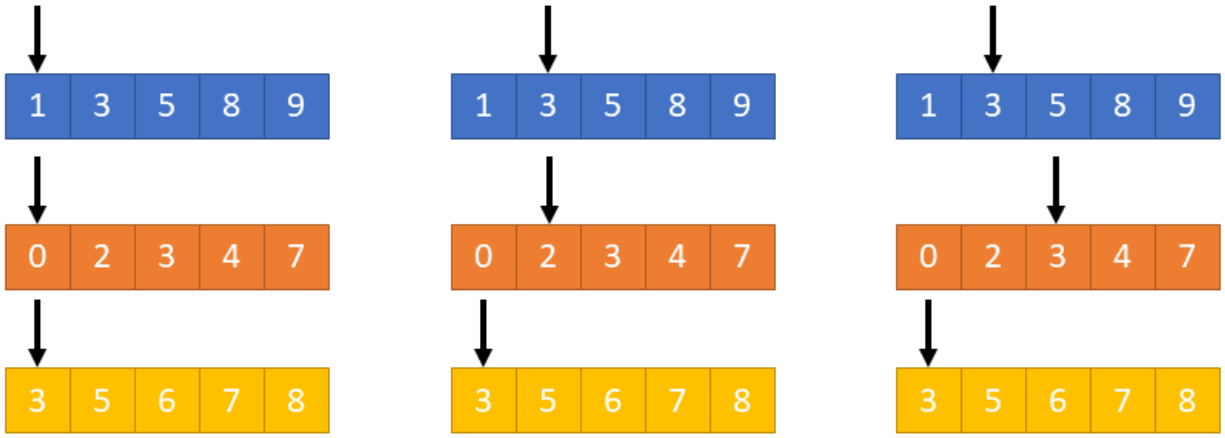
Generally, this means that our idea of a pattern is now a list of set operations to perform at each level, and a set of constraints that must be true for each embedding. We currently construct these constraints by hand, then verify them by a brute-force verifier to ensure that no double counting occurs. We believe that better algorithms exist to find constraints, however that was not the focus of this work.

2.2.4 Efficient Set Operations

In GPM, we frequently need to perform set operations, namely intersect and difference. Thus, it is pivotal that these operations are efficient.

Naively, if we are given two edgelists, we must perform an $O(n^2)$ iteration across both edgelists to find all intersects. In other words, we must iterate over each edge in set 1 and then find it in edge 2. This does not scale well; for E edgelists of length n, our runtime explodes into $O(n^E)$.

However, if we pre-sort all of our edgelists, then set intersection can be performed much more efficiently. For N inputs, we scan through all edgelists simultaneously, but only move forwards if our data is not the maximum number. Thus, we synchronously march through all edgelists, and can find intersects as they come up. This is shown in Figure 8, and has a runtime of $O(n * E)$.



(a) Initial edgelist pointers (b) Edgelist pointers after one iteration (c) Edgelist pointers after two iterations

Figure 8: Efficient set intersect on sorted edgelists

This algorithm works because if our pointed-to element is in the set, then all other pointers must be looking at numbers less than or equal to this number. Otherwise, since the lists are sorted, they don't contain the number of interest.

Set difference on sorted edgelists is similarly efficient. We maintain two scan pointers like before, but only emit successes when the minuend is not contained in the subtrahend.

2.2.5 State Explosion

Edge-induced graph pattern mining is embarrassingly parallel, but the amount of intermediate state generated quickly grows. To avoid constant paging of the intermediate data, one needs to effectively limit the amount of parallelism. This is much like a classic tradeoff between *depth first search* (DFS) and breadth first search (BFS): DFS does not generate as much intermediate state when it is deep in the tree as BFS would. For Excavator, this means that we should start mining BFS-style (as the pseudocode above showed) in order to maximize cache utilization and edgeread streams. However, once we run low on memory to store the fringe, we switch to DFS to clear some intermediate state away. This allows us to maximize performance while limiting state explosion.

2.3 FPGAs

Field Programmable Gate Arrays (FPGAs) are chips that can be reconfigured on-the-fly. FPGAs can be used to implement highly specific, highly expressive logic designs, giving bit-level control to the programmer. This has led to them being used to accelerate a wide variety of workloads, such as graph processing, neural network inference, and RTL prototyping. While FPGAs were originally designed as a circuit simulator, the reconfigurability and speed of FPGAs has led them to emerge as first class accelerators in cloud environments.

2.3.1 FPGA Architecture

FPGAs consist of *fabric*, a general term used to describe the heterogeneous resources available for developer use. These building blocks include Lookup Tables (LUTs), Block RAMs (BRAMs), and Digital Signal Processing units (DSPs), as well as a reconfigurable interconnect between resources. Each of these building blocks excels at performing different tasks; LUTs implement reconfigurable logic and registers, BRAMs provide local memory, and DSPs are specialized math units. When designing an application that will run on an FPGA, the hardware compiler will automatically map different logic to different resources; the programmer can override these if necessary.

Datacenter FPGAs have a specific relationship between their fabric (the FPGA chip) and the rest of the board. Specifically, datacenter FPGA boards contain DRAM and some baked-on memory controllers that are accessible

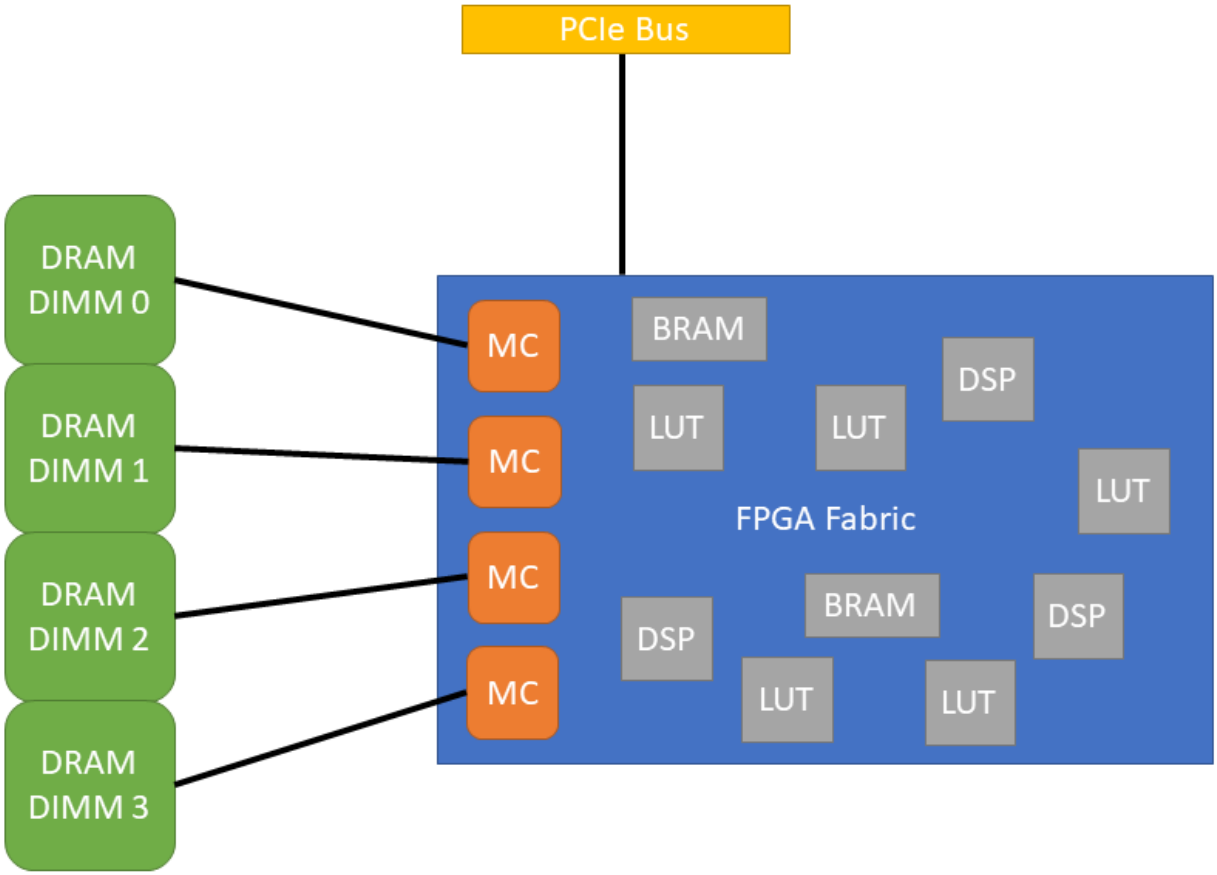


Figure 9: Datacenter FPGA Architecture

from within the fabric. Additionally, they provide a PCIe port so that the card can interface with the rest of the system. This is the style of FPGA that Excavator targets, and is illustrated in Figure 9.

