FPGAs as General-Purpose Accelerators for Non-Experts via HLS: The Graph Analysis Example

Pedro Filipe Silva
Faculty of Engineering, University of Porto
Porto, Portugal
pedro.filipe.silva@fe.up.pt

João Bispo, Nuno Paulino
INESC-TEC
and Faculty of Engineering, University of Porto
Porto, Portugal
jbispo@fe.up.pt, nuno.m.paulino@inesctec.pt

2021 International Conference on Field-Programmable Technology (ICFPT) | 978-1-6654-2010-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICFPT52863.2021.9609832

***Abstract*—We discuss the concept of *FPGA-unfriendliness*, the property of certain algorithms, programs, or domains which may limit their applicability to FPGAs. Specifically, we look at graph analysis, which has recently seen increased interest in combination with High-Level Synthesis, but has yet to find great success compared to established acceleration mechanisms. To this end, we make use of Xilinx's Vitis Graph Library to implement Single-Source Shortest Paths (SSSP) and PageRank (PR), and present a custom kernel written from the ground up for Distinctiveness Centrality (DC, a novel graph centrality measure). We use public datasets to test these implementations, and analyse power consumption and execution time. Our comparisons against published data for GPU and CPU execution show FPGA slowdowns in execution time between around 18.5x and 328x for SSSP, and around 1.8x and 195x for PR, respectively. In some instances, we obtained FPGA speedups versus CPU of up to 2.5x for PR. Regarding DC, results show speedups from 0.1x to 3.5x, and energy efficiency increases from 0.8x to 6x. Lastly, we provide some insights regarding the applicability of FPGAs in FPGA-unfriendly domains, and comment on the future as FPGA and HLS technology advances.**

I. Introduction

The recent history of FPGAs shows parallels to that of general-purpose computing. If one views the FPGA as a *programmable computing device*, rather than simply a prototyping tool, the abstraction level of its programming mechanisms has been steadily increasing. From the earliest one-time fuse-based devices, to logic synthesis and EDA, hardware engineers have been able to build increasingly complex systems thanks to such advances in abstraction. High-Level Synthesis (HLS) is now reaching the software mainstream [1], and the gap between software and hardware engineering grows ever smaller at the FPGA boundary. Could the FPGA ever be thought of as a general-purpose accelerator? Could it combine ease of programming with efficiency rivalling that of devices such as GPUs or Tensor Processing Units (TPUs)?

Our work focuses on applying FPGAs to High-Performance Computing (HPC) applications. To that end, we restrict ourselves to server-grade FPGA boards. In order to simplify discourse, we refer to an FPGA acceleration board simply as an "FPGA".

A. FPGA-Unfriendliness

We define *FPGA-unfriendliness* (or simply *unfriendliness*) as the property carried by certain algorithms, programs, or entire application domains which makes their implementation/acceleration using FPGA technology difficult, less performant, or both, when compared to more traditional execution platforms [2]. Several factors related to the FPGA technology and ecosystem (whether intrinsic to FPGAs themselves or not) contribute to this unfriendliness:

1) Hardware limitations (e.g. low clock rates, limited global memory bandwidth, limited local memory capacity);
2) Synthesis limitations: at the HLS level, mostly refers to abstraction gaps between the high-level language source (e.g. C/C++) and the low-level target (e.g. Verilog); at the RTL level, refers to lack of abstraction which hinders the implementation of complex algorithms;
3) Software/IP limitations: lack of dedicated frameworks/libraries for unfriendly algorithms; libraries not sufficiently optimised; missing features in runtime, shell/board support packages, or development environments;

These are *originating* factors. A problem must be *affected* by these factors to be classified as unfriendly. Graph analysis, for instance, is clearly an unfriendly domain, due to, e.g. high random memory access rates – see [2], [3] for more details.

We do not claim FPGAs have no place in the software field: on the contrary, the overall outlook regarding current technology is quite positive [4]. We believe, however, that significant hardware knowledge is still a necessity to create FPGA applications in unfriendly domains. We additionally posit that high-level tools such as libraries and frameworks, which raise the abstraction level even further, are pivotal to enable viable acceleration in said domains – besides the unfriendliness aspect, a domain expert should not be expected to descend onto the kernel level too often – making the greater part of our focus evaluating instances of said tools.

