Language and Hardware Acceleration Backend for Graph Processing

Andrey Mokhov[†], Alessandro de Gennaro[†], Ghaith Tarawneh[†], Jonny Wray[‡],
Georgy Lukyanov[†], Sergey Mileiko[†], Joe Scott[†], Alex Yakovlev[†], Andrew Brown[§]
[†]Newcastle University, UK
[‡]e-Therapeutics, Oxford, UK
[§]University of Southampton, UK

***Abstract*—Graphs are important in many applications however their analysis on conventional computer architectures is generally inefficient because it involves highly irregular access to memory when traversing vertices and edges. As an example, when finding a path from a source vertex to a target one the performance is typically limited by the memory bottleneck whereas the actual computation is trivial.**

This paper presents a methodology for embedding graphs into silicon, where graph vertices become finite state machines communicating via the graph edges. With this approach many common graph analysis tasks can be performed by propagating signals through the physical graph and measuring signal propagation time using the on-chip clock distribution network. This eliminates the memory bottleneck and allows thousands of vertices to be processed in parallel. We present a domain-specific language for graph description and transformation, and demonstrate how it can be used to translate application graphs into an FPGA board, where they can be analysed 1000x faster than on a conventional computer cluster.

I. Introduction

Network science is an emerging field combining models, theories and methods across application areas that involve processing large-scale graphs, e.g. social, telecommunication and biological networks [1]. A lot of on-going research is dedicated to the discovery of new algorithms for processing graphs, particularly focusing on improving their asymptotic complexity, where the runtime of an algorithm is characterised using the *big $\mathcal{O}$ notation* and constant factors are ignored. For example, *breadth-first search*, a textbook algorithm for computing the shortest path between two vertices in an unweighted graph, takes $\mathcal{O}(|V| + |E|)$ time to process a graph with the vertex set V and the edge set E, e.g. see [2]. Here the big $\mathcal{O}$ notation hides the constant factors c_1 and c_2 in the more precise runtime bound $c_1|V|+c_2|E|$ for the sake of convenience and simplicity. The research on graph algorithms, therefore, tends to focus on improving the asymptotic complexity while paying little attention to constant factors.

In this paper we take an orthogonal approach: our primary focus is the improvement of the underlying constant factors by physically embedding graphs in silicon and making vertices communicate locally and directly, which is radically different from conventional graph processing in software that involves an indirect traversal of a graph data structure stored in memory. We do not claim any asymptotic improvements to classic graph algorithms, but we demonstrate that the proposed approach can provide 3-4 orders of magnitude speedups for sizable real-life graphs compared to a software implementation.

We automate the presented approach by developing a graph transformation language that allows the user to parse application graphs, manipulate them using common graph operations, write them into an FPGA in the form of a circuit netlist, and execute graph analysis queries.

It is important to note that the presented approach is suitable only for those applications where the overhead associated with embedding the graph into an FPGA is negligible compared to the graph processing runtime. For example, executing a single shortest-path query takes only a fraction of the time required to synthesise the FPGA netlist. One therefore needs to execute thousands of such queries in order to achieve any improvement compared to a conventional software implementation. Our case study requires the execution of millions of graph analysis queries, which justifies the upfront cost of graph embedding.

The contributions of the paper are:

- We present a domain-specific language for manipulating and embedding graphs in silicon in Section II. The language is implemented in Haskell [3], which allows us to reuse standard functional programming abstractions to efficiently manipulate graphs.
- Execution of graph analysis queries requires support of an on-FPGA infrastructure that is developed in Section III.
- We demonstrate the presented methodology on the example of biological network analysis in Section IV.

Future research directions are discussed in Section V.

II. Graph Transformation Language

This section presents a framework for constructing, manipulating, and embedding graphs in silicon, which is implemented on top of the algebraic-graphs library [4]. The framework provides a domain-specific language for graph transformation, a parser for GraphML files, and a hardware synthesis engine for generating VHDL netlists. It is open-source and publicly available under the MIT license [5].

