

Numerical mathematics on FPGAs using CλaSH

Comparison to conventional solution methods

Martijn Bakker

Committee:

Dr.ir. J. Kuper

Dr. R.M.J. van Damme

Dr.ir. J. Broenink

Computer Architecture for Embedded Systems Electrical Engineering,
Mathematics and Computer Science (EEMCS)

University of Twente

This thesis is submitted for the degree of

Bachelor of Science

Advanced Technology

July 2015

Abstract

This is where you write your abstract ...

Acknowledgements

Christiaan Baaij - making CλaSH

Table of contents

1	Introduction	1
1.1	Project goals	1
1.2	FPGAs	1
1.2.1	What is an FPGA?	1
1.2.2	How does it work and what are common usages?	2
1.2.3	System-on-a-chip	2
1.3	Solver theory	2
1.4	Functional programming	2
1.4.1	What is functional programming?	2
1.4.2	Using FP for numerical mathematics	2
1.4.3	Example: Numerical solutions of ODEs in Haskell	3
1.5	Data transfer from and to the FPGA	5
1.6	CλaSH	5
1.6.1	Designing hardware in a functional way	5
1.6.2	Mealy machines	5
2	Results	7
2.1	Euler's method	7
2.1.1	Performance	7
2.1.2	Accuracy	7
2.2	Second order Runge-Kutta	7
2.2.1	Performance	7
2.2.2	Accuracy	7
2.3	4th order Runge-Kutta	7
2.3.1	Performance	7
2.3.2	Accuracy	7
2.4	Comparison with CPU implementations	7
2.5	Comparison with GPU implementations	7
3	Conclusion	9
4	Discussion	11
4.1	sdfasdf	11

References	13
Appendix A Haskell source code for numerical solutions of ODEs	15
Appendix B Installing the CUED class file	21
Appendix C Installing the CUED class file	23

Acknowledgements

first row
second row
and so on

Chapter 1

Introduction

1.1 Project goals

From the start, this project has had two main goals: firstly, obtaining information on the feasibility and the advantages and disadvantages of performing numerical mathematics directly on (programmable) hardware, the FPGA. Secondly: figuring out whether higher-order functions are of much use for numerical mathematics. As per usual, having main goals spawns off several minor goals which support the main parts. Both supporting goals are about simplifying the process of configuring FPGAs: an easy way of setting up projects with complicated IO requirements and furthermore, developing a tool-chain integration which turns the long process of compiling and deploying your FPGA project into the execution of a single command.

Alongside these concrete goals the underlying theme is to do as much work as possible in C λ aSH, a language developed by Christiaan Baaij at the CAES group at the University of Twente. Further elaboration on C λ aSH can be found in section 1.6.

1.2 FPGAs

1.2.1 What is an FPGA?

An FPGA (Field Programmable Gate Array) is a chip in which you can specify the hardware yourself. In contrast to regular programming in which you generate a long list of instructions which are executed sequentially on a fixed chip configuration, the FPGA allows you to specify exactly which wire (signal) leads where and what operation should be applied to that signal. This approach to programming can have several advantages. The first one arises from the large opportunities for parallelism. Every part of the FPGA can be executing a meaningful computation simultaneously, whereas processors are bound by the amount of physical cores they have in the amount of truly concurrent instruction executions possible. Secondly, a conventional processor only has a fixed instruction set. Using an FPGA you can define your own instructions (subcircuits), again providing a possible improvement in computational speed. When comparing FPGA

1.2.2 How does it work and what are common usages?

An FPGA is built up from several distinct element types:

1. Logic blocks
2. Programmable interconnects
3. IO blocks

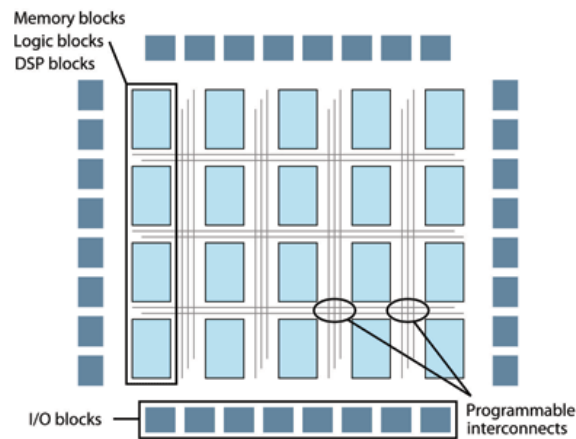


Fig. 1.1 FPGA fabric.

