Performance-driven System Generation for Distributed Vertex-Centric Graph Processing on Multi-FPGA Systems

Nina Engelhardt, C.-H. Dominic Hung and Hayden K.-H. So
Department of Electrical and Electronic Engineering, University of Hong Kong, Pokfulam, Hong Kong
Email: {nengel, chdhung, hso}@eee.hku.hk

Abstract—**In this paper, we present a multi-FPGA graph processing framework and an accompanying performance model. Our framework emphasizes programmability, requiring minimal user input beyond providing the application kernel and the dataset. The framework predicts the performance of the system based on the algorithm characteristics and problem size and automatically selects the optimal FPGA configuration. We implement our system on an experimental 4-FPGA platform and compare the results to the predicted performance.**

Index Terms—**Graph Processing, Multi-FPGA Architecture, Performance Modelling**

I. Introduction

With the emergence of big data and the successes of custom hardware in other areas of machine learning, interest has increased in the use of nontraditional computing platforms such as FPGAs for graph algorithms. Graph algorithms are different from most other workloads. They are irregular, highly interconnected workloads characterized by a very low computation-to-communication ratio. This plays to the strengths of FPGAs, which have a more distributed architecture with low single-thread performance but high parallelism and tight integration of high-speed interfaces.

FPGAs are notoriously challenging to program, and multi-FPGA systems only more so. It is therefore important to be able to judge in advance whether the performance improvement of an FPGA system would be worth committing to even a prototype implementation. Surveying the various implementations presented in the literature can give some idea of the capabilities of specific platforms, but much of the knowledge gained is rapidly obsolete. There are also few multi-FPGA implementations in the literature, all of which have been evaluated in simulation only[1], [2].

In this paper, we present a performance model of graph processing on multi-FPGA systems. From an analysis of the execution model of synchronous message-passing-based vertex-centric graph frameworks, we derive the computational, memory, and network limitations and construct a performance model. We extend the previously presented GraVF framework[3] with multi-FPGA capabilities, and use it to evaluate the performance of an example platform.

This work was supported in part by the Research Grants Council of Hong Kong (Project GRF 17245716), and the Croucher Foundation (Croucher Innovation Award 2013).

Organization of this paper: Section II situates our work in the context of previous efforts on FPGA-based graph frameworks. Section III describes how we model system performance. Section IV presents a concrete implementation of our system on a 4-FPGA platform and compares it to the predictions of our model. Section V discusses the limitations of our framework and model. Section VI concludes this paper.

II. Background and Related Work

An early exploration of multi-FPGA frameworks for graph processing is GraphStep[1], but due to the limited resources of chips from a decade ago, they projected needing hundreds of FPGAs. More recently, ForeGraph[2] simulated a 4-FPGA system and investigated optimizations to more efficiently use the available memory and network bandwidth.

Several single-FPGA systems have focused on improving the programmability aspect. Nurvitadhi et al. proposed GraphGen[4], the first framework for vertex-centric graph processing. However, the FPGA accelerator it automatically generates from a hardware implementation of the user kernel processes all vertices sequentially. GraphSoC[5] investigated a softcore-based approach where the vertex kernel is turned into a custom instruction. In our previous work[3], we have shown how to automatically generate custom graph processing architectures targeting single FPGAs from only a short, user-friendly kernel implementation such as shown in listing 1.

```
self.comb += [
  visited.eq(self.state_in.parent != 0),
  If(visited,
    self.state_out.parent.eq(self.state_in.parent),
    self.state_out.active.eq(self.state_in.active)
  ).Else(
    self.state_out.parent.eq(self.sender_in),
    self.state_out.active.eq(1)
  ),
  self.state_valid.eq(self.valid_in),
  self.nodeid_out.eq(self.nodeid_in),
  self.ready.eq(self.state_ack)
]
```

Listing 1. BFS kernel gather logic. No change if the vertex was previously visited, otherwise the message sender becomes parent. The remaining stages for this algorithm contain no logic, merely return their inputs unmodified.

Our improved framework uses a multi-FPGA platform to increase the size of graph that the system is able to process, while adopting a productivity-focused approach that encodes as much platform knowledge as possible within the framework.

This allows an application domain expert to produce efficient graph processing accelerators without needing to acquire systems expertise.

III. Modelling System Performance

Our framework encodes system knowledge in the form of a performance model that can predict the best system configuration for a given algorithm and problem size. The performance model is based on an analysis of the system structure. In this section, we will first give a general overview of the architecture, and then analyse the upper bounds on system throughput we can derive from this organization.

