# dfesnippets: An Open-Source Library for Dataflow Acceleration on FPGAs

Paul Grigoras(✉), Pavel Burovskiy, James Arram, Xinyu Niu, Kit Cheung, Junyi Xie, and Wayne Luk

Department of Computing, Imperial College London, London, UK
{paul.grigoras09,w.luk}@imperial.ac.uk

**Abstract.** Highly-tuned FPGA implementations can achieve significant performance and power efficiency gains over general purpose hardware. However the limited development productivity has prevented mainstream adoption of FPGAs in many areas such as High Performance Computing. High level standard development libraries are increasingly adopted in improving productivity. We propose an approach for performance critical applications including standard library modules, benchmarking facilities and application benchmarks to support a variety of use-cases. We implement the proposed approach as an open-source library for a commercially available FPGA system and highlight applications and productivity gains.

## 1 Introduction

Highly tuned FPGA implementations can achieve performance and power efficiency gains for many problems [1]. However, development productivity is limited compared to other acceleration alternatives such as GPUs or Xeon Phi processors [2].

Recently, higher-level programming facilities based on High Level Synthesis [3,4] or domain specific languages [5–7] have improved productivity of FPGA development significantly. High-quality standard development libraries are becoming essential to improve productivity further. However, FPGA development environments may not provide standard development libraries. Fundamental operations such as floating point reductions may not be supported, and depending on the available resources and desired performance are nontrivial to implement, as we show in Sect. 2.

It is therefore necessary to provide well-designed component libraries to facilitate the development of applications and tools. However, in addition to these facilities, and as a point of departure from conventional approaches, given the performance-critical nature of the FPGA environment, component and application benchmarks should also be part of the library to facilitate the development of high-performance designs. To increase developer productivity for FPGA accelerators at all levels, libraries might provide: *(1)* library components which serve as the building blocks for developing real-world applications. These library components should be efficient in terms of latency, throughput and resource usage,

© Springer International Publishing AG 2017
S. Wong et al. (Eds.): ARC 2017, LNCS 10216, pp. 299–310, 2017.
DOI: 10.1007/978-3-319-56258-2_26

and provide a useful and customisable interface; *(2)* benchmarking utilities which aid in tasks such as determining system performance and resource utilisation. These utilities are essential for rapid prototyping, and assessing the scalability and feasibility of FPGA designs; *(3)* applications which can be used as benchmarks, or case studies for framework and tool development. These applications can also be adapted to accelerate closely related problems, considerably reducing development time.

In this work we present `dfesnippets`,[1] the first community driven open-source library for Maxeler DataFlow Engines (DFEs). The library is available under the MIT License. Table 1 provides an overview of the components:

1. A library component which contains useful reusable cores such as reduction, sorting and I/O circuits; these cores are tested and optimised, and have been used in several published designs [8–11].
2. A benchmarking component which facilitates quantitative evaluation and simplifies the process of modelling and estimating resource and performance properties, speeding up the design process.
3. An application component which provides a collection of full applications to be used as case-studies for the development of frameworks and tools for FPGA based programming. These applications have been used in several research projects and publications [12–16].

**Table 1.** Overview of components in `dfesnippets`

| Component | Block | Refs |
|---|---|---|
| Library | Input/output – ALBP, Inter-FPGA | [8] |
| | Linear algebra – SpMV, power iteration | [17,18] |
| | Reductions – tree, PCBT, LogAdder | [9] |
| | Sorting – bitonic sorter | [13] |
| Benchmarks | Infiniband/PCIe throughput | – |
| | Custom memory controller throughput | – |
| | Default memory controller throughput | – |
| | Component resource utilisation | – |
| Applications | Quantitative phase imaging | [19] |
| | Genetic sequence alignment | [16,20] |
| | Monte carlo finite difference option pricing | [13] |
| Software utilities | Build tool (python) | – |
| | Project template tool (python) | – |
| | Results extraction (python) | – |
| | Sparse matrix utilities (C++) | – |
| | Scheduling utilities (C++) | – |

[1] https://github.com/custom-computing-ic/dfe-snippets.

