From exaflop to exaflow

Tobias Becker,* Pavel Burovskiy,* Anna Maria Nestorov,† Hristina Palikareva,* Enrico Reggiani,† and Georgi Gaydadjiev*
*Maxeler Technologies Ltd, UK
†Politecnico di Milano, Italy

***Abstract*—Exascale computing is facing a gap between the ever increasing demand for application performance and the underlying chip technology that does no longer deliver the expected exponential increases in CPU performance. The industry is now progressively moving towards dedicated accelerators to deliver high performance and better energy efficiency. However, the question of programmability still remains. To address this challenge we propose a dedicated high-level accelerator programming and execution model where performance and efficiency are primary targets. Our model splits the computation into a conventional CPU-oriented part and a highly efficient fully programmable data flow part. We present a number of systematic transformations and optimisations targeting Maxeler dataflow systems that typically yield one to two orders of magnitude improvements in terms of both performance and energy efficiency. These significant gains are enabled by addressing fundamental algorithmic properties and on-demand numerical requirements. This approach is demonstrated by a case study from computational finance.**

I. Introduction

Data centres experience salient demand for computational performance driven by a vast amount and spectrum of applications ranging from conventional HPC to Cloud Computing, while at the same time facing stagnating performance gains and energy efficiency challenges inherent to conventional computing. In particular, power and energy consumption are a growing concern as they directly impact operating cost which is further escalated by the need to manage the dissipated heat through wasteful cooling systems. As data centres push performance towards exascale level, increasing energy efficiency is becoming one of the leading concerns [1].

As a result of the above, energy-efficient accelerators are becoming more commonplace in industry. In particular, reconfigurable accelerators, e.g. FPGA based ones, are becoming increasingly relevant to the industry due to their flexibility, high throughput and superior energy efficiency. A noteworthy example is the recent introduction of FPGA F1 instances at Amazon Web Services. However, challenges in accelerator programmability still persist. Conventional hardware design is too complex and low level, while some higher-level programming paradigms focus on simple portability while sacrificing performance. Domain-specific approaches have shown potential but are naturally not widely applicable.

Maxeler is pioneering a novel approach based on the principles of dataflow and systolic arrays where application experts in science, engineering or finance can develop and customise their algorithms using a convenient high-level language, targeting Maxeler's multiscale dataflow computers. The design process involves splitting the application in a control-oriented part and a dataflow-oriented part. The dataflow plane is fully programmable and supports custom optimisations, enabling the developer to focus on maximising system performance and efficiency. In this paper we outline the general dataflow model that is used in Maxeler's systems. By shifting away from the mindset where systems are scaled out by adding more and more conventional nodes, and adopting a flow-oriented paradigm, we are able to achieve a significantly higher degree of efficiency, enabling the next step towards exascale computing. Furthermore, we cover a range of optimisations typical for dataflow systems that demonstrate the benefit of challenging conventional practices such as using floating point for all non-integer type calculations.

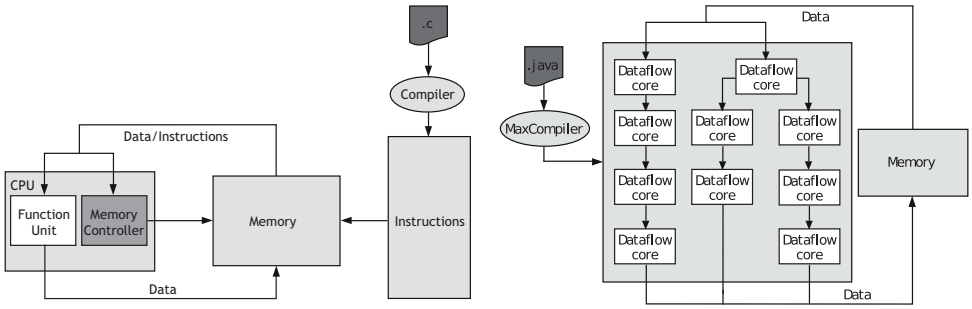
II. Principles of Maxeler dataflow computing