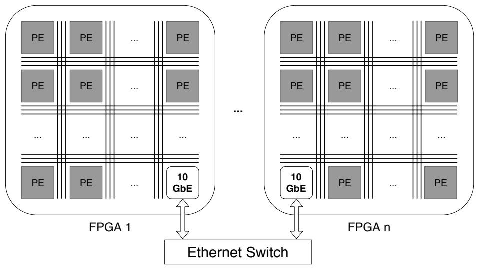

Fig. 1. System overview

We modify our previously presented GraVF [3], [6] framework by adding an external interface to the communication network that allows messages to be sent to a different FPGA over 10Gb Ethernet. The features of this framework relevant to our performance analysis are the following:

The basic computing unit is the *processing element* (PE). Each PE is assigned a set of vertices for which it will execute the vertex kernel. The PE stores the data associated with this vertex by the algorithm, and the edge data indicating the vertex's neighbors. For larger graphs, the edge data may be exported to off-chip memory, so each PE can request edge data from a shared memory controller.

In the course of execution, the PEs will send and receive messages for their assigned vertices. These messages have to be transported to the destination PE, which may be located on a different FPGA. Within an FPGA, the PEs are connected by a crossbar network. This allows all PEs to send a message in the same cycle, so that the internal network bandwidth will never limit PE performance. Messages destined for a PE on a different FPGA are routed towards an external interface. Inter-FPGA connections can be realized either directly or via a switch. For the following analysis, we assume connection via a switch that can accommodate all available FPGA boards. It can provide a maximum total network throughput of $BW_{network}$ to all boards combined. Fig. 1 shows an overview of this system architecture.

The goal of this performance model is to maximise the overall system throughput (T_{sys}) by determining the values of the following two variables: the number of FPGAs (n_{FPGA}) and the number of PEs per FPGA ($n_{PE/FPGA}$). System throughput is measured in traversed edges per second (TEPS), a common measure of performance for graph processing. While traversing one edge, the PE has the following data transfers: it receives one message, loads one edge, and sends one message. As the number of PEs in an FPGA increases, either the step of loading the edges from memory or the sending and receiving of messages over the network becomes the rate-limiting factor.

TABLE I
Definitions

Variables	
n_{FPGA}	number of FPGAs to use
$n_{PE/FPGA}$	number of PEs per FPGA
Dependent Variables	
T_{FPGA}	throughput per FPGA (edges/s)
T_{sys}	total system throughput (edges/s)
System Parameters	
CPE_{PE}	cycles per edge (edges^{-1})
f_{clk}	operating frequency (Hz)
BW_{if}	network interface bandwidth (bits/s)
BW_{if}	total network bandwidth (bits/s)
BW_{mem}	memory interface bandwidth (bits/s)
M_{board}	memory capacity per FPGA (bits)
Algorithm-dependent Parameters	
m_{vertex}	data storage per vertex (bits)
$m_{message}$	message size (bits/edge)
m_{edge}	edge size (bits)
$p_{msg/TE}$	messages per traversed edge (edges^{-1})
Dataset-dependent Parameters	
$\lvert V\rvert$	number of vertices in the input graph

The system evaluates which of the following factors limits the overall performance:

A. Processing element throughput.

Each processing element can traverse a limited number of edges per second. Analogous to a processor's CPI (cycles per instruction), we define the processing element's CPE (cycles per edge) as the average number of cycles it takes to traverse an edge. It is determined by a combination of architectural features, input graph features and message patterns. Experimentally, we determine the CPE of our PEs to vary between 1.2 and 2.

The cumulative throughput limit L_{PE} of all PEs in the system is expressed in the following formula:

$$T_{sys} \leq L_{PE} = n_{FPGA} \times n_{PE/FPGA} \times f_{clk}/CPE_{PE} \quad (1)$$

For the maximum number of PEs that fits in an FPGA, we obtain the computational limit of the FPGA, $L_{PE_{max}}$.

B. Memory bandwidth.

When using off-chip memory to store the adjacency lists, the FPGA throughput T_{FPGA} may also be limited by the memory bandwidth. Traversing one edge requires loading one edge of size m_{edge} from memory:

$$T_{FPGA} \times m_{edge} \leq BW_{mem}$$

From which, by multiplying with the number of FPGAs in the system, we derive the memory interface limit L_{mem}:

$$T_{sys} \leq L_{mem} = n_{FPGA} \times \frac{BW_{mem}}{m_{edge}} \tag{2}$$

C. Network interface bandwidth.

The PEs have to exchange messages among each other. A messsage will be sent for each traversed edge. This message also has to be received on another FPGA, leading it to traverse an interface twice. In the absence of elaborate partitioning, a message is equally likely to be sent to any other PE. Therefore, a fraction $\frac{n_{FPGA}-1}{n_{FPGA}}$ of messages has to traverse the inter-FPGA network. Each FPGA's throughput T_{FPGA} is limited by its network interface(s):

$$2 \times T_{FPGA} \times \frac{n_{FPGA}-1}{n_{FPGA}} \times m_{message} \leq BW_{if}$$

