Journal of Big Data
a SpringerOpen Journal

CASE STUDY   Open Access

An AppGallery for dataflow computing

Nemanja Trifunovic[1], Veljko Milutinovic[1*], Nenad Korolija[1] and Georgi Gaydadjiev[2]

*Correspondence:
vm@etf.rs
[1] School of Electrical Engineering, University of Belgrade, Bulevar Kralja Aleksandra 73, 11120 Belgrade, Serbia
Full list of author information is available at the end of the article

Abstract
This paper describes the vision behind and the mission of the Maxeler Application Gallery (AppGallery.Maxeler.com) project. First, it concentrates on the essence and performance advantages of the Maxeler dataflow approach. Second, it reviews the support technologies that enable the dataflow approach to achieve its maximum. Third, selected examples of the Maxeler Application Gallery are presented; these examples are treated as the final achievement made possible when all the support technologies are put to work together (internal infrastructure of the AppGallery.Maxeler.com is given in a follow-up paper). As last, the possible impact of the Application Gallery is presented and the major conclusions are drawn.

Keywords: Dataflow, Supercomputing, Maxeler, Applications

Background

A rule is that each and every paradigm-shift idea passes through four phases in its lifetime. In the phase #1, the idea is radicalized (people laugh on it). In the phase #2, the idea is attacked (some people aggressively try to destroy it). In the phase #3, the idea is accepted (most of those who were attacking it, now keep telling around that it was their idea, or at least that they always were supportive of that idea). In the final phase #4, the idea is considered as something that existed forever (those who played roles in initial phases are already dead, physically or professionally, by the time the fourth phase started).

The main question for each paradigm-shift research effort is how to make the first two phases as short as possible? In the rest of this text, this goal is referred to as the “New Paradigm Acceptance Acceleration” goal, or the NPAA goal, for short.

The Maxeler Application Gallery project is an attempt to achieve the NPAA goal in the case of dataflow supercomputing. The dataflow paradigm exists on several levels of computing abstraction. The Maxeler Application Gallery project concentrates on the dataflow approach that accelerates critical loops by forming customized execution graphs that map onto an reconfigurable infrastructure (currently FPGA-based). This approach provides considerable speedups over the existing control flow approaches, unprecedented power savings, as well as a significant size reduction of the overall supercomputer. This said, however, dataflow supercomputing is still not widely accepted, due to all kinds of barriers, ranging from the NIH syndrome (Not Invented Here), through 2G2BT (Too Good To Be True) till the AOC syndrome (Afraid Of Change) widely present in the high-performance community.

The technical infrastructure of the Maxeler Application Gallery project is described in [1]. The paper at hand describes: (1) the essence of the paradigm that the Maxeler Application Gallery project is devoted to (2) the whereabouts of the applications used to demonstrate the superiority of the dataflow approach over the control flow approach, and (3) the methodologies used to attract as many educators and researchers as possible, to become believers into this "new" and promising paradigm shift.

This paper is organized as follows: in the next section we introduce the essence of the paradigm. Next, we introduce the support technologies required by the paradigm. Thereafter selected examples incorporated into the Maxeler Application Gallery project are presented along with the details of the selected examples. A separate section is dedicated on the expected impact while the conclusions summarize the paper. In addition, Figs. 1, 2 illustrate the structure and impact of a typical AppGallery.Maxeler.com example.

The major references of relevance for this paper are: [2] about trading off latency and performance [3], about splitting temporal computing (with mostly sequential logic) and spatial computing (with combinational logic only) [4], about the Maxeler dataflow programming paradigm [5], about selected killer applications of dataflow computing [6], and [7] about the essence of the paradigm [8], about algorithms proven successful in the dataflow paradigm, and [9] about the impacts of the theories of Cutler, Kolmogorov, Feynman, and Prigogine. See [10–12] for issues in arithmetic, reconfigurability, and testing.

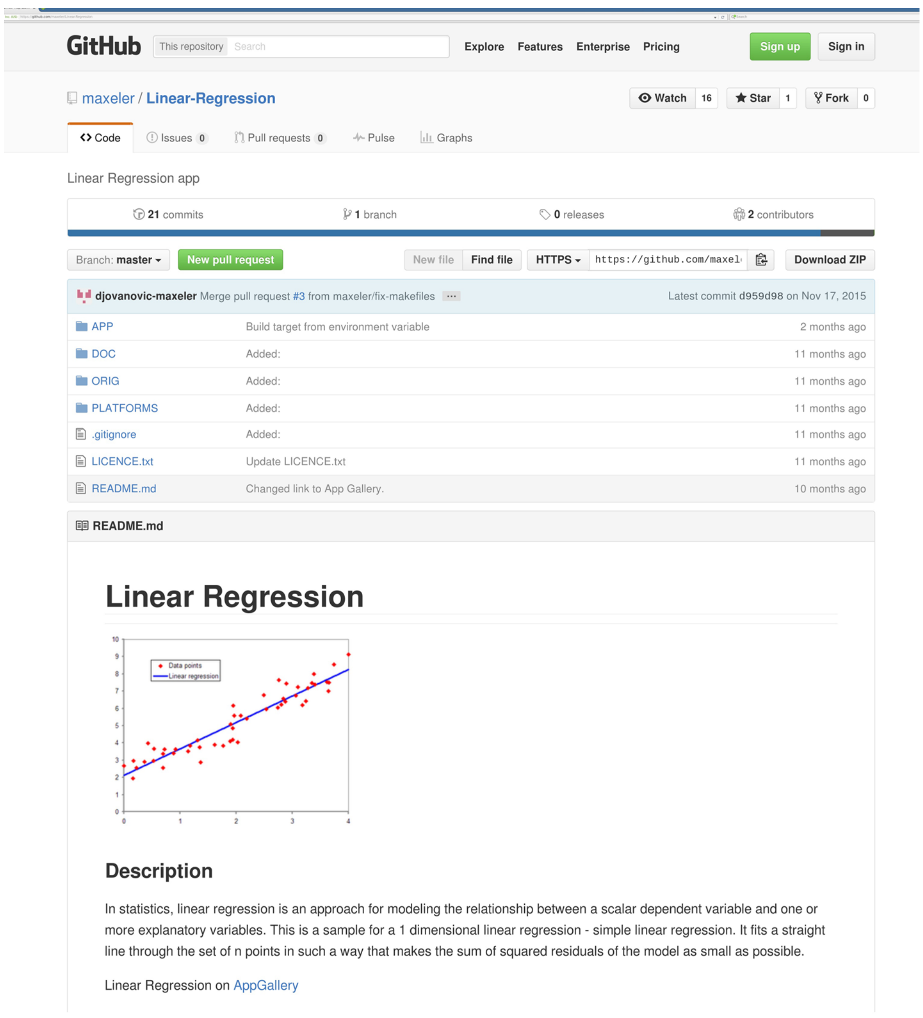
Case description

The dataflow supercomputing paradigm essence

At the time when the von Neumann paradigm for computing was formed, the technology was such that the ratio of arithmetic or logic (ALU) operation latencies over the communication (COMM) delays to memory or another processor t(ALU)/t(COMM) was extremely large (sometime argued to be approaching infinity).

In his famous lecture notes on Computing, the Nobel Laureate Richard Feynman presented an observation that in theory, ALU operations could be done with zero energy, while communications can never reach zero energy levels, and that speed and energy of computing could be traded. In other words, this means that in practice, the future technologies will be characterized with t(COMM)/t(ALU) extremely large (in theory, t(COMM)/t(ALU) approaching infinity), which is exactly the opposite of what was the case at the times of von Neumann. That is why a number of pioneers in dataflow supercomputing accepted to use the term Feynman Paradigm for the approach utilized by Maxeler computing systems. Feynman never worked in dataflow computing, but his observations made many to believe into the great future of the dataflow computing paradigm (along with his involvement with the Connection Machine design).

Obviously, when computing technology is characterized with extremely large t(COMM)/t(ALU), the control flow machines of the multi-core type (like Intel) or the many-core type (like NVidia) could never be as fast as the dataflow machines like Maxeler, for one simple reason: Buses of the control flow machines will never become of zero length, while many edges of the execution graph can easily be made zero length.



maxeler / Linear-Regression

Linear Regression app

21 commits 1 branch 0 releases 2 contributors

https://github.com/maxel

djovanovic-maxeler Merge pull request #3 from maxeler/fix-makefiles … Latest commit d959d98 on Nov 17, 2015

APP	Build target from environment variable	2 months ago
DOC	Added:	11 months ago
ORIG	Added:	11 months ago
PLATFORMS	Added:	11 months ago
.gitignore	Added:	11 months ago
LICENCE.txt	Update LICENCE.txt	11 months ago
README.md	Changed link to App Gallery.	10 months ago

README.md

Linear Regression

Description

In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable and one or more explanatory variables. This is a sample for a 1 dimensional linear regression - simple linear regression. It fits a straight line through the set of n points in such a way that makes the sum of squared residuals of the model as small as possible.

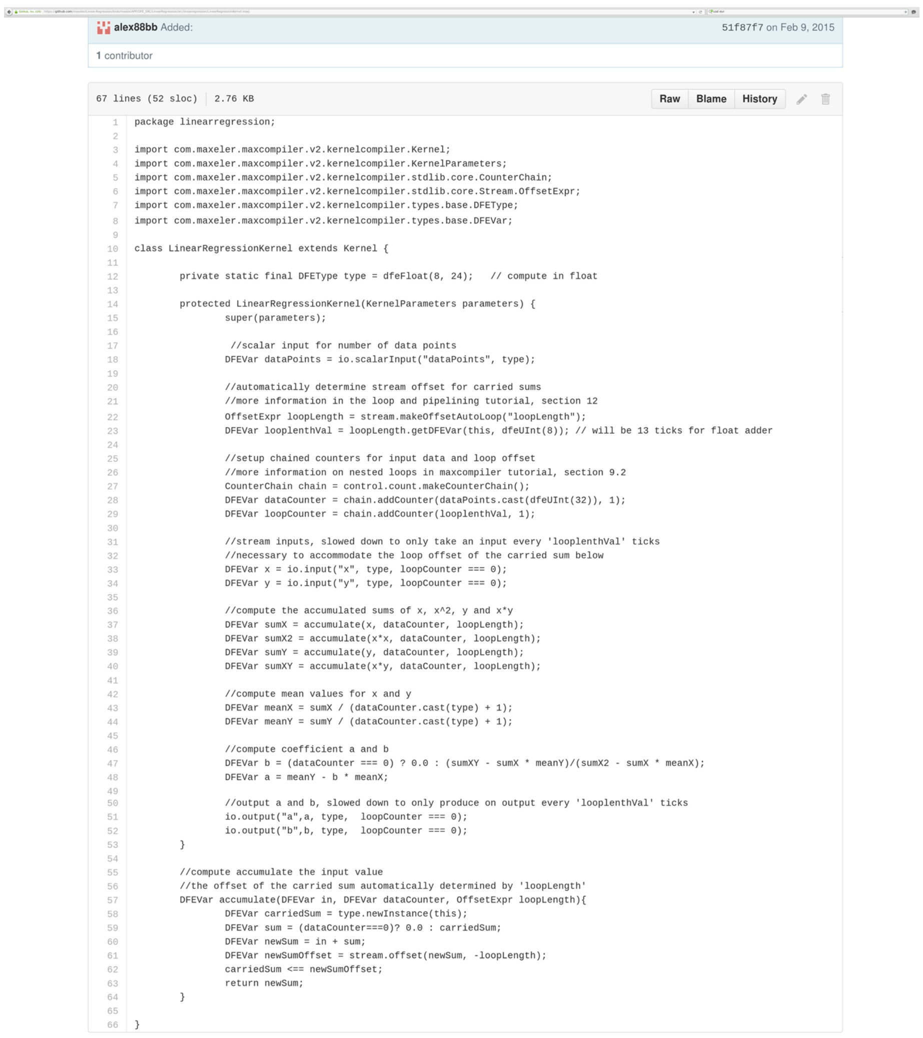
Linear Regression on AppGallery

Fig. 1 GitHub repository of the linear regression application

The main technology related question now is: "Where is the ratio of t(ALU) and t(COMM) now?"

