A Template-Based Design Methodology for Graph-Parallel Hardware Accelerators

Andrey Ayupov, Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Steven Burns, and Ozcan Ozturk

***Abstract*—Graph applications have been gaining importance in the last decade due to emerging big data analytics problems such as web graphs, social networks, and biological networks. For these applications, traditional CPU and GPU architectures suffer in terms of performance and power consumption due to irregular communications, random memory accesses, and load balancing problems. It has been shown that specialized hardware accelerators can achieve much better power and energy efficiency compared to the general purpose CPUs and GPUs. In this work, we present a template-based methodology specifically targeted for hardware accelerator design of big-data graph applications. Important architectural features that are key for energy efficient execution are implemented in a common template. The proposed template-based methodology is used to design hardware accelerators for different graph applications with little effort. Compared to an application-specific high-level synthesis (HLS) methodology, we show that the proposed methodology can generate hardware accelerators with up to 18x better energy efficiency and requires less design effort.**

I. Introduction

Customizing hardware for a target domain of applications can lead to significant power and performance improvements. Especially in the dark silicon era, hardware accelerators play an important role to significantly improve the energy efficiency of compute systems. The basic idea is to integrate specialized hardware accelerators targeted for frequently executed workloads so that these workloads can be executed orders of magnitude more efficiently compared to general purpose CPUs. IBM's Wire-Speed Processor [11] is an example, where a number of fixed-function hardware accelerators are integrated with processing cores.

The importance of graph applications have been increasing in the last decade especially due to emerging big data problems such as knowledge discovery for web data, social network analysis, natural language processing, and bioinformatics. The typical objective is to extract information from interactions (graph edges) between different entities (graph vertices). Graph analysis is known to be different from traditional grid-based high performance computing because of irregular communication, little data locality, low computation to communication ratio, frequent synchronization requirements, and hard-to-predict work assignment [13].

The performance bottleneck of big data graph applications is typically the DRAM access latency due to the low compute-to-memory ratios and random memory access patterns [7]. Most previous works on hardware accelerators assume that data resides at a local memory with fixed latency. This is typically ensured by processing one partition at a time and overlapping computation of the current partition with the communication of the next partition. However, the large graphs of big data applications and the poor temporal/spatial locality of these real-world graphs make the aforementioned techniques impractical. Instead, an accelerator is expected to generate many concurrent DRAM requests to be able to hide long (typically hundreds of cycles) latencies and fully utilize the available DRAM bandwidth. It has been shown that hardware accelerators can operate at power levels that are much lower than the state-of-the-art multi-core CPUs [26].

Let us consider a design company that needs to choose which algorithms to accelerate in hardware, e.g. for custom server chips. Let us also assume that this decision will be made based on analyzing the performance benefits and the power/area costs for each application. Following the traditional RTL-based design methodology can lead to weeks or months of development time just to obtain accurate power, performance, and area estimation for one application. This development time can be acceptable for final hardware implementation. However, if there are many potential applications that need to be evaluated this way, this methodology becomes impractical. Instead, one can use High Level Synthesis (HLS) tools to model each application at a higher level to reduce the development and debug time. Although HLS tools are very effective for applications with regular compute patterns [15, 23, 27, 30], they still require low level modeling - hence high development costs - for irregular applications such as graph algorithms, as will be discussed in Section III.

In this paper, we propose a template-based automation methodology for the design of hardware accelerators for graph applications. For this, we rely on the software abstraction models proposed for distributed graph processing frameworks [20].

The contributions of this paper can be summarized as follows:

- We show the limitations of HLS-based accelerator design methodologies for irregular graph algorithms.
- We propose a template-based design methodology for iterative graph algorithms. Since the template is created only once and utilized across many applications, the design effort is amortized. This allows easily incorporating into the generated accelerators important architectural features that are expensive to develop.
- We propose a design-space exploration algorithm that is specifically targeted for graph accelerator architectures. This helps designers choose the desired tradeoff between per-

A. Ayupov, T. Kim, and S. Burns are with the Strategic CAD Labs of Intel Corporation, Hillsboro, OR.

S. Yesil is with the University of Illinois at Urbana-Champaign, IL.

M. M. Ozdal and O. Ozturk are with Bilkent University, Ankara, Turkey.

This work was partially supported by European Union H2020-MSCA-IF Program under grant number 704476.

formance and area/power without the need for low-level parameter tuning.
- We provide a detailed power, performance, and area comparison of graph accelerators generated using 22nm industrial libraries for three different applications. The comparison is done between hardware generated by 1) direct HLS methodology using traditional bulk-synchronous models, 2) the proposed template-based methodology, and 3) a state-of-the-art CPU system. We show that the direct HLS methodology requires more design effort to implement a simpler accelerator architecture, while the proposed methodology can hide the architectural complexities within the common template and achieve up to 18x better energy efficiency.

The rest of the paper is organized as follows. In Section II, we provide background on irregular graph applications and summarize the previous studies in this area. In Section III, we outline the challenges of using an HLS methodology to design accelerators for graph applications. We also describe a generic pipeline for a baseline HLS architecture in that section. In Section IV, we briefly describe the proposed architecture for irregular graph applications. Our proposed template-based design methodology is presented in Section V. Finally, our experimental results are provided in Section VI.

II. Background and Related Work

Efficient execution of a graph algorithm requires both high throughput computation and work efficiency. Throughput of computation can be defined as the number of vertices or edges processed per unit time, and existing studies typically focus on this metric. On the other hand, work efficiency is defined based on the number of vertices or edges processed to complete a given task. This metric is especially important for iterative graph algorithms where a certain convergence criteria needs to be satisfied for completion. There are two factors that directly affect work efficiency of iterative graph algorithms: *asynchronous execution* and *active vertex set*.

In a bulk synchronous implementation of a graph algorithm, there are global barriers between iterations, and only the data from the previous iteration can be used. In contrast, an asynchronous implementation allows vertices to access the latest data from neighbors, allowing them to see the updates done in the same iteration. It was shown that asynchronous execution can lead to about 2x faster convergence for some graph algorithms [20, 25].