1.2.3 System-on-a-chip

1.3 Solver theory

Euler

Take derivative, multiply with timestep, add to initial value, repeat until end.

Runge-Kutta methods (second order)

Take derivatives at some time points, multiply with proper coefficients, add all to initial value, repeat until end.

Stability of integration schemes

1.4 Functional programming

1.4.1 What is functional programming?

1.4.2 Using FP for numerical mathematics

Functional languages have several properties which make them suitable for the purpose of solving problems in numerical mathematics. First and foremost, Haskell, being based on λ -

calculus is very close to mathematics. The useful mathematical properties here are *referential transparency*, easy *partial function application* and being a *declarative language*. Referential transparency implies that a variable only has a constant value which is the same everywhere in the program. This prevents that changing a variable might have influence on another computation as a side effect and it corresponds to mathematical notation. For instance, in an imperative programming like C you could write $i = i + 1$, which is a mathematical impossibility and therefore not allowed in Haskell. Partial function application is another very useful concept. Often in numerical mathematics, you want to create or process a function. You need a function that has another function as return value. For instance, take a function which requires two arguments. After only applying a single argument, the object returned still needs the second argument in order to compute the final value. This is exactly according to the definition of a function: An object that still needs arguments before being able to return its final value. Being a *declarative language* means that you write code that specifies what you want to accomplish, not how to get there. This concept is again borrowed from mathematics. You put in a set of function definitions and Haskell will figure out how to actually compute the value you request according to those definitions. This property of declarativity also has the result that Haskell is a terse language whilst remaining easy to understand. Secondly, Haskell has a very strong type system. The type system has three main advantages. It becomes very easy to swap out and replace functions as long as you make sure that the types are the same. The Haskell compiler will start to assert errors immediately whenever you feed it something which does not make sense or could be ambiguous which is very useful when writing programs. By having a look at the types of a Haskell program it becomes very straightforward to see what the program does and how it works, which is very useful when attempting to understand your own or someone else's code. Lastly, a property which is often very important for numerical mathematics: Haskell is fast. According to the *Computer Language Benchmarks Game* [1], Haskell is almost on par with Java and Fortran but significantly faster than Python and Matlab (not shown), two languages which are often used for numerical mathematics nowadays. There is still a performance gap of around a factor 3 between Haskell and C (the reference), hence if speed is of the absolute highest concern C is still a valid option.

1.4.3 Example: Numerical solutions of ODEs in Haskell

As mentioned before, the types in Haskell reveal lots of information about the structure and functionality of the program. The three main types constituting the numerical solver for ordinary differential equations are listed above.

Listing 1.1 Main types for the ODE solver

```
1  type Equation = ODEState -> D_ODEState
2  type Scheme   = TimeSettings -> Equation -> ODEState -> ODEState
3  type Solver    = Scheme -> TimeSettings -> Equation -> ODEState -> [ODEState]
```

Equation

In essence, a differential equation is a mapping (function) from a certain state of the system to the change of this system. This is also what the type signature of `Equation` signifies, a mapping from an `ODEState` to a `D_ODEState`. This generic set up allows the specification of any ODE for solving. The implementation in pure Haskell of a simple ODE is given in listing 1.2, which corresponds to the equation $x' = -x$. However, this representation is not very elegant and a lot of the code is performing unboxing of the types. Using property that this equation is linear, it is possible to use an utility method which takes as input a matrix and returns the Haskell differential equation function belonging to that matrix. The same can be done for heterogeneous linear systems using a different utility function, which does not only takes a matrix as input but also a list of functions representing the heterogeneous part of the equation. The example code for this can be seen in appendix A.