Designing applications for hardware presents unique challenges. First, the design must compile to a target clock frequency. Higher clock frequencies can enable more performance, but seriously degrade programmability. Second, the developer faces spacial constraints as well as timing constraints: typically, routing wires takes above 80% of the clock time. The interfaces to DRAM and PCIe live on fixed places in the fabric, influencing the design. Third, although the compiler generally takes care of what logic should map to which hardware units, a developer must keep in mind the basic blocks available to them to save themselves from future timing-related issues.

However, there are some key advantages to FPGAs as well. Primarily, their reconfigurability allows us to optimize on a per-graph and per-pattern basis in a way that a hard-coded ASIC chip cannot. Furthermore, their ease of deployment on environments like Amazon F1 means much lower financial overhead to get mining.

3 Prior Work

3.1 Software Approaches

The majority of prior work in Graph Pattern Mining has been focused on software solutions. Some of these systems include [24, 5, 18, 6, 25, 10, 9], which mostly propose new programming paradigms for GPM as well as provide software GPM optimizations.

Peregrine [18]introduces the idea of pattern-aware GPM, an important technique in reducing the search space. This pattern-aware approach leads to optimized execution plans for various input patterns, hinting that hardware solutions for GPM must be equally as flexible.

Sandslash [5]discusses how a hybrid DFS/BFS approach can be utilized to balance the amount of available parallelism with the amount of intermediate state. It initially operates in a BFS manner to produce opportunities for other threads to explore in a DFS manner.

Accelerating set intersection has been studied [16, 29, 17, 20, 23, 19, 11], with the primary focus being on using SIMD instructions in modern CPUs. SIMD instructions can be used to filter out elements before handing off control to more specialize code blocks optimized for specific input sizes. The key observation is that for many set intersections, the input sizes are very small. This follows the similar pattern we observed about how a small fraction of vertices account for the majority of compute time.

3.2 Hardware Approaches

While hardware-based approaches to GPM are still in their infancy, GPM benefits from the wealth of research on accelerating more traditional graph algorithms [13, 8, 22, 28, 15, 26]. Set intersection, a key kernel in many GPM implementations, has been extensively studied for acceleration [16, 29, 17, 20, 23, 19, 11, 3]. A key takeaway is that the optimally performing solution depends on if the inputs are of a similar size or not.

GRAMER [27] presents an architecture (implemented on an FPGA) whose primary contribution is the use of a unique cache hierarchy that sub-divides caches based on both data type (vertices, edges, other) and priority (high vs. low). Their design takes advantage of the fact that high priority data accounts for a majority of memory accesses during the runtime of a GPM workload.

FlexMiner [7] introduces a custom data structure for caching set intersection operations, specialized set acceleration units, and an ancestor stack for rolling back state on embeddings that fail to match. Fingers [4] provides higher performing implementations of most of the concepts introduced in FlexMiner. SISA [3] implements all GPM algorithms as set operations and proposes a set-centric architecture.

Excavator's base architecture borrows concepts from many of these design, but is unique in its ability to intelligently balance the amount of generated intermediate state and parallelism. Additionally, we propose deployment on FGPAs rather than using FPGAs as fast ASIC simulators. This allows us to tailor the data path to each unique pattern and graph.

4 Excavator Design

4.1 Initial Design

Our initial design of Excavator was to translate the pseudocode from figure 3.2.2 into hardware. This resulted in a system that looked like Figure 10.

However, there are some significant flaws with this design. Primarily, the N=# extenders, which take an N-1 long embedding and create an N long embedding, are fixed in functionality and not modular. This means that building an N=5 design would entail creating an entirely new logical unit. There are exponentially more N length embeddings than N-1 length embeddings, so each extender will produce exponentially more output, causing it to not be able to take in more inputs for a long time. This is a phenomenon known as *backpressure*. Although this image gives the appearance of a pipelined design, in reality each extender will face so much backpressure that it ends up performing much like a sequential depth first search.

Additionally, the repeated Symmetry filters became a waste of area, as the backpressure problem meant that only one would be enabled at a time.

4.2 Goals

The initial design shaped our goals to become:

- Exploit the reconfigurable nature of an FPGA to optimize on a per-graph and per-pattern basis
- Utilize local memory and caching as much as possible to save DRAM bandwidth, which we aim to make the ultimate limiting factor
- Design a modular solution that scales automatically with the size of embeddings and patterns.

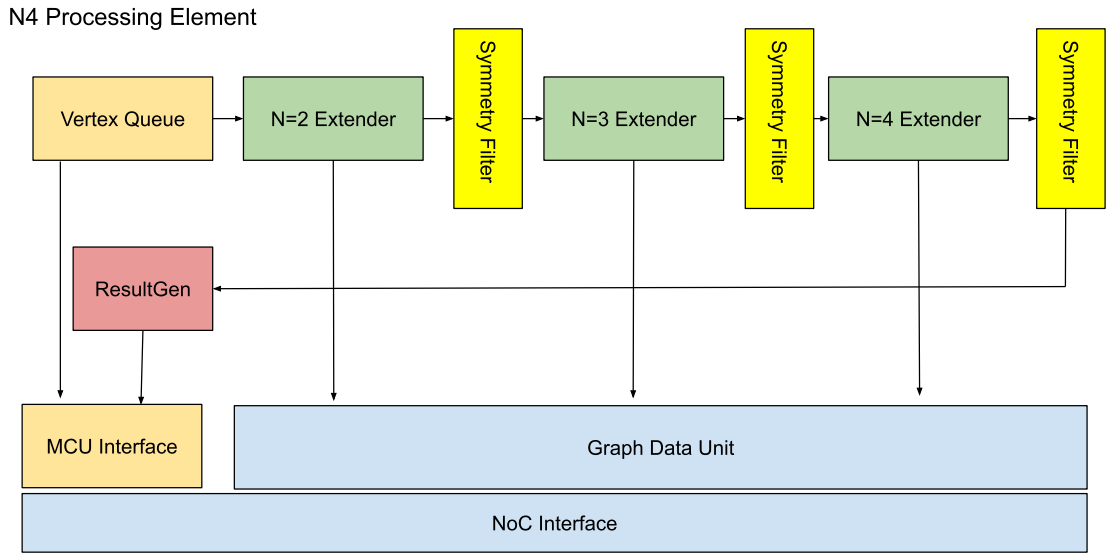


Figure 10: Our initial GPM design

- Fully pipeline our system and design around backpressure
- Make the most out of every memory transfer
- Achieve performance close to ASIC-based solutions

4.3 Excavator Overview

We propose Excavator, a novel FPGA-based graph pattern mining accelerator that improves upon state of the art ASIC-designs [7, 27] and takes advantage of FPGA-specific optimizations.

Our system, shown in Figure 11, is designed for Amazon F1 FPGA instances and builds on the AmorphOS FPGA Operating System, which provide infrastructure for handling application control and data [cite]. Execution begins with the host machine DMA (direct memory access) transferring the input graph to the FPGA. The host then programs the *Master Control Unit* (MCU) with control information (search pattern, address of input graph, etc...) before sending a start signal.