Another implementation aspect that affects work efficiency of iterative graph algorithms is whether all vertices are processed in every iteration or not. It was shown in the aforementioned works that vertices converge at different speeds and significant work efficiency can be achieved by not processing the vertices that converge earlier than others. For this purpose, the set of *active* (not-yet-converged) vertices can be maintained. The architectural requirements for such graph applications were expressed in [25] and it was shown that the features that lead to better work efficiency may lead to lower throughput on multi-core CPU systems.

Other related work can be summarized as follows. Graph processing is a widely studied problem at both software and hardware levels. A wide range of distributed software frameworks are available, such as Google's Pregel [21] and Graphlab [20]. Software-level optimizations for existing massively parallel architectures have been proposed such as [19] and [34]. There have also been several proposals for accelerators for specific graph applications. Both PageRank [22] and variations of breadth-first-search and single-source shortest path algorithms [6, 8, 17, 31, 33] have been implemented for FPGAs and ASICs. In addition, there are other studies that attempt to provide abstract templates for regular graph processing [12, 24]. However, these models focus on a bulk synchronous execution model and do not consider the work efficiency aspects described above. Furthermore, they are not well-suited for irregular graph applications with poor-locality of memory accesses.

III. HLS-Based Accelerator Design

Although HLS tools can be used effectively to design accelerators for compute-oriented applications with regular memory access patterns, it is hard to use them directly on the software implementations of irregular graph applications. First of all, it is not viable to store very large graphs in local memories of an accelerator. Furthermore, as mentioned in Section I, processing one partition at a time is not practical due to the irregular memory access patterns. Hence, the designed accelerator needs to be able to make requests to system memory and be able to hide access latency by scheduling multiple memory requests concurrently. In this section, we describe how to model graph accelerators using HLS tools directly as an alternative to the proposed template based methodology.

In this paper, we assume that the input graphs are stored in the well-known Compressed Sparse Row (CSR) format, where two arrays are used to store the graph topology, denoted in this paper as *VertexInfo* and *EdgeInfo*. Entry *VertexInfo[i]* stores the topology information for vertex *i*: the corresponding edge offset in *EdgeInfo* array and the number of incoming/outgoing edges (for directed graphs). Similarly, *EdgeInfo[i]* stores the neighboring vertex index for edge *i*. In addition, the application data associated with each vertex and/or edge is stored in arrays *VertexData* and *EdgeData*.

For HLS implementations, we have implemented a simple memory subsystem to access DRAM. This model supports multiple outstanding requests to enable memory-level parallelism so that long DRAM latencies can be hidden. For processing vertices, we have adopted the bulk synchronous processing (BSP) model of execution due to its simplicity. In this model, two copies are stored for each vertex/edge data object, corresponding to the previous and current iterations. When a vertex needs to access the data of a neighbor vertex/edge, it reads the data from the previous iteration. Since iterations are separated by barriers, race conditions are avoided.

The microarchitecture we use for the HLS-based design of an application-specific graph accelerator can be represented as the pipeline shown in Figure 1. In this pipeline, vertices are processed in order, however there can be up to 128 vertices being processed at different stages of the pipeline at the same time. Processing many vertices in parallel results in many memory requests originating from different stages of the pipeline and helps utilize the available DRAM bandwidth. We were able to achieve close to peak memory bandwidth utilization with this architecture for the different accelerators we implemented.

Figure 1 shows a generic pipeline without implementation details. This pipeline needs to be implemented for each graph

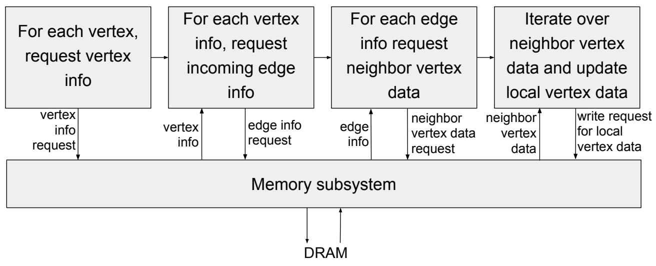
For each vertex, request vertex info
For each vertex info, request incoming edge info
For each edge info request neighbor vertex data
Iterate over neighbor vertex data and update local vertex data
vertex info request
vertex info
edge info request
edge info
neighbor vertex data request
neighbor vertex data
write request for local vertex data
Memory subsystem
DRAM

Figure 1: Architecture for application-specific HLS designs

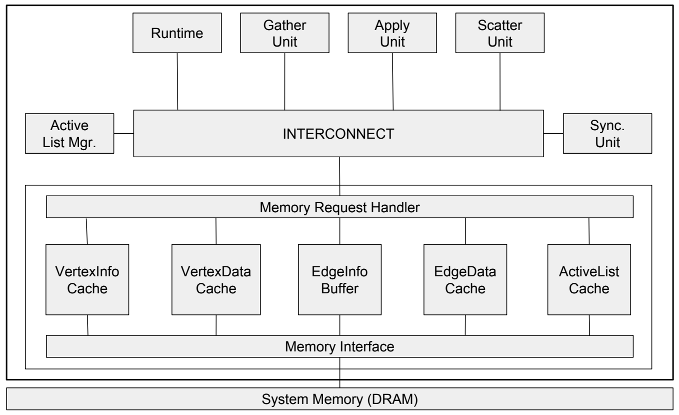

Figure 2: Accelerator Unit proposed in the preliminary version [26].

application separately by designers. Here, the VertexInfo and EdgeInfo requests can be made streaming, because the algorithms implemented process all vertices and edges in order. Neighbor vertex data is the only data type that is accessed from memory in an irregular fashion, but it is the performance bottleneck. According to our experiments, a generic cache of size up to 64KB does not provide any significant advantage due to the majority of cache accesses resulting in cache misses. Thus, we do not include a cache in the architecture of the application-specific accelerators we implemented.

Note that we chose to implement a BSP execution model instead of implementing the work-efficient asynchronous model described in Section II. Asynchronous execution requires synchronization between concurrently executing vertices and edges to ensure sequential consistency. This requires a locking mechanism for vertices to avoid Read-After-Write (RAW) and Write-After-Read (WAR) hazards. Vertex locking in turn requires out-of-order execution support for vertex processing that also makes the architecture more complicated.