Although we will not cover this in greater detail due to lack of space, the library also contains: *(1)* header only C++ libraries implementing useful functionality for managing and benchmarking DFE projects ranging from timing utilities to APIs for reordering sparse matrix data in preparation for FPGA execution; *(2)* tools for creating and managing projects such as to compile, generate and manage multiprocess and multi-node hardware compilation, and automatically extract and tabulate resource usage and generate reports; *(3)* comprehensive, automated test suite, testing each design to ensure it is functionally correct and it meets timing and resource usage constraints.

A community driven library of open-source implementations, component and application benchmarks can increase the productivity of researchers and professional programmers. It can also improve the quality of results, and pave the way for broader FPGA adoption in areas where productivity has been a key limiting factor, such as High Performance Computing.

## 2 Library Components

Library components are the building blocks for developing more complex real world applications. `dfesnippets` includes a range of components such as generic reduction, I/O blocks, linear algebra blocks (sparse product, matrix vector and matrix-matrix-multiply, power iteration kernels, sparse matrix vector multiplication for banded matrices), and generic configuration and connectivity utilities such as inter-FPGA communication blocks. Despite being fundamental components, they are challenging to implement on FPGAs due to the resource constrained nature and high emphasis on performance and resource efficiency.

To be used effectively in large scale designs, library components must be parametric, provide a useful interface, and be efficient in terms of latency, throughput, and resource utilisation. Pure encapsulation, in software terminology, is difficult to achieve, therefore the internals of many cores may have to be customised in order to fit into the resource and performance constraints of a particular application. This makes source code availability important for component reuse.

We implement `dfesnippets` for the Maxeler FPGA platform [21]. The platform constitutes of a hardware implementation, a compiler from a high-level dataflow language, MaxJ, to FPGA bitstream, and a runtime environment. MaxJ [22] provides explicit control of the design of the hardware architecture itself, which is critical in delivering good performance and effectively exploiting customisation opportunities available for FPGA designers. It is conceptually close to Verilog, but with increased productivity due to the abstraction of low-level vendor IPs; MaxJ provides good support for software-only simulation and interfacing with many available programming languages. These features make MaxJ a good choice for implementing an open-source library: it provides a high level of control and flexibility without being verbose while the similarity to other hardware description languages simplifies porting components to other languages. In the rest of this Section we provide a more in-depth look at certain components of `dfesnippets`. For a full list please see the project page[2].

[2] https://github.com/custom-computing-ic/dfe-snippets.

### 2.1 Reductions

A *reduction* is the application of an associative binary operator to an initial value and a list of values in order to collapse the list to a single value. The deeply pipelined nature of the arithmetic units on FPGAs, such as those for floating point, make reduction operators non-trivial to implement, and much research has gone into efficient reduction circuits [23–25].

Reduction implementations must balance throughput, resource usage and latency. From this perspective, we can define at least three types of reduction circuits. First, a fully parallel *reduction tree* which can reduce $k$ values per clock cycle, assuming fully pipelined operators. Trees have the highest throughput, but also the largest resource usage of O(k). To reduce large data streams of size $n$, reduction trees of size $n$ are required. This is fast but not practical from a resource utilisation standpoint. Second, a *C-slowed accumulator* may reduce one value per clock cycle, with a latency depending on the latency $L$ of the reduction operator. This is a resource efficient approach requiring a single reduction operator, but the throughput is limited: one value per clock cycle may make the reduction circuit a performance bottleneck of the entire design. For example a modern FPGA architecture may read 48 double precision values from DRAM per clock cycle. Also, the C-slowed accumulator does not fully compute the reduction, as $L$ partial sums are left to be reduced in the pipeline. Third, more complex reduction circuits have been proposed such as the *partially compacted binary reduction tree*, PCBT [26]. These blocks are more complex to implement but can achieve good resource efficiency when high throughput is not a concern. Circuits such as the PCBT solve the issue of only partially reducing the data set, and they typically require more resources than a C-slowed implementation but fewer resources than a tree.

Many designs, such as implementations of sparse matrix vector circuits [8], iterative solvers [10] and power iteration kernels [17] may require a combination of all three circuits to achieve maximum performance: *(1)* a full tree performs the initial reduction at high throughput reducing $k$ values per cycle, *(2)* each output of the reduction tree is fed and accumulated in a C-slowed accumulator and finally *(3)* each output of the C-slowed accumulator may be reduced using a PCBT.