Thus we get the network interface limit L_{if}:

$$T_{sys} \leq L_{if} = \frac{BW_{if}}{2 \times m_{message}} \times \frac{n^2_{FPGA}}{n_{FPGA}-1} \tag{3}$$

D. Total network bandwidth.

The overall network bandwidth of the switch is limited. The cumulative messages sent by all the FPGA boards cannot exceed it:

$$n_{FPGA} \times T_{FPGA} \times \frac{n_{FPGA}-1}{n_{FPGA}} \times m_{message} \leq BW_{network}$$

$n_{FPGA} \times T_{FPGA} = T_{sys}$, therefore we have the network limit $L_{network}$:

$$T_{sys} \leq L_{network} = \frac{BW_{network} \times n_{FPGA}}{(n_{FPGA}-1) \times m_{message}} \tag{4}$$

Overall, taking into account all four factors, we can derive the predicted maximum throughput of an FPGA:

$$T_{sys} = \min(L_{PE}, L_{mem}, L_{if}, L_{network}) \tag{5}$$

Since we would not want to computationally limit the system while there are resources available, in a first step we assume we use the maximum number of PEs available, replacing L_{PE} with the constant $L_{PE_{max}}$.

$L_{PE_{max}}$, L_{mem} and L_{if} increase with n_{FPGA}, whereas $L_{network}$ decreases with n_{FPGA}. Therefore, there are four candidates for the optimum value of T_{sys}. As the network limits are not defined for $n_{FPGA} = 1$, it might be the best solution. Otherwise, it will be obtained at the point where the network limit and one of the other three limits intersect:

$$L_{PE} = L_{network} \text{ or } L_{mem} = L_{network} \text{ or } L_{if} = L_{network}$$

Each of the cases is solved individually for n_{FPGA}, and the resulting values of T_{sys} compared to find out which of the four is the real solution. Rounding up to the nearest integer if necessary, we obtain the value of n_{FPGA}.

If L_{mem} or L_{if} is found to be the limiting factor, once the number of FPGAs is chosen, the number of PEs per FPGA can be lowered to save power. Substituting the values of T_{sys} and n_{FPGA} calculated in the previous step into equation 1 gives a lower bound on the number of PEs per FPGA necessary to maintain the same performance:

$$n_{PE/FPGA} \geq \frac{T_{sys} \times CPE_{PE}}{n_{FPGA} \times f_{clk}}$$

In this manner, the framework chooses the optimal values for the number of FPGAs (n_{FPGA}) and the number of PEs per FPGA ($n_{PE/FPGA}$), optimizing first for throughput and then for power.

IV. Experimental Results

We compare the predictions by our model to the results obtained on a system consisting of 4 AlphaData ADM-PCIE-7V3 Virtex-7 FPGA boards interconnected with 10Gb Ethernet by a Dell Networking S6000 40GbE Switch. For use in our model, we need to determine the effective network interface bandwidth, which is measured by sending the same data repeatedly and counting the amount of received packets in a specified time interval. $BW_{network} = 6.7$ Gbps. The operating frequency is set to 100MHz.

We run a set of three algorithm kernels to evaluate the performance of the system: PageRank (PR), Breadth First Search (BFS), and Connected Components (CC). All three are widely used in the literature for this purpose, allowing our work to be compared easily to others.

We use two input graphs for our experiments. The first is a scale-free graph generated using the RMAT generator from the graph500 benchmark[7] with scale 15 and edgefactor 16. (This size has been chosen as the largest that fits in block RAM of a single FPGA.) The second is a uniform graph with the same number of vertices and edges.

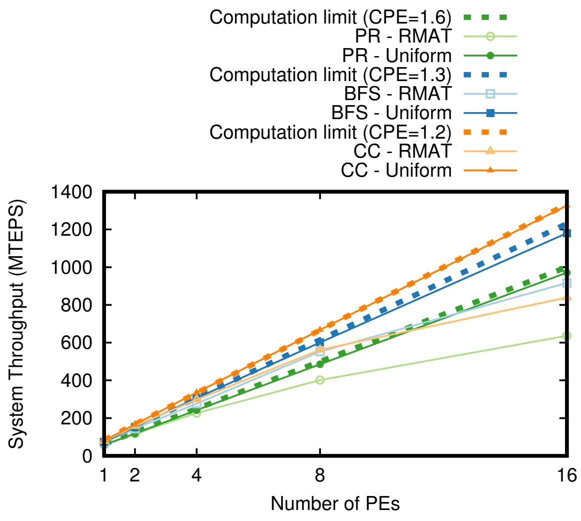

Fig. 2. Single-FPGA PE performance

1) Computation bound: First, we explore the computational limitations by varying the number of PEs on a single ADM-PCIE-7V3 board. Fig. 2 shows the computation limit predicted by the model, and the observed performance for both the scale-free (RMAT) and the uniform graph. The performance on the uniform graph closely matches the predicted performance, but the increasing imbalance in computational workload (over 20% at 16 PE) affects the performance on scale-free graphs.