According to the above mentioned sources, the energy needed to do an IEEE Floating Point Double Precision Multiplication will be only 10pJ around the year 2020, while the energy needed to move the result from one core to another core of a multi-core or a many-core machine is 2000pJ, which represents a factor of 200×. On the other hand, moving the same result over an almost-zero-length edge in a Maxeler dataflow machine, in some cases, may take less than 2pJ, which represents a factor of 0.2×.

Therefore, the times have arrived for the technology to enable the dataflow approach to be effective. Here the term technology refers to the combination of: the hardware technology, the programming paradigm and its support technologies, the compiler technology, and the code analysis, development, and testing technology. These technologies are shed more light at in the next section.



```
package linearregression;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.OffsetExpr;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class LinearRegressionKernel extends Kernel {

        private static final DFEType type = dfeFloat(8, 24);   // compute in float

        protected LinearRegressionKernel(KernelParameters parameters) {
                super(parameters);

                 //scalar input for number of data points
                DFEVar dataPoints = io.scalarInput("dataPoints", type);

                //automatically determine stream offset for carried sums
                //more information in the loop and pipelining tutorial, section 12
                OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
                DFEVar looplenthVal = loopLength.getDFEVar(this, dfeUInt(8)); // will be 13 ticks for float adder

                //setup chained counters for input data and loop offset
                //more information on nested loops in maxcompiler tutorial, section 9.2
                CounterChain chain = control.count.makeCounterChain();
                DFEVar dataCounter = chain.addCounter(dataPoints.cast(dfeUInt(32)), 1);
                DFEVar loopCounter = chain.addCounter(looplenthVal, 1);

                //stream inputs, slowed down to only take an input every 'looplenthVal' ticks
                //necessary to accommodate the loop offset of the carried sum below
                DFEVar x = io.input("x", type, loopCounter === 0);
                DFEVar y = io.input("y", type, loopCounter === 0);

                //compute the accumulated sums of x, x^2, y and x*y
                DFEVar sumX = accumulate(x, dataCounter, loopLength);
                DFEVar sumX2 = accumulate(x*x, dataCounter, loopLength);
                DFEVar sumY = accumulate(y, dataCounter, loopLength);
                DFEVar sumXY = accumulate(x*y, dataCounter, loopLength);

                //compute mean values for x and y
                DFEVar meanX = sumX / (dataCounter.cast(type) + 1);
                DFEVar meanY = sumY / (dataCounter.cast(type) + 1);

                //compute coefficient a and b
                DFEVar b = (dataCounter === 0) ? 0.0 : (sumXY - sumX * meanY)/(sumX2 - sumX * meanX);
                DFEVar a = meanY - b * meanX;

                //output a and b, slowed down to only produce on output every 'looplenthVal' ticks
                io.output("a",a, type,  loopCounter === 0);
                io.output("b",b, type,  loopCounter === 0);
        }

        //compute accumulate the input value
        //the offset of the carried sum automatically determined by 'loopLength'
        DFEVar accumulate(DFEVar in, DFEVar dataCounter, OffsetExpr loopLength){
                DFEVar carriedSum = type.newInstance(this);
                DFEVar sum = (dataCounter===0)? 0.0 : carriedSum;
                DFEVar newSum = in + sum;
                DFEVar newSumOffset = stream.offset(newSum, -loopLength);
                carriedSum <== newSumOffset;
                return newSum;
        }

}
```

Fig. 2 Code of the linear regression kernel written in MaxJ language

The paradigm support technologies

As far as the hardware technology, a Maxeler dataflow machine consists of either a conventional computer with a Maxeler PCIe board (e.g., Galava, Isca, Maia etc. all collectively referred as DataFlow Engines—DFEs), or is a 1U box[1] that could be stacked into a 40U rack, or into a train of 40U racks.

As far as the programming paradigm and the support technologies, the Maxeler dataflow machines are programmed in space, rather than in time, which is the case with typical control flow machines. In the case of Maxeler (and the Maxeler Application Gallery project), the general framework is referred to as OpenSPL (Open Spatial Programming Language, http://www.OpenSPL.org), and its specific implementation used for the development of AppGallery.Maxeler.com applications is called MaxJ (shorthand for Maxeler Java).

[1] 1U refers to the minimal size in rack-mount servers; the number (×U) indicates the size of the rack-mount.

As far as compilation, it consists of two parts: (a) Generating the execution graph, and (b) Mapping the execution graph onto the underlying reconfigurable infrastructure. The Maxeler compiler (MaxCompiler) is responsible for the first part. The synthesis tools from the FPGA device vendor are responsible for the second part. As interface between these two distinct phases VHDL is used.

As far as the synergy of the compiler and the internal "nano-accelerators" (small add-on at the hardware structure level, invisible at the level of the static dataflow concept, and invisible to the system designer, but under the tight control of the compiler), the most illustrative is the fact that the compiled code for a board based on Altera or Xilinx chips is much slower compared to the code compiled for a Maxeler board having the same type and the same number of Altera or Xilinx chips. This is due to many different "nano-accelerators" present at the Maxeler board. These are invisible on the concept level and invisible to the dataflow programmer, but extremely important for the concept implementation, visible to the Maxeler compiler, and it knows how to utilize them. These add-ons are here referred to as "nano-accelerators".

As far as the tools for analysis, development, and testing, they have to take care of the following issues: (a) When the paradigm changes, the algorithms that previously were the best ones, according to some criterion, now are not any more, so the possible algorithms have to be re-evaluated, or new ones have to be created; (b) When the paradigm changes, the usage of the new paradigm has to be made simple, so the community accepts it, with as short as possible delays, and (c) When the paradigm changes, the testing tools have to be made compatible with the needs of the new paradigm, and the types of errors that occur most frequently.

The best example of the first is bitonic sort, which is considered among the slowest in control flow and among the fastest in dataflow. The best example of the second is WebIDE. Maxeler.com with all its features to support programmers. The most appropriate example of the third is related to the on-going research aimed at including a machine learning assistant that analyses past errors and offers hints for the future testing-oriented work.

All the above support technologies have to be aware of the following: If the temporal and the spatial computing are separated (hybrid computing with only combinational logic in the spatial part), and if latency is traded for better performance (maximal acceleration even for the most sophisticated types of data dependencies between different loop iterations), benefits occur only if the support technologies are able to utilize them!

Presentation of the Maxeler application gallery project

The next section of this paper contains selected examples from the Maxeler Application Gallery project, visible at the web (AppGallery.Maxeler.com). Each example is presented using the same template.

The presentation template, wherever possible, conditionally speaking, includes the following elements: (a) The whereabouts (b) The algorithm implementation essence supported with a figure, (c) The coding approach changes made necessary by the paradigm changes, supported by GitHub details, (d) The major highlights, supported with a figure, (e) The advantages and drawbacks, (f) The future trends, and (g) Conclusions related to complexity, performance, energy, and risks leading to possible problems in the domains of complexity, performance, energy, and creation of new risks that recursively open a new round of all the above.

Discussion and evaluation

Selected Maxeler application gallery examples

The following algorithms will be briefly explained: brain network, correlation, classification, dense matrix multiplication, fast Fourier transform 1D, fast Fourier transform 2D, motion estimation, hybrid coin miner, high speed packet capture, breast mammogram ROI extraction, linear regression, and n-body simulation. These twelve applications were selected, based on the speed-up provided. All these were published recently; however, they represent results of decade-long efforts.

The problem of transforming control flow algorithms to dataflow algorithms is not new. It can be found in the theory of systolic arrays. For example, the Miranker and Winkler's method [13], which is an extension of the Kuhn's method [14], is best suited for transforming algorithms consisting of loops with constant execution time and dependencies within iterations, to systolic arrays, but it also supports other algorithms. Figure 3 demonstrates this method applied to the problem of dataflow programming.

Although various methods help in transforming algorithms from control flow to dataflow, a programmer has to be somehow aware of the main characteristics of the underlying hardware in order to produce optimal code. For example, if one has to process each row of a matrix separately using dataflow engines (DFEs), where each element depends on the previous one in the same row, the programmer should reorder the execution, so that other rows are being processed before the result of processing an element is given to the next element in the same row.

Therefore, one can say that changing the paradigm requires changing the hardware in order to produce results quicker with lower power consumption, but also changing our way of thinking. In order to do that, we have to think big, as Samuel Slater, known as "The Father of the American Factory System". Instead of thinking how we would solve a single piece of the problem, we should rather think of how a group of specialized workers would solve the same problem, and then focus on the production system as a whole, imagining the worker in a factory line. For state of the art, see [15–16].

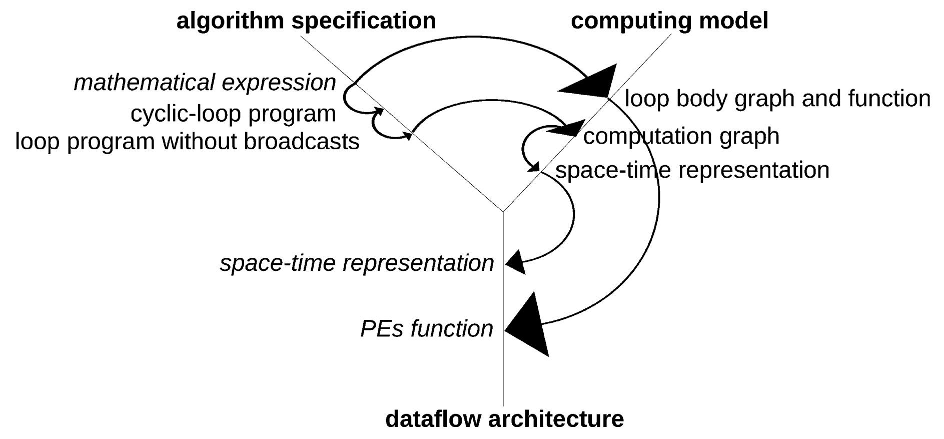
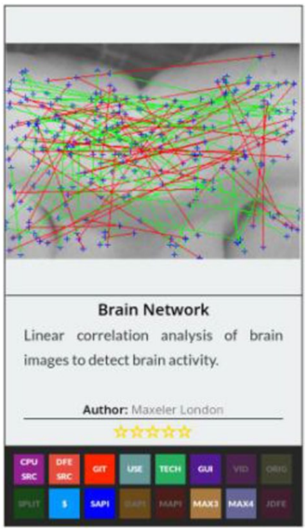


Fig. 3 The Miranker and Winkler's method

Brain network

In July 2013, Maxeler has developed a brain network simulator for extracting network activity by extracting a dynamic network from brain activity of slices of a mice brain (application icon is given in Fig. 4). The application implements a linear correlation analysis of brain images to detect brain activity, as shown in Fig. 5. Figure 6 depicts a part of the linear correlation kernel that exploits available parallelism potentials of dataflow engines (DFEs). Figure 7 depicts details of the implementation. While the Maxeler code is more complicated than the one for the control flow algorithm, the achievements in power reduction and speed are noticeable. DFEs have higher potentials for further development comparing to conventional processors, making it possible to develop a dynamic community analysis. For a data center processing, a MaxNode with four MAX3 DFEs is equivalent in speed to 17 high-end Intel nodes, where the energy consumption of each MaxNode is comparable to Intel nodes.