A. The dataflow computing paradigm

The multiscale dataflow computing paradigm successfully deployed by Maxeler in commercial systems fundamentally differs from conventional processors which are control-flow centric. This approach is illustrated in Figure 1 and represents an evolution of dataflow and systolic array concepts [2], [3]. A conventional processor operates by reading and decoding instructions, loading the required data, performing an operation on the data, and returning the result to memory. This process is iterative in nature and requires complex control mechanisms that manage the operation of the processor. The dataflow execution model is greatly simplified in comparison. Data is streamed from memory into the chip where arithmetic operations are performed by chains of functional units (dataflow cores) statically interconnected in a structure that corresponds to the implemented functionality. This dataflow structure performs computations entirely without instructions or control mechanisms; instead, data simply flows from one functional unit directly to the next one. Each dataflow core performs only a simple operation such as addition or multiplication and hence thousands of operations can fit on the chip surface.

Unlike a control-flow oriented processor where operations are computed on a time-shared functional unit ("computing in time"), the complete dataflow computation is laid out in space over the entire chip ("computing in space"). Dependencies in the computation are resolved statically at compile time, and because there is no data-dependent behaviour present at run time, the entire dataflow computation can be mapped onto a deeply pipelined compute structure. This model of computation eliminates a lot of components that are typically necessary in a processor such as instruction decoding logic, branch prediction or general purpose caches. Instead of managing the execution, all resources on the chip can be

978-3-9815370-8-6/17/$31.00



(a) Control-flow system (b) Data-flow system

Fig. 1. A comparison between a conventional control-flow oriented processor (a) and a Maxeler DFE (b).

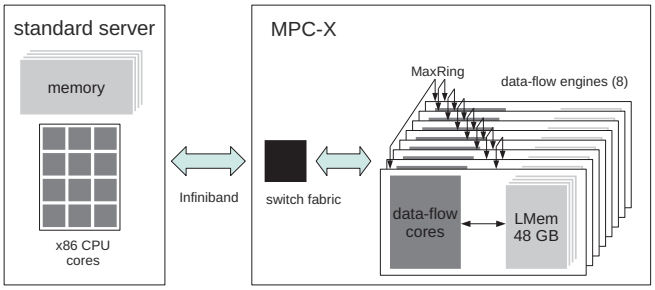


Fig. 2. Architecture of an MPC-X node with eight dataflow engines. Multiple MPC-X nodes and CPU nodes are connected through an Infiniband network.

now dedicated to performing useful computations and hence a dataflow computer has far greater efficiency in terms of computations per chip area or computations per energy.

B. Dataflow computing systems

Maxeler realises this dataflow-oriented computing approach by describing applications in terms of their dataflow execution model and mapping them onto dedicated dataflow engines (DFEs). The current generation MAX4 DFEs are based on a large Altera Stratix-V FPGA that provides the reconfigurable computing substrate for the dataflow cores. The FPGA is combined with large amounts of DRAM memory (currently between 48-96 GB), providing very high memory capacity for large computational problems. This memory is called Large Memory (*LMem*). In addition, the FPGA itself also provides embedded on-chip memories which are spread throughout the chip's fabric and can be used for low-latency buffering of local compute values. These embedded memories are called Fast Memory (*FMem*) as they can be accessed with an aggregated bandwidth of several terabytes/second. This is an important factor for both the performance and energy efficiency of DFE computations because data can be kept locally where it is needed and accessed at very high rates.

DFEs are highly efficient for large-scale computations with a static structure due to the elimination of sequential execution and control as well as the optimisation of memory access and data movements. However, DFEs are not a good platform for small-scale computations with control-dominated dynamic behaviour. Hence, DFEs are combined with conventional CPUs in a heterogeneous high-performance computing system. DFEs are PCIe cards that can be plugged into any conventional server. However, a far more flexible system architecture is illustrated in Figure 2. In Maxeler MPC-X series systems, eight DFE cards are incorporated into a dense, industry standard 1U chassis, forming a pure dataflow appliance. This system is connected to conventional CPU servers via an Infiniband network. Inside the MPC-X, the DFEs are also directly connected through MaxRing.