Listing 1.2 Example equation for exponential decay

```

1  eq_exponential :: Equation
2  eq_exponential state = [-x !! 0]
3  where
4    x = xs state
```

SolveMethod

The `SolveMethod` performs the actual computations on what the next value of the solution should be: the integration scheme. In order to obtain this next state, the scheme needs three input values: It needs information on the timing constraints of the solution, in this case it needs the time step. Furthermore, it needs the equation itself and it requires the state of the system at t_n in order to be able to determine the state of the system at $t_{n+1} = t_n + \Delta t$.

The most straightforward integration scheme is called forward Euler, given in equation 1.1. Listing 1.3 depicts the translation of the mathematical expression 1.1 to Haskell. Even though some list operations have been inserted (**zipWith** and **map**), the structure is still recognizable. It computes the change in state, multiplies this with the time step obtained in line 6 and adds the initial state in line 4. Lastly, the integration scheme returns the new state of the equation, consisting of a list of x-values and a corresponding time value. Implementations of different solvers (eg. 4th order Runge-Kutta) can be found in appendix A.

Listing 1.3 Example code for the forward Euler integration scheme

```

1  euler :: Scheme
2  euler time equation initState = ODEState newX newT
3  where
4    newX    = zipWith (+) (xs initState) dX
5    dX      = map (timestep *) (equation initState)
6    newT    = (t initState) + timestep
7    timestep = dt time
```

$$\vec{x}(t + \Delta t) \approx \vec{x}(t) + \frac{d\vec{x}(t)}{dt} \Delta t \quad (1.1)$$

Solver

The Solver function in listing 1.4 acts as the main interface to the program. You specify a SolveMethod, the TimeSettings (containing the time step and the time at which to stop solving), the equation itself and an initial condition. The Solver will then return a list of states of the system. As is very common in functional programming, the Solver has been defined recursively. Line 4 is where the magic happens: the solution list is defined to be the initial condition, followed by the solution list with the new state (computed by the integration scheme on line 6) as initial condition. Additionally, there is a comparison in line 7 which ends the recursion whenever the time of the solution exceeds the maximum time value, set in the TimeSettings.

The solutions of a wide range of equations, both linear and non-linear, both homogeneous and heterogeneous and using the input matrix utility functions have been plot with suitable initial conditions to show their behavior in figure 1.2.

Listing 1.4 The main controlling function

```

1  solve :: Solver
2  solve solvemethod time equation initState
3    | end = []
4    | otherwise = initState : solve solvemethod time equation newState
5  where
6    newState = solvemethod time equation initState
7    end = (t initState) > (tMax time)

```

$$\text{Exponential} \quad x(t)' = -x(t) \quad (1.2)$$

$$\text{Simple harmonic} \quad x(t)'' = -x(t) \quad (1.3)$$

$$\text{Cosine hyperbolic} \quad x(t)' = \frac{\sqrt{x(t)^2 - a^2}}{a} \quad (1.4)$$

$$\text{Simple harmonic} \quad \vec{x}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{x}(t) \quad (1.5)$$

$$\text{Simple forced harmonic} \quad \vec{x}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{x}(t) + \begin{bmatrix} \sin(t) \\ e^{-t} \end{bmatrix} \quad (1.6)$$