Brain Network

Linear correlation analysis of brain images to detect brain activity.

Author: Maxeler London

Fig. 4 Application icon for brain network

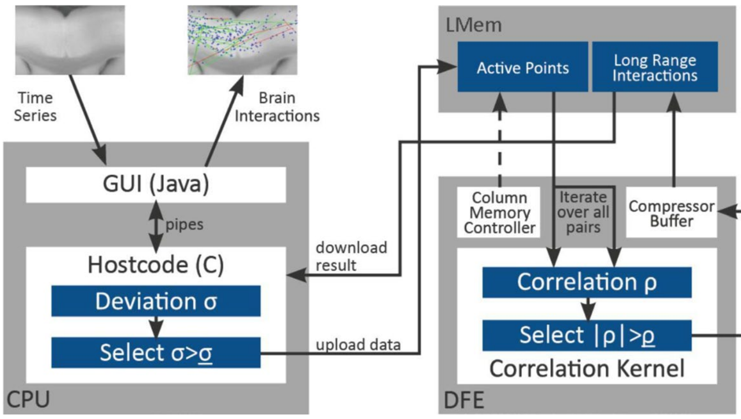


Fig. 5 The brain network algorithm

```
for (int i=0; i<num_pipes; ++i){
        //Get the temporal series on current pipe
        DFEVector<DFEVar> temporal_series_row = row_buffer[i]["temporal_series"];

        //Calculate E[ab] starting with summation...
        //Use DSPs to implement "multiply and add" (9bitsx9bits -> 1 DSP -> 30 DPSs)
        DFEVar average_product  = constant.var(0);
        for (int j=0; j<window; ++j){
                average_product += temporal_series_row.get(j).cast(dfeInt(9))*
                temporal_series_column.get(j).cast(dfeInt(9));
        }
        //Obtain E[ab] dividing by window...
        //implemented as product of inverse in order to use DSPs (18x20 on single DSP)
        average_product *= constant.var(dfeFixMax(18, 1.0/window,
                SignMode.TWOSCOMPLEMENT), 1.0/window);
        //Calculate product between average E[a]*E[b] (24x24 still on DSP)
        DFEVar product_between_average = ((DFEVar)row_buffer[i]["average"]) *
                ((DFEVar)column_point["average"]);
        //Calculate covariance as E[ab]-E[a]*E[b]
        DFEVar covariance = average_product - product_between_average;

        //Calculate product between standard deviations std(a)*std(b)...
        //mapped on DSPs (24x24 on two DSPs)
        DFEVar product_between_deviations = (((DFEVar)row_buffer[i]["standard_deviation"]) *
                ((DFEVar)column_point["standard_deviation"]));
        //Reduce resources used for division by squash optimization
        //optimization.pushSquashFactor(0.5);
        //optimization.popSquashFactor();

        //Finally, calculate the correlation (range [-1,1] rounded with 30 fractional bits)
        optimization.pushFixOpMode(Optimization.bitSizeExact(32), Optimization.offsetExact(-30),
                MathOps.DIV);
        correlation[i] = covariance/product_between_deviations;
        optimization.popFixOpMode(MathOps.DIV);
}
```

Fig. 6 The kernel code

Correlation

In February 2015, Maxeler has developed a dataflow implementation of the statistical method to analyse the relationship between two random variables, by calculating sums of vectors given as streams (application icon is given in Fig. 8). Figure 9 depicts calculating the sum of a moving window of elements by adding a new element and substituting the old one from the previous sum. Figure 10 depicts crucial parts of the code that can be accelerated using DFEs. Figure 11 depicts which sums are calculated using CPU and which ones are calculated using DFEs. Dataflow achieves better performances in computation of sums of products, both in speed and power consumption, while some additional hardware is needed for implementing the code using the Maxeler framework. Comparing MAX2 and MAX4 Maxeler cards performance, as well as Intel's CPU development in the same period (2005 and 2010, respectively), we can expect higher increase in processing power in the dataflow computing, enabling much more than 12 pipes to

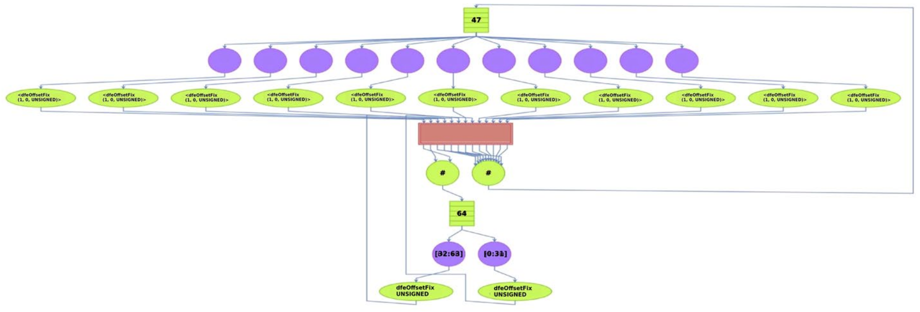
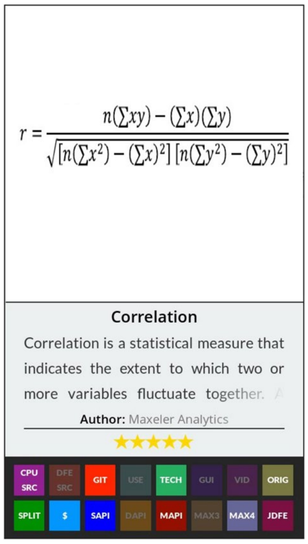


Fig. 7 Details of the implementation

be placed on the chip, while at the same time memory bandwidth will further increase the gap between dataflow and control flow paradigms. In order to achieve good price-performance ratio, one needs to have a relatively big stream of data, which is usually the case.



$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n(\sum x^2) - (\sum x)^2][n(\sum y^2) - (\sum y)^2]}}$$

Correlation

Correlation is a statistical measure that indicates the extent to which two or more variables fluctuate together.

Author: Maxeler Analytics

★★★★★

Fig. 8 Application icon for correlation

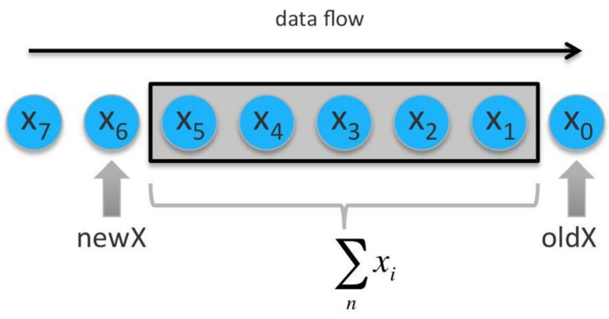
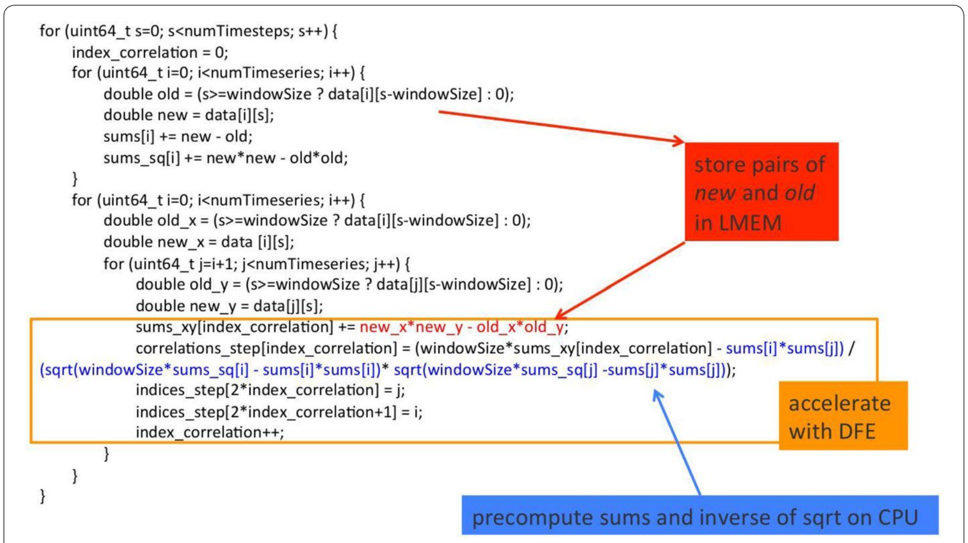


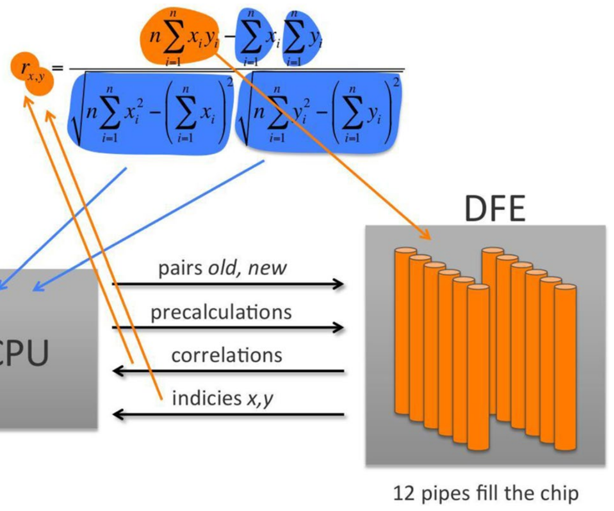
Fig. 9 The moving sum algorithm



```
for (uint64_t s=0; s<numTimesteps; s++) {
    index_correlation = 0;
    for (uint64_t i=0; i<numTimeseries; i++) {
        double old = (s>=windowSize ? data[i][s-windowSize] : 0);
        double new = data[i][s];
        sums[i] += new - old;
        sums_sq[i] += new*new - old*old;
    }
    for (uint64_t i=0; i<numTimeseries; i++) {
        double old_x = (s>=windowSize ? data[i][s-windowSize] : 0);
        double new_x = data [i][s];
        for (uint64_t j=i+1; j<numTimeseries; j++) {
            double old_y = (s>=windowSize ? data[j][s-windowSize] : 0);
            double new_y = data[j][s];
            sums_xy[index_correlation] += new_x*new_y - old_x*old_y;
            correlations_step[index_correlation] = (windowSize*sums_xy[index_correlation] - sums[i]*sums[j]) /
(sqrt(windowSize*sums_sq[i] - sums[i]*sums[i])* sqrt(windowSize*sums_sq[j] -sums[j]*sums[j]));
            indices_step[2*index_correlation] = j;
            indices_step[2*index_correlation+1] = i;
            index_correlation++;
        }
    }
}
```

store pairs of *new* and *old* in LMEM
accelerate with DFE
precompute sums and inverse of sqrt on CPU