Such a system can dynamically allocate large numbers of DFEs to multiple applications, providing high scalability and flexibility for large-scale systems. This platform has demonstrated one to two orders of magnitude gains in performance and energy efficiency over conventional CPU servers in a range of application domains [4], [5]. In the context of the H2020 project EXTRA, we currently explore how to move towards exascale performance with such a platform [6].

C. Dataflow programming

The efficiency of dataflow computing is closely linked to its dedicated programming model that exposes many optimisation options. It is beyond the scope of this paper to cover the full programming model; here we simply introduce some of the basic principles, followed by a description of selected optimisations and their impact on a practical application.

DFEs are highly efficient for carrying out the large-scale, compute-intensive parts of an application while conventional CPUs are more suitable for control-intensive tasks. Porting an application onto a dataflow system therefore requires the application code to be split into a host application and a dataflow part. The basic logical architecture is illustrated in Figure 3. The DFE part contains one or more dataflow kernels that perform the accelerated computations. It also contains a manager that is responsible for managing the connections between kernels, DFE off-chip memory, CPU host memory and various interconnects such as PCIe, Infiniband and MaxRing. The CPU is responsible for setting up and controlling the computation on the DFE as well as for performing pre-computations or control-oriented tasks.

Developing a practical dataflow application typically starts with a conventional CPU implementation that needs to be accelerated. After identifying the performance-critical parts of the application, the designer ports these parts to the DFE by describing their dataflow models in a Java-based meta-language called MaxJ. MaxJ programming adopts Java syntax but is in principle different from regular Java programming

or other imperative programming paradigms. Compiling MaxJ code does not produce a Java application; instead, it leads to the generation of dataflow kernels. Both the compute kernels handling the data-intensive part of the application and the associated manager, which orchestrates data movement between kernels and external interfaces, are written using MaxJ. Developing a dataflow application therefore involves implementing three parts:

1) A CPU host application typically written in C/C++, Matlab, Python, R, etc;
2) A number of dataflow kernels written in MaxJ;
3) A manager described in MaxJ.

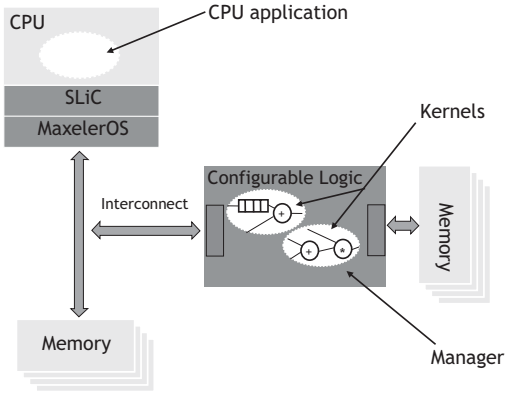


Fig. 3. Logical architecture of a dataflow computing system with a CPU and a DFE.

Maxeler developed MaxCompiler to compile MaxJ kernels and manager code into a binary that can be loaded and run on the DFEs. This DFE binary needs to interface with a CPU host application. Therefore, Maxeler also provides several software components for seamless integration with CPU host applications as well as run-time management functionality. The software platform consists of the following:

1) MaxCompiler, a custom-built general-purpose compiler for developing dataflow applications in MaxJ;
2) Simple Live CPU interface (SLiC) API to seamlessly integrate calls to DFEs into CPU host applications;
3) MaxelerOS, a runtime layer between the SLiC API, the Linux operating system and the DFE hardware, which manages CPU-DFE interactions in a transparent way.