Our aim is therefore to both empirically evaluate the performance of unfriendly graph algorithms against traditional execution platforms (CPU and GPU), and to provide useful insight – to both experts and non-experts – on the implementation of a new algorithm (DC). Note that while the poor performance of FPGA *vs.* GPU algorithms for *most* graph applications is commonly referred to in the field, very few publications actually demonstrate this with data.

II. Experimental Setup

A. Hardware

All tests were executed in a single machine, equipped with:

- ASRock TRX40 Creator Motherboard
- 32 GB DDR4 RAM
- AMD Ryzen Threadripper 3960X CPU
- NVIDIA GeForce GTX 1650 Super GPU
- Xilinx Alveo U250 FPGA Data Centre Accelerator Card

When comparing only time-performance, the machine was used as-is. However, when comparing time- *and* energy-performance, the graphics card was removed and the operating system was executed at runlevel 3. Idle power draw for the system in this state is approximately 115.5 W when the FPGA is programmed with the 201830_2 shell (during PR and CPU executions) and 124.5 W when programmed with the gen3x16_base3 shell (during all other instances).

B. Software

The machine is equipped with: Ubuntu 18.04 LTS (Linux version 4), Xilinx Runtime Library (XRT) 2.9.317, and Xilinx Vitis 2020.2.

C. Instruments

Power consumption was measured using a UT230B power meter. Accuracy for power measurement is ±1%. This device has no data-logging feature, so we report peak power draw.

D. Algorithms, Libraries, and Implementations

All host-side code was compiled with `gcc` 7.5.0 with `-O3`. All device-side code was compiled with `-O3`[1].

1) PageRank: PageRank (PR) was implemented using the Vitis Graph Library 2020.2 PR kernel [5], atop custom host-side code. Note that the kernel implements a *weighted* version of PR, while we're strictly evaluating the more common unweighted version, so we set all edge weights to one before executing.

2) Single-Source Shortest Paths: Single-Source Shortest Paths (SSSP) was likewise implemented using the Vitis Graph Library 2020.2 SSSP kernel [5] atop custom host-side code. We use the version which does not return predecessor data.

3) Distinctiveness Centrality:

$$D_1(i) = \sum_{j=1, j\neq i}^{n} w_{ij} \log_{10} \frac{n-1}{g_j^{\alpha}} \qquad (1)$$

To the best of our knowledge, no accelerator for Distinctiveness Centrality (DC) currently exists. Several DC algorithms are given by [6]. We have implemented DC1, which (vertex-wise) uses only data from a vertex and its neighbours. The value of DC1 for a vertex i is given by Equation 1, where w_{ij} is the weight of the edge between i and j (or zero if no such edge exists), n is the total number of vertices, g_j is the degree of vertex j, and α is a tuning parameter.

[1]All host- and device-side code, along with evaluation datasets, is available at https://doi.org/10.5281/zenodo.5155449.

TABLE I
Dataset Shorthands and Origin

Shorthand	Full Name	Source
SO	soc-orkut	[7]
SO^T	soc-orkut (transposed)	[7]
LJ	soc-LiveJournal1	[8]
USA	USA-road-d.USA	[9]
RMAT	RMAT22	[10]
R4	r4-2e23	[10]
CO	com-Orkut	[8]

As an optimisation, our kernel receives as input a Compressed Sparse Row version of the graph (as for the SSSP kernel) and reverses the computation, processing edge sets for each vertex as they appear in the matrix – in essence, computing each addend of the sum in Equation 1. This requires a nested vertex-edge loop structure.

The kernel thus outputs one result per edge: this result is binned according to its destination vertex and is then accumulated, as a histogram, by the host.

As a consequence, the final vertex-wise value only corresponds to DC1 if the graph is undirected and its matrix symmetric. Otherwise, the output relates only to incoming edges and is dubbed in-DC1.

The device-side uses dataflow. It is optimised to the level of a non-expert with *introductory* knowledge: it separates M-AXI interfaces via bundling, pipelines inner execution/read/write loops (with II=1), and uses a single compute unit which receives the entire dataset at once. The entire workflow, including kernel compilation, was performed in Vitis 2020.2.