A. Graph Data Type

Graphs are represented by the abstract data type `Graph a`, where `a` corresponds to the type of the graph vertices. For example, `Graph Int` and `Graph String` are graphs whose vertices are integers and alpha-numeric strings, respectively.

```haskell
-- Read and parse a GraphML file describing a network
readGraphML :: FilePath -> IO Network

-- Print a network
print :: Network -> IO ()

-- Synthesise a network into a hardware circuit, write the result to a VHDL file
writeVHDL :: Network -> FilePath -> IO ()

-- Merge a list of proteins into a single protein complex
mergeVertices :: [Protein] -> Protein -> Network -> Network

-- Split a protein complex into a list of proteins
splitVertex :: Protein -> [Protein] -> Network -> Network

-- Compute the subgraph induced by a given protein predicate
induce :: (Protein -> Bool) -> Network -> Network
```

Fig. 1: Overview of main functions of the framework API specialised to protein-interaction networks.

```
> g1 <- readGraphML "example.graphml"
> print g1
edges [("A","B"), ("B","C"), ("B","D"), ("C","E"), ("D","E")]

> g2 = mergeVertices ["C","D"] "CD" g1
> print g2
edges [("A","B"), ("B","CD"), ("CD","E")]

> g3 = splitVertex "CD" ["C","D"] g2
> print g3
edges [("A","B"), ("B","C"), ("B","D"), ("C","E"), ("D","E")]

> relevantProtein p = p `notElem` ["A","D","CD"]
> induce relevantProtein g3
edges [("B","C"), ("C","E")]

> writeVHDL g3 "circuit.vhdl"
```

Fig. 2: Interactive graph transformation session.

Our main case study presented in Section IV is dedicated to the analysis of *protein-interaction graphs*, further referred to simply as *networks*, for the purpose of drug discovery. Vertices of these networks are *proteins*, and edges are known protein interactions. The following two type synonyms are defined for convenience:

```haskell
type Protein = String
type Network = Graph Protein
```

That is, a `Protein` is represented simply by its name, and a `Network` is a graph whose vertices are proteins.

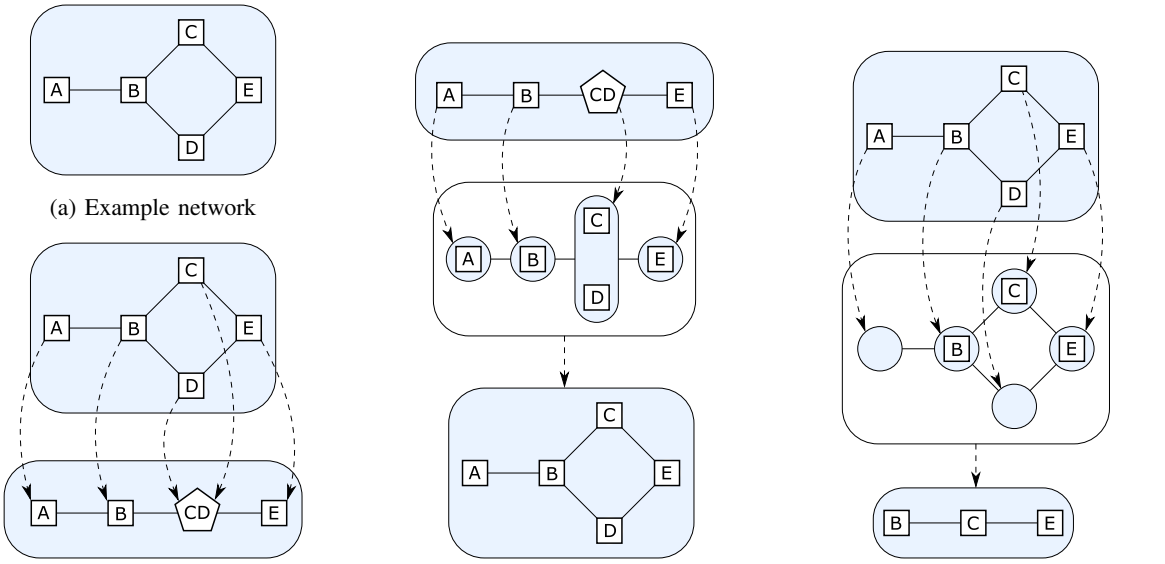
The developed graph transformation language is fully polymorphic with respect to the type of graph vertices, however we only discuss protein-interaction networks in the rest of the section for the sake of simplicity. We refer an interested reader to the framework documentation [5], which provides specifications and examples of using fully polymorphic versions of the functions we discuss.