Let us now consider the fundamental aspects of effective dataflow programming for large-scale applications. All operations within a DFE are naturally parallel, and any operation specified in MaxJ code is parallel unless explicitly specified as sequential. During kernel development, a developer would focus on realising large degrees of parallelism and pipelining as well as performing a range of dataflow-specific optimisations.

To illustrate this concept, we show how a simple computation is described in MaxJ and present the resulting dataflow graph. The C code snippet in Figure 4 illustrates how a simple computation is carried out in a loop. The compiled version of this code will instruct a CPU to carry all steps sequentially.

Figure 5 shows the MaxJ code that performs the same computation on a DFE. Unlike an imperative programming

```
for (i = 0; i < numDataElements; i++)  {
    float x = input[i];
    float y = x * x + 3 * x + 17;
    output[i] = y;
}
```

Fig. 4. C code of a simple computation inside a loop.

language that describes changes in program state, MaxJ describes the structure of a static dataflow graph which computes results by streaming data through it. A new kernel is created by extending the Kernel object which is provided by the MaxJ libraries. The original loop over data is removed and replaced by streaming inputs and outputs (`io.input` and `io.output`). The inputs and outputs allow full customisation of the used data types which will be explained in greater detail in Section III. Here we use standard single-precision floating point. Inside the kernel we use DFEVar objects to represent data streams. The fact that we create a *static* dataflow graph has a number of important implications that need to be reflected at MaxJ level: (i) all conventional Java types such as `int` or `float` are statically evaluated and represent constants; (ii) `for` loops can be used to create unrolled compute structures; (iii) conditionals are also statically evaluated and can be used to control build options; (iv) dynamic control can be achieved through a mux construct that computes all options of a conditional statement and selects the appropriate output. A full description can be found in [7].

```
class SimpleCalc extends Kernel {
    SimpleCalc() {
        DFEVar x = io.input("x", dfeFloat(8,24));
        DFEVar y = x * x + 3 * x + 17;
        io.output("y", y, dfeFloat(8,24));
    }
}
```

Fig. 5. A MaxJ description that generates the dataflow implementation shown in Figure 6.

Figure 6 illustrates the dataflow graph obtained after compiling the MaxJ kernel described above. It is a pipelined structure where all arithmetic operations are carried out in parallel. All dataflow core operators can also be customised and optimised for their input format, making them far more efficient than programmable ALUs in a CPU. The presented example is for illustrative purposes only; practical dataflow implementations can have thousands of arithmetic operators.

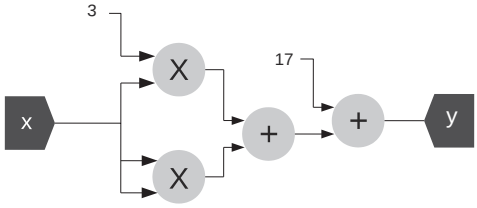


Fig. 6. A dataflow implementation for the computation inside the loop body.

III. Dataflow-oriented optimisations

As outlined in the previous section, developing a dataflow kernel in MaxJ entails building an application-specific compute structure which can be tailored to the exact requirements of the algorithm. This model cuts across many layers of abstraction and exposes a number of optimisation options

which are typically not available in conventional software design. A non-exhaustive list includes custom number formats, compute vs. local memory look up, and degree of loop unrolling. In this section we consider the following optimisation schemes in greater detail: dataflow kernel merging, modelling floating-point with fixed-point number formats, and function approximations.

As explained in section II, a dataflow design often consists of multiple kernels. A simple reason for this might be to increase parallelism and throughput by replicating a kernel several times until all chip resources are in use. However, in some cases not all kernels are active at the same clock cycle meaning that hardware resources can be shared between them. In MaxCompiler this can be exploited with an optimisation called kernel merging [8]. Figure 7 illustrates the concept: dataflow graphs from two different kernels are merged into a single dataflow graph using additional muxes. A Kernel Merger tool is available as part of the Maxeler Standard Library (MaxPower) [9].