Similarly, we chose not to implement the active vertex set support described in Section II. Implementing active vertex set support requires a special data structure that has at least one bit (active vs. converged) for all vertices in the graph, which needs to be stored in system memory. Given frequent accesses to the active list, some special caching capability is required with coherence support to guarantee data consistency for potentially multiple vertices requesting and modifying the vertex state from active to converged and vice versa.

Implementing asynchronous execution and active vertex set support efficiently would require cycle-level micro-architectural modeling of complex synchronization mechanisms and special data structures. Using a high-level model (e.g. SystemC) is not much different from RTL for such complex modules, and hence the benefit of HLS tools would be limited. Hence, we chose not to implement them because they are not representative of a typical HLS-based design flow.

Note that designing each accelerator using low-level RTL models requires too much design effort and is not in line with the objectives of this paper. Because of this reason, the application-specific accelerators designed using the HLS methodology of this section will be used as a baseline in our experiments (Section VI)

IV. Proposed Architecture

The preliminary version of this paper has proposed several microarchitectural features to achieve both high throughput and high work efficiency for asynchronous execution of graph applications [26]. The basic idea is to allow processing tens/hundreds of vertices/edges to be able to hide long access latencies to main memory. The potential hazards and race conditions between simultaneously executed vertices/edges are handled through special mechanisms in the architecture.

Figure 2 shows the internal architecture of a single accelerator unit proposed. This is a loosely-coupled accelerator connected to the system DRAM directly. As shown in the figure, the accelerator architecture consists of several components, which will be explained here briefly. Active List Manager (ALM) is responsible for keeping the set of active vertices that need to be processed before convergence. Runtime Unit (RT) receives vertices from ALM and schedules them for execution based on the resource availability in the system. RT sends the next vertex to Sync Unit (SYU) to start its execution. SYU is responsible for making sure that all edges and vertices are processed such that sequential consistency is guaranteed. SYU checks and avoids the potential read-after-write (RAW) and write-after-read (WAR) hazards. Then, SYU sends the vertices to the Gather Unit (GU), which is the starting point of vertex program execution. It executes the *gather* operation as will be discussed in Section V-B. An important feature of GU is that it can process tens of vertices and hundreds of edges to hide long access latencies to the system memory. It can also switch between many small-degree vertices and few large-degree vertices for dynamic load balancing. After GU is done with the gather operation of a vertex, its state data is sent to the Apply Unit(APU), which performs the main computation for the vertex. After APU is done, vertex data is passed to the Scatter Unit where the *scatter* operation (Section V-B) will be done. In this stage, the neighboring vertices can be activated (i.e. inserted into the active list) based on application-specific conditions. Similar to GU, SCU also processes tens/hundreds of vertices/edges concurrently. In addition to the computational modules, there is a special memory subsystem, consisting of caches for different data types, specifically optimized for graph applications.

Readers can refer to the preliminary version [26] for low-level details of the proposed architecture. In this extended version, the main focus is on the design automation aspects that could not be included in the preliminary version.

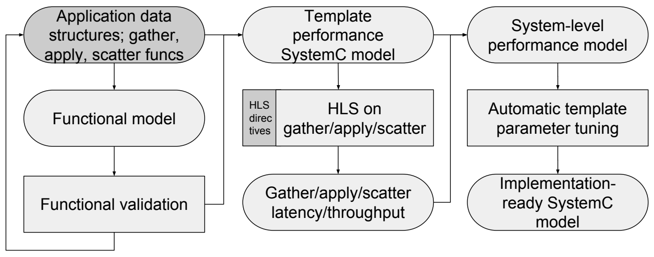
Application data structures; gather, apply, scatter funcs
Functional model
Functional validation
Template performance SystemC model
HLS directives
HLS on gather/apply/scatter
Gather/apply/scatter latency/throughput
System-level performance model
Automatic template parameter tuning
Implementation-ready SystemC model

Figure 3: The proposed template-based design flow. Color-coding: dark gray – user code, light gray – template/auto-generated models

PageRank(Input graph: (V, E))

```
1. for each unconverged vertex $v \in V$
2.     $sum = 0$
3.     for each vertex $u$ for which $(u \to v) \in E$
4.         $sum = sum + \frac{r_u}{d_u}$
5.     $r_v^{new} = \frac{(1-\beta)}{|V|} + \beta \cdot sum$
6.     if $|r_v^{new} - r_v| > \varepsilon$ then
7.         for each vertex $w$ for which $(v \to w) \in E$
8.             activate $w$
9.     $r_v = r_v^{new}$
```

Figure 4: Pseudo code of the PageRank (PR) algorithm.

V. Template-Based Methodology

In this section, we describe our template-based design methodology to implement accelerators for irregular graph applications. This template is created only once and is utilized across many applications. Hence, it is affordable to incorporate important architectural features into this template such as asynchronous execution support, lightweight synchronization mechanisms, and active vertex set, as described in Section IV. While these features significantly improve energy efficiency of the synthesized accelerators, the template-based design methodology reduces the design effort substantially.

A. Design flow

The proposed design flow is shown in Figure 3. Observe that there is a clear separation between the template implementation and the application specific models in this methodology. In Figure 3, the dark gray color corresponds to the user code, and the light gray corresponds to the template code and the automatically generated models in the design flow. The template code comes with functional and performance SystemC models. The functional model does not include microarchitectural details of the final implementation and thus is not representative of the performance of an accelerator. It is primarily used to validate functionality of the user application. The performance model is a cycle-accurate SystemC model that has all the microarchitectural details of the template modules, except the user-defined functions. Section V-D provides more details about these two models.

In the proposed design flow, the user specializes the template by providing application specific data types and implementation of some predefined functions in C language (see Section V-B for details). As an example, the user-specified code is less than 50 lines for the PageRank application. In addition, the user is responsible for providing HLS directives for the application specific functions to specify operations such as loop unrolling and pipelining. These directives can be specified as pragmas in the user source code. It is important to point out that the user has no access to the template code during this process, hence (s)he only needs to deal with the short snippets of the application specific code.