B. Reading and Parsing GraphML Files

GraphML is a popular file format for graph storage that is used in the drug discovery field. The framework supports reading and parsing GraphML files using the `readGraphML` function, whose type is shown in line 2 of Fig. 1. The graph transformation language is embedded in Haskell and is therefore a purely functional language, requiring all side-effects to be explicitly reflected in function types. In the case of the `readGraphML` function, the type says that the function takes the path to a GraphML file as the input and returns the resulting network as the output, performing some side-effects during the execution (specifically, `IO` operations such as reading a file). Most of the functions of the framework API shown in Fig. 1 are pure, i.e. they have no side-effects, which improves the testability and scalability of the framework.

The following GraphML file describes a graph with 5 vertices and 5 edges that we will use as a running example in this section. The graph is shown in Fig. 3(a).

```xml
<graph id="G" edgedefault="undirected">
  <node id="A"/>
  <node id="B"/>
  <node id="C"/>
  <node id="D"/>
  <node id="E"/>
  <edge source="A" target="B"/>
  <edge source="B" target="C"/>
  <edge source="B" target="D"/>
  <edge source="C" target="E"/>
  <edge source="D" target="E"/>
</graph>
```



(a) Example network

(b) Merging vertices

(c) Splitting vertices

(d) Computing an induced subnetwork

Fig. 3: Examples of transforming protein-interaction networks.

C. Transforming Graphs

This section describes examples of graph transformation supported by the presented language. Fig. 2 shows an example of an interactive graph manipulation session, and Fig. 3 illustrates the transformations. We cover transformations relevant to the drug discovery case study, namely vertex merging and splitting, as well as computing induced subnetworks. The library provides more functionality – see the documentation.

The session starts with parsing the example GraphML file (see line 1 of the session in Fig. 2), which is given the name `g1` and printed using the function `print`.

Vertex merging can be used to model the formation of *protein complexes* in the context of protein-interaction networks. A protein complex is a structure where two or more proteins physically come together and form a functional unit, the complex, where all the components are required for the complex to function. To merge a list of proteins into a complex, one can use the `mergeVertices` function, as demonstrated in lines 5-7 of Fig. 2. The implementation is based on the standard functional programming abstraction called *functor* [6]: we apply a *mapping* function to each vertex of a given graph, as illustrated in Fig. 3(b), and if two vertices are mapped into the same target they are merged.

Protein complexes may be unstable and their dissociation could be modelled by splitting the corresponding vertex of the network preserving its connectivity. Lines 9-11 of the session demonstrate the use of the function `splitVertices` to undo the effect of merging vertices C and D. Note that the resulting graph `g3` coincides with the original graph `g1`. Vertex splitting is implemented using another standard functional programming abstraction called *monad* [6][7], where each vertex of the graph can be *substituted with a subgraph*, subsequently flattening the result, as illustrated in Fig. 3(c).

An important step in the drug discovery process is the identification of proteins that are relevant to a specific biological process, and discarding the remaining ones from the network under consideration, i.e. computing the *induced subnetwork* on the set of relevant proteins. This can be achieved using the `induce` function, see lines 13-15 in the session. The example predicate `relevantProtein` discards proteins A, D and CD, and can be implemented as follows:

```
relevantProtein :: Protein -> Bool
relevantProtein p = p `notElem` ["A","D","CD"]
```

Here the function `notElem` returns `True` if the given element does not belong to the given list. Note that discarding the non-existent vertex CD is allowed.