Another optimisation scheme that is often highly beneficial is replacing conventional floating-point number formats with reduced-precision or fixed-point formats. Unlike conventional hardware with pre-fabricated hardware ALUs supporting only a limited range of numeric data types, DFEs can instantiate both floating and fixed-point arithmetic units of arbitrary bit widths, tailored to the dynamic range of numeric variables of a target application on some representative input data set. This includes both IEEE-compatible floating point types with custom mantissa widths and fixed-point types of any (reasonable) bit widths with arbitrary location of the decimal point.

It is a common misconception that floating-point arithmetic always provides the most accurate numerical results. In many real applications fixed-point arithmetic provides higher accuracy than calculations in IEEE double precision floating point, e.g. when the dynamic range of a variable requires using the same exponent of an IEEE double precision representation and one needs more than 53 bits of mantissa for accurately representing a fraction.

MaxJ lets the user choose mixed precision, which stands for using both fixed and floating point data types in the same compute kernel. Moreover, MaxJ provides extremely powerful and easy to use language support to mixed precision computation. The user can declare the custom data type via, e.g., a declaration `dfeFloat(11, 46)` (an IEEE floating-point type with an 11-bit exponent and a 46-bit mantissa) or `dfeFix(3, 9, SignMode.UNSIGNED)` (an unsigned fixed-point type with a 3-bit integer and a 9-bit fractional part). Casting from a floating-point data type to a fixed-point data type (and vice versa) is also an integral part of MaxCompiler.

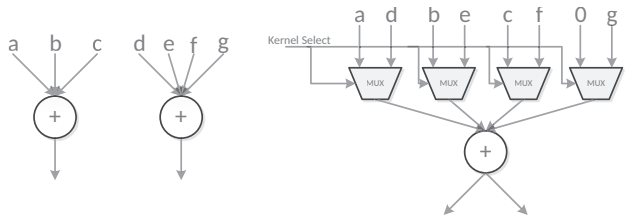


Fig. 7. Merging two dataflow graphs from different kernels.

Another example of a dataflow-specific optimisation that cuts across abstraction layers is related to function approximations. When a user evaluates elementary functions such as `exp` and `log` for a given numerical argument, the output is computed using function approximations which are frequently implemented in software or as processor microcode. Dataflow kernels provide the advantage of pipelining implementation of such approximations and producing a result every cycle, compared to the processors for which it may easily span 10 or more clock cycles in a simple case. In addition, a composition of elementary functions, such as $\exp\left(-x^2\right)$ or $\sin\left(\sqrt{1-\ln x^2}\right)$ on conventional hardware is predominantly implemented via several consecutive evaluations of elementary functions and arithmetic operations [10].

The dataflow optimisation exploits the fact that approximating the entire function composition may be possible with a similar amount of compute resources as approximating each elementary function on its own. Typically abstraction layers hide the approximation of elementary functions from the user. By removing this barrier we can build a single compute kernel approximating the target function composition at a cost similar to that of approximating just one elementary function. In the next section we show that building the piece-wise polynomial approximation to a relatively complex non-elementary function can be done accurately and resource efficiently.

IV. Case study

Computational finance is an application domain with ever increasing performance requirements, driven by both competition between financial institutions as well as new regulatory requirements. As many applications in finance are pushing conventional HPC technology to its limits, reaching exascale performance would be desirable and using efficient dataflow-oriented accelerators could enable this leap in the near future. By adopting a dataflow-oriented development approach, employing dedicated optimisations and replacing conventional floating-point formats whenever possible, significant gains over conventional implementations can be achieved.

In this case study we focus on the normal cumulative distribution function (NCDF) used in many financial applications, e.g. the Curran's approximation of the Asian option pricing model, with target accuracy being less than one penny per million pounds (9 decimal places of precision). For performance reasons, this application needs the instantiation of several copies of the NCDF hardware block, which creates a challenge of resource efficiency.