Fig. 10 Extracting the most compute-demanding code for acceleration using DFEs (dataflow engines)

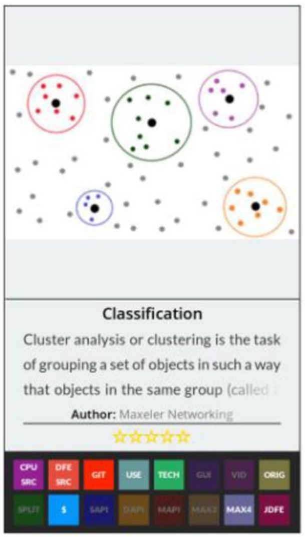


$$r_{x,y} = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}\sqrt{n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2}}$$

Fig. 11 Details of the implementation

Classification

In March 2015, Maxeler developed a dataflow implementation of the classification algorithm, by engaging multiple circuits to compute the squared Euclidean distances between points and classes and to compare these to the squared radii (application icon is given in Fig. 12). Figure 13 depicts the classification algorithm. Figure 14 depicts a part of the CPU code responsible for classifying, where the most inner loop can be parallelized using the dataflow approach. Figure 15 depicts parallel processing of n dimensions of vectors p and q using DFEs. The dataflow paradigm offers both better bandwidth and better computing performances, but only if one provides a relatively big stream of data. Otherwise, a flow throughout the PCIe towards the DFEs may last longer than executing corresponding code on the CPU. Maxeler already offers many more data streams


Classification

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called

Author: Maxeler Networking

Fig. 12 Application icon for classification

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

Fig. 13 The classification algorithm

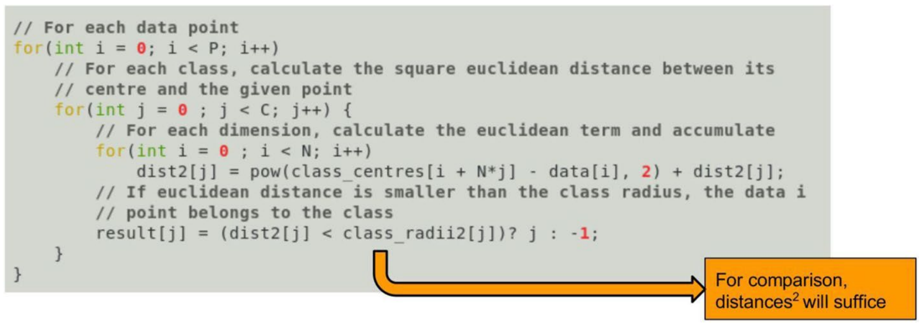

```
// For each data point
for(int i = 0; i < P; i++)
    // For each class, calculate the square euclidean distance between its
    // centre and the given point
    for(int j = 0 ; j < C; j++) {
        // For each dimension, calculate the euclidean term and accumulate
        for(int i = 0 ; i < N; i++)
            dist2[j] = pow(class_centres[i + N*j] - data[i], 2) + dist2[j];
        // If euclidean distance is smaller than the class radius, the data i
        // point belongs to the class
        result[j] = (dist2[j] < class_radii2[j])? j : -1;
    }
}
```

Fig. 14 The CPU code for classification

that could feed DFEs, comparing to the conventional CPUs, and it is expected that this gap will grow in the future. Noticeable differences in computing paradigms are obvious, since the dataflow implementation of the algorithm uses 160 ROMs in parallel to fill up the chip. In the DFE context, ROMs (Read Only Memory) refer to custom look-up tables used to implement complex non-linear functions.

Dense matrix multiplication

In February 2015, Maxeler published a dataflow implementation of the dense matrix multiplication algorithm, by splitting vector–vector multiplications on DFEs, in order

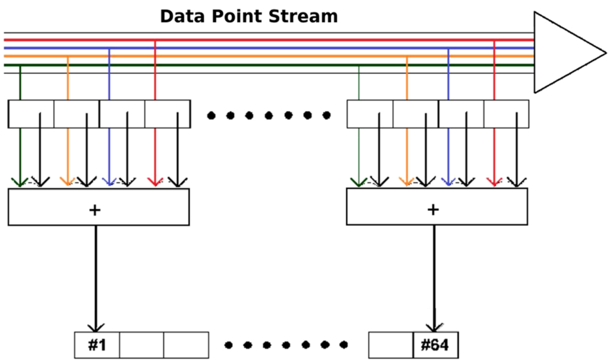
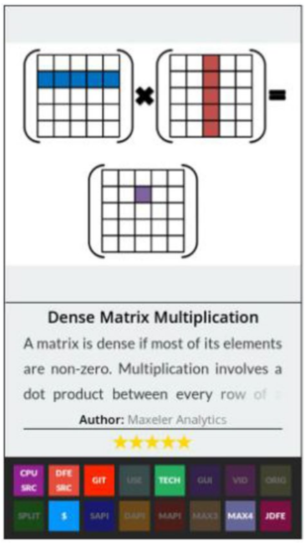


Fig. 15 Streaming the data through DFEs

to increase the performance (application icon is given in Fig. 16). Figure 17 depicts the matrix–matrix multiplication algorithm. Figure 18 depicts the Maxeler manager code responsible for connecting streams to and from the DFEs. Figure 19 depicts parallel processing of parts of the matrix on the DFEs. The dataflow paradigm reduces complexity of the problem of multiplying matrices, but the price of that is sharing results between DFEs. Problems including multiplying matrices tend to increase in time, forcing us to switch between control flow algorithms and dataflow algorithms. Once the matrix size exceeds thousand elements, single MAX4 MAIA DFE at 200 MHz with 480 tile size performs around 20 time faster than the ATLAS SSE3 BLAS on the single core of Intel Xeon E5540.


Dense Matrix Multiplication

A matrix is dense if most of its elements are non-zero. Multiplication involves a dot product between every row of

Author: Maxeler Analytics

Fig. 16 Application icon for dense matrix multiplication

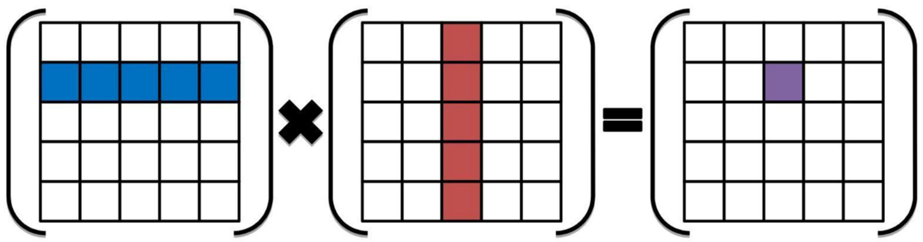


Fig. 17 The matrix–matrix multiplying algorithm

```
public GemmManager(GemmEngineParameters params) {
                    super(params);
                    configBuild(params);

                    DFEType type = params.getFloatingPointType();

                    int tileSize = params.getTileSize();

                    addMaxFileConstant("tileSize", tileSize);
                    addMaxFileConstant("frequency", params.getFrequency());

                    DFELink a = addStreamFromCPU("A");
                    DFELink b = addStreamFromCPU("B");

                    addStreamToCPU("C") <== TileMultiplierKernel.multiplyTiles(this, "TM", type, tileSize, a, b);
}
```

Fig. 18 The Maxeler manager connecting DFEs for matrix multiplication

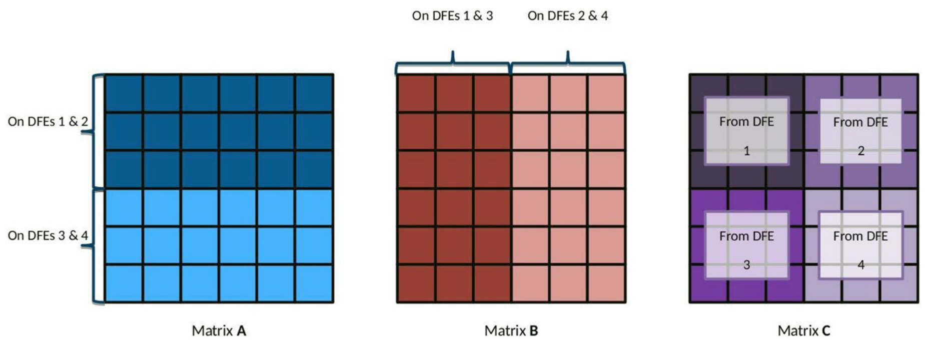
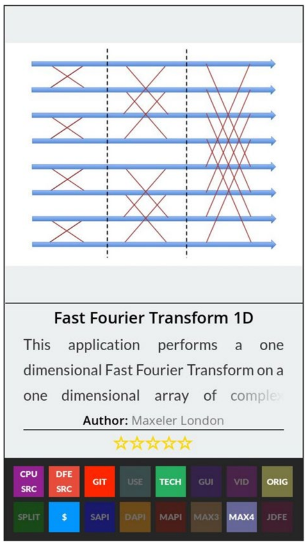


Fig. 19 Dividing of multiplications over several DFEs

Fast Fourier transform 1D

In April 2015, Nils Voss and Maxeler developed a dataflow implementation of the algorithm that performs Fast Fourier transform (FFT) by exploiting the dataflow parallelism (application icon is given in Fig. 20). Figure 21 depicts the 1D FFT process that transforms a function to a series of sinusoidal functions it consists of. Figure 22 depicts the Maxeler Kernel responsible for Fourier Transform. Figure 23 depicts parallel processing of signal components using DFEs. The dataflow paradigm offers better performances comparing to CUDA, since DFEs are much closer to each other, which leads to a faster communication. The only constraint is the size of a chip, i.e., number of available DFEs. Since computers are used for signal processing for a long time already, and



Fast Fourier Transform 1D

This application performs a one dimensional Fast Fourier Transform on a one dimensional array of comple

Author: Maxeler London

Fig. 20 Application icon for the 1D Fast Fourier transform

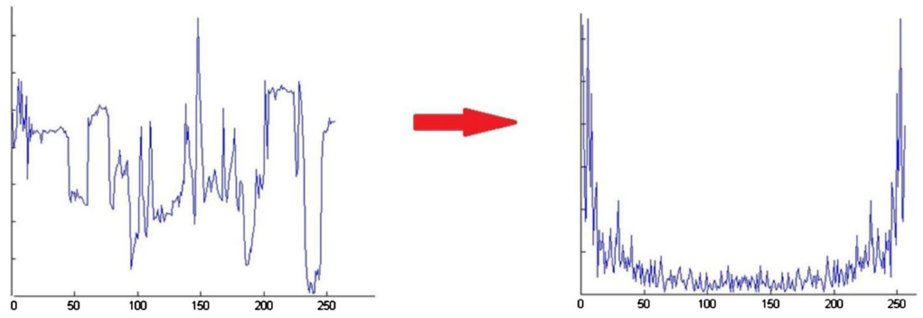


Fig. 21 The Fourier transform process

```
public class FftKernel extends Kernel {
        public FftKernel(KernelParameters kp, FftParams params) {
                super(kp);

                DFEComplexType type = new DFEComplexType(params.getBaseType());
                DFEVectorType<DFEComplex> vectorType = new
                        DFEVectorType<DFEComplex>(type, 4);

                DFEVector<DFEComplex> in = io.input("fft_in", vectorType);
                FftFactory4pipes fftFactory = new FftFactory4pipes(this, params.getBaseType(),
                                                                        params.getN(), 0);
                io.output("fft_out", vectorType) <== fftFactory.transform(in, 1);
        }
}
```