The implementation of `induce` can also be expressed in terms of the monad abstraction, where discarded vertices are substituted with empty subgraphs, which effectively removes these vertices from the graph – see Fig. 3(d).

Another reason to remove a vertex in the drug discovery context is to account for the introduction of a drug into the system, which can bind to certain proteins, preventing them from participating in their usual interactions. The removal of a vertex `v` can be expressed as computing an induced subgraph on all vertices but `v`.

A graph transformation session is typically ended by saving the result in a GraphML file or embedding it into an FPGA for accelerating its analysis, as described in Section III. Graph embedding is performed by the function `writeVHDL`.

The presented graph transformation language relies on powerful functional programming abstractions, such as functor and monad, which allows the user of the framework to implement common graphs transformations in a concise and clear manner, and reuse existing functional programming libraries.

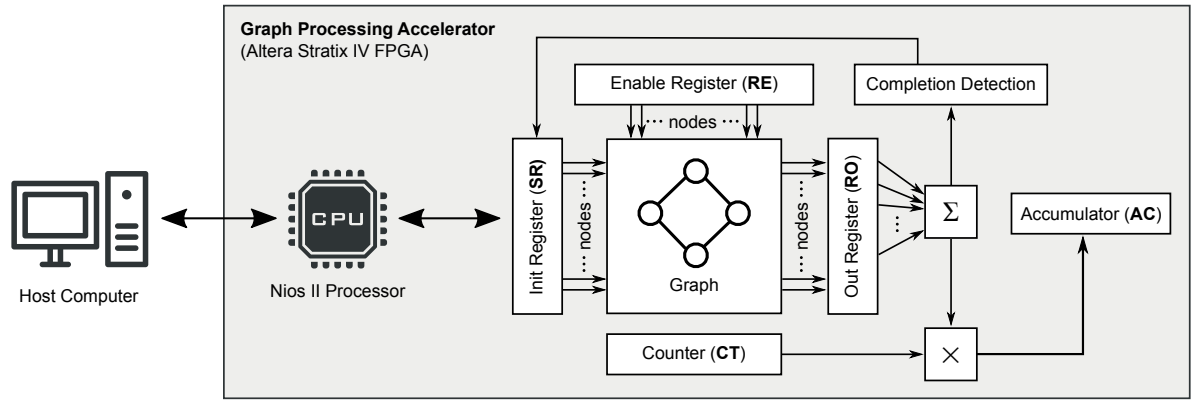


Fig. 4: Architecture of graph processing accelerator. An instance of the graph to be processed is synthesised and implemented with additional control circuitry and an on-board NIOS II processor. Host computer communicates with the accelerator via a JTAG interface and can initiate or read computation results using an Application Programming Interface (API) for graph processing.

