Language and Hardware Acceleration Backend for Graph Processing

Andrey Mokhov[1(✉)], Alessandro de Gennaro[1], Ghaith Tarawneh[1], Jonny Wray[2], Georgy Lukyanov[1], Sergey Mileiko[1], Joe Scott[1], Alex Yakovlev[1], Andrew Brown[3]

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 analyzed up to 1000× faster than on a conventional computer.

Keywords: graph processing, average shortest path, breadth-first search, hardware acceleration, FPGA, drug discovery, domain-specific language, Haskell

1 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 characterized using the *big 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 $O(|V|+|E|)$ time to process a

✉ andrey.mokhov@ncl.ac.uk
[1]Newcastle University (UK) · [2]e-Therapeutics, Oxford (UK) · [3]University of Southampton (UK)

graph with the vertex set V and the edge set E, e.g. see [2]. Here the big 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 2-3 orders of magnitude speedups for sizable real-life graphs compared to a conventional software implementation. We automate the presented approach by developing a graph transformation language that allows the user to parse application graphs, perform common graph transformations, write graphs 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 synthesize 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 idea of using FPGAs to accelerate graph processing is not new. However, to the best of our knowledge, proposed approaches have thus far been predominantly based on the conventional Turing-style computing paradigm where one or several processors manipulate a representation of the graph in shared memory. They achieve acceleration through innovative memory architectures that exploit the flexibility of FPGAs [3][4] and/or through custom processing cores optimised for graph algorithms [5][6]. In this paper we abandon the conventional data-compute separation: we synthesize both the data (the graph) and the compute (the mathematical formulation of the breadth-first search algorithm) together into an FPGA. This solution is not general-purpose, but can achieve much higher acceleration factors.

The contributions of the paper are:

- We present a domain-specific language for manipulating and embedding graphs in silicon in Section 2. The language is implemented in Haskell [7], which allows us to reuse existing functional programming abstractions for graph parsing and manipulation.
- Execution of graph analysis queries requires support of an on-FPGA infrastructure that is developed in Section 3.
- We demonstrate the presented methodology using biological network analysis as an example in Section 4.

Future research directions are discussed in Section 5.

2 Graph Transformation Language

This section presents Centrifuge, a framework for constructing, manipulating, and embedding graphs in silicon, where they can be analyzed using the accelerator presented in Section 3. The framework provides a domain-specific language for graph transformation, a parser for GraphML files, and a hardware synthesis engine for generating VHDL netlists. Centrifuge is open-source and available at [8].

2.1 Proteins and Networks

Centrifuge is built on *algebraic graphs* [9], which allows us to reuse existing functional programming abstractions, and formally prove the correctness of graph transformation algorithms. 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 ByteString` are graphs whose vertices are integers and arbitrary identifiers, respectively. Our case study (Section 4) is dedicated to *protein-interaction graphs*, further referred to as simply as *networks*, for the purpose of drug discovery. Vertices of these networks are *proteins*, and edges are known *protein interactions*. The following type synonyms are defined for convenience:

```
type Protein = ByteString
type Network = Graph Protein
```

That is, a `Protein` is represented simply by its identifier (typically, an alpha-numeric string), 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 [8], which provides examples of using fully polymorphic versions of the function we discuss.

2.2 Reading and Parsing GraphML Files

GraphML is a popular XML-based file format for graph storage supported by most graph processing engines. 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). The Centrifuge API is pure, which improves the testability and scalability of the framework.

```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
```