4.3.1 Processing Elements

Our design broadly consists of some *Processing Elements* (PEs), which are individual units capable of mining any pattern. To accomplish this, they are assigned *Vertex Tasks* by the *Master Control Unit* (MCU). These vertex tasks define a range of vertices which we want to mine on that PE. The MCU intelligently schedules vertex tasks onto idle PEs. Each PE can communicate with DRAM in order to read the graph through the *Network on Chip* (NoC).

4.3.2 Decoupled Translation-Execute

Figure 12 shows the layout of a PE. As mentioned in Section 2.1.1, the two primary graph operations are vertex translates and edge reads. In the old design, these were fully synchronous, sequential systems. However, in our new design, we decouple vertex translation and edgereads. This explains the main segregations in Figure 12.

The design consists of two primary units: the Embedding eXtension Units (EXUs) and Set Operation Units (SOUs). Each EXU schedules work onto the SOUs, and performs lookups via the Vertex Lookup Units in orange. The Graph Data Unit abstracts as many memory ports as we need onto the NoC. Each of these components is connected via a crossbar, allowing any unit to communicate with any other unit.

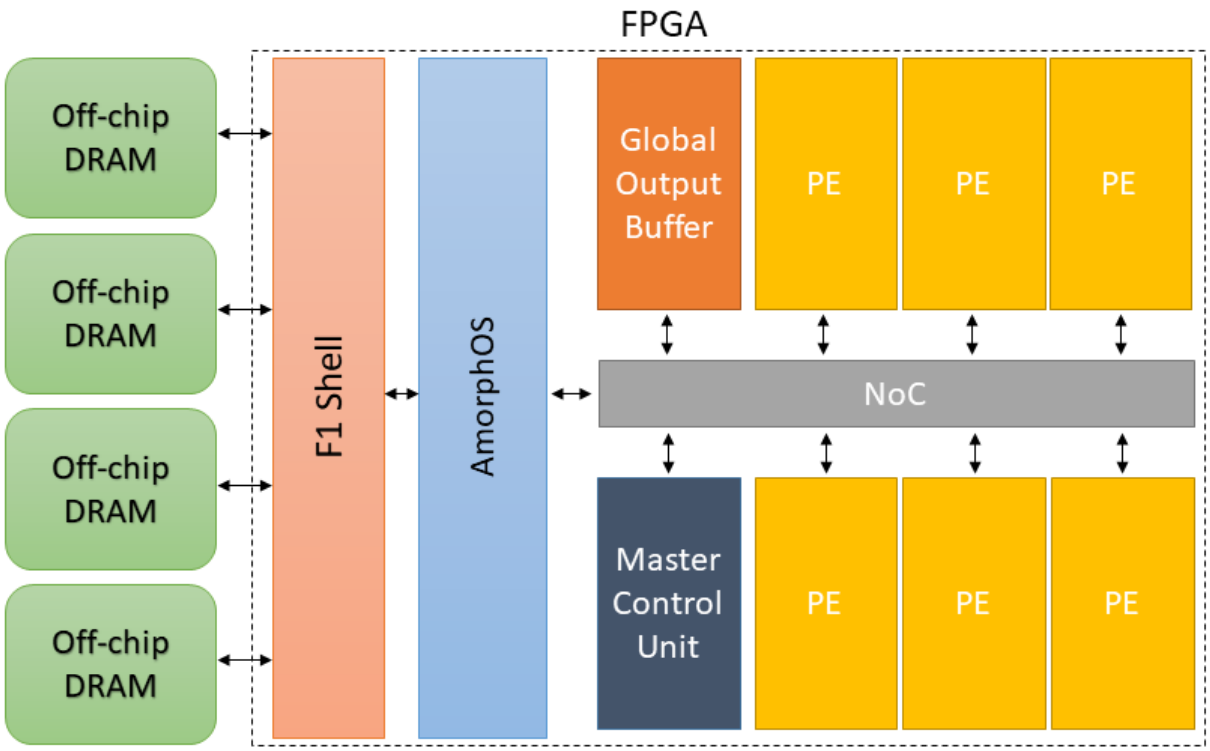


Figure 11: High level overview of Excavator

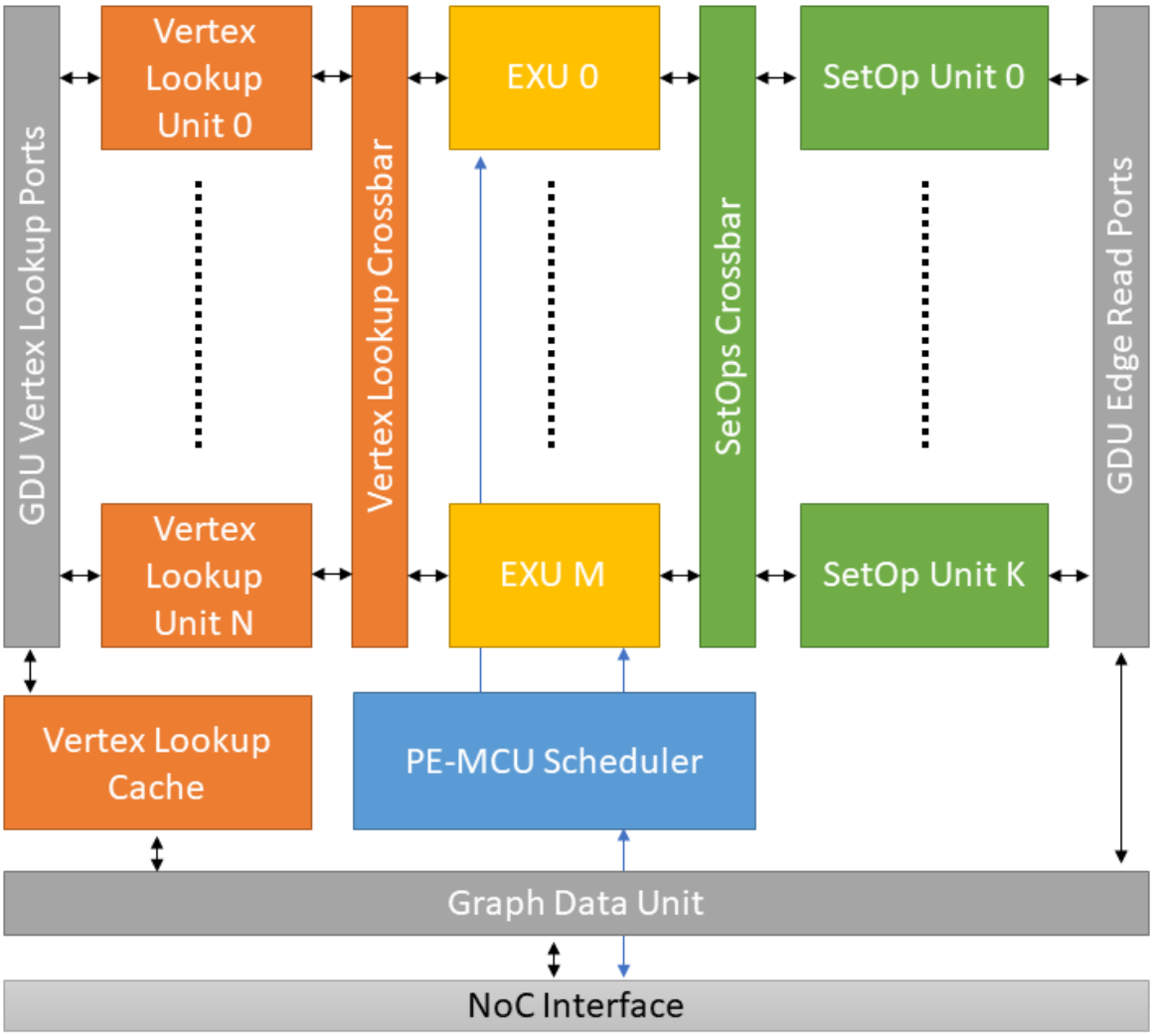


Figure 12: A Processing Element (PE)