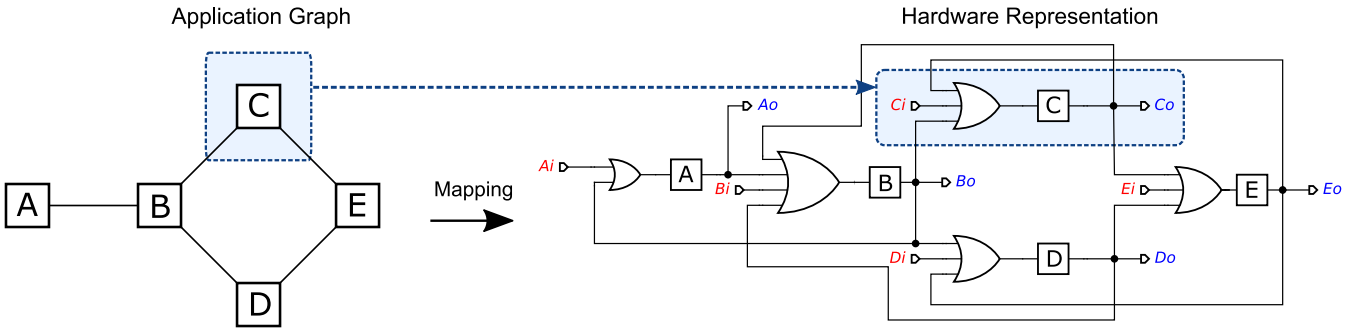


Fig. 5: Mapping a graph to a digital circuit for implementing on an FPGA.

III. Embedding Graphs into FPGAs

While the presented language is an expressive and powerful tool for manipulating graph descriptions, executing graph algorithms in software is generally inefficient due to the memory bottleneck. To this end we present an FPGA-based hardware acceleration backend that supplements the language described in Section II. The backend provides significant improvement in shortest path calculations compared to an optimised software implementation (we compare the two in Section IV).

Even though there are other hardware-based solutions for accelerating graph computations including many-core/cluster-based systems [8][9] and GPUs [10][11], we argue that FPGAs are better suited for the following reasons:

1) They are more cost effective than clusters of commodity general purpose processors.
2) They allow a direct mapping between network elements and physical silicon structures (i.e. vertices can be mapped to flip-flops and edges to interconnect paths, as will be discussed shortly). This increases both the scale and absolute performance of computations compared to more abstract network representations that are necessary in GPU and DSP implementations.
3) They are programmable, so a single device can be used to analyse multiple networks, unlike ASICs. This is particularly useful if the underlying network is frequently updated (e.g. due to acquisition of new data).

A. General Architecture

An architectural overview of the accelerator is shown in Fig. 4. At the core, the accelerator consists of an *in silico* instance of the graph to be processed, synthesised by mapping vertices to individual memory elements (flip-flops) and edges to combinational paths between these elements. The resulting hardware graph is encapsulated by the control circuitry to enable/disable selected vertices, coordinate computation, and read shortest-path computation results. An on-chip software processor (NIOS II) is also included to communicate with the host computer and provide an Application Programming Interface (API) for graph processing.

B. Graph Traversal in Hardware

The basic idea behind representing graphs using flip-flops and combinational paths is that we wish to perform graph traversal by propagating logic high values between flip-flops. The logic state of each flip-flop therefore indicates whether a given vertex has been visited (logic high) or not (logic low). To propagate a "visited" state between flip-flops, we OR the outputs of all vertex neighbours and use it as an input to the vertex flip-flop. This mapping scheme is illustrated in Fig. 5.

Using this hardware representation, shortest path calculation from a starting vertex S is performed as follows. Initially, all vertex flip-flops except for S are reset (indicating an unvisited state). On the first clock cycle following the initial state, the

"visited" state of S propagates to its immediate neighbours. The newly-visited vertices then propagate this state to their own neighbours in the following cycle and so on until the graph is fully traversed (i.e. when all vertices have been visited). In short, this computation is a classic breadth-first search where each iteration is performed in a clock cycle and involves visiting flip-flops by changing their state to logic high.

Note that the maximum number of clock cycles required to traverse the graph is equal to the graph *diameter*, which is often very small for real-life graphs. For example, biological networks in our case study comprise thousands of vertices yet their diameter is typically around 5 due to the *small-world phenomenon*. These networks can therefore be traversed in few clock cycles, which is faster than a single memory access on a commodity computer. This forms the basis for the significant acceleration factors reported in Section IV.