Our implementation of the PCBT is shown in Fig. 1. It consists of a chain of blocks, each implementing its own level of a binary reduction tree using a state machine, a buffer and an adder. The state machine has two states: *no arguments* and *one argument*. When one argument is present in the buffer and the second argument is an enabled input, the adder produces their sum as the `output` signal with the `valid` signal high, then flushes the internal state to *no arguments*. At the transition to the *one argument* state the output `valid` is low, whilst the `output` is a sum of a stored argument with zero. The `valid` signal of a block is connected to the `enable` signal of the next block so that each level of the PCBT is waiting for the complete accumulation at the previous level. It also enables the PCBT to stall but preserve its internal state if necessary (when `enable` is low). The external `reset` signal forces all state machines to produce the `valid`

signal high regardless of their internal state, thus finalising the reduction with whatever number of inputs are internally present in our PCBT circuit. This enables accumulating an arbitrary number of terms in a reduction set.

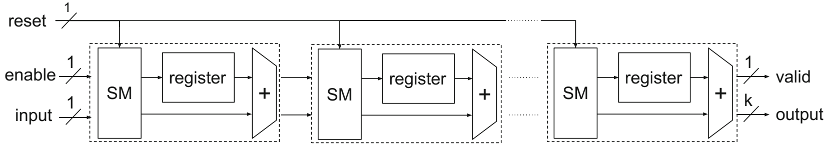


**Fig. 1.** PCBT based on state machines

To conclude the reduction case study, we note that, in principle, the reduction operation is probably one of the most fundamental building blocks required for implementing more complex applications. However, due to the broad range of design choices with varying throughput, resource utilisation and functionality, this operation is not trivial to implement. Having easy access to multiple variants of reduction circuit, as provided by `dfesnippets`, can therefore improve productivity.

### 2.2 Input and Output Blocks

I/O blocks are commonly used to manage the connection between the computational kernel implemented on the FPGA accelerator and off-chip components such as DRAM, the host CPU (PCIe, Infiniband), or other FPGA devices.

In the case of DRAM and CPU communication, the I/O blocks may be required to convert the fixed width output interface of the communication channel to a different input width of the computational kernel. This is a common requirement, particularly for applications which process an irregular, runtime dependent input size at each cycle such as a sparse matrix vector multiplication kernel [8]. The I/O blocks are required to be efficient from a resource utilisation perspective but the logic they implement is often complex and the control heavy nature does not map well to dataflow style accelerators and languages. If unoptimised, these blocks can use substantial on-chip resources, particularly memory resources such as BRAMs.

Blocks such as the Arbitrary Length Burst Proxy (ALBP) included in `dfesnippets` and used in previous work [8] can help address these issues. The ALBP architecture contains $k$ FIFOs to store bursts retrieved from off-chip memory. Once a burst is retrieved, data are pushed in the FIFOs such that the $i$-th element of a burst is assigned to FIFO $o+i \bmod k$. $o$ is the position after processing the previous burst. $m_t < k$ data items may be simultaneously requested from the ALBP by the compute kernel, where $m_t$ is runtime-determined. If fewer than $k$ items are requested, the output is zero-padded to the fixed width $k$ to match with the fixed, regular $k$ width of the compute kernel's input interface.

Other I/O blocks may be required for inter-device communication. In computing clusters with multiple FPGAs, light-weight and easy-to-use communication modules for inter-FPGA data transfer help to reduce the latency and overhead of the whole system. In a number of HPC systems, such as the Maxeler system used in this work, direct built-in inter-FPGA links are used for transferring large amounts of data with low latency. `dfesnippets` builds on top of these inter-FPGA links to implement an interchangeable communication kernel which creates a one-dimensional systolic array, unrolled across multiple FPGAs. The kernel uses counters to keep track of the amount of data sent and received, which are used to control input and output switches, allowing data to be rapidly transferred among the FPGAs. `dfesnippets` implements an all-to-all broadcasting protocol using this systolic array by alternating the direction of the data transfer in successive turns. Interchangeable inter-FPGA communication modules provide greater flexibility in distributing workload among the accelerators, hence such a library of modules is extremely useful for applications requiring large-scale multi-FPGA systems.