2) Network bound: In our second experiment, we run the benchmark on a system of four ADM-PCIE-7V3 cards. Fig. 3 shows the results of using 1, 2, 3 or 4 FPGAs.

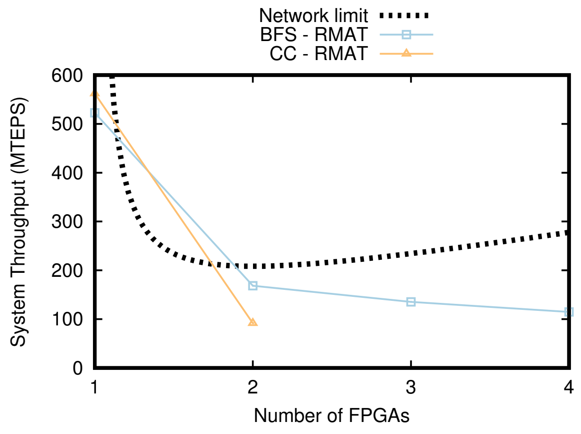

Fig. 3. Performance when increasing the number of FPGA boards

Due to the limited network interface bandwidth, the expected incremental improvement of adding more FPGAs is small. The observed performance is lower than expected, most likely because the worsening computational imbalance not accounted for in the model overwhelms the small gains. However, the model predicts correctly that using a single FPGA will lead to the best performance.

V. Discussion

A. Limitations

The model uses an upper bound on performance for its estimations. Various factors can reduce the performance, such as the effect of workload imbalance shown above. This might lead the model to mispredict the most efficient configuration, generally by underestimating the number of PEs needed. If the magnitude of the effect is known, this can be mitigated by factoring it into the CPE value.

Another aspect of the model is that it is entirely throughput-based, and as such cannot account for performance limits imposed by latency. Latency can become an issue especially for small datasets, or when the computation consists of a large amount of supersteps with very few active nodes each round. In the former case, a software framework would probably have good enough performance that acceleration would not be considered. In the latter case, an asynchronous framework would be recommended for this type of algorithm, rather than a synchronous one like explored here. However, the border of what constitutes a "too small" dataset is much higher for a multi-FPGA solution than for a single FPGA, as the superstep synchronization requires one round-trip over the network. To keep full throughput, enough messages need to be exchanged within a superstep to fully utilize the network resource over this period. While it would be possible to add this calculation to the model, it requires inputs that cannot easily be determined without running the computation (number of supersteps and amount of messages per superstep). Future work could investigate heuristics to determine the point when latency rather than throughput becomes limiting.

B. Extending the model to larger platforms

For a larger scale system, the assumption of a switch with enough ports would no longer hold. Instead, traditionally boards within a rack would share a switch be connected to other racks via a second layer switch. Intra-rack bandwidth would then be larger than inter-rack bandwidth. This situation mirrors the intra-FPGA versus inter-FPGA communication, leading to essentially the same analysis as in section III-C, but with FPGA in the place of PE and rack in the place of FPGA. The model is therefore easily extensible to hierarchical systems.

VI. Conclusion

In this paper, we present a multi-FPGA framework for graph processing and an accompanying performance model. We explore the model's behavior and the framework's real-life behavior across multiple configurations. The model cannot account for all of the factors influencing performance, most notably those that depend on features of the input graph, but it nevertheless arrives at the correct conclusion as to the best configuration in the cases examined. Some areas for future improvements to increase the accuracy of the model are highlighted. While spreading the computation across multiple FPGAs is not a winning proposition on our demonstration problem set, our multi-FPGA framework is a necessity to process larger graphs that cannot fit into a single FPGA. Moreover, our framework is poised to take advantage of future technology improvements such as 100GbE that will lift the limitations imposed by our current network equipment.

References

[1] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. Knight Jr., and A. DeHon, "GraphStep: A system architecture for sparse-graph algorithms," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2006.

[2] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017.

[3] N. Engelhardt and H. K. H. So, "GraVF: A vertex-centric distributed graph processing framework on FPGAs," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[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 *22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014, pp. 25–28.

[5] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015.

[6] N. Engelhardt and H. K.-H. So, "Towards flexible automatic generation of graph processing gateware," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, June 2017.

[7] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph500," *Cray Users Group (CUG)*, 2010.