C. Calculating Average Shortest Path

The accelerator is designed primarily to calculate the average shortest path $\vartheta(G)$ over all pairs of source and destination vertices (a, b) where $a \neq b$, or

$$\vartheta(G) = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{j \neq i}^{N} D(v_i, v_j), \tag{1}$$

where N is the number of vertices and $D(a, b)$ is the shortest distance between vertices a and b. For better correspondence with the hardware implementation, presented next, we reformulate $\vartheta(G)$ as

$$\vartheta(G) = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{k=1} k \times C(v_i, k), \tag{2}$$

where $C(a, k)$ is the number of vertices at a distance k from vertex a. In this case the inner loop terminates when $C(v_i, k) = 0$ since this implies $C(v_i, h) = 0$ for $h > k$.

D. Implementation Details

We now describe in more details how the accelerator computes ϑ. The graph circuit interfaces with three registers: an initialisation shift register (**SR**), an enable register (**RE**) and an output register (**RO**). Register **SR** initializes vertex values at the beginning of each traversal while **RE** enables/disables selected vertices and **RO** detects which vertices have been visited during the current traversal step. Additionally, a counter **CT** maintains the step count during each traversal.

Computing ϑ involves N traversals, each amounting to calculating the inner sum in (2). During each step (of each traversal), the number of bits in **RO** is multiplied by **CT** and the result is added to an accumulator **AC**. Each traversal is completed when **RO** $= 0$ (i.e. when no new vertices are visited). After N traversals, each starting from a different vertex, the value held in **AC** is divided by $N(N-1)$ to obtain ϑ.

Register **SR** initializes the graph in preparation for a traversal operation. As discussed earlier, all vertices except for the starting vertex S are initialized in an unvisited state. The value of **SR** is therefore a one-hot encoding of the index of S.

The accelerator is meant to be used in applications where selected graph vertices can be disabled and the impact of this on ϑ can be determined. Register **RE** provides this functionality; it is an N-bit register that can be prepopulated by the user (via API calls). Any 0 bit entries in this register will disable the corresponding vertices during the traversal process, effectively removing them from the graph.

The accelerator is controlled by a host computer; computations are started, monitored and their results are read via API calls. This provides a programmatic interface enabling the accelerator to be used as a step in an automatic quantitative workflow involving manipulating a base graph via vertex removal and evaluating the impact by re-calculating ϑ. Using the language presented in Section II, an input graph can be converted into VHDL code and synthesized into an FPGA within the accelerator framework. Therefore, developers can read a graph description, synthesize and implement the accelerator, and then use it to analyse the graph, all while remaining within the same programming environment.

IV. Case study

The presented graph transformation language and accompanying hardware implementation have been applied successfully by e-Therapeutics, a pioneer in drug discovery, to accelerate their analyses of protein interactions networks. We discuss this use case here.

A. Computational Drug Discovery: An Overview

Biological systems can be modeled at different levels of abstraction by extensive networks of interactions. At the base level, molecular interactions give rise to a rich set of interactions at the cellular level while the latter mediate higher forms of interactions and so on. If normal cellular function arises from the molecular interactions within the cell then disease mechanisms can be considered as emerging from collections of pathological interactions that only occur in the disease state [12][13]. If the cellular mechanisms underlying (certain) diseases can be described as a complex system then drug discovery aimed at combating those diseases can be considered as the identification of agents that have a significant effect when used to perturb those systems.

Approaching the discovery of new therapeutic agents from this direction has a number of theoretical benefits, such as being better placed to address complex diseases that arise due to interactions between multiple components, to tackle inherent cellular robustness mechanisms [14], potentially reduce the capacity to develop resistance [15], and aid in the discovery of personalized therapeutics [16]. The robustness and resilience properties of complex systems implies they can withstand the failure, or functional perturbation, of small numbers of their constituent elements. Thus, substantial levels of change in system behavior can only occur when multiple elements are perturbed simultaneously. The fact that linear superposition does not hold implies that the identification of such element collections is not trivial, and is certainly not as simple as choosing those that appear most important

TABLE I: Resource Utilization and Performance Comparison for Six Protein Network Implementations