1.5 Data transfer from and to the FPGA

1.6 CλaSH

1.6.1 Designing hardware in a functional way

1.6.2 Mealy machines

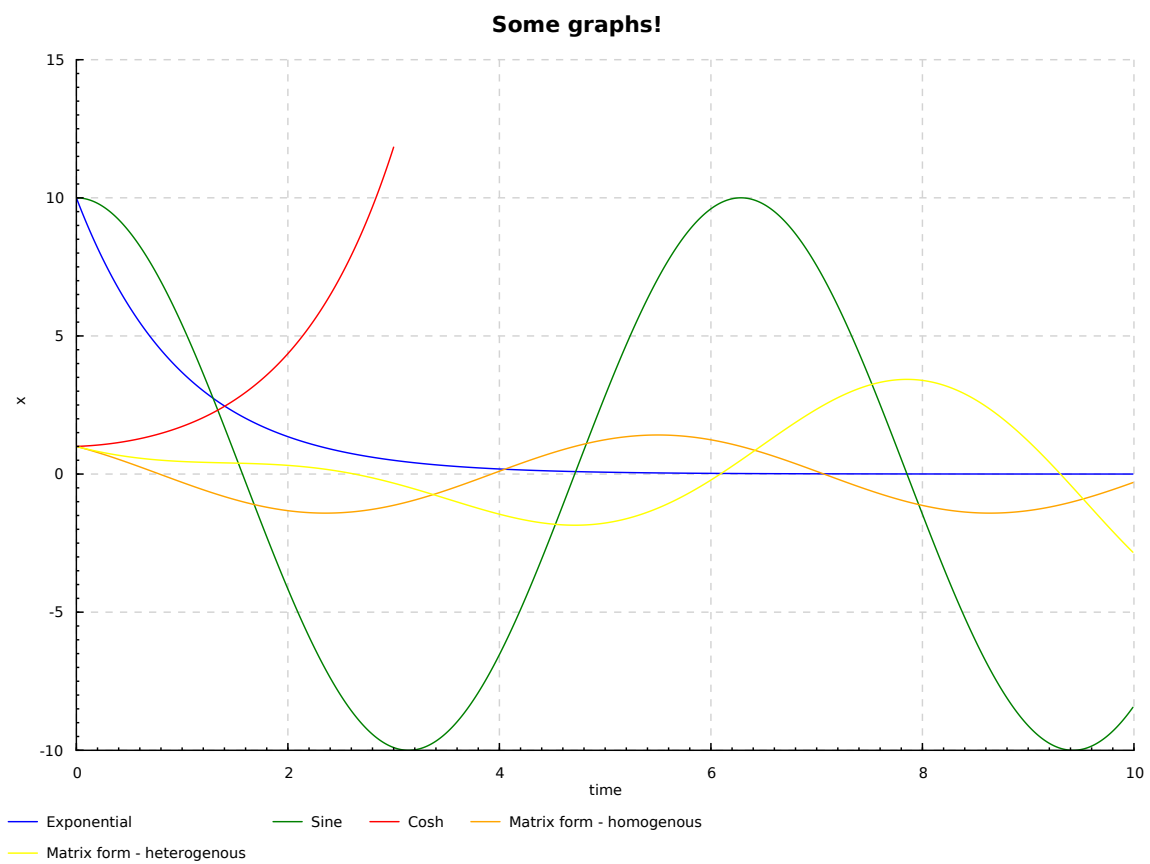


Fig. 1.2 Graphs

Chapter 2

Results

2.1 Euler's method

2.1.1 Performance

2.1.2 Accuracy

2.2 Second order Runge-Kutta

2.2.1 Performance

2.2.2 Accuracy

2.3 4th order Runge-Kutta

2.3.1 Performance

2.3.2 Accuracy

2.4 Comparison with CPU implementations

2.5 Comparison with GPU implementations

Chapter 3

Conclusion

Chapter 4

Discussion

4.1 sdfasdf

References

- [1] Computer language benchmarks game. <http://benchmarksgame.alioth.debian.org/u32q/which-programs-are-fastest.html>, 2015. [Accessed:].

Appendix A

Haskell source code for numerical solutions of ODEs

Listing A.1 Solver.hs

```
1
2 module Solver where
3
4 import Prelude
5 import SolverTypes
6
7 import SolverEquations
8 import SolverSolvers
9 import SolverPlotter
10 import SolverPresets
11
12 ----- CALLERS
13 -- general form, stop after a certain time
14 solve :: Solver
15 solve solvemethod time equation initState
16   | end = []
17   | otherwise = initState : solve solvemethod time equation newState
18 where
19   newState = solvemethod time equation initState
20   end = (t initState) > (tMax time)
21
22 sol_start = solve rk4 initTimeSettings
23
24 solution_expo = sol_start eq_exponential initODEState
25 solution_sine = sol_start eq_sine initODEState
26 solution_cosh = solve rk4 initTimeSettings2 eq_cosh initODEState2
27 solution_homo = sol_start (eq_linear_homo_const sinematrix) initODEState2
28 solution_hetr = sol_start (eq_linear_hetr_const sinematrix funcvec) initODEState2
29
30
31 testPlot = plotSolutions [s1,s2,s3,s4,s5] "Some graphs!"
32 where
33   s1 = (solution_expo, "Exponential")
```

```

34     s2 = ( solution_sine , "Sine")
35     s3 = ( solution_cosh , "Cosh")
36     s4 = ( solution_homo , "Matrix form – homogenous")
37     s5 = ( solution_hetr , "Matrix form – heterogenous")