Fig. 22 The DFE code for the 1D Fast Fourier transform

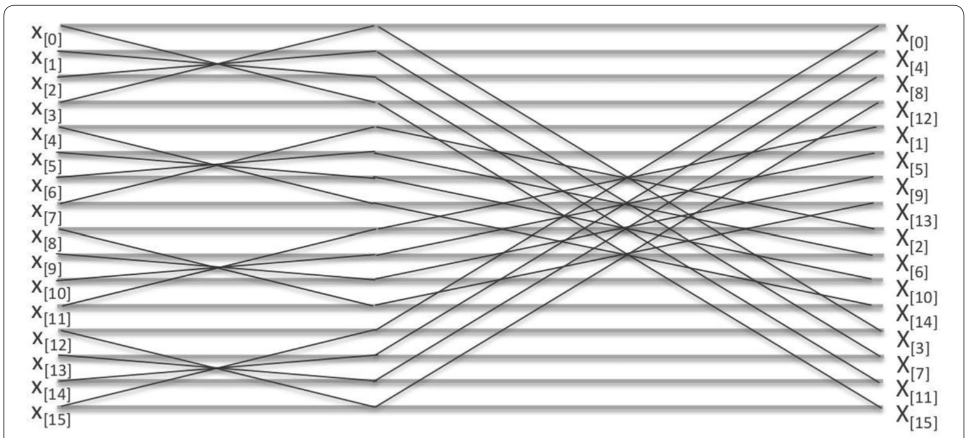
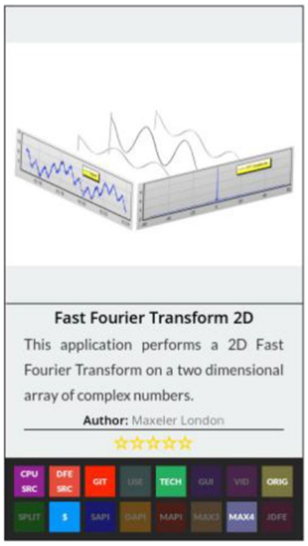


Fig. 23 Parallelizing of data processing using DFEs

signal frequencies are constantly increasing, processing power should increase over time accordingly. This leads to dataflow processing. The presented application adapts the radix-4 decimation-in-time algorithm implemented in the class FftFactory4pipes, which is more efficient than using a radix-2 algorithm.

Fast Fourier transform 2D

In April 2015, Nils Voss and Maxeler developed a dataflow implementation of the algorithm that performs 2D FFT using 1D FFT (application icon is given in Fig. 24). Figure 25 depicts the 2D FFT algorithm for a signal h (n, m) with N columns and M rows. Figure 26 depicts the Maxeler Kernel responsible for FFT 2D. Figure 27 depicts using 1D FFT for calculating 2D FFT. While DFEs offer faster signal processing, communications between DFEs and the CPU may in some cases last too long. In order to have results in time, processing may have to use FPGAs directly, or a CPU with a lower


Fast Fourier Transform 2D

This application performs a 2D Fast Fourier Transform on a two dimensional array of complex numbers.

Author: Maxeler London

Fig. 24 Application icon for classification

$$\hat{h}(k,l) = \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} e^{-i(\omega_k n + \omega_l m)}\, h(n,m)$$

$$h(n,m) = \frac{1}{NM}\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} e^{i(\omega_k n + \omega_l m)}\, \hat{h}(k,l)$$

Fig. 25 The 2D Fast Fourier transform process

```
public class FftKernel extends Kernel {
        public FftKernel(KernelParameters kp, FftParams params) {
                super(kp);

                DFEComplexType type = new DFEComplexType(params.getBaseType());
                DFEVectorType<DFEComplex> vectorType =
                        new DFEVectorType<DFEComplex>(type, 4);

                DFEVar numMat = io.scalarInput("num_matrices", dfeUInt(32));
                DFEVar inputEnable = control.count.simpleCounter(32)
                        < params.getM()*params.getN()/4*numMat;
                DFEVector<DFEComplex> in = io.input("fft_in", vectorType, inputEnable);

                FftFactory4pipes fftFactoryRows =
                        new FftFactory4pipes(this, params.getBaseType(), params.getN(), 0);
                FftFactory4pipes fftFactoryCols =
                        new FftFactory4pipes(this, params.getBaseType(), params.getM(), 0);

                DFEVector<DFEComplex> fftRowsOut = fftFactoryRows.transform(in, 1);

                TransposerMultiPipe transRowsToCols =
                        new TransposerMultiPipe(this, fftRowsOut, params.getN(), params.getM(), true);

                DFEVector<DFEComplex> fftColOutput =
                        fftFactoryCols.transform(transRowsToCols.getOutput(), 1);

                TransposerMultiPipe transColsToRows =
                new TransposerMultiPipe(this, fftColOutput, params.getM(), params.getN(), true);

                io.output("fft_out", vectorType) <== transColsToRows.getOutput();
        }
}
```

Fig. 26 The DFE code for 2D FFT

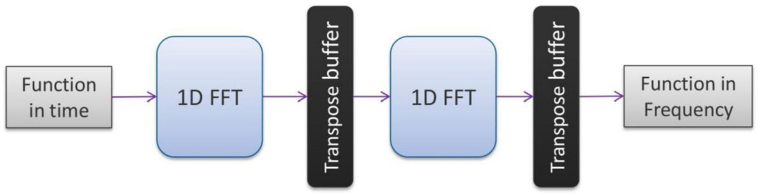
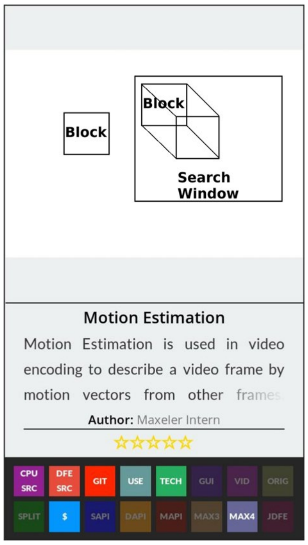


Fig. 27 Implementing the 2D FFT using 1D FFT data processing on DFEs

precision. As time passes, frequencies of signals tend to become higher, requiring 2D FFT to run faster, in order to process signals in real-time. The application presented adapts the radix-4 decimation-in-time algorithm, implemented in the class FftFactory4pipes, which is more efficient than using a radix-2 algorithm.

Motion estimation

In April 2015, Maxeler developed a dataflow implementation of the motion estimation algorithm based on comparing reference blocks and moved blocks (application icon is given in Fig. 28). Figure 29 shows the pseudo-code of the motion estimation algorithm. The DFE code from Fig. 30 computes, in parallel, the sum of absolute differences between source blocks and the reference block for 4×4 sub-blocks. Figure 31 presents relevant components of the system that incorporates Deblocking Filter besides motion estimation into motion compensation and then selects either the corresponding output or intraframe prediction for further processing. The dataflow approach offers faster parallel image processing comparing to control flow counterparts at the cost of transferring frames from the CPU to DFEs. Today's security cameras tend to capture more and more



Motion Estimation

Motion Estimation is used in video encoding to describe a video frame by motion vectors from other frames

Author: Maxeler Intern

☆☆☆☆☆

Fig. 28 Application icon for motion estimation

```
Iterate on each source block, i
        W = corresponding search window,
        Initialize min to maximum integer for all sub-blocks
        Iterate on each reference block, j in W
                curr = SAD of i and j and their sub-blocks
                For each sub-block, if min > curr
                        min = curr
                        minBlock = j
        For each sub-block, output minBlock in the form a motion vector
```

Fig. 29 The pseudo-code of the motion estimation algorithm

```
for (int blockRow = 0; blockRow < blockSize / 4; blockRow++) {
        for (int blockCol = 0; blockCol < blockSize / 4; blockCol++) {
                List<DFEVar> srcBlock = new ArrayList<DFEVar>();
                List<DFEVar> refBlock = new ArrayList<DFEVar>();
                for (int row = 0; row < 4; row++) {
                        for (int col = 0; col < 4; col++) {
                                srcBlock.add(source[row + blockRow * 4][col + blockCol * 4]);
                                refBlock.add(reference[row + blockRow * 4][col + blockCol * 4]);
                                sadList[row*4 + col] = absDiff(source[row + blockRow * 4][col +
                                blockCol * 4], reference[row + blockRow * 4][col + blockCol * 4]);
                        }
                }
                output[0][blockCol + blockRow * blockSize / 4] = adderTree(sadList);
        }
}
```

Fig. 30 The kernel code of the motion estimation algorithm

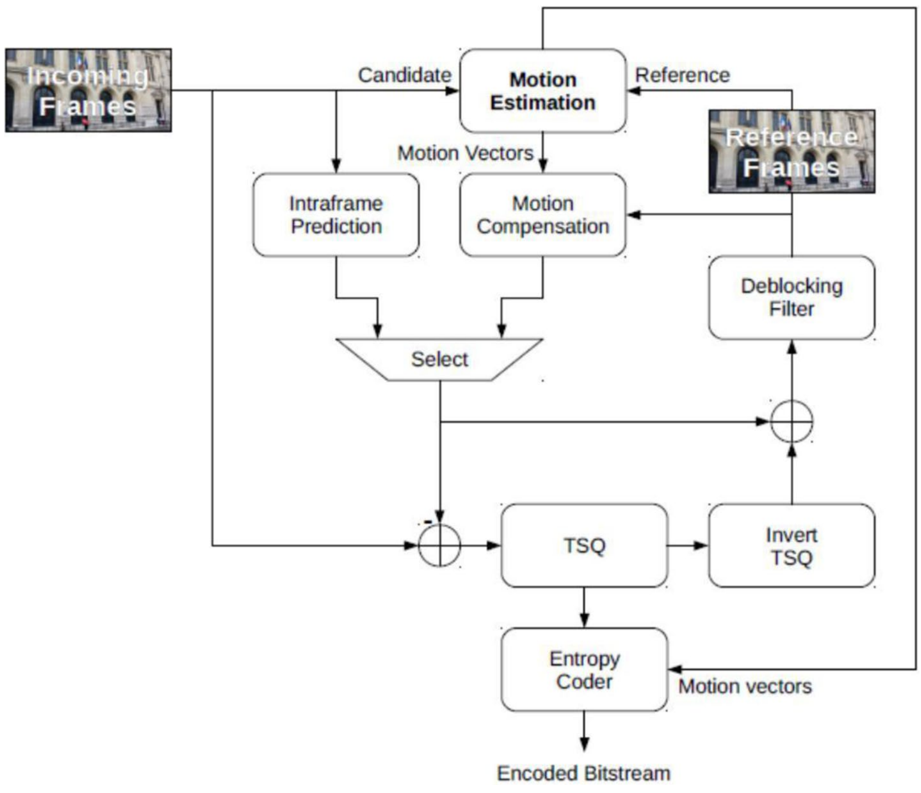


Fig. 31 Implementing the motion estimation using DFEs

events that need to be processed in order to extract valuable information. By having two read ports for the buffer, one can read blocks from two rows in parallel, which allows the kernel to start reading the first block of a row before the reading of the previous row is finished, and thus, further reducing the energy consumption per frame and further increasing the performance.