Once the template is specialized, an HLS flow is used to generate RTL for the user-defined functions for timing characterization. After the latency and throughput values for each user function are computed, they are back-annotated into the performance models to produce a system-level performance model for this accelerator. This model is then used for design space exploration, details of which will be given in Section V-E. After automatic tuning of the template parameters, the implementation-ready SystemC models can be synthesized to produce RTL using a standard HLS flow. During synthesis, the unused parts of our template are automatically discarded.

B. Programming Interface & Data Structures

The proposed template creates an abstract representation for the application program and graph data. For our programming interface, we have adopted the well known Gather-Apply-Scatter (GAS) model from [20] which is originally designed for large-scale distributed graph processing. In this model, the user needs to divide the vertex program into three logical building blocks as follows:

- ***Gather:*** The neighboring edges and/or vertices of the current vertex are processed to compute an accumulated value in this stage. The user needs to provide an accumulation function corresponding to one neighbor only.
- ***Apply:*** Main computation is done for the current vertex in this stage by using the accumulated value from the *Gather* stage. The user provides the compute function for the vertex.
- ***Scatter:*** The neighbor vertices are activated if necessary and the data computed in *Apply* stage is distributed to the neighboring vertices and/or edges. The user needs to provide the scatter function for a single neighbor only.

In addition to the three logical blocks, the programming interface includes helper functions. Both *Gather* and *Scatter* have two helper functions each: *GatherInit* and *ScatterInit* functions are used to initialize the corresponding states for the *Gather* and *Scatter* stages; *GatherFinish* and *ScatterFinish* functions update the associated data for the current vertex and finalize the states for the *Gather* and *Scatter* stages.

To store the graph data in memory and to facilitate communication between computational blocks, the template has eight different data types: *PrivateVertexData* stores data associated with a vertex that can be accessed by only the corresponding vertex; *SharedVertexData* stores data associated with a vertex that can be accessed by the neighboring vertices; *VertexData* is a combination of PrivateVertexData and SharedVertexData; *EdgeDataG* corresponds to the data associated with an incoming edge of a vertex and is used in the Gather stage; *EdgeDataS* corresponds to the data associated with an outgoing edge of a vertex and is used in the Scatter stage; *GatherState* stores the intermediate state used in the Gather iterations and it is also passed to the Apply stage; *ApplyState* is the state computed

GlobalParams:	ε:Real *// error threshold*
	rankOffset:Real *//* $= r_u/d_u$
	β:Real
PrivateVertexData:	oneOverDegree:Real *//* $= 1/d_v$
SharedVertexData:	scaledPR:Real *//* $= r_v/d_v$
VertexData:	**PrivateVertexData** $\cup$ **SharedVertexData**
ApplyState:	doScatter:Bool
GatherState:	accumPR:Real *// accumulated rank value*

Figure 5: Data types of PageRank application for our template.

```
gather_init()
  Output GatherState:gst
1.    $gst.accumPR = 0$  // initialize accumulated rank

gather()
  Input SharedVertexData:nvd  // neighbor's data
  Output GatherState:gst
2.    $gst.accumPR += nvd.scaledPRValue$

apply()
  Input GatherState:gst
  Input/Output VertexData:lvd  // local vertex data
  Output ApplyState:ast
3.    $newRank = rankOffset + \beta * gst.accumPR$
4.    $newRankScaled = newRank * lvd.oneOverDegree$
5.    if $|newRankScaled - lvd.scaledPR| > \varepsilon$
6.    then
7.      $ast.doScatter = true$
8.    end
9.    $lvd.scaledPR = newRankScaled$

scatter()
  Input ApplyState:ast
10.   if $ast.doScatter == true$
11.   then
12.     $activateNeighVtx = true$
13.      // send activation for neighbor
14.   end
```

Figure 6: Pseudocode of the PageRank application for our template.

in the Apply stage and passed to the Scatter stage; and finally *ScatterState* stores the intermediate state used in the Scatter edge iterations.

These data types can be categorized as *graph data* and *local data* where *GatherState*, *ApplyState*, and *ScatterState* are *local data* and others are *graph data*. The *local data* types are used as intermediate states and for transferring data between different compute blocks. On the other hand, *graph data* types are used to store the graph structure in memory.

Note that all user-level data structures and functions are specified using plain C, without SystemC constructs.

C. Case Study: PageRank (PR)

PageRank is an important web page ranking algorithm. PR calculates the importance of a page by analyzing the other pages that are pointing to it. PR works on a sparse graph of web-pages. A generic implementation of PR is shown in Figure 4. Here, r_u denotes the page rank of vertex u; d_u denotes the degree of (# of edges connected to) vertex u. The parameters β and ε denote the rank scaling coefficient and the error threshold, respectively.

The corresponding implementation of PR using our programming interface is shown in Figures 5 and 6. Here, the global parameters such as the error threshold (ε), initial rank offset (*rankOffset* = $(1-\beta)/|V|$), and the rank scaling coefficient (β) are specified in the *GlobalParams* structure. The *gather* stage starts by initializing the page rank value for the current vertex to zero (*gather_init()*). Then, for every incoming edge, the page rank value is accumulated (*accumPR*) using the neighbor's scaled rank value (*nvd.scaledPRValue*). In the *apply* stage, the new page rank value is computed using the formula in line 5 of Figure 4. If the page rank value has changed beyond the given threshold, then the corresponding flag (*doScatter*) is set and sent to the scatter stage as part of the apply state. A true value for this flag triggers the scatter loop to be processed over all outgoing edges of the current vertex. In the *scatter* function corresponding to an edge, activating the neighboring vertex is decided based on the value of the *doScatter* flag. In other words, if the rank value of the current vertex has changed beyond the ε threshold, all outgoing neighbors are scheduled for future execution. Note that even though our programming interface is based on C language, only pseudo codes are provided here for clarity of the figures.

D. Functional and performance simulators

The functional and performance simulators are based on the functional and the performance models from Figure 3 respectively. The functional model implements the gather-apply-scatter pattern in a serial fashion. The performance model is a cycle-accurate model and is used to estimate actual performance of the accelerator. Due to its simplicity, the runtime of the functional simulator is much faster than the performance simulator. Therefore, we expect the user to validate the functionality of the application data types and functions with this model.