### 2.3 Other Blocks

In addition to the components presented in this section, `dfesnippets` includes a range of components such as sorting and linear algebra blocks, more generic configuration and connectivity utilities and a substantial number of CPU based functionalities, to handle pre-processing and integration of the accelerator designs within larger application frameworks. Leveraging these blocks, there are many possibilities for developing, optimising and including more library components within the proposed approach, which will further increase the productivity and applicability.

## 3 Benchmarking

Benchmarking utilities are especially helpful for the research community. They help establish a baseline for the system performance or resource efficiency, facilitate quick estimation and prototyping (for example to assess the scalability of various designs with respect to memory bandwidth, resources etc.), provide sanity checks and highlight empirically the impact of some optimisations which may not be entirely transparent to the end user. Two types of benchmarks are particularly important for FPGA development: *(1)* performance benchmarks which can be used to measure the throughput and latency of FPGA designs and memory and interconnect subsystems *(2)* resource utilisation benchmarks which demonstrate the resource efficiency of particular cores and are essential for assessing the scalability and feasibility of FPGA designs.

**Performance.** `dfesnippets` provides three system level performance benchmarks which can be used to measure the achievable throughput of various links. The *Default DRAM Benchmark* instantiates a default memory controller, with

customisable clock frequency which reads and writes data in a linear access fashion. This can be used to determine the peak memory bandwidth performance of a given device, which can serve as a baseline for measuring the achieved performance of user applications. The *Custom DRAM Benchmark* instantiates a more complex design with a custom memory command generator and associated host code to drive the benchmarking. This can be used for evaluating the memory access speed using custom memory commands and linear access patterns. It fetches parallel data streams from DRAM and then routes them to DRAM and/or host, behaviour which is configurable by the user. The major configuration options are parameterised so users can change the number of bursts per command, size of memory to access, width of memory interface and number of parallel DRAM streams to match existing properties in their own designs. This enables rapid experimentation with application specific data placement and access scheduling techniques to improve DRAM performance. The *Infiniband/PCIe DRAM Benchmark* instantiates a simple pass through design which matches the PCIe input width (128 bits). Together with the associated software to run on the CPU, the design can be used to measure throughput over the CPU to FPGA interconnect.

The library allows users to easily adjust the number of measurements, data size, memory controller frequency, on-chip frequency, and architecture for each benchmark. This reduces the possibility for error and promotes good practices.

**Resource Utilisation.** `dfesnippets` includes a synthetic resource utilisation benchmark to measure the resource usage of various blocks using the MaxCompiler builtin resource usage annotations. These reports are openly available as part of the library and can provide the basis for rapid resource usage estimation models without the need to sit through long compilation times. This can greatly reduce the time to prototype designs. The benchmarks are provided for both the Xilinx Virtex 6 based Vectis boards and the Stratix V Maia boards. This provides a quick method to highlight differences between the two (such as different resource usage profile of DSPs) or provide insight into hidden properties, which can probably only be discovered by significant empirical exploration, such as the considerable resource savings achieved by reducing pipelining factors on the Stratix V Maia boards.

## 4 Applications

`dfesnippets` also includes a set of full applications which can be used as reusable components in other applications or as benchmarks and case studies for framework and tool development. The broader availability of such applications can help researchers and developers focus more on their area of expertise and avoid typical pitfalls stemming from the complexity of designing FPGA based applications. These applications themselves contain reusable blocks which can be adapted in other designs, or can be reused directly in other applications, perhaps as one stage of a complex pipeline or multi FPGA design. Overall, the availability of these larger designs can increase the productivity of researchers and tool developers.

**Genomic Data Analysis.** dfesnippets includes an FM-index [27] design which can be used to accelerate a variety of genomic data analysis applications such as sequence alignment [28], sequence assembly [29], and reference-based compression [30]. Several works which make use of the FM-index design have been published [15,16]. The FM-index is a full-text compressed index which supports substring searching in time proportional to the search string length. The FM-index is built upon the Burrows-Wheeler transform [31], a permutation of a text generated from its Suffix Array [32].

A single FPGA outperforms the fully-optimised software version running on dual Intel Xeon CPUs with 16 threads. The largest performance improvement is for the hg38 data set where the FPGA is *3.7 times faster* than the software version. With the performance gains presented, the FM-index design has great potential for integration into many genomic data analysis applications or to be used as a reference benchmark application for tool development.