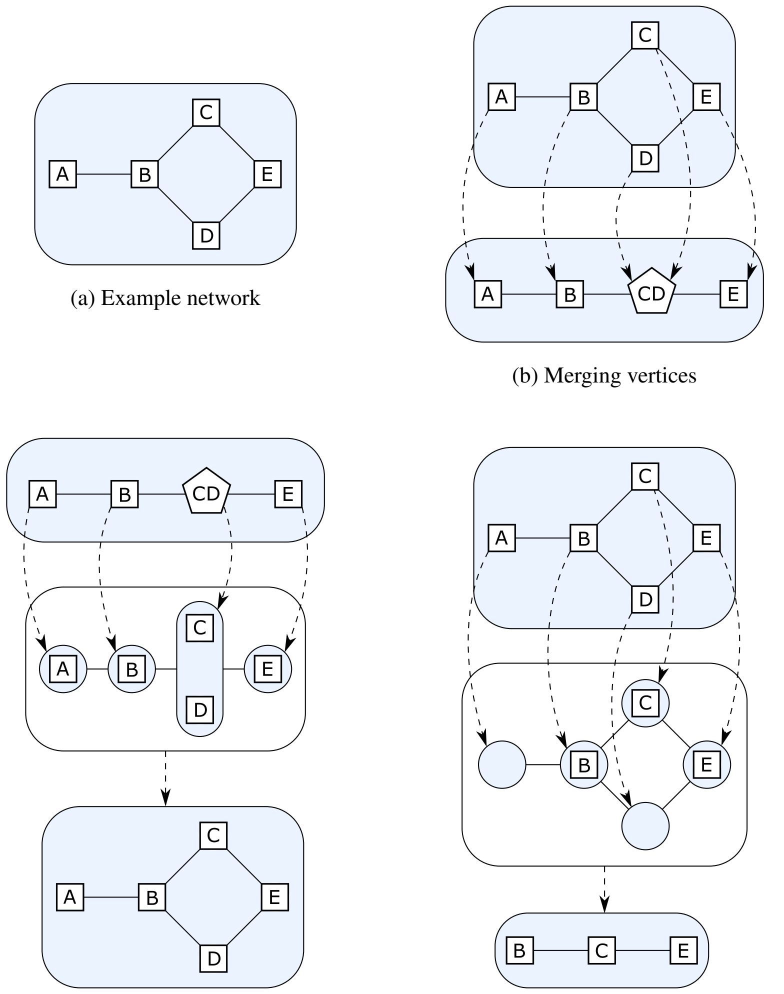
Fig. 1: Overview of graph manipulation functions of the framework API specialized to protein-interaction networks.

```haskell
-- Read an example network from a GraphML file
> g1 <- readGraphML "example.graphml"
> print g1
edges [("A","B"), ("B","C"), ("B","D"), ("C","E"), ("D","E")]

-- Merge proteins C and D into a protein complex CD
> g2 = mergeVertices ["C","D"] "CD" g1
> print g2
edges [("A","B"), ("B","CD"), ("CD","E")]

-- Undo the previous transformation by splitting CD into constituent proteins
> g3 = splitVertex "CD" ["C","D"] g2
> print g3
edges [("A","B"), ("B","C"), ("B","D"), ("C","E"), ("D","E")]

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

-- Map the network into a circuit netlist
> writeVHDL g3 "circuit.vhdl"
```

Fig. 2: An example interactive graph transformation session. See Fig. 3 for illustrations.

2.3 *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: vertex merging and splitting, as well as computing induced subnetworks. The framework provides more functionality – see the documentation.



(a) Example network

(b) Merging vertices

(c) Splitting vertices

(d) Computing an induced subnetwork

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

The session starts with parsing a GraphML file (line 1 of the session in Fig. 2). The obtained network is given the name `g1` and printed using the function `print`; the network is shown in Fig. 3(a).

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* [10]: we use `fmap` to 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. The implementation is given below.

```
-- Merge a list of vertices 'vs' into a vertex 'u'
mergeVertices :: Eq a => [a] -> a -> Graph a -> Graph a
mergeVertices vs u = fmap $ \x -> if x `elem` vs then u else x
```

Protein complexes may be unstable and their dissociation could be modeled 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* [10][11], where each vertex of the graph can be *substituted with a subgraph*, subsequently flattening the result, as shown in Fig. 3(c). The substitution function is denoted by the operator >>= in the code below.

```
-- Split a vertex 'u' into a list of vertices 'vs'
splitVertex :: Eq a => a -> [a] -> Graph a -> Graph a
splitVertex u vs g = g >>= \x -> if x == u then vertices vs else vertex x
```

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) and the code below.

```
-- Discard vertices that do not satisfy a given predicate 'p'
induce :: (a -> Bool) -> Graph a -> Graph a
induce p g = g >>= \x -> if p x then vertex x else empty
```

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 3. 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, as well as reuse existing functional programming libraries. Further examples of graph transformation can be found in [9].