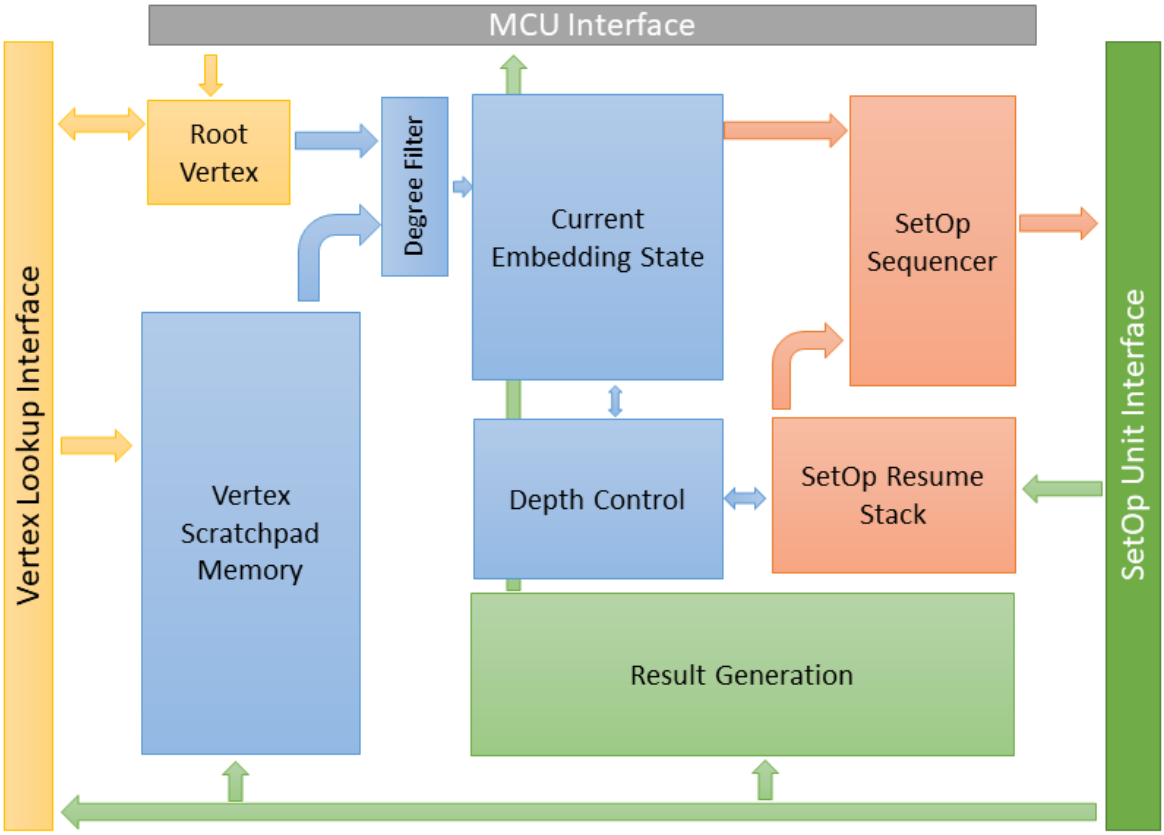


Figure 13: An Embedding eXtension Unit (EXU)

4.3.3 Embedding eXtension Units

The Embedding eXtension Units are the brain of the pattern mining process. Each EXU will receive a Vertex Task, which gives it a range of root nodes to iterate over. For each root node, we apply the pattern-aware mining algorithm described in Section 2.2.2. This means that the EXU is responsible for holding a current state and the list of unexplored embeddings that need extending. Additionally, the EXU translates vertices returned by the SOU asynchronously, and stores the returned edgelist pointer within the embedding itself. We call this a tagged embedding.

The internals of an Embedding eXtension Unit are shown in Figure 13. Each EXU communicates with the Vertex Translation Units and the Set Operation Units to perform mining. For instance, let's assume we are mining for triangles. To begin, we have some root node A. First, we ask a Set Op Unit to stream all of the edges of A, which we will call B. We need somewhere to store the B vertices while we choose one of them to explore deeper. For this, we will place all of the returned vertices into the scratchpad. Additionally, we send a request to translate all of the potential B vertices, and we place those results in the scratchpad. We can then pick a candidate B, and proceed one level deeper into the algorithm. The pattern for finding triangles is to look for the set intersect between vertices A and B, so we send off A and B to the SOU to stream back potential C vertices. If these pass the filtering stages, then they are returned to the MCU as triangles. Once this B vertex has been exhausted, we move on to the next one; once all B vertices are exhausted, we move on to a new root node.

This workflow is shown in Figure 14, where each yellow box shows the EXU's current embedding and each blue box shows the EXU's scratchpad. In general, the EXU state diagram is shown in Figure 15.

Of course, the scratchpad is a limited sized memory. If we perform a set operation and run out of room for results, we must be able to store the location to resume from later. This is stored in the Resume Stack. By not