For performance comparison, we use a single-threaded C++ version following the same algorithm implemented for FPGA execution.

E. Datasets, Parameters, and Comparison Benchmarks

Table I shows the used datasets, their source, and corresponding shorthands. Table II shows the algorithms and performance data source for each dataset, along with relevant metadata. $|E|$ means "number of edges"; $|V|$ means "number of vertices"; *Sym.* indicates whether the graph/matrix is symmetric: the edge counts given are the actual edge counts present in the matrix representation of the dataset.

When weighted graphs were required to execute SSSP, we used the random weights generated by the authors.

III. Results

We evaluate the FPGA implementation using two metrics: time- and energy-performance. We source time-performance data from [10] and [11], as well as our own tests for DC1. Energy-performance data is sourced solely from our tests. In all instances, except DC1 as the host performs some of the workload, we exclude data transfer times and report only execution times. In accordance with sourced data, all SSSP tests were executed using the first vertex as the source, and

TABLE II
AUTHOR-ALGORITHM-DATASET CORRESPONDENCE

Author	Algorithm	Dataset	$\|E\|$	$\|V\|$	Sym.
Graphit [11]	SSSP	SO^T	106M	3.00M	No
Graphit	PR	SO	213M	3.00M	Yes
Graphit	SSSP	LJ	68.4M	4.85M	No
Graphit	PR	LJ	85.6M	4.85M	Yes
Zheng [10], *(this)*	SSSP, PR, DC1	USA	57.7M	24.0M	No
Zheng, *(this)*	SSSP, PR, DC1	RMAT	33.0M	4.19M	No
Zheng, *(this)*	SSSP, PR, DC1	R4	34.0M	8.39M	No
(this)	DC1	CO	234M	3.07M	Yes

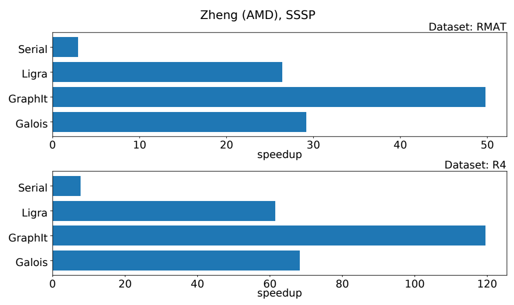

Fig. 1. Execution time for SSSP – CPU over FPGA, per dataset and framework.

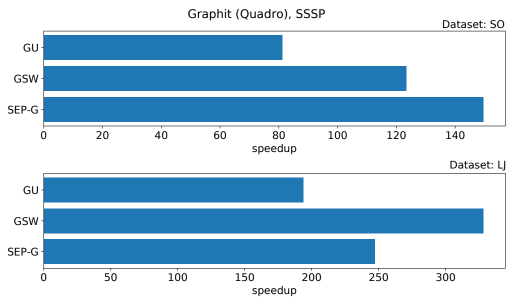

Fig. 2. Execution time for SSSP – GPU over FPGA, per dataset and framework.

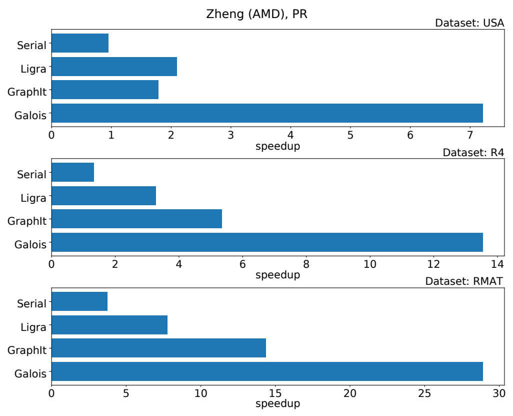

Fig. 3. Execution time for PR – CPU over FPGA, per dataset and framework.