3 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 2. The backend provides significant improvement in shortest path calculations compared to an optimised software implementation (we compare the two in Section 4).

Even though there are other hardware-based solutions for accelerating graph computations including many-core/cluster-based systems [12][13] and GPUs [14][15], we argue that FPGAs are better suited for the following reasons:

1. They are more cost effective than clusters of 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).

3.1 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, synthesized by mapping vertices to individual memory elements (flip-flops) and edges to combinational paths between these elements. The resulting hardware graph is encapsu-

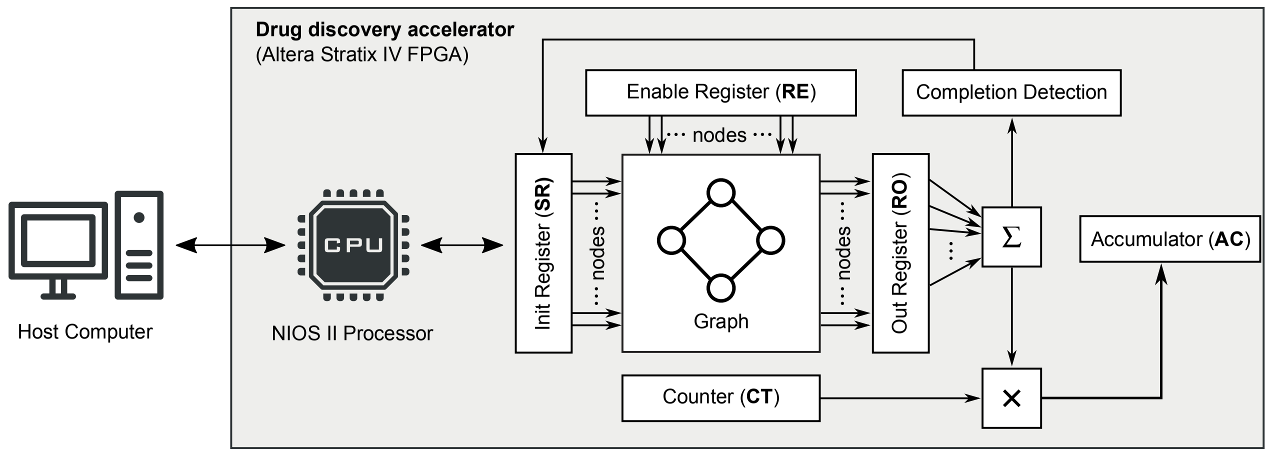


Fig. 4: An architectural overview of the developed hardware accelerator for graph processing. An instance of the graph (for example, a protein-interaction network) to be processed is synthesised with additional control infrastructure and an on-board NIOS II processor (Altera IP). A host computer communicates with the accelerator via the JTAG interface and can initiate a computation or read computation results using an Application Programming Interface (API) for graph processing. The accelerator calculates the *average shortest path* of the graph, as described in detail in Section 3.3. In short, it computes the all-pairs shortest path by measuring signal propagation delay between vertices in terms of clock cycles. The graph can be reconfigured by using the *enable register* that can enable/disable a specified set of vertices, which allows the user to analyse every subgraph of the initially synthesized graph without expensive FPGA resynthesis.

lated 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.

3.2 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 neighbors and use it as an input to the vertex flip-flop. This mapping scheme is illustrated in Fig. 5.