```

Listing A.2 SolverTypes.hs

```

1  module SolverTypes where
2
3  import Prelude
4
5  type NumRepr = Float
6  type D_ODEState = [NumRepr]
7
8  data ODEState = ODEState { xs :: [NumRepr]
9    , t :: NumRepr
10 } deriving (Show)
11
12 data TimeSettings = TimeSettings { dt :: NumRepr
13   , tMax :: NumRepr
14 } deriving (Show)
15
16 type SubFunction = (NumRepr -> NumRepr)
17
18 type Equation = ODEState -> D_ODEState
19 type Scheme   = TimeSettings -> Equation -> ODEState -> ODEState
20 type Solver   = Scheme -> TimeSettings -> Equation -> ODEState -> [ODEState]

```

Listing A.3 SolverEquations.hs

```

1  module SolverEquations where
2
3  import Prelude
4  import SolverTypes
5  import SolverPresets
6
7  -- Exponential :  $y = A * \exp(-t)$ 
8  --  $y' = -y$ 
9
10 --  $x0' = -x0$ 
11 eq_exponential :: Equation
12 eq_exponential state = [-x !! 0]
13   where
14     x = xs state
15
16
17
18 -- Sine :  $y = A * \sin(\omega * t)$ 
19 --  $y'' = -y$ 
20
21 --  $x0' = x1$ 
22 --  $x1' = -x0$ 
23 eq_sine :: Equation

```

```

24 eq_sine state    = [x0,x1]
25   where
26     x = xs state
27     x0 = x !! 1
28     x1 = - (x !! 0)
29
30
31
32 -- Hyperbolic cosine :  $y = a \cosh((x - b)/a)$ 
33 --  $y' = \sqrt{y^2 - a^2}/a$ 
34
35 --  $x0' = \sqrt{x0^2 - a^2}/a$ 
36 eq_cosh :: Equation
37 eq_cosh state = [x0]
38   where
39     x = xs state
40     x0 = sqrt((x !! 0)^2 - a^2)/a
41     a = 0.99
42
43 -- Arbitrary homogenous system
44 --  $y' = Ay$ 
45 eq_linear_homo_const :: [[NumRepr]] -> ODEState -> D_ODEState
46 eq_linear_homo_const matrix state = map (rowmult y) matrix
47   where
48     y = xs state
49
50 rowmult :: [NumRepr] -> [NumRepr] -> NumRepr
51 rowmult vec1 vec2 = sum $ zipWith (*) vec1 vec2
52
53 -- Arbitrary heterogenous system
54 --  $y' = Ay + F$ 
55 eq_linear_hetr_const :: [[NumRepr]] -> [SubFunction] -> ODEState -> D_ODEState
56 eq_linear_hetr_const matrix vector state = zipWith (+) (map (rowmult y) matrix) (map ($time) vector)
57   where
58     y = xs state
59     time = t state

```

Listing A.4 SolverPresets.hs

```

1 module SolverPresets where
2
3 import Prelude
4 import SolverTypes
5
6 unity :: [[NumRepr]]
7 unity = [[1,0,0],[0,1,0],[0,0,1]]
8
9 sinematrix :: [[NumRepr]]
10 sinematrix = [[0,1],[-1,0]];
11
12 vec :: [NumRepr]
13 vec = [4,3,2]
14