We first examine a naïve DFE implementation of NCDF employing double precision, then we analyse a version optimised with Kernel Merger and finally we evaluate a piecewise polynomial approximation of NCDF based on the Remez algorithm [11].

A. Baseline approximation of NCDF

The normal cumulative distribution function is defined as:

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} exp(-z^2/2)dz$$

Case	Add	Mul	Div	Exp
Approximation Interval 1	11	12	1	0
Approximation Interval 2	19	21	2	1
Approximation Interval 3	14	17	4	1

TABLE I
NUMBER OF ARITHMETIC OPERATIONS IN NCDF APPROXIMATION DEVELOPED FOR CPUS.

This is an example of a non-elementary function (one that cannot be represented exactly as a composition of elementary functions), which requires extra effort in approximation.

As a baseline, we take a piece-wise approximation of the NCDF function developed for evaluation on CPUs in IEEE double precision. Figure 8 illustrates the approximation strategy: the x axis is split into 3 intervals and the function is approximated on each interval separately. Table I reports the total number of arithmetic operations and exponential function evaluations used in this approximation. Later in this section we compare the approximation accuracy of this double-precision evaluation on CPU with approximations built on DFE in floating and fixed-point types of various bit widths.

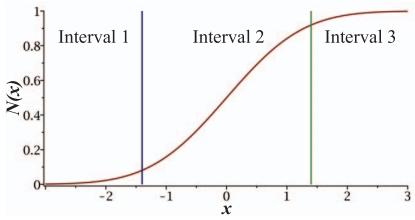


Fig. 8. Normal cumulative distribution function (NCDF).

B. Naïve and Kernel Merger implementations

The naïve dataflow implementation of NCDF evaluates in a single dataflow kernel all three branches present in the CPU code (where each branch evaluates a different approximation within its corresponding interval), and selects a single result depending on the value of the input argument x. This approach is inefficient since all three branches must be mapped onto the DFE, even if only one of them is active at each cycle.

In order to optimise resource utilisation we first employ Kernel Merger. We implement the three function approximations as separate kernels and select an active kernel at run time at each cycle based on the value of x.

In order to explore the trade-off between resource usage and resulting approximation accuracy for a given data type, we conducted precision analysis of all internal variables inside the Kernel Merger based NCDF evaluation kernel. We analyse both fixed and floating-point data types, both of them with different bit widths. The analysis indicates that the fixed-point representation requires a 16-bit integer part to avoid overflows. Further in this section all fixed-point data types are signed.

Figure 9 compares the resource utilisation of the naïve and Kernel Merger based implementations for different data types (32 bit-width is not considered since it does not match the precision constraints). We use MaxCompiler 2016.1.1 with default values for both pipelining and DSP factors (1.0 and 0.5, respectively). In all cases the Kernel Merger optimisation reduces the amount of used hardware resources. For example,

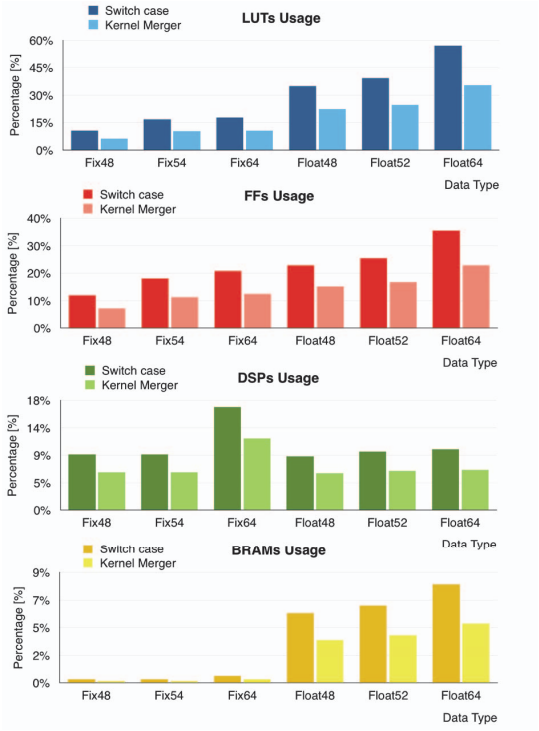


Fig. 9. Resource usage comparison between the naïve and Kernel Merger based implementation for the following data types: Fix48(16, 32), Fix54(16, 38), Fix64(16, 48) (in brackets: number of integer bits, then number of fractional bits), and Float48(11, 37), Float52(11, 41), Float64(11, 53) (in brackets: number of exponent bits, then number of mantissa bits).