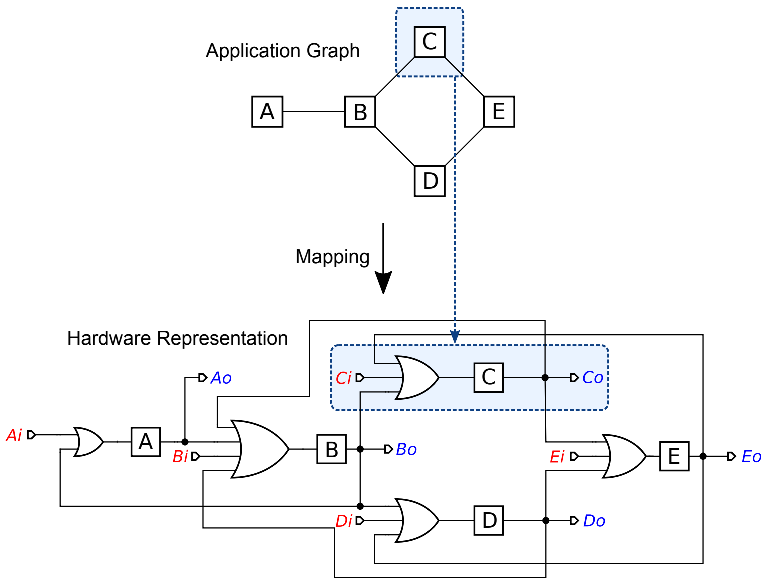


Fig. 5: Mapping a graph to a digital circuit for implementing on an FPGA.

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 neighbors. The newly-visited vertices then propagate this state to their own neighbors 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 similar to the classic breadth-first search algorithm, but all vertices in a new layer are discovered in parallel in a single clock cycle, which 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 4.

3.3 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 $h > k$.

3.4 Implementation Details

We now describe in more details how the accelerator computes ϑ. The graph circuit interfaces with three registers: an initialization 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. This approach is inspired by Conditional Partial Order Graphs [16][17], which are used to efficiently represent large graph families in hardware.

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 2, 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 analyze the graph, all while remaining within the same programming environment.

4 Case Study

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

4.1 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 [18][19]. 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 [20], potentially reduce the capacity to develop resistance [21], and aid in the discovery of personalized therapeutics [22]. 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 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 [23]. 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. Percolation theory applied to complex networks [24] is concerned with exploring the change in network structure and behavior due to perturbation of collections of system elements. A key result [25] 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.

4.2 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 [25] and average shortest path [26]. Both measures rely on calculating the shortest path between each pair of network vertices, or those in the largest connected component if the network becomes fragmented.

Practical percolation experiments used in drug discovery at e-Therapeutics involve calculating the impact of removing various protein sets on networks spanning thousands of proteins. There can be hundreds of thousands protein sets whose removal impact must be evaluated, resulting in a very large number of shortest path calculations. As such, performance improvements in network analysis can shorten drug discovery time dramatically.

4.3 Biological Networks on FPGAs

We attempted to synthesize accelerator instances containing networks of various sizes using the approach described in Section 3. The networks were protein interactomes used by e-Therapeutics to evaluate the effects of different drug candidates on complex cellular systems. Our motivation was: (i) to compare accelerator performance to a software implementation and (ii) to test the scalability of our hardware implementation using benchmarks from an industrial application. The networks ranged from very small (3 vertices, 2 edges) to considerably large (3487 vertices, 115898 edges) and all had a small diameter (up to 5).

Table 1 summarizes accelerator resource utilization and performance for the six networks. We found that the FPGA device we used (Altera Stratix IV - EP4SGX230) was not sufficiently large to accommodate the largest network (n5). 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. In general, we expect that FPGA interconnect density will impose an upper bound on the scalability of our approach for many real-world networks with comparable degrees of connectivitiy. Nevertheless, this upper bound is in fact not very restrictive; we were still able to fit substantially large networks. In our case, n5 is actually the largest within its class of protein interaction networks at e-Therapeutics and so the FPGA device we used was able to accommodate all but the largest network (up to n4 which had 1628 vertices and 53406 edges).