**Monte Carlo Finite Difference Option Pricing.** dfesnippets includes a multi-FPGA dynamic Monte Carlo design for bond options pricing. This design is particularly useful as a case study for resource management frameworks or environments for FPGAs [13,14] as it demonstrates good scalability and performance. To accelerate the payoff evaluation for the bond option, the Monte Carlo paths and the payoff evaluation functions are implemented on the FPGA accelerator. The finite difference method [33] is applied to solve the resulting equations and estimate the payoff of the bond at some time in the future.

The design operates in a map-reduce fashion, using OpenMP to parallelise the calls to the Maxeler API which load and execute the Monte Carlo evaluation over a configurable number of FPGAs. A final reduction step is implemented on the CPU to aggregate the results of the computation corresponding to different Monte Carlo paths. On the FPGA accelerator, an optimised random number generator [34] is used to generate the random numbers required for the Monte Carlo computation. A baseline design with 4 parallel processing elements, uses less than 20% of the resources on the Virtex 6 chip of the Maxeler Vectis DFE. This makes the design easy to place and route, and therefore ideal for experimental workloads where a short iteration time is essential, for example when developing tests and benchmarks for more complex tools. The design achieves linear scalability [13] and can therefore be used to benchmark load distribution tools, scheduling strategies and cloud-like environments for heterogeneous systems, such as FPGAs.

**Quantitative Phase Imaging (QPI) on FPGAs.** dfesnippets also includes a block for image processing based on the newly developed quantitative asymmetric-detection time-stretch optical microscopy (Q-ATOM) which offers ultrafast and high-sensitivity quantitative phase cellular imaging. Retrieved phase images provide essential information of cells and potentially benefits medical diagnostics. However, performing backend phase retrieval and cell image classification is extremely computationally intensive. With the aid of FPGAs researchers can push QPI phase retrieval and cell image classification to near real-time speed [19].

The QPI phase retrieval and cell image classification design is composed of a spatial domain module, a frequency domain module and a linear SVM classifier. The spatial domain module performs background subtraction, intensity normalization and complex phase shift extraction. The frequency domain module performs low-pass filtering to reduce noises and retrieves final phase images. The Winograd 16-point algorithm is used in the frequency domain module to perform forward and inverse 2D fast Fourier transform (FFT). The sequential Winograd algorithm has low resource consumption and is suitable for a wide range of applications involving frequency spectrum analysis.

The QPI application has *a throughput of 32.08 GOPS* when running on a single Altera Stratix V GS 5SGSD8 FPGA [19], which is equivalent to retrieving and classifying around 2497 phase images of 256 $\times$ 256 size. Classification accuracy of unstained and live human chondrocytes (OAC), human osteoblasts (OST) and mouse fibroblasts (3T3) increases when using retrieved phase images.

## 5 Evaluation

`dfesnippets` totals approximately 6000 lines of CPU utilities and tests and 7000 lines of MaxJ in the library and benchmarking components and 4000 lines of CPU and MaxJ code in the applications components. We estimate the development time of each library component to be of the order of one to two weeks while the development effort for applications is on the order of 1–2 months. Both library and application development usually involve two developers, of which one is typically experienced (more than two years) in the MaxJ programming language.

Even in a relatively high level language such as MaxJ, approximately 600 lines of library code including comments are required to implement the three alternative reduction strategies described in Sect. 2.1 plus an additional 700 lines for setting up the CPU test bench that is vital to verify the correctness of these implementations, particularly for the more complex designs. By using `dfesnippets` almost 1300 lines of code can be replaced by several lines to instantiate the required reduction circuits directly in the user design. Therefore the productivity gains resulting from the proposed library component of our approach are substantial, particularly since reduction circuits are generic blocks, commonly used in many applications. Table 2 shows several applications where we have used `dfesnippets` and observed a substantial reduction in source lines of code (SLOC) for the hardware design.