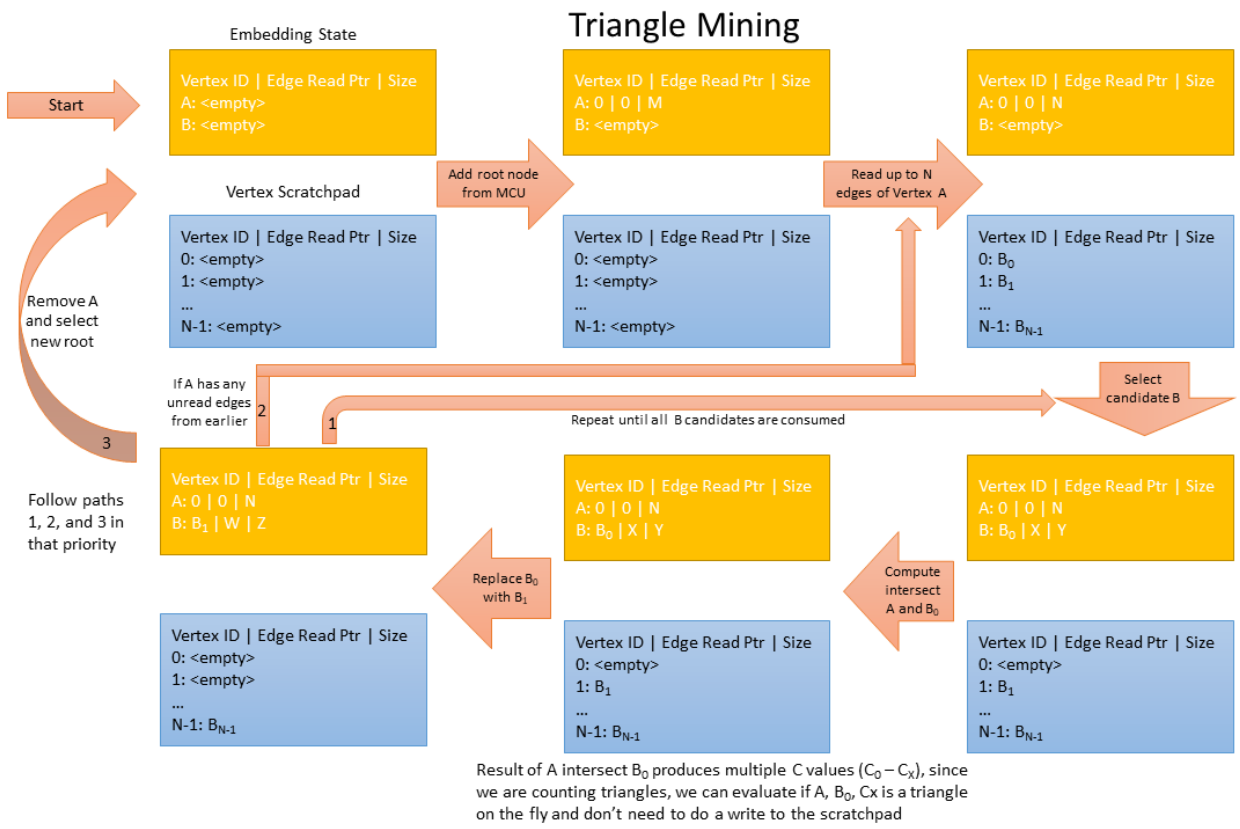


Figure 14: An EXU's states during triangle mining

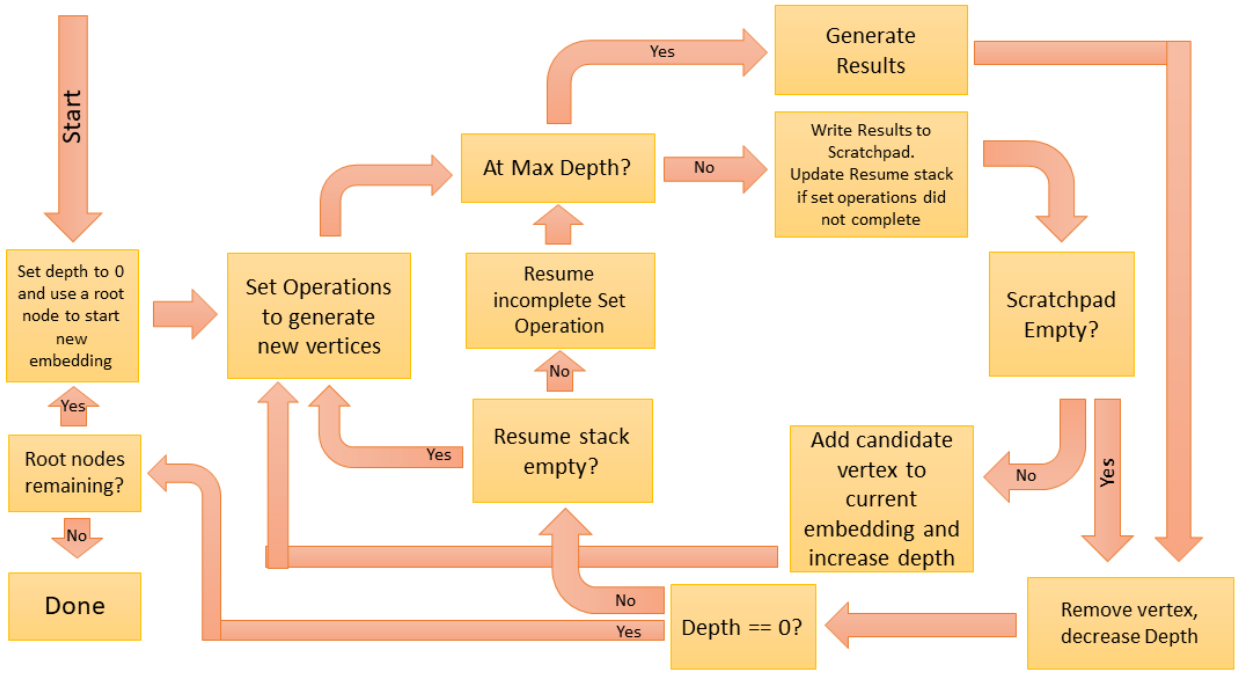


Figure 15: The state machine diagram of an EXU

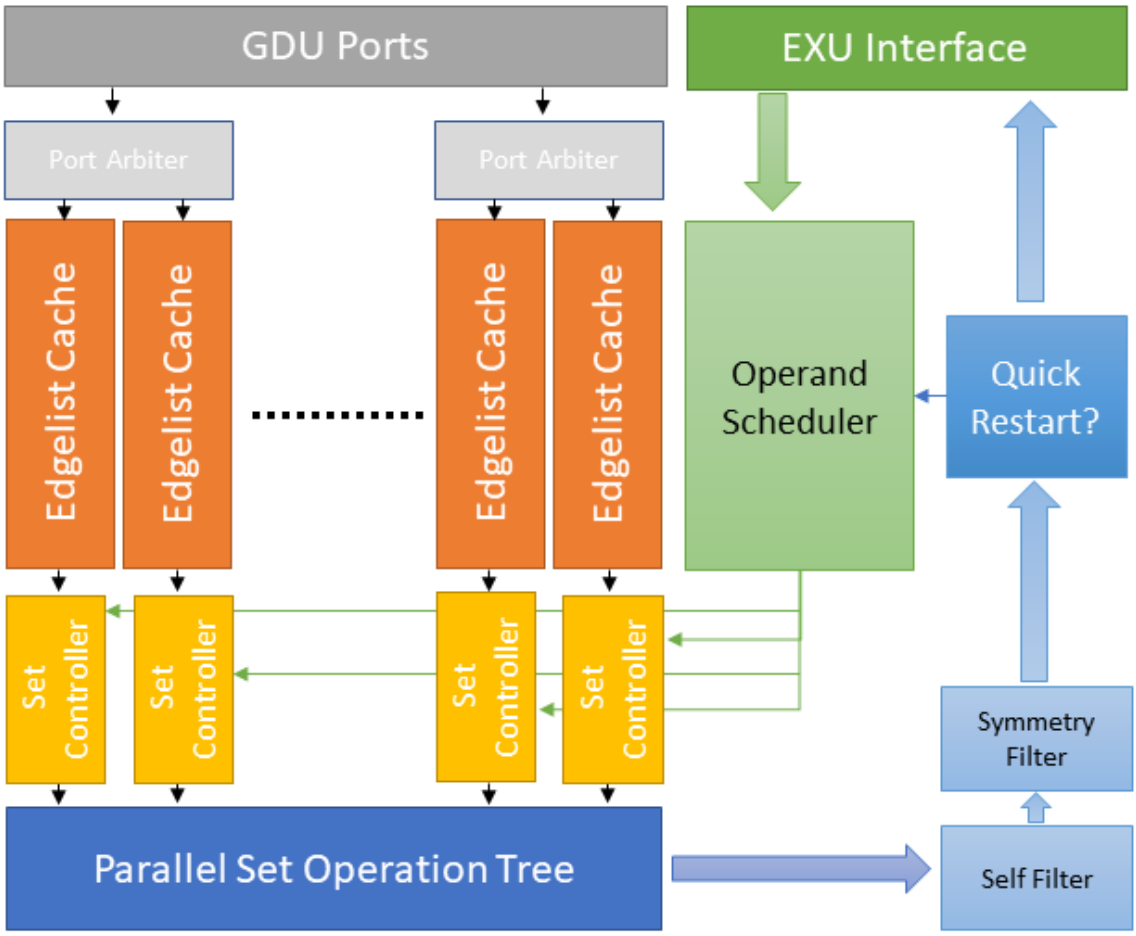


Figure 16: A Set Operation Unit

writing out extra results to DRAM, we save memory bandwidth and enable a piecemeal-style intermediate state generation.

Additionally, the Embedding eXtension Unit is partially responsible for filtering out vertices to fix the double counting problem described in Section 2.2.3.

4.3.4 Vertex Cache

We implement a small Vertex Cache between the NoC and AmorphOS interface that allows for us to reuse vertex translations. Many of our graphs obey the *power law*, where the degree distribution of vertices resembles a nonlinear curve. According to the 80-20 principle, this tells us that roughly 80% of our vertex translates will be to the same 20% of vertices. Hence, a global cache for translations is justified. We currently use a direct-mapped cache, although it may be worthwhile to explore other options.

4.3.5 Set Operation Unit

The Set Operation Unit is the compute of the graph pattern mining process. The design is shown in Figure 16. Each pattern can be defined by sequential stages of set operations, so we allow the EXU to program the SOU depending on our current depth into an embedding.