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 1. This is because neighboring vertices had to be mapped to more distant flip-flops to accommodate the entire network and worst-case propagation delay increased correspondingly. Nevertheless, we found that we could achieve a target clock frequency higher than 100 MHz for one of the largest networks (n4). 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 our reference software implementation (a single-threaded program written in C++, running on Intel i7-6700HQ 2.60 GHz CPU, 16 GB RAM and 6 MB cache), the throughput of average path calculations using our hardware accelerator (running at 100 MHz) was higher by 1-3 orders of magnitude. Even though larger networks required more cycles to calculate ϑ, the relative performance of the accelerator with these networks was higher (compared to software). Again, this is

Table 1: Resource utilization and performance comparison for six protein-interaction network benchmarks on the Altera DE4 board (FPGA: Stratix IV EP4SGX230).

Resource Utilization of the Network shows the resources used only by the hardware representation (HW) of the corresponding network. The number of *Dedicated Registers* instantiated on FPGA for the HW networks matches the number of vertices in a network: every protein is mapped to a flip-flop register (see Figure 5). We found that our biggest benchmark n5 cannot be synthezised into the FPGA, because the number of protein interactions exceeds the available FPGA interconnections in some parts of the chip (see the *Peak Interconnect Usage* entries). For this reason, some of the entries in the table are missing (marked by $-$) and others were estimated (marked by $*$).

Resource Utilization of the Prototype shows the amount of resources used by the entire drug discovery prototype. The amount of logic utilization due to the extra control circuitry and the NIOS II software processor is not negligible, but it is not the cause of the network n5 synthesis failure. In fact, QUARTUS also fails to synthesize n5 on its own.

Operating Parameters show the power consumption of the full FPGA prototype as estimated by the *PowerPlay Power Analyser* tool. We also show the maximum working frequency at which each network can be clocked (calculated without having the extra accelerator logic around), and the frequency that we fixed for the prototype. The prototype frequency and *processing cycles* (i.e. number of cycles needed to calculate ϑ) are used to determine the prototype performance.

Finally, the **Performance** part of the table shows the obtained acceleration figures relative to our baseline software implementation in C++. See Section 4.3 for further discussion.

Network	**n0**	**n1**	**n2**	**n3**	**n4**	**n5**
Vertices	3	15	87	349	1628	3487
Edges	2	42	804	6456	53406	115898

Resource Utilization of the Network

Dedicated registers (FFs)	3	15	87	349	1628	3487
Lookup Tables (LUTs)	3	29	250	1541	11054	24878
Logic Utilisation	1%	1%	1%	1%	8%	18%*
Average Interconnect Usage	∼0%	∼0%	∼0%	0.4%	6%	16%*
Peak Interconnect Usage	∼0%	∼0%	1%	18%	79%	95%*

Resource Utilization of the Prototype

Dedicated registers (FFs)	1272	1362	1753	3089	9510	18850
Lookup Tables (LUTs)	1347	1453	1954	4192	18114	38759
Logic Utilisation	1%	1%	1%	3%	12%	–
Average Interconnect Usage	0.6%	0.6%	0.7%	1%	11%	27%*
Peak Interconnect Usage	15%	15%	18%	31%	90%	120%*

Operating Parameters

Power Consumption (mW)	956	956	961	993	1238	–
Network Frequency (MHz)	>1000	>1000	426	195	122	–
Prototype Frequency (MHz)	100					
Processing Cycles (per network)	32	215	1206	4793	22772	48371

Performance

Software Throughput (networks/sec)	10^5	$>10^4$	1176	56	3	∼0.88
FPGA Throughput (networks/sec)	$>10^6$	$>10^5$	82918	20863	4391	2067*
Acceleration Factor	∼10×	∼10×	70×	372×	1463×	2349×*

a trend that we expected; our approach scales much better with respect to network size compared to a software implementation.

This trend can be also observed in Fig. 6, where we show the execution time of 10 different ϑ calculations on the network n4. In each of these calculations, a different number of edges (decided randomly) is enabled, from 10% to 100%. While software execution time scales linearly with the number of network edges, the execution time of the accelerator is approximately constant. Moreover, enabling more network edges tends to reduce network diameter and therefore has the counter-intuitive effect of reducing execution time.