Network	**n0**	**n1**	**n2**	**n3**	**n4**	**n5**
Vertices	3	15	87	349	1628	3487
Edges	2	42	804	6456	53406	115898
Resource Utilization						
LUTs	54	120	477	2237	14093	31249
Memory Bits	63	134	445	1512	6658	14082
Logic Utilization (LUTs + memory)	<1%	<1%	1%	2%	11%	25%
Average Interconnect Usage	<1%	<1%	<1%	<1%	8%	25%
Peak Interconnect Usage	<1%	<1%	2%	16%	75%	91%
Operating Parameters						
Maximum Frequency (MHz)	706	283	246	159	131	107
Processing Cycles (per network)	32	215	1206	4793	22772	48371
Performance						
Software Throughput (networks/sec)	105	>104	1176	56	3	∼0.88
FPGA Throughput (networks/sec)	>107	>106	204290	33186	5489	2205
Acceleration Factor	∼100×	∼100×	173×	592×	1829×	2505×

individually. In the context of drug discovery this leads to the concept of the identification of collections of multiple proteins, that, when perturbed simultaneously, can have a large effect on biological function. The experimental identification of effective protein sets within intact cellular systems is tricky due to both experimental limitations and combinatorial explosion. Conversely, computational approaches are ideally suited for problems plagued by combinatoric issues.

e-Therapeutics has developed a practical, *in silico*, systems based approach to drug discovery based on the above principles [17]. Networks can be considered a mathematical abstraction of a complex system [1] and have become a very useful tool for modeling the molecular interactions within a cell and guiding the understanding of integrated biological function [18]. Networks, therefore, are an ideal computational approach to use for modeling cellular disease mechanisms. Percolation theory applied to complex networks [19] is concerned with exploring the change in network structure and behavior due to perturbation of collections of system elements. A key result [20] demonstrates that certain networks, including biological networks, show tolerance to random vertex failure but are vulnerable to targeted attacks. As such, network percolation forms a computational framework to develop analysis approaches aimed at the identification of effective protein sets in disease networks.

B. Drug Discovery by Network Analysis

The impact I of a perturbation, or removal, of vertices from a network is defined as

$$I_X = \frac{|\zeta_n - \zeta_0|}{\zeta_0}$$

where ζ_n is the value of network measure X when n vertices have been removed from the network.

Numerous measures X have been used in studies of network percolation and robustness with two commonly used measures being network diameter [20] and the average shortest path [21]. Both these measures rely on the calculation of all shortest paths between every pair of vertices in the network, or those in the largest connected component if the network becomes fragmented. For unweighted graphs, which are typically used in the drug discovery context, a breadth-first search is the most efficient solution to obtain all shortest paths between a pair of vertices with a time complexity of $\mathcal{O}(|V| + |E|)$, where $|V|$ and $|E|$ are the numbers of vertices and edges, respectively. The practical use of percolation experiments during the drug discovery approach at e-Therapeutics necessitates the calculation of impact on a network of multiple different proteins sets with the total number of protein sets into the hundreds of thousands. As such, improvements in calculation speed of the core impact measures based on shortest path calculations would have a major positive effect on reducing computational bottlenecks in the discovery process.

C. Biological Networks on FPGAs

We synthesized six protein interaction networks, used by e-Therapeutics, and implemented them on an FPGA as part of the graph analysis accelerator described in Section III. The networks are used to evaluate and compare the effects of different drug candidates on complex cellular systems. The impact of each drug is evaluated by disabling selected vertices that the drug is known to perturb and then calculating ϑ. Our motivation was: i) to compare accelerator performance to a software implementation, and ii) to test the scalability of our hardware implementation using realistic networks used in an industrial application. The networks ranged from very small (3 vertices, 2 edges) to considerably large (3487 vertices, 115898 edges) and had a small diameter (around 5).