Additionally, since ports in the Set Operation Units are frequently reused (i.e., for a triangle, the same vertex A will be edgeread in SOU port 0 many times), we also implement an port-local Edge Cache to save memory

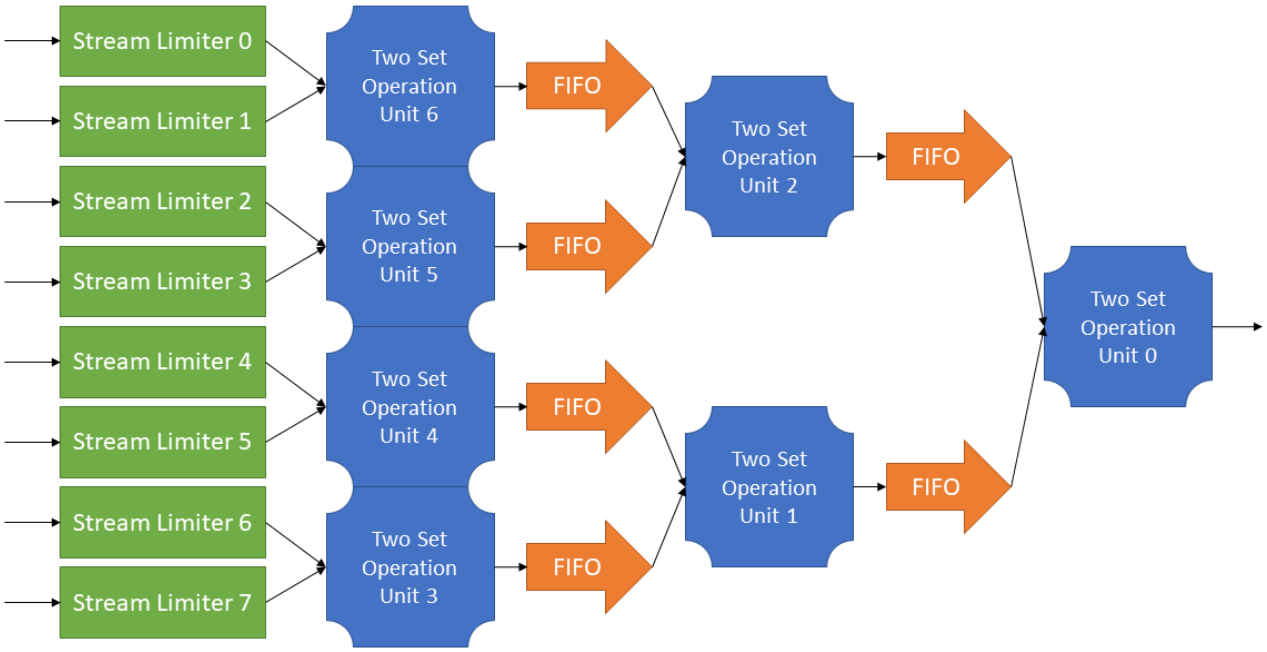


Figure 17: A Set Operation Tree

bandwidth. We choose a port-local edge cache over a global edge cache (like in GRAMAR [27]) because there is little similarity in edge reads across EXUs.

Parallel set operations are performed according to Section 2.2.4 using sorted edgelists, where each input stream can only progress when its data is less than the other stream. This is implemented in the Parallel Set Operation Tree (SOT), which lies at the heart of the Set Operation Unit.

Figure 17 shows the design of the SOT. Since the Embedding eXtension Unit's scratchpad is limited in size, a core operation of the SOU is to produce only as many results as the EXU can hold, and to resume from the same point later. This is the task of the Stream Limiters (SLs). Then, the Two Set Operation Units (TSOs) perform the necessary comparison and progress the correct stream every cycle. Finally, this design contains buffers to ease timing pressure.

4.3.6 Two Set Operation Units

Each TSO is configured by the EXU to behave as an intersect unit or a difference unit. Additionally, if it notices that only one input stream is *alive*, it turns into a passthrough. Set operations are performed by comparing values in two child streams. If those two values are equal, then we can progress both streams and produce an output. Otherwise, we progress the stream with the lower value and do not produce any output. By applying TSOs recursively in this tree pattern, we can easily scale to any size pattern.

4.3.7 Stream Limiters

One crucial property of $A \cap B$ is that we can only have maximum $min(|A|, |B|)$ elements in our resulting set, where $|A|$ denotes the length of set A. This means that by limiting our input sets to only what we have space for in our EXU, we can guarantee that we won't produce too many results.

It is worthwhile to consider a runahead execution model, where as many results are produced as possible until we run out of space. However, this design has a few flaws. Primarily, it is difficult to figure out where to resume from, because the Stream Limiters have run ahead some undefined amount. Additionally, it can overuse memory bandwidth, which is a precious resource in our design.

Instead, our design only allows X edges through each SL. Whenever a TSO throws away an edge on one stream because it is not a result, that subtree is notified to increase its X value by one, allowing one more additional edge

through. Once $X == 0$ for any SL and all of the FIFOs are empty, we can never produce any more output, and so we are done. Each SL is tasked with keeping track of the offset into the edgelist, which is returned to the EXU for resuming.

Additionally, Set Difference introduces some challenges. Firstly, we can no longer use the X value in a SL if it goes into the subtrahend of a set difference. This is because one edge in the subtrahend can produce practically unlimited results, depending on the size of the minuend. In this case, we allow the subtrahend to freely produce as many values as it can. However, edgelist resuming becomes harder; for instance, when performing $A - (B \cup C)$, we may freely produce some intermediate values $B \cup C$. However, when we run out of room, we need to know where to resume B and C from. The naive approach is just to reset them to offset 0, because we trust that the minuend saves its position correctly, and on a resume the subtrahend will quickly catch up to its previous position. However, a more sophisticated approach is to bound the current runaway of a stream by the amount of intermediate FIFO space present in the tree. In our current design with a FIFO depth of 2, this allows us to waste at most 4 edge reads.

4.3.8 Graph Data Unit

The NoC interface gives one port to memory; however, each Embedding eXtension Unit and Set Operation Unit within a PE needs their own port. The Graph Data Unit (GDU) is responsible for exposing virtual NoC ports to a PE. This allows each EXU and SOU to use one virtual port, meaning that we can scale the number of EXUs and SOUs per PE without needing to change their interfaces to worry about NoC port sharing.

4.3.9 Load Balancing Across PEs

To avoid double counting, we must employ vertex filtering. For an N-clique, these filters generally look like $A > B > C > D$.... This means that many more patterns will be found at higher vertices than low ones, so we cannot statically partition the vertex space and expect an equal distribution of work. Instead, the MCU splits up the work into fixed-size *chunks*, and distributes the tasks round-robin style into each PE's vertex task FIFO. Whenever there is space in a FIFO, it sends a new vertex task, until all tasks have been distributed. This ensures that all PEs remain working throughout all of the computation.

4.3.10 Early Filtration

Since we know that set operation results must be less than a certain vertex ID thanks to our filters (described in Section 2.2.3), we can compute the maximum viable vertex before each Set Operation Unit call. Because our edgelists are sorted, we can early terminate a set controller's edgeread as soon as we find a vertex greater than our maximum viable vertex. This allows for large computation savings at the beginning of a workload, since large degree vertices will only incur small amounts of edgereads.

For situations where early filtration is impossible (e.g. mixing greater than and less than in the pattern's symmetry breaking scheme), the SOU's outputs are put through two filters. The first, the *self filter*, filters out all vertices already present in the embedding. This is because we define an embedding to not contain any duplicate vertices. The second, the *symmetry filter*, employs any arbitrary symmetry breaking scheme within an embedding. If these filters succeed in removing embeddings, then the SOU will *quick restart* until enough embeddings are generated to fill the scratchpad.

4.4 FPGA-Specific Optimizations

We believe that an FPGA is the correct device to target for graph pattern mining acceleration. This is primarily due to the fact that graphs are extremely heterogeneous data structures:

- *Wide range of degree distributions:* Most graphs that we evaluate on obey a power-law distribution, where many vertices have low degree and few vertices have high degree. This implies that our architecture should optimize for these two extrema. However, given that our EXU Scratchpad size, Edge Cache size, Vertex Cache size, number of PEs, number of EXUs/PE, and number of SOUs/PE are reconfigurable parameters, one could tune our system to fit their graph's needs.