The proposed template architecture is configurable through microarchitectural parameters. It is possible to vary these parameters for an application and observe the effects on performance by running SystemC simulation of the performance model. Since the exploration space is large, we also propose an automatic design space exploration framework that will be explained in the next subsection.

E. Design Space Exploration

There are several microarchitectural parameters in the template proposed. The first set of parameters is the number of vertices/edges that can be processed concurrently in the gather/scatter stages. As mentioned earlier, processing multiple vertices/edges in an accelerator unit allows memory level parallelism to hide the DRAM access latencies. However, a temporary execution state needs to be stored locally in the accelerator unit corresponding to each vertex and edge processed. In other words, increasing these parameter values can improve performance in exchange for a larger local storage. The other set of parameters are the cache sizes corresponding to different graph data types. If a certain data type has good access locality for an application, it makes sense to increase the corresponding cache size to improve performance in exchange for extra area and power.

```
Design_Space_Exploration()

1.  Metric $m_1 \leftarrow$ maximize $T/A$
2.  $(T_0,A_0) \leftarrow$ OptimizeParams($m_1$)
3.  $i \leftarrow 1$
4.  for each $\alpha_i$ in the input parameter set do
5.      Define metric $m_2$: maximize $(T/T_0 - \alpha_i A/A_0)$
6.      $(T_i,A_i) \leftarrow$ OptimizeParams($m_2$)
7.      Add $(T_i,A_i)$ to the Pareto curve
8.      $i \leftarrow i+1$
9.  return Pareto curve
```

Figure 7: The high-level algorithm for design space exploration.

The tradeoff between performance and area/power is application specific and depends on different factors. For example, consider an application for which the main performance bottleneck is the gather operation (e.g. *PageRank*). It makes sense to increase the number of vertices/edges that can be processed concurrently in the gather stage for such an application. On the other hand, if the VertexData accessed during the gather operation has poor memory access locality, it makes sense to keep the corresponding cache size small to save area and power. The exact parameter values chosen should depend on both the application characteristics and the desired tradeoffs. In particular, the designers need to consider how much extra area/power they are willing to pay for a certain amount of increase in performance.

When all the microarchitecture parameters are considered, the exploration space is quite large, and tuning the parameters requires an inherent knowledge of both the application characteristics and the template microarchitecture. We propose an automatic design space exploration methodology to shield the designers from these details. For a given application, our methodology generates a Pareto curve between performance and area/power at different design points. After that, the designer can choose the desired tradeoff and the corresponding parameters for the accelerator.

In the rest of this section, we use throughput (i.e. number of vertices or edges processed per second) as proxy for performance and the hardware area as proxy for power. However, our methodology is applicable for different metrics as well. To estimate the throughput of an accelerator for a specific set of parameters, we run our performance simulator long enough such that the effect of the warm-up period on the measured performance is negligible. To estimate the area corresponding to a set of parameters, we first characterize the area impact of each parameter by running synthesis (for compute unit parameters) and using the available cache models (for cache size parameters).

For the purpose of generating a Pareto curve, it makes sense to maximize a linear function of throughput (T) and area (A) such as: $(T - \alpha A)$. By varying the α parameter, one can generate different design points with different tradeoffs. The advantage of such a formulation is that it has a well-defined mathematical property: The slope of the Pareto curve at a particular point must be equal to the α value used to generate that point. However, the disadvantage is that such a function is not intuitive because T and A have different units and it is not clear how to choose the range of α values.

For this reason, we propose an exploration methodology, as shown in Figure 7. In this methodology, we use an iterative optimization algorithm (denoted as *OptimizeParams* in the figure) to compute the parameters that maximize a given objective metric. This is a greedy algorithm that starts from an initial design point, evaluates an incremental change for each parameter and chooses the change that maximizes the given metric at every iteration. The iterations continue until the improvement obtained is below a certain threshold. Our experiments have demonstrated that such a greedy algorithm is sufficient in our exploration framework. The architectural parameters we considered in this framework are as follows: the number of concurrently processed vertices and edges in the gather and scatter stages (four different parameters), the sizes of the caches for *VertexInfo*, *VertexData*, *EdgeInfo*, and *EdgeData* (four different parameters).

In the first step of the proposed exploration methodology in Figure 7, the well-known power-delay-product metric[1] is used to compute the reference values for throughput and area, denoted as T_0 and A_0. Then, we perform a number of iterations to generate a Pareto curve around the reference point. For this, we use a linear optimization metric where each term is normalized with respect to the reference values. The α value in this metric determines the tradeoff between the change in throughput and area. In our experiments, the α value is varied between 0 and 5. Here, $\alpha = 0$ is one extreme, where the objective is to maximize throughput without considering the impact on area, whereas $\alpha = 5$ is the other extreme, where every $x\%$ increase in area must be compensated by at least $5x\%$ improvement in throughput. By normalizing throughput and area with their reference values, we believe that the optimization metric and the meaning of the α parameter has become more intuitive for the users.

Note that each α value considered in Figure 7 generates a point on the Pareto curve. The Pareto curve generated for the PageRank accelerator is shown in Figure 12 as an example.

VI. Experimental Study

A. Applications

In our experiments, we have evaluated three graph applications: Single-Source Shortest Path (SSSP), Stochastic Gradient Descent (SGD), and PageRank (PR). Brief descriptions of SSSP and SGD are given below, while the detailed description of PR was given in Section V-C.

Single Source Shortest Path (SSSP): The distance values of all vertices are initialized to be infinity except for the source vertex, which has distance of zero by definition. A vertex updates its distance value based on the distance values of its neighbors using the formula in expression (1). In our template implementation, the distance value of a vertex is stored in the corresponding vertex data. When the distance value of a vertex is changed, the neighbors of the vertex are activated.

$$dist^{t+1}(v) = min_{(u,v)\in E} dist(u) + weight(u,v) \tag{1}$$

[1]As mentioned earlier, we use area as proxy for power and 1/throughput as proxy for delay in this section. Hence the actual maximization metric becomes T/A.