Table I summarizes accelerator resource utilization and performance for the six networks. We found that the FPGA device we used (Altera Stratix IV - EP4SGX230) was sufficiently

large to accommodate the largest network (n5). For this network, combinational logic and register utilization represented a considerable but still permitting percentage of device resources (25%) while peak interconnect utilization approached the device's limit (91%). This result is not surprising given that the high degree of connectivity in biological networks cannot be matched by the planar interconnect network of an FPGA. Many real-world networks have comparable degrees of connectivity and so we expect that the scalability of our hardware implementation will be upper-bounded by FPGA interconnect density. Nevertheless, n5 is the largest within its class of proteins interaction networks used at e-Therapeutics and so our hardware implementation and choice of FPGA device have been sufficient for this particular application.

The increase in network scale also meant that the clock frequency at which the network could be clocked was lower, a trend clearly visible in Table I. This is because neighboring vertices had to be mapped to more distant flip-flops to accommodate the entire network and worst-case propagation delay had to be increased correspondingly. Nevertheless, we found that we could achieve a target clock frequency higher than 100 MHz for the largest network (n5). Another upshot of increasing network scale is that the number of cycles to calculate ϑ also increases since shortest path computations have to be repeated for a larger number of vertices. This decreased performance further compared to smaller networks.

Compared to a software implementation, the throughput of average path calculations using our hardware accelerator was higher by 2 to 4 orders of magnitude. Even though larger networks could be clocked at lower frequencies and required more cycles to calculate ϑ, the relative performance of the accelerator with these networks was higher (compared to software). Again, this is a trend that we expected; our approach scales much better with respect to network size compared to a software implementation. The performance benefit is therefore more prominent when processing larger networks.

V. Conclusions and future research

The paper presented a domain-specific language for graph construction and transformation, and a hardware acceleration backend for processing graphs on FPGAs. We demonstrate 1000x acceleration compared to a conventional software implementation used in the drug discovery industry.

Our future research will focus on investigating the applicability of the developed technology to graph processing in other domains, where graphs are typically fixed apart from minor perturbations and can therefore be embedded into FPGAs. One such example is *smart grids*, where vertices and edges are rarely added or removed. Deep learning networks are also suitable for embedding in hardware and it would be instructive to compare the developed technology to those produced by other groups in this active research area. We believe that the presented graph transformation language may be particularly well-suited for compiling machine-learning networks developed using frameworks such as Tensorflow [22] into hardware. Finally, with the advent of cloud FPGA technology it becomes possible to provide easily accessible and highly scalable graph manipulation and processing infrastructure for a wide range of applications and users, which is our long-term goal.

References

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[3] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.

[4] Haskell library algebraic-graphs. Home page and documentation on Hackage: http://hackage.haskell.org/package/algebraic-graphs.

[5] Removed for blind review.

[6] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2012.

[7] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

[8] Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[9] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[10] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John 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*, page 11. ACM, 2016.

[11] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[12] T. Ideker and R. Sharan. Protein networks in disease. 18(4):644–652.

[13] Eric E. Schadt. Molecular networks as sensors and drivers of common human diseases. 461(7261):218–223.

[14] Hiroaki Kitano. A robustness-based approach to systems-oriented drug design. 6(3):202–210.

[15] Tianhai Tian, Sarah Olson, James M. Whitacre, and Angus Harding. The origins of cancer robustness and evolvability. 3(1):17.

[16] Rui Chen and Michael Snyder. Systems biology: Personalized medicine for the future? 12(5):623–628.

[17] Removed for blind review.

[18] Albert-László Barabási and Zoltán N Oltvai. Network biology: Understanding the cell's functional organization. 5(2):101–13.

[19] D S Callaway, M E Newman, S H Strogatz, and D J Watts. Network robustness and fragility: Percolation on random graphs. 85(25):5468–71.

[20] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. 406(6794):378–382.

[21] Paolo Crucitti, Vito Latora, Massimo Marchiori, and Andrea Rapisarda. Efficiency of scale-free networks: Error and attack tolerance. 320:622–642.

[22] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Google Research, White Paper*, 2016.