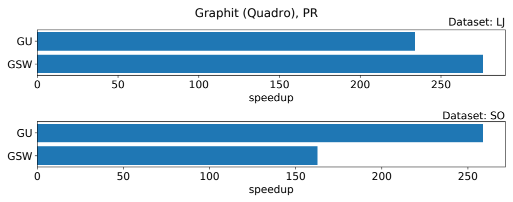

Fig. 4. Execution time for PR – GPU over FPGA, per dataset and framework.

all PR tests use the same epsilon value and maximum iteration count. All DC1 tests were executed with $\alpha = 5$.

Time-performance is displayed as the *relative speedup of the CPU/GPU implementation over the FPGA implementation*, except for DC1, where we show FPGA over CPU results. Energy-performance is given as the ratio between absolute Watt-second values. Note that both sources give results for two execution platforms. Our figures show speedups versus the most performant.

A. SSSP versus High-Performance Frameworks

Figure 1 shows SSSP time-performance versus [10]'s benchmarks of CPU frameworks (as well as unoptimised serial execution). We were unable to execute SSSP using the USA dataset after five hours – this may be be due to a conjunction of the kernel's internal use of a queue, FPGA-unfriendliness, and the graph's extreme sparsity – so it is excluded from our comparisons. Note that speedups for R4 are approximately two times lesser versus RMAT, corresponding to the ratio between vertex counts for the two datasets – thus likely indicating the SSSP kernel's suboptimal handling of increasingly sparse matrices.

Figure 2 paints a similar picture – note that SO^T is less sparse than LJ – although the GPU frameworks seem to be affected by the issue differently.

B. PageRank versus High-Performance Frameworks

Figure 3 displays results for FPGA vs. CPU PR execution. The FPGA implementation bests serial execution for USA, and slowdowns versus the other frameworks are significant, but manageable (possible excepting Galois). We note that speedups for USA versus the less-performant Intel machine given in [10] are positive for both serial execution *and* the Ligra framework.

Less encouraging are the results versus the highly optimised GPU frameworks shown in Figure 4. Both frameworks outperform FPGA execution by a large factor, as in SSSP. Our absolute figures for FPGA execution show time per iteration for SO of around 4 times the equivalent for LJ, indicating sparsity has a significant impact as well.

C. DC1 vs. Serial C++ Implementation

Figure 5 shows DC1 time- and energy-performance given as a relative speedup (or equivalent "energy factor") of FPGA execution over CPU execution. As the FPGA implementation involves computation by the host, we include host execution times (as well as memory transfer times) and host execution

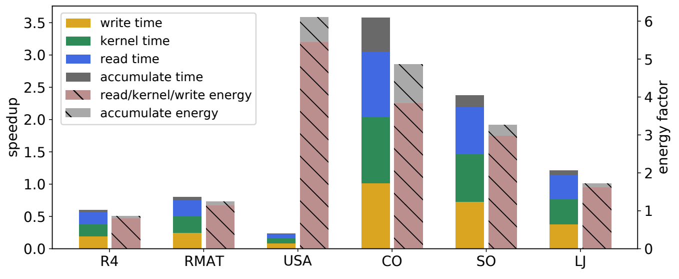

Fig. 5. Execution time/total energy expenditure, left to right, for DC1 – FPGA over CPU, per dataset.

power draw in our formula for total energy consumption. The percentage of time/energy spent during each stage is shown by differently-coloured bar segments (where a larger segment means more time/energy). One caveat is that power draw during memory transfer is higher, despite the transfer itself often being quick – especially for small datasets such as RMAT and R4 – making energy measurement during this stage difficult. Our total energy consumption numbers take this spike into account, but energy factor figures for RMAT and R4 may be skewed in favour of the CPU implementations. Similarly, FPGA execution times for all datasets except USA (11.3 s) are under 5 s. This is not sufficient to stabilise FPGA power draw measurement after memory transfer, again skewing results towards CPU execution in these instances.

Of note is our evaluation of the largest dataset, CO. Results for this dataset show a clear preference for less-sparse matrices. This is due to the deep pipeline which processes edges *per vertex* (and so favours vertexes with greater edge counts). USA, by far the sparsest matrix, has the worst speedup but the second best energy factor, as power draw has had time to stabilise enough to overcome poor performance.