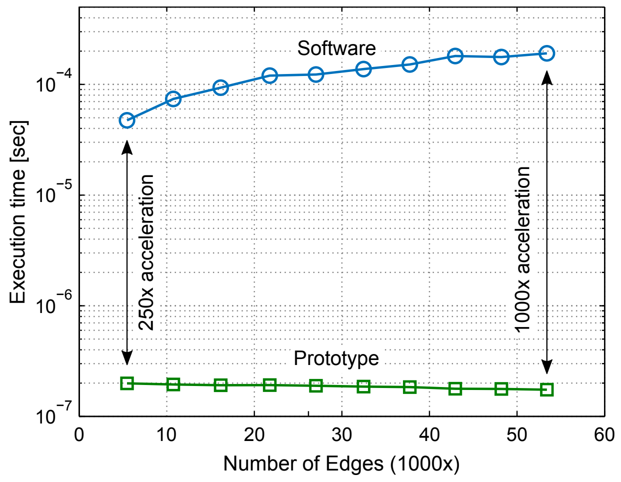


Fig. 6: Execution time of an analysis run on the network n4 at varying number of edges.

5 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 1000× acceleration compared to a conventional software implementation on real-life drug discovery benchmarks.

The main idea behind our approach is to physically embed the application graph into silicon, where each vertex is mapped into a tiny core consisting of few gates, which can communicate concurrently and directly, avoiding the memory bottleneck.

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 efficiently analyzed using the presented approach. One such example is *smart energy 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 [27] 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.

Acknowledgements

This research was funded by EPSRC Impact Acceleration Account (EP/K503885/1, project FANTASI), EPSRC Programme Grant POETS (EP/N031768/1), Newcastle University and e-Therapeutics PLC.

References

1. R. Albert and A.-L. 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. B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *Int'l Conference on Field-Programmable Technology*, 2011.
4. E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. Graphgen: An FPGA framework for vertex-centric graph computation. In *Int'l Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2014.
5. N. Kapre. Custom FPGA-based soft-processors for sparse graph acceleration. In *Int'l Conference on Application-specific Systems, Architectures and Processors*, pages 9–16. IEEE, 2015.
6. M. Lin, I. Lebedev, and J. Wawrzynek. High-throughput Bayesian computing machine with reconfigurable hardware. In *Int'l Symposium on Field Programmable Gate Arrays*, pages 73–82, 2010.
7. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.
8. Centrifuge project. GitHub page: https://github.com/tuura/centrifuge.
9. A. Mokhov. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of the International Symposium on Haskell*. ACM, 2017.
10. M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2012.
11. P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
12. N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

13. S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE, 2011.
14. 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*, page 11. ACM, 2016.
15. P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.
16. Andrey Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.
17. A. Mokhov and A. Yakovlev. Conditional Partial Order Graphs: Model, Synthesis, and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.
18. T. Ideker and R. Sharan. Protein networks in disease. 18(4):644–652.
19. E. E. Schadt. Molecular networks as sensors and drivers of common human diseases. 461(7261):218–223.
20. H. Kitano. A robustness-based approach to systems-oriented drug design. 6(3):202–210.
21. T. Tian, S. Olson, J. M. Whitacre, and A. Harding. The origins of cancer robustness and evolvability. 3(1):17.
22. R. Chen and M. Snyder. Systems biology: Personalized medicine for the future? 12(5):623–628.
23. M. P. Young, S. Zimmer, and A. V. Whitmore. Chapter 3. Drug Molecules and Biology: Network and Systems Aspects. In J. R. Morphy and C. J. Harris, editors, *RSC Drug Discovery*, pages 32–49. Royal Society of Chemistry.
24. 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.
25. R. Albert, H. Jeong, and A.-L. Barabási. Error and attack tolerance of complex networks. 406(6794):378–382.
26. P. Crucitti, V. Latora, M. Marchiori, and A. Rapisarda. Efficiency of scale-free networks: Error and attack tolerance. 320:622–642.
27. M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Google Research, White Paper*, 2016.