To illustrate the productivity gains achievable by the applications components we note that recent software frameworks such as experimental compilers [12] and resource management frameworks [13] for FPGA based systems can utilise these applications directly as benchmarks. Prototypes for these projects require 4123 and 3880 lines of code respectively, while the benchmarks require 2050 and 2924 lines of code respectively. Therefore a substantial productivity gain comes from the ability to directly reuse these benchmarks and avoid spending substantial time on redeveloping complex designs. We estimate the development time of application components in `dfesnippets` to be between 1–2 months

**Table 2.** Examples of projects using dfesnippets and estimated productivity improvement measured in a reduction in source lines of code (SLOC), including only the hardware components and thus excluding comments, test code, CPU interfaces etc.

| Application | Components used | SLOC reduction |
|---|---|---|
| SpMV tuning framework [8] | Reductions, I/O, configuration | 710 |
| Biomedical acceleration [15] | Bitwise operation, FM index | 361 |
| FEM accelerator [11] | Reductions, I/O | 490 |
| Elastic cloud framework [13] | Option pricing app benchmark | 590 |
| Linear solver [10] | Benes network, reductions | 250 |
| SpMV accelerator [18] | Reductions, I/O | 490 |

each for an experienced MaxJ developer. These applications often require complex, specialised and state of the art blocks such as high throughput random number generators, Fast-Fourier Transforms, and custom memory controllers. Such blocks are not only complex and non-trivial to optimise for FPGA implementation, they are also difficult to develop and debug. It is clear that from a tool developer perspective, it is not productive to spend as much time developing the benchmark as developing the tool itself.

Not only is the development time reduced substantially by avoiding the need to redevelop benchmarks, but the parametric design supports customisation effectively, leading to additional productivity gains. All applications can be built with minimal configurations to verify correctness or with full replication and optimisations to verify performance and energy efficiency. This approach simplifies debugging and testing in the early stages of project development by reducing the compilation time.

## 6 Conclusion

We present `dfesnippets`, an open source library of reusable components: cores, benchmarks, applications and tools. It improves the productivity of FPGA development by providing fundamental blocks for any real world application as well as system, component and application benchmarks. By providing `dfesnippets` directly as open source software to the research community, we hope that a substantial improvement in productivity can be achieved. This may pave the way for supporting exciting and sophisticated research and applications, while enhancing the adoption of FPGAs in High Performance Computing, embedded systems and other domains.

**Acknowledgement.** The support of UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, the Maxeler University Programme, Altera, Intel and Xilinx is gratefully acknowledged.

## References


1. Todman, T.J., Constantinides, G.A., Wilton, S.J., Mencer, O., Luk, W., Cheung, P.Y.: Reconfigurable computing: architectures and design methods. IEE Proc.-Comput. Digit. Tech. **152**(2), 193–207 (2005)
2. Jones, D.H., Powell, A., Bouganis, C., Cheung, P.Y.: GPU versus FPGA for high productivity computing. In: Proceedings of the FPL, pp. 119–124 (2010)
3. Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., Cong, J.: AutoPilot: a platform-based ESL synthesis system. In: Coussy, P., Morawiec, A. (eds.) High-Level Synthesis, pp. 99–112. Springer, Heidelberg (2008)
4. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the FPGA, pp. 33–36. ACM (2011)
5. Kulkarni, C., Brebner, G., Schelle, G.: Mapping a domain specific language to a platform FPGA. In: Proceedings DAC, pp. 924–927. ACM (2004)
6. George, N., Lee, H., Novo, D., Rompf, T., Brown, K.J., Sujeeth, A.K., Odersky, M., Olukotun, K., Ienne, P.: Hardware system synthesis from domain-specific languages. In: Proceedings of the FPL, pp. 1–8. IEEE (2014)
7. Cong, J., Sarkar, V., Reinman, G., Bui, A.: Customizable domain-specific computing. IEEE Des. Test Comput. **28**(2), 6–15 (2011)
8. Grigoras, P., Burovskiy, P., Luk, W.: CASK: open-source custom architectures for sparse kernels. In: Proceedings of the FPGA, pp. 179–184 (2016)
9. Grigoras, P., Burovskiy, P., Hung, E., Luk, W.: Accelerating SpMV on FPGAs by compressing nonzero values. In: Proceedings of the FCCM (2015)
10. Chow, G., Grigoras, P., Burovskiy, P., Luk, W.: An efficient sparse conjugate gradient solver using a benes permutation network. In: Proceedings of the FPL (2014)
11. Burovskiy, P., Grigoras, P., Sherwin, S.J., Luk, W.: Efficient assembly for high order unstructured FEM meshes. In: Proceedings of the FPL (2015)
12. Grigoras, P., Niu, X., Coutinho, J., Luk, W., Bower, J., Pell, O.: Aspect driven compilation for dataflow designs. In: Proceedings of the ASAP (2013)
13. Grigoras, P., Tottenham, M., Niu, X., Coutinho, J.G.F., Luk, W.: Elastic management of reconfigurable accelerators. In: Proceedings of the ISPA, pp. 174–181. IEEE (2014)
14. Coutinho, J.G.F., Pell, O., O'Neill, E., Sanders, P., McGlone, J., Grigoras, P., Luk, W., Ragusa, C.: HARNESS project: managing heterogeneous computing resources for a cloud platform. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) ARC 2014. LNCS, vol. 8405, pp. 324–329. Springer, Heidelberg (2014). doi:10.1007/978-3-319-05960-0_36
15. Arram, J., Pflanzer, M., Kaplan, T., Luk, W.: FPGA acceleration of reference-based compression for genomic data. In: Proceedings of the ICFPT, pp. 9–16. IEEE (2015)
16. Arram, J., Luk, W., Jiang, P.: Ramethy: reconfigurable acceleration of bisulfite sequence alignment. In: Proceedings of the FPGA, pp. 250–259. ACM (2015)
17. Burovskiy, P., Girdlestone, S., Davies, C., Sherwin, S., Luk, W.: Dataflow acceleration of Krylov subspace sparse banded problems. In: Proceedings of the FPL, pp. 1–6. IEEE (2014)
18. Grigoras, P., Burovskiy, P., Luk, W., Sherwin, S.: Optimising sparse matrix vector multiplication for large scale FEM problems on FPGA. In: Proceedings of the FPL, pp. 1–9. EPFL (2016)