Host (accumulate) time is shorter than all other stages, for all datasets. This suggests improvements to the application should mostly focus on the kernel itself. The fact that sparser graphs (USA, RMAT, R4) suffer from poor performance suggests a possible line of improvement to the main kernel execute cycle, which is vertex-oriented (and thus pays a fixed time-cost for each vertex): reshaping the computation to a fully pipelined edge-oriented cycle with no nesting. In addition, as the CSR-based DC1 formulation is fully data-parallel (vertex-wise), several compute units can be instantiated at the same time, each processing one chunk of the full dataset.

IV. Discussion and Conclusion

Overall, our results paint a negative picture. FPGA implementations of SSSP and PR significantly underperform when compared to execution with GPU accelerators, which are, for the most part, the current state-of-the-art in HPC graph analysis. Regarding CPU execution, results for SSSP are similarly poor, while results for PR may indicate a good opportunity for further development. We have also compared our own implementation of the DC1 algorithm against a serial C++ version. These results too show promise, especially with low-sparsity matrices, but one must keep in mind that, e.g., GPU DC1 may eventually outperform our FPGA code as well.

Discussion on what FPGAs are *suitable* for is commonplace, especially as the focus for HLS shifted from strict dataflow to general-purpose programming. Our intention for this text (as well as our greater study) *is to inform such discussion*. As we've mentioned, we believe higher-level tooling is key if FPGAs are to ever be suited for unfriendly applications (and thus to general-purpose computing at large). Tooling *over* HLS (not involving a library) is, in essence, a shifting of the HLS source language from C/C++ to domain-specific DSLs. At another level, it has been proposed to shift the HLS *target* from Verilog/VHDL to an intermediate representation (e.g. MLIR-based [12]) and/or Calyx [13]). This covers levels 2 and 3 of our unfriendliness classification (see Subsection I-A). Improvements at the hardware level are also a possibility (we note, for instance, the appearance of FPGA boards with High-Bandwidth Memory, which we plan to explore in the future).

Currently, a non-HLS expert must expend time learning the particularities of FPGA and HLS technology before being able to optimise their unfriendly applications much further than, e.g., the level at which we have optimised DC1 in this work. Our discussion on DC1 gives possible avenues for improvement, but these are based on *acquired knowledge* of these particularities – inaccessible to beginners. We thus hold that, if HLS is to become a fully usable technology for domain experts, even in unfriendly domains, vendors must prioritise lowering the burden of knowledge imposed on beginners.

References

[1] J. Lant, J. Navaridas, M. Lujan, and J. Goodacre, "Toward FPGA-Based HPC: Advancing Interconnect Technologies," *IEEE Micro*, vol. 40, no. 1, pp. 25–34, Jan. 2020.

[2] P. F. Silva, J. Bispo, and N. Paulino, "Building Beyond HLS: Graph Analysis and Others," in *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE '21)*, Virtual USA, Apr. 2021.

[3] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler, "Graph Processing on FPGAs: Taxonomy, Survey, Challenges," *arXiv:1903.06697 [cs]*, Apr. 2019.

[4] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 5, pp. 898–911, May 2019.

[5] Xilinx, "Vitis Graph Library," Xilinx.

[6] A. Fronzetti Colladon and M. Naldi, "Distinctiveness centrality in social networks," *PLOS ONE*, vol. 15, no. 5, p. e0233276, May 2020.

[7] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," 2015.

[8] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014.

[9] DIMACS, "9th DIMACS Implementation Challenge - Shortest Paths," 2006.

[10] R. Zheng and S. Pai, "Efficient Execution of Graph Algorithms on CPU with SIMD Extensions," in *2021 IEEE/ACM Int. Symp. Code Gener. Optim.* Seoul, Korea (South): IEEE, Feb. 2021, pp. 262–276.

[11] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, "Compiling Graph Applications for GPUs with GraphIt," in *2021 IEEE/ACM Int. Symp. Code Gener. Optim.* Seoul, Korea (South): IEEE, Feb. 2021, pp. 248–261.

[12] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM Intl. Symp. Code Gener. Optim*, 2021, pp. 2–14.

[13] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proc. 26th ACM Intl. Conf. Arch. Supp. Prog. Lang. Oper. Syst.* Virtual USA: ACM, Apr. 2021, pp. 804–817.