in the case of a Float64 data type Kernel Merger saves 14.6% of LUTs, 12.14% of FFs, 3.15% of DSPs and 3.07% of BRAMs available on a single MAX4 DFE.

C. Implementation based on piece-wise polynomials

We also implement NCDF using the piece-wise polynomial approximation generated by the Remez algorithm. We split the x axis into 1024 intervals and approximate NCDF on each interval with its own 4^{th} order polynomial. The dataflow kernel implementing this approximation strategy consists of a lookup table that stores the coefficients for each polynomial, and the straightforward implementation of a Horner rule evaluating the polynomial in fixed-point arithmetic. For this implementation strategy, the dynamic range analysis of the representative input data set suggests using only 4 bits for the integer part. Hence, we build this kernel using the Fix34, Fix40 and Fix48 data types, all of them with a 4-bit integer part. Note that for the same bit width as in the Kernel Merger based implementation we obtain more accurate data representation by only *changing the numerical algorithm*; this is due to the additional bits dedicated to representing the fractional part and the fewer bits used for the integer part of the fixed-point representation.

Figure 10 compares the resource utilisation of piece-wise NCDF approximation and the approximation of a single exponential function. It shows that evaluating both mathematical functions on DFEs requires comparable amount of on-chip resources despite the fact the NCDF is not an elementary function. It also shows together with Figure 9 that choosing

the evaluation strategy specific for DFEs results in substantial savings of on-chip resources.

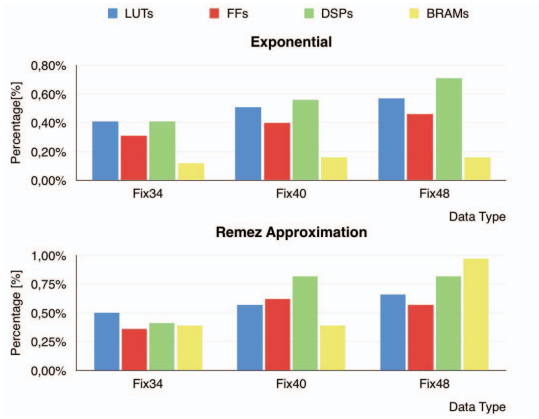


Fig. 10. Resource utilisation of piece-wise polynomial approximation of NCDF versus an exponential function approximation using the following data types: Fix34(4, 30), Fix40(4, 36) and Fix48(4, 44).

D. Numerical accuracy versus data representation

In this section we compare the numerical accuracy of the NCDF evaluation achieved with Kernel Merger and piece-wise polynomial approximation. The numerical accuracy of the naïve and Kernel Merger implementations are identical because they yield effectively the same compute flow. The analysis is based on the *average relative error* of approximation computed against the reference data set representative to the Asian option pricing model. We define the average relative error as $\frac{1}{N}\sum_{k=1}^{N}\frac{1}{D_k}|C_k - D_k|$, where N is the dimension of the input data set (100 000 in our case) and D_k and C_k are the NCDF values computed on DFE and CPU respectively.