Hybrid Coin Miner

A coin miner application that can mine SHA-256 based coins and Scrypt based coins simultaneously.

Author: Maxeler London

Fig. 32 Application icon for hybrid coin miner

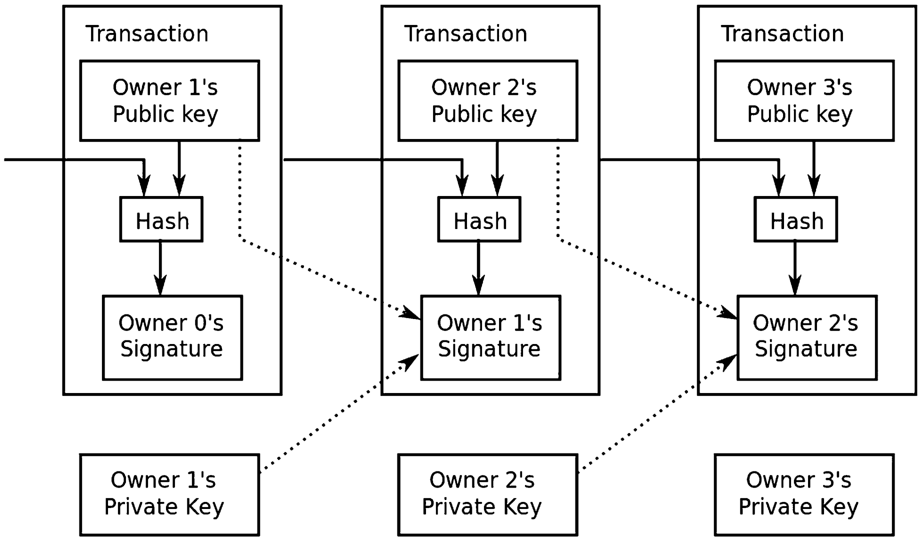


Fig. 33 The used simplified chain of bitcoin ownership

Hybrid coin miner

In March 2014, Maxeler developed a dataflow implementation of the algorithm that performs hybrid coin miner, an application that runs both SHA-256 and Scrypt algorithms simultaneously on DFEs (application icon is given in Fig. 32). Figure 33 depicts the simplified chain of bitcoin ownership. Figure 34 depicts the Maxeler kernel code responsible for a 224-bit comparison that has to be processed over and over again. Figure 35 shows the block diagram connecting DFEs with relevant blocks of the system. In case of coin mining, DFEs do not offer as good speedup as it might be expected, but the power consumption is approximately one order of magnitude lower. A 224-bit comparison is huge, and currently cannot be auto-pipelined by MaxCompiler, so it is done manually by

```
final int B = 16;
final int N = a.getType().getTotalBits();
if (N < B) {
            throw new RuntimeException("Size must be a multiple of " + B);
} else if (N == B) {
            return (a < b);
} else {
            DFEVar aHi = a.slice(N-B, B).cast(dfeUInt(B));
            DFEVar bHi = b.slice(N-B, B).cast(dfeUInt(B));
            DFEVar aLo = a.slice(0, N-B).cast(dfeUInt(N-B));
            DFEVar bLo = b.slice(0, N-B).cast(dfeUInt(N-B));
            return (aHi < bHi) | ((aHi === bHi) & ltPipelined(aLo, bLo));
}
```

Fig. 34 The DFE code for a 224-bit comparison

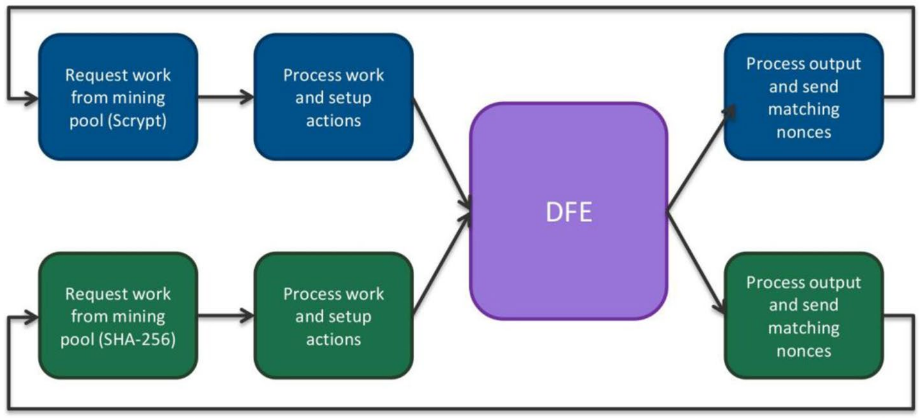
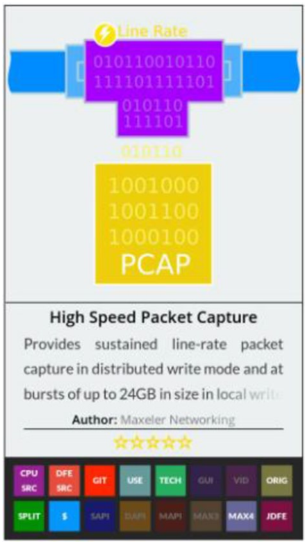

Request work from mining pool (Scrypt)
Process work and setup actions
DFE
Process output and send matching nonces
Request work from mining pool (SHA-256)
Process work and setup actions
Process output and send matching nonces

Fig. 35 Connecting DFEs with relevant blocks of the system

breaking it into a many 16-bit comparisons and chaining the results, thus reducing the energy consumption. Meanwhile, the technology continues improving. The FPGAs don't enjoy a 50×–100× increase in the mining speed, as the transition from CPUs to GPUs; however, a typical 600 MH/s graphics card consumed more than 400 w of power, a typical FPGA mining device would provide a hash-rate of 826 MH/s at only 80 w of power.

High speed packet capture

In March 2015, Maxeler developed a high speed packet capture algorithm by using DFEs to process packets (application icon is given in Fig. 36). Figure 37 depicts the capturing process. Figure 38 depicts the Maxeler CaptureKernel. Figure 39 depicts the high speed packet capture implementation. The Maxeler DFEs offer faster processing, but the communication has to flow to the DFEs and later from DFEs to the CPU. As time passes, network traffic grows exponentially, as well as the need for processing the packets for various purposes, requiring FPGAs to be used in order to increase the speed and reduce the energy consumption. The JDFE's 192 Gb LMEM used as a buffer allows bursts of up to ~20 s of lossless 10Gbps capture from a single port, reducing the energy consumption per Gb of processed packets.



High Speed Packet Capture

Provides sustained line-rate packet capture in distributed write mode and at bursts of up to 24GB in size in local writ

Author: Maxeler Networking

☆☆☆☆☆

Fig. 36 Application icon for classification

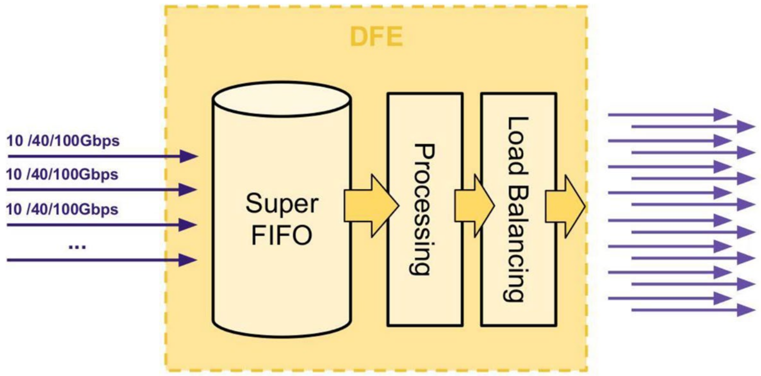


Fig. 37 The high-speed packet capturing process

```
CaptureKernel( KernelParameters parameters, Types types )
{
        super(parameters);
        this.flush.disabled();
        EthernetRXType frameType = new EthernetRXType();
        DFEStruct captureData = types.captureDataType.newInstance(this);
        NonBlockingInput<DFEStruct> frameInput = io.nonBlockingInput("frame", frameType,
                constant.var(true), 1, DelimiterMode.FRAME_LENGTH, 0,
                NonBlockingMode.NO_TRICKLING);
        DFEStruct timestamp = io.input("timestamp", TimeStampFormat.INSTANCE);
        frameInput.valid.simWatch("valid");
        frameInput.data.simWatch("frame");
        timestamp.simWatch("timestamp");
        captureData[CaptureDataType.FRAME] <== frameInput.data;
        captureData[CaptureDataType.TIMESTAMP] <== timestamp;
        io.output("captureData", captureData, captureData.getType(), frameInput.valid);
}
```

Fig. 38 The capture kernel

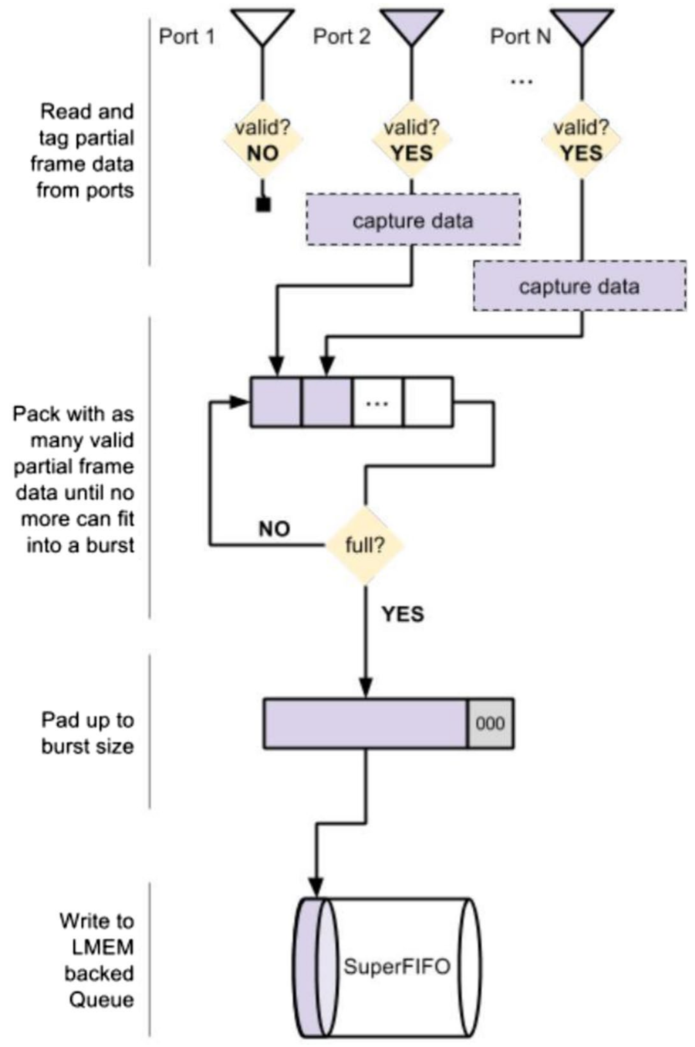
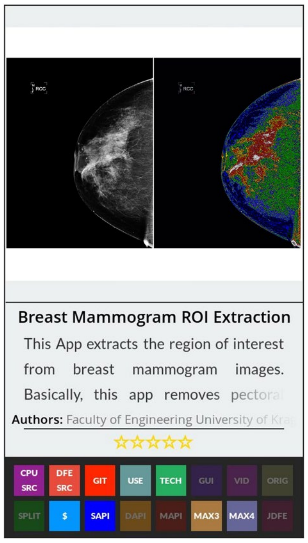