- *Edge Types:* While all of our graphs exclusively contain *undirected edges*, many real-world graphs contain *directed edges*, where each edge has a direction (think an arrow). Additionally, many real world graphs contain self-edges, where edges can point to their own source vertices. There also exist graphs with weights on each edge. While Excavator does not support these graphs at the moment, a reconfigurable deployment mechanism makes implementing these features feasible.
- *Property Values:* Some graphs contain properties associated with each vertex or edge. Our architecture can expand to support this via the filter pipelining inside of the EXU. However, an ASIC would have to always support these properties, slowing down the case when the graph does not have properties.
- *Data Widths:* Since we know our graph's number of vertices and edges, we can optimize our bit widths of various wires. This has a trickle-down impact on the rest of the FPGA design, allowing for smaller adder units and local storage.
- *Graph Format:* Our current system supports the CSR graph format. However, there exist a multitude of formats with and without properties that graphs can come in. We optimize Excavator's current design for CSR, but it can be changed to optimize for other formats.
- *Output Behavior:* While some applications of GPM are to count the number of instances of a pattern, some applications need to list off all of those patterns. This involves writing back to memory, a potentially routing-expensive action. Hence, we currently only count the number of instances, but logically adding a pattern output buffer and writeback would be simple. Since we are only counting, we optimize our logic to not include this writeback path.

5 Excavator Implementation

This section explores how our materialization of Excavator differs from the proposed design, as well as other optimizations that we made.

5.1 Load Balancing

We did not have time to implement a global Master Control Unit. Instead, the host acts as the MCU, sending vertex tasks to each PE. This leverages the host-communication abilities in AmorphOS. We still distribute tasks in round-robin chunks, but we suffer communication overhead. To provide a fair comparison against other, simulated designs that do not have this overhead, we measure the runtime directly within a PE.

5.2 Number of PEs, Number of EXUs per PE

AWS F1 FPGAs give us 256 bytes per cycle, which currently translates to 64 edges per cycle. To fully utilize this, we must have 64 EXUs and SOUs constantly translating or performing edgelist reads. Currently, we intend to split these into 16 PEs, with 4 EXUs and SOUs per PE. However, due to fabric constraints, we have only been able to construct 16 PEs with 2 EXUs and SOUs per PE. Additionally, to simplify logic, each EXU is tied to its own SOU. However, we hope to add an EXU-SOU scheduler that can optimize requests based on the status of the SOU edge caches.

5.3 Graph Placement

Each of our evaluated graphs is less than 16GB. Since Amazon F1 FPGAs contain 4 16GB DRAM banks, we replicate the graph across each DIMM. We then restrict each PE to a specific DIMM. This allows us to avoid the logic overhead of instantiating a NoC to DRAM interconnect. We assign 4 PEs to each DRAM DIMM.

Graph	# Vertices	# Edges	Max Degree
AstroPh (As) [21]	18.8 K	198 K	504
Patents (Pa) [14]	3.8 M	16.5 M	793
LiveJournal (Lj) [21]	4.8 M	42.9 M	20,333
Orkut (Or) [21]	3.1 M	117.2 M	33,313

Table 1: Graphs used to evaluate Excavator

5.4 Filtration

We currently do not perform early filtration as described in Section 4.3.10. Instead, we only filter results as they come out of the SOU and back into the EXU. However, we expect that much of the early filtration benefits can also be gained by sorting a graph by the degree of each vertex, so that large vertices come later. We have yet to experiment with this design.

5.5 Edge Cache Data Placement

One disadvantage of Excavator is that an input pattern must know the microarchitecture in order to fully optimize runtime. For example, the edge caches (Section 4.3.5) work best when a certain embedding's vertex stays in the same SOU tree port. In order to optimize this further, we plan to introduce Anti-Difference (where $AD(A, B) = B - A$) as a scheduling option, allowing for more opportunities for vertices to not have to change SOU tree ports. We believe that a software plan generator could come up with the ideal edge cache ports for each embedding vertex in future work.

6 Evaluation

6.1 Benchmark Graphs and Patterns

We evaluate Excavator using the same patterns used in [4] and [7]. These are: 3-(triangle), 4-, and 5- clique (tc, 4cl, 5cl), tailed triangle (tt), 4-cycle (square) (cyc), and diamond (dia). Additionally, we evaluate on the same datasets, listed in Table 1. We did not evaluate on Mico [12] or Youtube [21] because prior work disagreed on the properties of these graphs (e.g. disagreed on the number of vertices).

Since we compare against GRAMAR, we currently present only the patterns which they have comparisons for. These are: tc, 4cl, and 5cl. To show our ability to scale, we also include data for 6cl. Finally, to show the effects of set difference, we also evaluate dia.

6.2 Implementations

Currently, we are able to compile with 12 PEs and 4 EXUs per PE. We set the Vertex Cache to hold 4096 vertex translations and the port-local Edge Caches to hold 64 edges maximum. Our scratchpad size is set to hold 64 elements, which we statically partition for each embedding depth.

6.3 Methodology

Our design runs on Amazon F1 instances. AWS F1 instances use 16 nm Virtex UltraScale+ XCVU9P chips [1], which internally contain over 1.1 million lookup tables (LUTs), 2.3 million flip-flops (FFs), and 6.8 thousand Digital Signal Processing (DSP) blocks [2] Our design compiles at 250MHz, the maximum supported by this card.

We evaluate the patterns tc, 4cl, and 5cl, because those are the patterns which GRAMAR [27]shows runtimes for. Additionally, we report 6cl data, to demonstrate our ability to scale.

We run each pattern on each graph, picking the best configuration of Excavator for that graph. All numbers reported are an average of 5 runs. To measure runtime, our MCU interface records the total time taken by each PE, and we take the maximum value as our runtime.

Pattern	Graph	Excavator	Gramar
tc	As	0.012	0.028
	Pa	0.459	3.09
	Lj	6.111	17.81
	Or	38.004	-
4cl	As	0.148*	0.27
	Pa	0.768*	3.74
	Lj	TODO	30.89
	Or	TODO	-
5cl	As	TODO	1.46
	Pa	TODO	4.06
	Lj	TODO	52.89
	Or	TODO	-

Table 2: Excavator vs. Gramar, runtime in seconds.
The *s represent slightly incorrect results (less than 0.001% off).

Pattern	Graph	Excavator
6cl	As	TODO
	Pa	TODO
	Lj	TODO
	Or	TODO
dia	As	TODO
	Pa	TODO
	Lj	TODO
	Or	TODO

Table 3: Excavator runtime in seconds.

6.4 Results

Table 2 shows our runtime. We compare directly with GRAMAR [27], the leading design implemented on FPGA. Note that GRAMAR did not run Orkut. Additionally, their reported Patents graph is roughly 30% smaller than what we found. Finally, GRAMAR runs on an Alveo U250 card [27], which is slightly bigger than the F1 instance FPGAs. They run at 200MHz, but that is a fault of their own system.

Table 3 shows our performance running 6cl and dia.

6.5 Analysis

We perform better than GRAMAR in every category. Additionally, we scale better than GRAMAR. Figure 18 shows how our runtime decrease over GRAMAR changes across graphs.

6.6 Microbenchmarks

In this section, we explore how the performance of Excavator scales according to internal knobs.

Figure 19 demonstrates how our performance scales with different numbers of PEs. This experiment was otherwise performed with the same properties as Section 6.2 on the Patents graph using Triangle Counting.

The runtime follows the expected linear speedup extremely closely. This tells us that we are not quite at the point where memory traffic has become an issue, meaning that we should strive to fit more PEs.