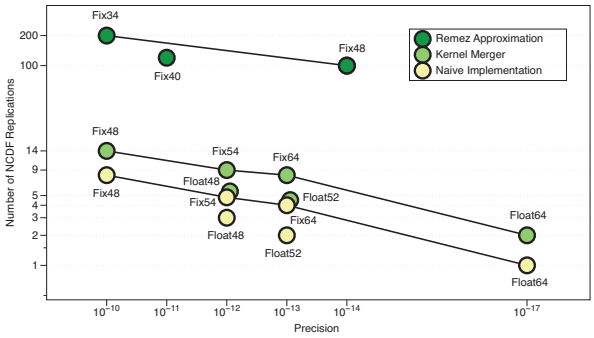


Fig. 11. Accuracy comparison between Naïve implementation, Kernel Merger, and Remez approximations. Lines connecting points show the Pareto optimal solutions for each implementation.

Figure 11 presents the trade-offs between the average relative error and the potential of replicating NCDF evaluation blocks until filling the chip for a given data representation and evaluation strategy. The data points only include the data types for which the NCDF evaluation satisfies the accuracy constraint. This shows that one can instantiate up to 200 copies of the NCDF block on the MAX4 DFE using Fix34 data type with average relative error of order 10^{-10}, while Kernel Merger based evaluator for the same error target can only be replicated 14 times using Fix48 data type.

The time for evaluating N samples with k NCDF instances on-chip at clock frequency of 200 MHz is $N/(k \cdot 2 \cdot 10^8)$. On a single core of a Xeon E5-2697 v2 CPU server, the execution time for 100 000 NCDF evaluations in double precision is 2.9 *ms*, while the naïve Float64 implementation on the DFE without any optimisations supports a single NCDF instance on the chip and takes 500μs, yielding a speed-up of $5.8\times$. Since the targeted CPU contains 12 cores, this speed-up is not competitive. However, after applying all optimisations (Remez approximation and using Fix34) we can now fit 200 NCDF instances on the DFE and still meet the accuracy constraint. If the inputs and outputs to NCDF need to be streamed onto the DFE from LMem, the design becomes IO bound and achieves a 258 times speed up with 45 instances of NCDF over a single CPU core. However, NCDF is often used as a component of larger financial computational models where data is kept on chip. In this case, IO limitations are not relevant and we can take advantage of the 200 times smaller size of the fully optimised NCDF implementation.

V. Conclusion

Moving to exascale performance poses many challenges to the industry and adopting a dataflow-oriented computing model can make the required performance level available with much greater space and power efficiency. We present the general dataflow computing paradigm and highlight several dedicated optimisations that cut through conventional layers of abstraction. Using a case study from computational finance, we show that a naïve dataflow implementation provides a $5.8\times$ speed up over a single CPU core, but after applying dedicated optimisations that replace floating point with fixed point and use a more efficient approximation technique, the DFE speed up is boosted to $258\times$ when streaming data in and out the chip. When evaluating the NCDF as an on-chip component to a larger financial design, the optimisations result in a $200\times$ smaller design than the original naïve solution.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison, *Exascale Computing Technology Challenges*. Springer, 2011, pp. 1–25.

[2] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov. 1980.

[3] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.

[4] L. Gan, H. Fu, C. Yang, W. Luk, W. Xue, O. Mencer, X. Huang, and G. Yang, "A highly-efficient and green data flow engine for solving Euler atmospheric equations," in *Field Programmable Logic and Applications, (FPL 2014)*. IEEE, 2014, pp. 1–6.

[5] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for accelerating short read alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. PP, no. 99, pp. 1–1, 2016.

[6] https://www.extrahpc.eu/.

[7] *Multiscale Dataflow Programming*, Maxeler Technologies, 2015.

[8] N. Voss, S. Girdlestone, O. Mencer, and G. Gaydadjiev, "Automated dataflow graph merging," in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2016.

[9] "Maxeler Standard Library (MaxPower)," https://github.com/maxeler/maxpower, accessed: 2016-11-30.

[10] J. F. Hart, *Computer Approximations*. Melbourne, FL, USA: Krieger Publishing Co., Inc., 1978.

[11] J. M. Muller, *Elementary functions*, 2nd ed. Birkhauser Boston, 2006.