Table I: Datasets used in our experiments.

Application	Dataset	# Vert.	# Edges
PageRank SSSP (Directed)	wg	916K	5.1M
	pk	1.6M	30M
	lj	4.8M	69M
	g24	16.8M	268M
SGD (Undirected)	1M	9.7K	1M
	10M	80K	10M

Stochastic Gradient Descent (SGD): SGD is a matrix factorization technique which is used in recommender systems. The aim of SGD is to compute a feature vector for each user and item based on the ratings provided as training data. Then, using these feature vectors, the missing ratings between other users and items can be estimated. This application takes a bipartite graph as input, where the vertices correspond to users and items, respectively, and the edges correspond to the given ratings in the training data. The algorithm starts with a random feature vector for each vertex, and it iteratively updates them using expressions (2) and (3). Here, the estimation error for rating r_{ui} is denoted as e_{ui} in (2), and the feature vector for vertex u (v_u) is updated based on the gradient function in (3). These functions are implemented in the *gather* stage of our template, where the gradient value for the current vertex is accumulated over all neighbors. Gather also keeps track of the maximum error observed, and the neighbors are updated in the *scatter* stage if the error is above a certain threshold.

$$e_{ui} = r_{ui} - \langle v_u, v_i \rangle \tag{2}$$

$$v_u = v_u + \gamma(e_{ui}v_i - \lambda v_u) \tag{3}$$

B. Experimental Setup

We have compared the performance of the generated accelerators with manually implemented HLS accelerators and a state of the art Ivy Bridge server system. Details of the execution environments are as follows:

1) CPU: The CPU measurements are collected on a 2-socket 24-core Ivy Bridge server with the following cache sizes per socket: 768KB of private L1, 3MB of private L2, and 30 MB of shared L3 cache. The total DRAM size is 132GB. For both PageRank and SSSP, we used the implementations from the Berkeley GAP benchmark [2]. Furthermore, we improved the PageRank implementation by adding a bit vector to keep track of the active vertices to improve convergence. For the SGD application, we used DSGD implementation from [14] since it is one of the best implementations available as stated in [29]. For all applications, we have used gcc 4.9.1 and enabled -O3 level optimizations. During execution, we set NUMA policy to distribute memory allocation for the graph to maximize the memory bandwidth utilization.

2) Template Acc: The generated accelerators are created by the proposed template-based methodology (Section V). The accelerators for all applications are composed of 4 Accelerator Units (AUs) and are customized per application.

3) HLS Acc: The accelerators are generated by the HLS flow from a manually-coded SystemC description (Section III). The SystemC code is implemented for each application independently and the memory subsystem code is reused. The memory subsystem consists of a simple load/store mechanism for each graph data type and supports the following features: packing/unpacking of streaming graph data types (e.g. *EdgeInfo* and *VertexInfo*) to/from cache lines; up to 128 outstanding cache line requests from/to the main memory, and the round-robin-based arbitration of multiple requests/responses from load/store units from/to the main memory.

Datasets: In our experiments, we have used datasets which are either obtained from well-known graph databases or generated by widely used tools. For example, Pagerank and SSSP work on directed graphs. We have selected *WebGoogle(wg)*, *soc-Pokec(pk)* and *soc-LiveJournal(lj)* datasets from SNAP [18] graph database. Additionally, we have generated a large graph (g24) by using the Graph500 [3] tool.

For SGD, two movie datasets are used from MovieLens [4] which have 1 million and 10 million movie ratings. Details of the selected graphs can be found in Table I.

C. Experimental Results

1) Estimation methodology

For the CPU experiments, we have run the applications on the native system and measured the runtime using OpenMP functions and the CPU power consumption using Running Average Power Limit (RAPL) [5] framework, which provides core and uncore power consumption values by reading the MSR registers. Since we are using a DDR4 memory model for our accelerators, we have provided DDR4 power consumption values for the baseline CPU system to make a fair comparison. For this purpose, we have generated DDR4 access traces that correspond to the same DDR3 bandwidth utilization of the base CPU system and fed them to the DRAMSim2 [28] tool.

For both the proposed template performance model and the application specific HLS baseline model, a commercial HLS tool was used to generate RTL from the SystemC models. We used an industrial 22*nm* technology library for standard cells and metal layers. The RTL is synthesized using a commercial physical-aware logic synthesis tool to produce the gate-level netlist as well as the timing reports that include the wire delay estimations. All accelerators operate at 1GHz frequency, and all designs can satisfy the timing constraints. For power estimations, the switching activity for all inputs and sequential elements are saved in SAIF format during RTL simulation. Then, a commercial power analysis tool is used to measure the static and dynamic power for the given switching activity files.

2) Power, Performance and Area Results

To estimate the power consumption of memory blocks in the template architecture, we have used well-known simulators CACTI [1] and DRAMSim2 [28]. We have used CACTI for estimating the power consumption of caches; however since our accelerator is synthesized for 22nm, we have scaled down the area and power values generated by CACTI from 32nm to 22nm. For scaling area, coefficient of 0.5 is used [9, 10], whereas for scaling power, coefficients of 0.569[16] (dynamic) and 0.8 [32] (leakage) are used. For DRAM power consumption of both template-based and the HLS accelerators, we have integrated the DRAMSim2 tool into our simulators and used the aforementioned DDR4 model for power and timing estimations.

Figure 8 reports the throughput of computation for the *template* and *HLS* accelerators, and the *24-core CPU* executions

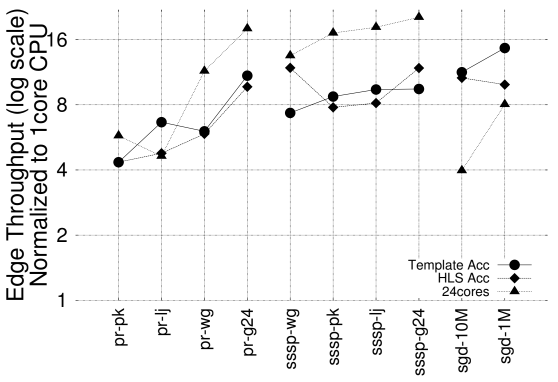

Figure 8: Edge throughput values normalized to 1 core CPU results. Note that Y-axis has log scale. Larger values are better.