Port 1
Port 2
Port N
...
Read and tag partial frame data from ports
valid? NO
valid? YES
valid? YES
capture data
capture data
Pack with as many valid partial frame data until no more can fit into a burst
...
NO
full?
YES
Pad up to burst size
000
Write to LMEM backed Queue
SuperFIFO

Fig. 39 The packet capturing process using DFEs

Breast mammogram ROI extraction

In April 2015, the University of Kragujevac developed a dataflow implementation of the algorithm that automatically extracts a region of interest from the breast mammogram images (application icon is given in Fig. 40). Figure 41 depicts the breast mammogram ROI extraction process that consists of pectorial muscle removal and background removal. Figure 42 depicts the main part of the Maxeler kernel responsible for the extraction. Figure 43 depicts the Maxeler manager responsible for the process. This DFEs-based approach offers faster signal processing, but the further work may include implementing other algorithms on DFEs that could also detect potential tumour. Breast cancer is more and more often the cause of death than ever before, requiring sophisticated computing techniques to be used in order to prevent the worst case scenario. The experimental results showed that there is a significant speedup in the breast mammogram ROI extraction process using the dataflow approach, around seven times, reducing the energy consumption at the same time.


Breast Mammogram ROI Extraction

This App extracts the region of interest from breast mammogram images. Basically, this app removes pectora

Authors: Faculty of Engineering University of Kra

Fig. 40 Application icon for breast mammogram ROI extraction

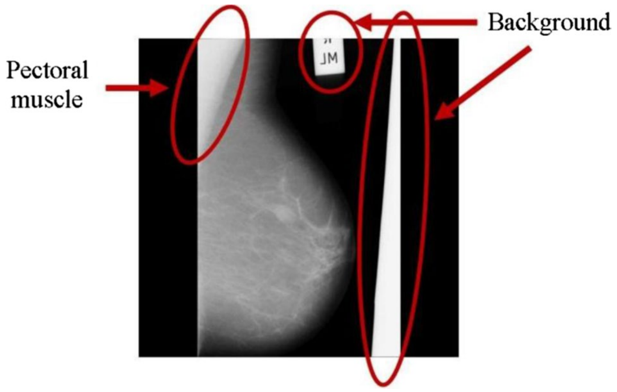


Fig. 41 The breast mammogram ROI extraction process

Linear regression

In December 2014, Maxeler developed a dataflow implementation of the linear regression algorithm by parallelizing calculations using DFEs (application icon is given in Fig. 44). Figure 45 depicts the linear regression algorithm. Figure 46 depicts the main part of the CPU code that could be accelerated by computing in parallel. Figure 47 depicts the process of calculating linear regression coefficients. This DFEs-based approach offers faster calculation even in the case when there is only a single dataset to demonstrate the principles. Linear regression is used for various purposes, including data mining, which is required for processing more and more data. The experimental results showed that there is a significant speedup in the linear regression algorithm using the dataflow approach, even with a small dataset, while the energy consumption is reduced.

```
DFEVar above_pixel = dfeUInt(32).newInstance(this);
DFEVar prev_pixel = stream.offset(image_pixel, -loopLength);
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(height, 1);
DFEVar j = chain.addCounter(width * loopLengthVal, 1);
DFEVar tempWhite = dfeUInt(32).newInstance(this);
DFEVar firstWhite = ((j < loopLengthVal)) ? 0 : tempWhite;
DFEVar result;
result = image_pixel;
result = ((i.eq(0)) & (firstWhite > 1)) ? 0 : result;
result = ((i > 0) & (firstWhite > 1) & (above_pixel < black)) ? 0 : result;
result = ((i <= height/10) & (image_pixel >= threshold)) ? 0 : result;
result = ((i > height/10) & (image_pixel >= threshold) & (above_pixel.eq(0))) ? 0 : result;
above_pixel <== stream.offset(result, -1024*loopLength);
tempWhite <== ((image_pixel >= black) & (prev_pixel < black) & (j >= loopLengthVal)) |
            ((j < loopLengthVal) & (image_pixel >= black)) ?
            stream.offset(firstWhite+1, -loopLength) : stream.offset(firstWhite, -loopLength);
```

Fig. 42 The DFE code for breast mammogram ROI extraction

N-body simulation

In May 2015, Maxeler developed a dataflow implementation of the algorithm that simulates effects on each body in the space on every other body (application icon is given in Fig. 48). Figure 49 depicts the amount of calculations of the n-body algorithm. Figure 50 depicts the main part of the n-body implementation on the DFE. Figure 51 depicts the simplified kernel graph. While the DFEs-based approach offers faster processing and higher memory bandwidth, the input data has to be reordered in order to achieve better performance. This algorithm is used in various fields, and the problem sizes keep growing. The experimental results showed that there is a significant speedup near to thirty times, with a trend of increasing with increasing the input data size.

Impact

The major assumption behind the strategic decision to develop the AppGallery.Maxeler.com is to help the community to understand and to adopt the dataflow approach. The best way to achieve this goal is to provide developers with access to many examples that are properly described and that clearly demonstrate benefits. Of course, the academic community is the one that most readily accepts new ways of thinking and is the primary target here.

The impact of the AppGallery.Maxeler.com is best judged by the rate at which the PhD students all over the world have started submitting their own contributions, after they had undergone a proper education, and after they were able to study from the examples generated with the direct involvement of Maxeler. The education of students is now based on the MaxelerWebIDE (described in a follow-up paper) and on the Maxeler Application Gallery. One prominent example is the Computing in space with the OpenSPL course at Imperial College taught at the Fall of 2014 for the first time (http://cc.doc.ic.ac.uk/openspl14/).

Another important contribution of the Maxeler Application Gallery project is to make the community understand that the essence of the term "optimization" in mathematics

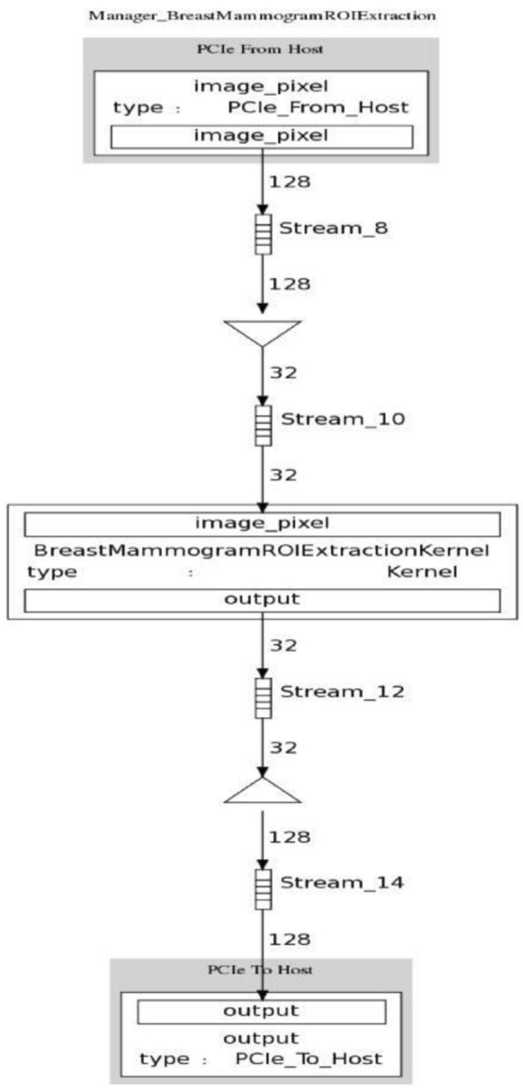
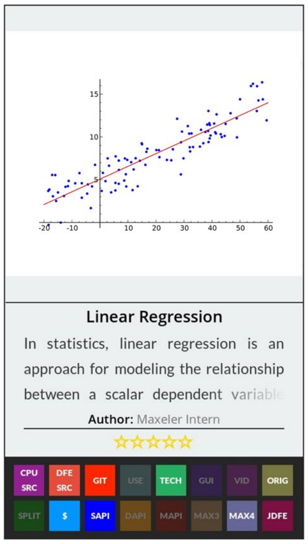


Fig. 43 The Maxeler manager responsible for breast mammogram ROI extraction

has changed. Before, the goal of the optimization was to minimize the number of iterations till convergence (e.g., minimizing the number of sequential steps involved) and to minimize the time for one loop iteration (e.g., eliminating time consuming arithmetic operations, or decreasing their number). This was so because, as indicated before, the ratio of t(ALU) over t(COMM) was very large.

Now that the ratio of t(ALU) over t(COMM) is becoming very small, the goal of optimization is to minimize the physical lengths of edges in the execution graph, which is a challenge (from the mathematics viewpoint) of a different nature. Another challenge is also to create execution graph topologies that map onto FPGA topologies with minimal deteriorations. This issue is visible from most of the Maxeler Application Gallery examples.



Linear Regression

In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable

Author: Maxeler Intern

Fig. 44 Application icon for linear regression

$$\min_{a,b} \sum_{i=1}^{n} (y_i - a - b \cdot x_i)^2$$

$$b = \frac{\sum_{i=1}^{n} (x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n} (x_i - \overline{x})^2} = \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n} \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right)^2}$$

$$a = \overline{y} - b \cdot \overline{x} = \frac{1}{n} \left(\sum_{i=1}^{n} y_i - b \sum_{i=1}^{n} x_i \right)$$

Fig. 45 The linear regression algorithm

```
void linearRegression(int dataPoints, float *x, float *y, float *a, float *b){
   int i;
   float sumX = 0;    // sum of x
   float sumX2 = 0;   // sum of x^2
   float sumXY = 0;   // sum of x*y
   float sumY = 0;    // sum of y

      for(i = 0; i < dataPoints; i++){
         sumX  +=  x[i];          //compute sums
         sumX2 +=  x[i]*x[i];
         sumXY +=  x[i]*y[i];
         sumY  +=  y[i];

         meanX = sumX / (i+1);   //compute mean values for x and y
         meanY = sumY / (i+1);

         b[i] = (sumXY - sumX * meanY) / (sumX2 - sumX * meanX); //compute b (slope)
         a[i] = meanY - b[i] * meanX;                            //compute a (intercept)
      }
}
```

Compute regression in a continuous fashion, i.e. compute a new *a*, *b* value pair for each added *x*, *y* data point.

Main computation in this loop, accelerated with DFE kernel.