19. Xie, J., Niu, X., Lau, A.K., Tsia, K.K., So, H.K.: Accelerated cell imaging and classification on FPGAS for quantitative-phase asymmetric-detection time-stretch optical microscopy. In: Proceedings of the ICFPT, pp. 1–8. IEEE (2015)
20. Arram, J., Tsoi, K.H., Luk, W., Jiang, P.: Hardware acceleration of genetic sequence alignment. In: Brisk, P., Figueiredo Coutinho, J.G., Diniz, P.C. (eds.) ARC 2013. LNCS, vol. 7806, pp. 13–24. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36812-7_2
21. Lindtjrn, O., Clapp, R.G., Pell, O., Mencer, O., Flynn, M.J.: Surviving the end of scaling of traditional micro processors in HPC. In: IEEE HOT CHIPS 22 (2010)
22. Pell, O., Mencer, O.: Surviving the end of frequency scaling with reconfigurable dataflow computing. SIGARCH Comput. Archit. News **39**(4), 60–65 (2011)
23. Morris, G.R., Zhuo, L., Prasanna, V.K.: High-performance FPGA-based general reduction methods. In: Proceedings of the FCCM, pp. 323–324 (2005)
24. Zhuo, L., Morris, G.R., Prasanna, V.K.: Designing scalable FPGA-based reduction circuits using pipelined floating-point cores. In: Proceedings of the ISPDP (2005)
25. Wilson, D., Stitt, G.: The unified accumulator architecture: a configurable, portable, and extensible floating-point accumulator. Trans. Reconfigurable Technol. Syst. (TRETS) **9**(3), 21 (2016)
26. Zhuo, L., Morris, G.R., Prasanna, V.K.: High-performance reduction circuits using deeply pipelined operators on FPGAs. IEEE Trans. PDS **18**(10), 1377–1392 (2007)
27. Ferragina, P., Manzini, G.: An experimental study of an opportunistic index. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 269–278. Society for Industrial and Applied Mathematics (2001)
28. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nat. Methods **9**(4), 357–359 (2012)
29. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Res. **22**(3), 549–556 (2012)
30. Zhang, Y., Li, L., Yang, Y., Yang, X., He, S., Zhu, Z.: Light-weight reference-based compression of FASTQ data. BMC Bioinform. **16**(1), 1 (2015)
31. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm (1994)
32. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
33. Mitchell, A.R., Griffiths, D.F.: The Finite Difference Method in Partial Differential Equations. Wiley, Hoboken (1980)
34. Thomas, D.B., Luk, W.: High quality uniform random number generation using LUT optimised state-transition matrices. Vlsi Sig. Process. **47**(1), 77–92 (2007)