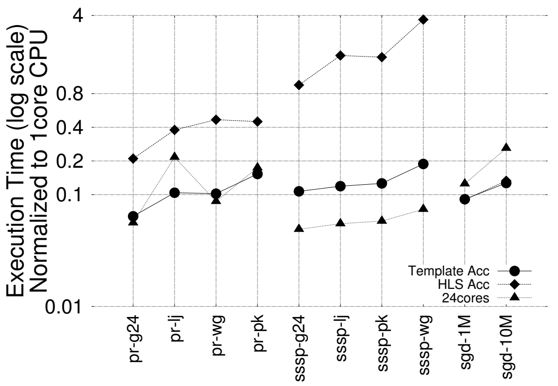

Figure 9: Execution times normalized to 1 core CPU results. Note that Y-axis has log scale. Smaller values are better.

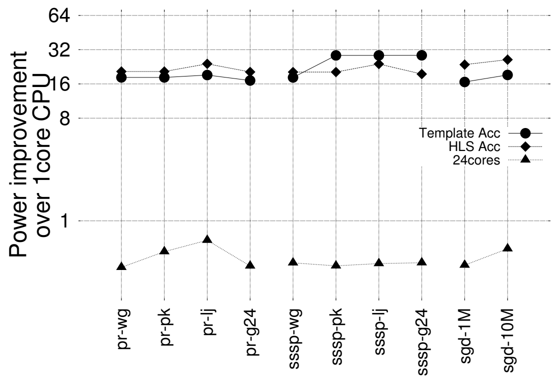

Figure 10: Power improvement, reported as 1-core power divided by the actual power for each run. Y-axis has log scale. Larger values are better.

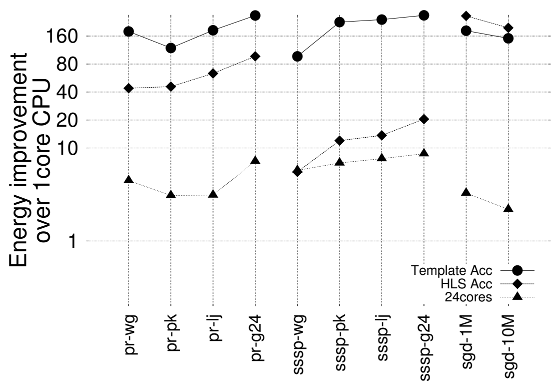

Figure 11: Energy improvement, reported as 1-core energy divided by the actual energy for each run. Larger values are better.

in terms of the number of graph edges processed per second. In Figure 9, the corresponding total execution times are given. The values in these figures are reported for each application-dataset pair and are normalized with respect to the corresponding 1-core Xeon CPU execution.

The results in these figures confirm that the edge throughput metric does not determine the performance by itself. Although both the template and HLS accelerators have comparable levels of edge throughput, the template accelerators are faster by a factor of up to 19x in terms of the total execution time. As was discussed in Section II, the total runtime depends on not only the throughput of computation but also the number of edges processed until convergence [20, 25]. Since the HLS accelerators do not support asynchronous execution and active vertex set, they end up processing many more edges until convergence as will be shown in Table II and explained below.

Power consumption comparison is shown in Figure 10. Power consumption for both the template and the HLS accelerators are dominated by the DRAM power. The computational units take less than 3% of the power values reported. Both accelerator implementations have 17-68x times less power consumption than the 24-core CPU runs. Although the power consumption of the template and HLS accelerators are similar, the template accelerators are much more energy efficient when the total execution times are taken into account. In particular, the template accelerators are up to 18x and 69x more energy efficient than the HLS accelerators and the 24-core CPU, respectively, as shown in Figure 11.

In Table II, we provide a summary of how the template accelerators compare to the HLS-based accelerators in terms of area, energy consumption, and total work metrics for all three applications. Area reported includes all the computational units, caches for the template accelerator and the memory subsystem for the HLS-based accelerator. Energy consumption and the total work metrics are measured for the largest graphs (g24 for PR, SSSP and 10M for SGD). The total work metric (the last row of Table II) is reported as the number of edges processed until convergence divided by the number of edges in the graph. It can be observed from the table that the template accelerators are more energy efficient than the HLS accelerators due to better work efficiency. Note that higher work efficiency is due to the architectural support for asynchronous execution and active vertex set. These extra features lead to 2-5 times larger areas for the template accelerators. Nevertheless, the accelerator areas are orders of magnitude smaller than the CPU area.

3) Design Space Exploration

To study the effectiveness of our design space exploration methodology, we have selected the PR application for case study. Pareto curve generated by the heuristic shown in Figure 7 is given in Figure 12. In this experiment, we swept the α values between 0 and 5.

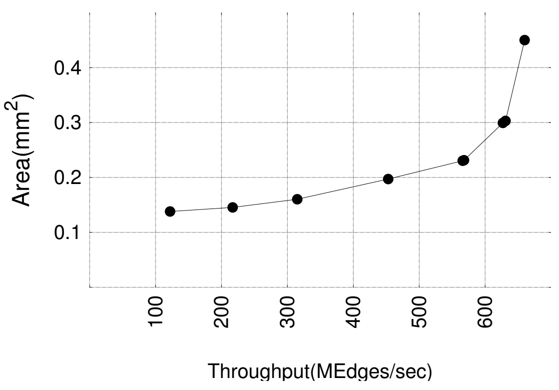

Figure 12: Pareto curve generated for PageRank using the proposed design space exploration algorithm.