Fig. 46 The possibility for acceleration

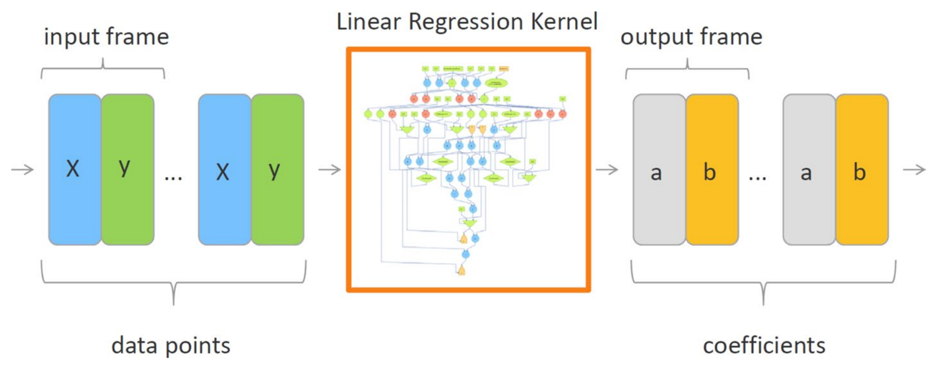
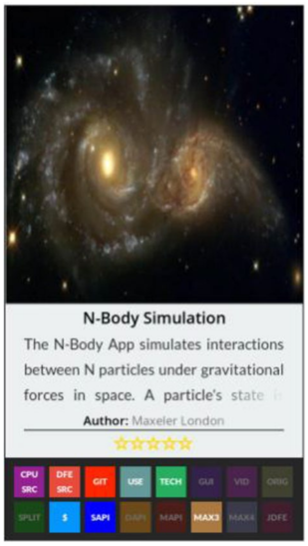


Fig. 47 The process of calculating linear regression coefficients



N-Body Simulation

The N-Body App simulates interactions between N particles under gravitational forces in space. A particle's state

Author: Maxeler London

Fig. 48 Application icon for classification

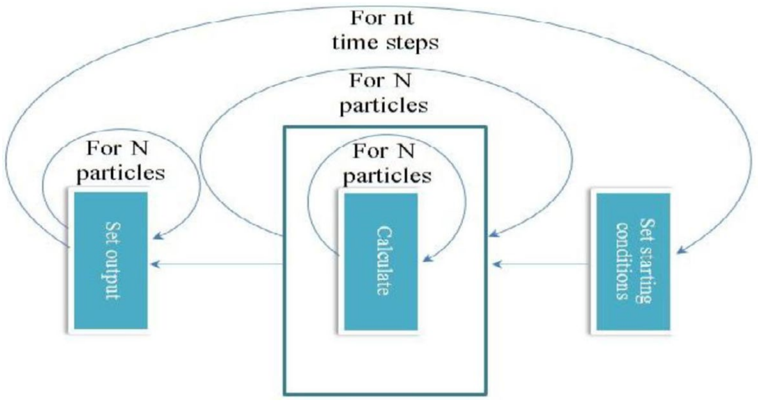


Fig. 49 The n-body algorithm

Finally, the existence of Maxeler Application Gallery enables different machines to be

```
...
// all the below are interleaved data streams
DFEVar rx = pjX - piX;
DFEVar ry = pjY - piY;
DFEVar rz = pjZ - piZ;
DFEVar dd = rx*rx + ry*ry + rz*rz + scalars.EPS;
DFEVar d = 1 / (dd * KernelMath.sqrt(dd));
DFEVar s = pjM * d;
DFEParLoop lp = new DFEParLoop (this, “lp”);
lp.set_inputs(3, dfeFloat(8,24), 0.0);
DFEVar accX = lp.feedback[0] + rx*s;
DFEVar accY = lp.feedback[1] + ry*s;
DFEVar accZ = lp.feedback[2] + rz*s;
lp.set_outputs(accX, accY, accZ);
...
```

Fig. 50 The DFE code of the n-body algorithm

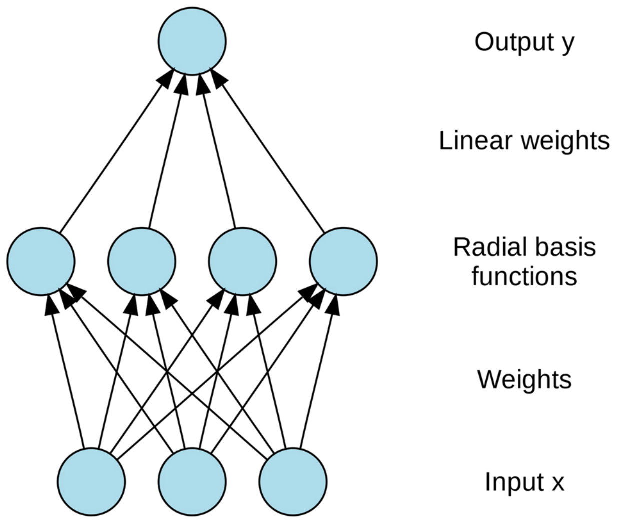


Fig. 51 The dataflow graph of the n-body algorithm

compared, as well as different algorithms for the same application to be studied comparatively. Such a possibility creates an excellent ground for effective scientific work.

Conclusions

This paper presented the Maxeler Application Gallery project, its vision and mission, as well as a selected number of examples. All of the examples were presented using a uniform template, which enables an easy comprehension of the new dataflow paradigm underneath.

This work, and the Maxeler Application Gallery it presents, could be used in education, in research, in development of new applications, and in demonstrating the advantages of the new dataflow paradigm. Combined with the MaxelerWebIDE, the Maxeler

Application Gallery creates a number of synergistic effects (the most important of which is related the possibility to learn by trying ones own ideas).

Future research directions include more sophisticated intuitive and graphical tools that enable easy development of new applications, as well as their easy incorporation into the Maxeler Application Gallery presentation structure. In addition, significant improvement of the debugging tools is underway.

Abbreviations

NPAA: new paradigm acceptance acceleration; FPGA: field-programmable gate array; NIH: not invented here; 2G2BT: too good to be true; AOC: afraid of change; ALU: arithmetic logic unit; COMM: latencies over the communication; PCIe: peripheral component interconnect express; DFEs: dataflow engines; 1U: the minimal size in rack-mount servers; CPU: central processing unit; ROM: read only memory; FFT: Fast Fourier transform; CUDA: compute unified device architecture; JDFE: Juniper dataflow engine; LMEM: Maxeler DRAM memory; ROI: region of interest.

Authors' contributions

NT has developed website http://appgallery.maxeler.com/#/. He has also developed some of the presented applications. VM coordinated the work and has written all chapters up to 3.1.1, as well as impact and conclusions sections. NK described presented applications. GG participated in organization of the AppGallery project and applications available on the web site. All authors read and approved the final manuscript.

Authors' information

Nemanja Trifunovic is with the Maxeler, London, UK. He has diploma from the School of Electrical Engineering, University of Belgrade, Serbia. He is within the 1 % of fastest studying students at the faculty. He was an intern in Google, NY, USA, Microsoft Development Center, Serbia, as well as Maxeler, London, UK.
Prof. Veljko Milutinovic, Fellow of the IEEE, received his PhD from the University of Belgrade, spent about a decade on various faculty positions in the USA (mostly at Purdue University), and was a codesigner of the DARPAs first GaAs RISC microprocessor. Now he teaches and conducts research at the University of Belgrade, in EE, MATH, and PHY/CHEM. His research is mostly in datamining algorithms and dataflow computing, with the stress on mapping of data analytics algorithms onto fast energy efficient architectures. For 7 of his books, forewords were written by 7 different Nobel Laureates with whom he cooperated on his past industry sponsored projects. He has over 60 IEEE or ACM journal papers, and about 4000 Google Scholar citations (including all misspellings of his name). He is a member of Academia Europea. He has taught DataFlow courses at Purdue, Indiana, Harvard, and MIT.
Nenad Korolija is with the faculty of the School of Electrical Engineering, University of Belgrade, Serbia. He received an Msc degree in electrical engineering and computer science in 2009. He was a teaching assistant at Software project management and Business communication: practical course. His interests and experience include developing software for high performance computer architectures and dataflow architectures. During 2008, he worked on HIPEAC FP7 project at the University of Siena, Italy. In 2013, he was an intern at the Google Inc., Mountain View, California, USA.
Georgi Gaydadjiev is a VP of Maxeler Technologies and a professor of the Chalmers University of Technology in Sweden. He is a Senior IEEE and ACM member. He is a general and program chair of many IEEE conferences. His awards include: ACM/SIGARCH 24th International Conference on Supercomputing (ICS'10) Tsukuba, Japan, June 2010 best paper award and Ramon y Cayal incorporaci.

Author details

[1] School of Electrical Engineering, University of Belgrade, Bulevar Kralja Aleksandra 73, 11120 Belgrade, Serbia. [2] Maxeler Technologies Ltd, 3-4 Albion Place, London W6 0QT, UK.

Competing interests

The authors declare that they have no competing interests.

Received: 11 November 2015 Accepted: 22 December 2015
Published online: 18 February 2016

References

1. Trifunovic N, et al. The MaxGallery Project. Advances in computers, vol 104. Springer. 2016.
2. Milutinovic V. Trading latency and performance: a new algorithm for adaptive equalization. IEEE Transactions on Communications. 1985.
3. Milutinovic V. Splitting temporal and spatial computing: enabling a combinational dataflow in hardware. Italy: The ISCA ACM Tutorial on Advances in SuperComputing, Santa Margherita Ligure; 1995.
4. Mencer O, Gaydadjiev G, Flynn M. OpenSPL: the Maxeler programming model for programming in space. UK: Maxeler Technologies; 2012.
5. Flynn M, Mencer O, Milutinovic V, Rakocevic G, Stenstrom P, Valero M, Trobec R. Moving from PetaFlops to PetaData. communications of the ACM. 2013. pp. 39–43.
6. Milutinovic V. The dataflow processing. Amsterdam: Elsevier; 2015.
7. Milutinovic V, Trifunovic N, Salom J, Giorgi R. The guide to dataflow supercomputing. USA: Springer; 2015.

8. The Maxeler dataflow supercomputing. The IBM TJ Watson Lecture Series. 2015.
9. The Maxeler dataflow supercomputing: guest editor introduction. Advances in computers, vol 104. Springer. 2016.
10. Vassiliadis S, Wong S, Gaydadjiev G, Bertels K, Kuzmanov G. The molen polymorphic processor. IEEE Trans Comput. 2004;53(11):1363–75.
11. Dou Y, Vassiliadis S, Kuzmanov GK, Gaydadjiev GN. 64-bit floating-point FPGA matrix multiplication. In: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays. 2005. pp. 86–95.
12. van de Goor AJ, Gaydadjiev GN, Mikitjuk VG, Yarmolik VN, March LR. A test for realistic linked faults. VLSI Test Symposium, 1996. In: Proceedings of 14th. 1996. pp. 272–280.
13. Miranker V. Space-time representations of computational structures. IEEE Comput. 1984; 32.
14. Kuhn P. Transforming algorithms for single-stage and VLSI architectures. In: Proceedings of Workshop on Interconnection Networks for Parallel and Distributed Processing. 1980.
15. Oriato D, Girdlestone S, Mencer O. Dataflow computing in extreme performance conditions. In: Hurson A, Milutinovic V (eds) Dataflow processing. Advances in computers, vol 96. Waltham: Academic Press; 2015. p. 105–138.
16. Stojanović S, Bojić D, Bojović M. An overview of selected heterogeneous and reconfigurable architectures. In: Hurson A, Milutinovic V (eds) Dataflow processing. Advances in computers, vol 96. Waltham: Academic Press; 2015. p. 1–45.