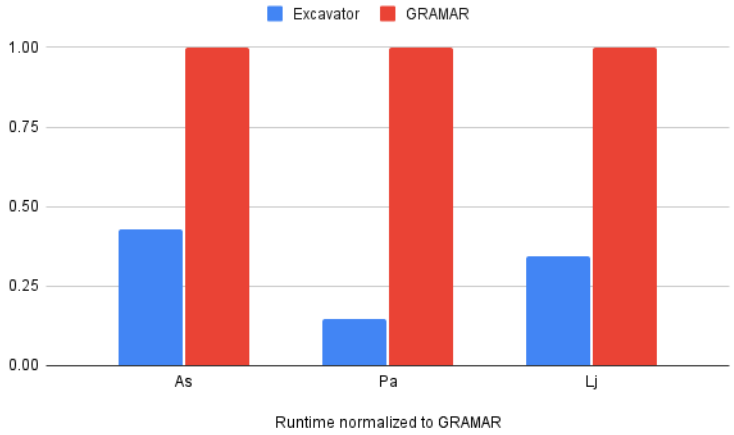


Figure 18: Runtime normalized to GRAMAR across graphs in TC

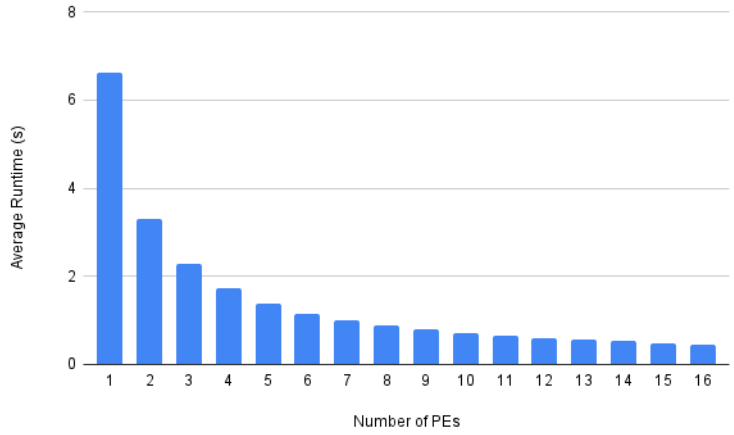


Figure 19: Runtime as a function of number of PEs

7 Conclusion

In this thesis, we present Excavator, a novel FPGA-based Graph Pattern Mining accelerator. Graph Pattern Mining is a problem of growing importance, with applications across the field of science. However, there is a lack of research in accelerating GPM. Since graphs come in many shapes and sizes, adaptability must be a key feature of a GPM accelerator. Thus, we propose that an FPGA-implementation of GPM makes more sense than an ASIC. FPGAs are significantly cheaper, involve less development overhead, and are deployable to cloud infrastructure. Additionally, we realize that we must carefully generate intermediate state as to not overwhelm memory bandwidth. With these realizations, we implement Excavator and show that it performs better than state of the art GPM accelerators on FPGAs in Triangle Counting. We hope that Excavator brings us one step closer to bridging the gap between ASIC and FPGA Graph Pattern Mining.

8 Acknowledgments

This work would not have been possible without the support of many. First, I'd like to thank Dr. Chris Rossbach for advising me for the past two years along with this project. His mentorship and classes have truly shaped who I am as a student. Secondly, I'd like to thank Dr. Calvin Lin for his fantastic mentorship throughout the past two years, and for being my Second Reader. His creative lab meetings and thoughtfulness for the individual show me how a leader should operate. I'd also like to thank Dr. Siddhartha Chatterjee for being on my Committee. Additionally, Ahmed Khawaja, Joshua Landgraf, and I co-created this project, so full credit to them as well. I'd like to thank them for being great teammates! Molly O'Neil has been a phenomenal mentor and an inspiration over the last year as well. Finally, I'd like to thank my friends and family who supported me throughout this journey.

9 References

[1] Amazon Web Services, Inc. Amazon EC2 F1 Instances. `https://aws.amazon.com/ec2/instance-types/f1/`. (Accessed on 04/24/2023).

[2] AMD Xilinx. UltraScale+ FPGAs Product Selection Guide (XMP103). `https://docs.xilinx.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide`, April 2023. (Accessed on 04/24/2023).

[3] Besta, M., Kanakagiri, R., Kwasniewski, G., Ausavarungnirun, R., Beránek, J., Kanellopoulos, K., Janda, K., Vonarburg-Shmaria, Z., Gianinazzi, L., Stefan, I., et al. 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* (2021), pp. 282–297.

[4] Chen, Q., Tian, B., and Gao, M. Fingers: Exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2022), ASPLOS '22, Association for Computing Machinery, p. 43–55.

[5] Chen, X., Dathathri, R., Gill, G., Hoang, L., and Pingali, K. Sandslash: A two-level framework for efficient graph pattern mining. *arXiv preprint arXiv:2011.03135* (2020).

[6] Chen, X., Dathathri, R., Gill, G., and Pingali, K. Pangolin: an efficient and flexible graph mining system on cpu and gpu. *Proceedings of the VLDB Endowment 13*, 10 (2020), 1190–1205.

[7] Chen, X., Huang, T., Xu, S., Bourgeat, T., Chung, C., and Arvind, A. Flexminer: a pattern-aware accelerator for graph pattern mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), IEEE, pp. 581–594.

[8] Dai, G., Huang, T., Chi, Y., Xu, N., Wang, Y., and Yang, H. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017), pp. 217–226.

[9] Danisch, M., Balalau, O., and Sozio, M. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference* (2018), pp. 589–598.

[10] Dias, V., Teixeira, C. H., Guedes, D., Meira, W., and Parthasarathy, S. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 1357–1374.

[11] Ding, B., and König, A. C. Fast set intersection in memory. *arXiv preprint arXiv:1103.2409* (2011).

[12] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment 7*, 7 (2014), 517–528.

[13] Gui, C.-Y., Zheng, L., He, B., Liu, C., Chen, X.-Y., Liao, X.-F., and Jin, H. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology 34*, 2 (2019), 339–371.

[14] Hall, B. H., Jaffe, A. B., and Trajtenberg, M. The nber patent citation data file: Lessons, insights and methodological tools, 2001.

[15] Ham, T. J., Wu, L., Sundaram, N., Satish, N., and Martonosi, M. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), IEEE, pp. 1–13.

[16] Han, S., Zou, L., and Yu, J. X. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 1587–1602.

[17] Inoue, H., Ohara, M., and Taura, K. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment 8*, 3 (2014), 293–304.

[18] Jamshidi, K., Mahadasa, R., and Vora, K. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[19] Katsov, I. Fast intersection of sorted lists using sse instructions, 2012.

[20] Lemire, D., Boytsov, L., and Kurz, N. Simd compression and the intersection of sorted integers. *Software: Practice and Experience 46*, 6 (2016), 723–749.

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

[22] Rahman, S., Abu-Ghazaleh, N., and Gupta, R. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 908–921.

[23] Schlegel, B., Willhalm, T., and Lehner, W. Fast sorted-set intersection using simd instructions. *ADMS@ VLDB 1* (2011), 8.

[24] Teixeira, C. H., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 425–440.

[25] Wang, K., Zuo, Z., Thorpe, J., Nguyen, T. Q., and Xu, G. H. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 763–782.

[26] Yan, M., Hu, X., Li, S., Basak, A., Li, H., Ma, X., Akgun, I., Feng, Y., Gu, P., Deng, L., et al. Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), pp. 615–628.

[27] Yao, P., Zheng, L., Zeng, Z., Huang, Y., Gui, C., Liao, X., Jin, H., and Xue, J. A locality-aware energy-efficient accelerator for graph mining applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), IEEE, pp. 895–907.

[28] Zhang, D., Ma, X., Thomson, M., and Chiou, D. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. *ACM SIGPLAN Notices 53*, 2 (2018), 593–607.

[29] Zhang, J., Lu, Y., Spampinato, D. G., and Franchetti, F. Fesia: A fast and simd-efficient set intersection approach on modern cpus. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), IEEE, pp. 1465–1476.