Table II: Comparison of the template and HLS accelerators

	PageRank		**SSSP**		**SGD**	
	Temp. Acc.	*HLS Acc.*	*Temp. Acc.*	*HLS Acc.*	*Temp. Acc.*	*HLS Acc.*
Area *(mm2)*	0.55	0.10	0.38	0.12	1.28	0.46
Energy *(Joule)*	10.57	29.05	0.81	10.54	0.68	0.46
#edges *(norm.)*	6.89	20	2.5	7	5	5

When we consider $\alpha = 0$, area does not have any impact on the metric $T - \alpha \cdot A$, thus we expect that only throughput will be optimized. On the other hand, larger α values increase the significance of the area term. Smaller area means smaller buffers and caches, and this causes two problems: (1) smaller number of edge slots and vertex rows imply less parallelism in GU and SCU; and it is harder to hide memory access latency, (2) smaller caches increase the number of memory accesses and impose higher memory latency. Therefore, we observe smaller throughput for large α values.

While high throughput values are desirable, different projects may have different design constraints. As shown in Figure 12, our heuristic provides a range of design points that a user can select based on specific constraints. However, we can say that design points that reside in the lower right corner of this figure would represent most desirable points in practice.

4) Design Effort Discussion

In Table III, we report the number of lines used to implement an application as a proxy for the effort invested in the accelerator design. For the template methodology, it includes all application functions (gather, scatter, apply) and the application data types. For HLS, we include the application specific code excluding the memory subsystem. We also provide the size of the code of the entire template for reference. In addition, the following are our design time estimates. The template architecture, design and validation took around 12 engineer-months. Application development for the template that includes gather/apply/scatter code development, parameter exploration, and HLS runs took about 1 engineer-week per application. The HLS design took about 3 weeks for each application. Note that for the template design, a user is not required to know the hardware details of the template implementation, whereas for the application specific HLS, the user needs to develop the microarchitecture of the accelerator from scratch. In other words, a software engineer can use our template to design an accelerator, whereas for the HLS methodology, hardware design experience and SystemC expertise is required.

Table III: Lines of code

	PageRank	SSSP	SGD
Application code for template	43	34	76
Application code for HLS	735	1333	1423
Template: functional model	753		
Template: performance model	38650		

VII. Conclusion

In this paper, we propose a template-based high-level design methodology specifically targeted at designing hardware accelerators for graph applications. The main idea is to provide a simple interface for design engineers, who can model a graph application by only defining the basic data structures and high-level functions. While the user-level models are defined per vertex and edge, the parallel hardware execution is handled by the architectural template that is common for many graph applications.

The proposed methodology allows a software designer to be directly involved in the accelerator design. Since the algorithm-specific functions gather/apply/scatter used in SystemC are captured in plain C++ language, a software designer can perform algorithm development and exploration by choosing what goes in the gather/apply/scatter stages and explore various parameters as shown in the paper. The rest of the architecture is captured in SystemC with low-level hardware specific optimizations, which does not have to be modified by an algorithm developer. For an ASIC implementation, hardware designers may still need to be involved in the later stages of the design implementation process such as logic synthesis and physical design. In addition to the ASIC-based design that the proposed methodology primarily targets, the architecture presented can also be used in reconfigurable hardware platforms such as FPGAs. In this case, the software engineer can manage the entire accelerator design process by specifying the gather/apply/scatter functions in C++ language and use FPGA-specific HLS tools to generate the FPGA bitstream.

In this paper, we show that our template-based methodology can reduce the design time significantly compared to conventional HLS. Furthermore, since the template is amortized over many applications, we can afford to incorporate complex optimizations specific to irregular graph applications such as synchronization, communication, latency tolerance, worklist maintenance, etc. into the proposed template. Our results show that these optimizations can generate up to 18x better energy efficiency compared to the accelerators generated using direct HLS, while the design effort is significantly lower.

References

[1] Cacti. http://www.hpl.hp.com/research/cacti.
[2] Gap benchmark suite code. https://github.com/sbeamer/gapbs.
[3] Graph500. www.graph500.org.
[4] Movielens dataset. http://grouplens.org/datasets/movielens/.
[5] Running average power limit. https://01.org/blogs/tlcounts/2014/running-average-power-limit-%E2%80%93-rapl.

[6] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 228–235, May 2014.

[7] S. Beamer, K. Asanovic, and D. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65, October 2015.

[8] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15, July 2012.

[9] M. Bohr. Silicon technology leadership for the mobility era. In *IDF*, 2012.

[10] M. Bohr. 14nm process technology: Opening new horizons. In *IDF*, 2014.

[11] J. Brown, S. Woodward, B. Bass, and C. Johnson. Ibm power edge of network processor: A wire-speed system on a chip. *Micro, IEEE*, 31(2):76–85, March 2011.

[12] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, J. Knight, T.F., and A. DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 143–151, April 2006.

[13] J. Feo. Graph analytics in big data. In *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2012*.

[14] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 69–77, New York, NY, USA, 2011. ACM.

[15] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195, May 2008.

[16] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. In *IEEE Micro*, 2011.

[17] G. R. Jagadeesh, T. Srikanthan, and C. M. Lim. Field programmable gate array-based acceleration of shortest-path computation. *IET Computers Digital Techniques*, 5(4):231–237, July 2011.

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

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Efficient and simplified parallel graph processing over CPU and MIC.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[22] S. McGettrick, D. Geraghty, and C. McElroy. An fpga architecture for the pagerank eigenvector problem. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 523–526, Sept 2008.

[23] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP(99):1–1, 2016.

[24] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin. Graphgen: An FPGA framework for vertex-centric graph computation. In *Proc. of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 25–28, Washington, DC, USA, 2014. IEEE Computer Society.

[25] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk. Architectural requirements for energy efficient execution of graph analytics applications. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 2015.

[26] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy efficient architecture for graph analytics accelerators. In *Proc. of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[27] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119, Oct 2014.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, Jan. 2011.

[29] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[30] B. C. Schafer and A. Mahapatra. S2cbench: Synthesizable systemc benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters*, 6(3):53–56, Sept 2014.

[31] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8, Sept 2015.

[32] O. Villa, D. R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. Scaling the power wall: A path to exascale. In *Proc. of Supercomputing*, 2014.

[33] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna. A message-passing multi-softcore architecture on fpga for breadth-first search. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 70–77, Dec 2010.

[34] J. Zhong and B. He. Medusa: A parallel graph processing system on graphics processors. *SIGMOD Rec.*, 43(2):35–40, Dec. 2014.