```

```

15 funcvec :: [SubFunction]
16 funcvec = [(\ t -> sin t ),(\ t -> exp (-t))]
17
18 initODEState = ODEState [10, 0.0] 0.0
19 initODEState2 = ODEState [1, -1] 0.0
20
21 initTimeSettings = TimeSettings 0.01 10
22 initTimeSettings2 = TimeSettings 0.01 3

```

Listing A.5 SolverHelper.hs

```

1 module SolverHelper where
2
3 import Prelude
4 import SolverTypes
5
6 sumLists :: [[NumRepr]] -> [NumRepr] -> [NumRepr]
7 sumLists [] factors      = []
8 sumLists (xs :[]) factors = map ((head factors)*) xs
9 sumLists (xs:xss) factors = zipWith (+) (map (head factors*) xs) (sumLists xss (tail factors))

```

Listing A.6 SolverPlotter.hs

```

1 module SolverPlotter where
2
3 import Prelude
4 import SolverTypes
5 import GHC.Float
6
7 import Graphics.Rendering.Chart.Easy
8 import Graphics.Rendering.Chart.Backend.Cairo
9
10 outProps = fo_format .~ PDF $ def
11
12 plotSolutions :: [(ODEState, String)] -> String -> String -> IO()
13 plotSolutions solutions title filename = toFile outProps filename $ do
14   layout_title .:= title
15   layout_x_axis . laxis_title .:= "time"
16   layout_y_axis . laxis_title .:= "x"
17   plotSolutions_help solutions
18
19
20 plotSolutions_help []      = error "empty list "
21 plotSolutions_help [sol]  = plotSolution sol
22 plotSolutions_help (sol : sols) = do
23   plotSolution sol
24   plotSolutions_help sols
25
26
27 plotSolution (solution, curveTitle) = plot $ line curveTitle [states]
28 where
29   states = reformData solution
30

```

```
31
32 reformData :: [ODEState] -> [(Double, Double)]
33 reformData states = map reformState states
34
35 reformState :: ODEState -> (Double, Double)
36 reformState state = (float2Double tVal, float2Double (x !! 0))
37   where
38     x = xs state
39     tVal = t state
```


Appendix B

Installing the CUED class file

\LaTeX .cls files can be accessed system-wide when they are placed in the `<texmf>/tex/latex` directory, where `<texmf>` is the root directory of the user's \TeX installation. On systems that have a local `texmf` tree (`<texmflocal>`), which may be named “`texmf-local`” or “`localtexmf`”, it may be advisable to install packages in `<texmflocal>`, rather than `<texmf>` as the contents of the former, unlike that of the latter, are preserved after the \LaTeX system is reinstalled and/or upgraded.

It is recommended that the user create a subdirectory `<texmf>/tex/latex/CUED` for all CUED related \LaTeX class and package files. On some \LaTeX systems, the directory look-up tables will need to be refreshed after making additions or deletions to the system files. For \TeX Live systems this is accomplished via executing “`texhash`” as root. \TeX users can run “`initexmf -u`” to accomplish the same thing.

Users not willing or able to install the files system-wide can install them in their personal directories, but will then have to provide the path (full or relative) in addition to the filename when referring to them in \LaTeX .

Appendix C

Installing the CUED class file

\LaTeX .cls files can be accessed system-wide when they are placed in the `<texmf>/tex/latex` directory, where `<texmf>` is the root directory of the user's \TeX installation. On systems that have a local `texmf` tree (`<texmflocal>`), which may be named “`texmf-local`” or “`localtexmf`”, it may be advisable to install packages in `<texmflocal>`, rather than `<texmf>` as the contents of the former, unlike that of the latter, are preserved after the \LaTeX system is reinstalled and/or upgraded.

It is recommended that the user create a subdirectory `<texmf>/tex/latex/CUED` for all CUED related \LaTeX class and package files. On some \LaTeX systems, the directory look-up tables will need to be refreshed after making additions or deletions to the system files. For \TeX Live systems this is accomplished via executing “`texhash`” as root. \TeX users can run “`initexmf -u`” to accomplish the same thing.

Users not willing or able to install the files system-wide can install them in their personal directories, but will then have to provide the path (full or relative) in addition to the filename when referring to them in \LaTeX .

