Performance Optimisation Strategies for Automatically Generated FPGA Accelerators for Biomedical Models

Ting Yu*†, Julian Oppermann‡, Chris Bradley†, Oliver Sinnen*
*Department of Electrical and Computer Engineering, University of Auckland
†Auckland Bioengineering Institute, University of Auckland
‡Embedded Systems and Applications Group (ESA), Technische Universitat Darmstadt, Germany
{ting.yu, c.bradley, o.sinnen}@auckland.ac.nz
{oppermann}@esa.tu-darmstadt.de

Abstract

Biomedical modelling that is mathematically described by Ordinary Differential Equations (ODEs), is often one of the most computationally intensive parts of simulations. With high inherent parallelism, hardware acceleration based on FPGAs has great potential to increase the computational performance of the ODE model integration, while being very power-efficient. ODoST, ODE-based Domain-specific Synthesis Tool, is a tool we proposed previously to automatically generate the complete hardware/software co-design framework for computing biomedical models based on CellML. Although it provides remarkable performance improvement and high energy efficiency compared to CPUs and GPUs, there is still a great potential for optimisation. In this paper we investigate a set of optimisation strategies including compiler optimisation, resource fitting and balancing, and multiple pipelines. They all have in common that they can be done automatically, hence can be integrated in our domain specific high level synthesis tool. We evaluate the optimised Hardware Accelerator Modules (HAMs) generated by ODoST on real hardware based on their resource usage, processing speed and power consumption. The results are compared with single threaded and multi-core CPUs with/without SSE optimisation and a graphics card. The results show that the proposed optimisation strategies provide significant performance improvement and result in even more energy efficient HAMs. Furthermore, the resources of the target FPGA device can be more efficiently utilised in order to fit larger biomedical models than before.

I. Introduction

CellML [1] is an open standard mark-up language based on XML that defines custom biomedical models based on differential algebraic equations (DAEs). CellML is used by a variety of tools to describe the DAE models, e.g., OpenCMISS, a general purpose computational library for solving field based equations with an emphasis on biomedical applications [2]. Such biomedical modelling often uses numerical integration of ODEs to simulate dynamic biomedical systems in order for researchers to understand different physiological functions. To simulate a model with a fine mesh size and running for millions of time steps, a significant amount of computation is required which can be very time consuming even with today's fastest CPUs [3].

In our previous study [4], we have designed a hardware accelerator with an FPGA-CPU heterogeneous architecture for biomedical models described by CellML. We call it HAM, which stands for Hardware Accelerator Module. The HAM provides an acceleration approach focusing on reconfigurable hardware aimed at high performance with reduced energy consumption. To reduce the effort in implementing accelerators from biomedical models, we have designed and implemented an ODE-based Domain-specific Synthesis Tool (ODoST) to automatically create the HAM accelerator framework from a CellML input [5]. We have evaluated three CellML models with diverse complexity. The results show that FPGAs can provide a highly energy efficient solution and remarkable processing performance compared to both multi-core processors and GPUs.

However, our previous study also shows some limitations. Large models, such as the model of human ventricular tissue, do not fit into our experimental device, the Terasic DE4 board, due to the exhaustive use of resources. Small to medium models are well for the DE4 board, but there are plenty of resources remaining idle which should be used for potential performance improvement.

In this paper, we propose three optimisation strategies to overcome or mitigate these limitations: equation optimisation, resource fitting and balancing, and multiple pipelines. We use these strategies to optimise the HAMs

before, and after, they are generated by ODoST. Two CellML models are used in the evaluation and results show that with multiple pipelines, medium sized models can achieve significant improvement in both processing speed and power efficiency and, for large models, equation optimisation and resource fitting/balancing techniques can assist the models to be fitted into a selected device.

The paper is organized as follows. We discuss related work in Section II. In Section III, we briefly discuss the HAM structure and ODoST that have been proposed in our previous study. We describe the three optimisation strategies in Section IV, V and VI. In Section VII, we evaluate these optimisation strategies in experiments. We conclude the paper in Section VIII.

II. Related Work

Biomedical models and simulations are used to understand normal and abnormal functions of animals and humans. Apart from CellML, other modelling languages such as the Mathematical Modelling Language (MML) [6] and the Systems Biology Markup Language (SBML) [7] have been developed for storing and interchanging biological mathematical models. Simulation tools have also been developed to simulate models written in those modelling languages. Among them, several tools emphasise large scale and continuum simulations that require high performance computation, such as the Cancer Heart and Soft Tissue Environment (CHASTE) [8] and OpenCMISS [2]. In our research, we have designed and built the accelerator model based on CellML which is intended to be used by OpenCMISS and other simulation packages.

Many case studies have used FPGAs to accelerate the simulation of biomedical models. Yoshimi et al. [9]'s accelerator of a fine-grained biochemical simulation achieved a 100x speedup compared to a single processor during that time. Osana et al. [10] developed a solver-based tool, ReCSiP, for biochemical simulations using Xilinx's XC2VP70-5 and reported a 50 to 80 times speedup compared to Intel's Pentium 4 processor. Thomas and Amano [11] proposed a pipelined architecture for a stochastic simulation of chemical systems and reported that their architecture was 30-100 times faster compared to a pure software simulator. de Pimentel and Tirat-Gefen [12] estimated that a real time simulation of a heart-lung system model was expected to be 90 times faster than a PC. However, their evaluation was calculated theoretically based on the performance of the multiplier on the device rather than a real implementation. Chen et al. [13] implemented a Runge-Kunta ODE solver using FPGAs and Simulink that resulted a 100x speedup compared to a 2.2 GHz desktop CPU. Most of these studies used a manual design and implementation to develop a specialised accelerator model. Manual design is impractical in biomedical/mathematical simulations since it is time consuming and requires hardware development skills, often not found in those researches with a biological background.

Due to the high requirement of development efforts and skills, many tools have been developed for implementing applications on FPGAs through High Level Synthesis (HLS) like SPARK [14], DRFM [15], GAUT [16], LegUp [17] and polyAcc [18], and also commercial level HLS tools like the Altera OpenCL SDK Altera [19], Xilinx's Vivado HLS [20] and Maxeler's MaxCompiler [21]. These tools are used to develop hardware circuits from a high level representation, e.g. C, Matlab, Java, etc. They prioritise functional equivalence with the input source code, and relies upon the hardware developer to transform their code in order to enhance the efficiency of their designs. In our research, we design and implement a domain specific synthesis tool called ODoST, which focuses on the ODE-based mathematical modelling and aims at creating the complete datapath of a given model including the data communication and software interfacing. This means the aim of ODoST is to generate a complete implementation of all parts, including the host software, the communication between host and FPGA and the acceleration module, which is ready for execution. ODoST is designed to be flexible so that optimisations can be easily adapted into the automatic generation process.

Equation or compiler optimisation strategies are widely discussed in the mathematical field. These strategies include the common subexpression elimination [22], partial product reduction [23] and multivariate Horner scheme optimisation [24]. We used these commonly used equation optimisation strategies as well as other strategies specified for the hardware resource usage. In our research, we develop an LLVM based implementation to optimise the original CellML C code to an optimised C code aiming at a lower hardware resource consumption.

A few tools or strategies have been developed in order to fit a large designs into target devices where high resources are utilised. Tessier and Giza [25] outlined a procedure to determine the appropriate partitioning of programmable logic and interconnection area to minimise overall device area. Liang et al. [26] formulated a module selection problem and discussed strategies to solve the problem. DeHon [27] presented a hierarchical array design to balance the interconnects and logic use. Each strategy is designed to satisfy a specified domain of problems and, in our study, we divide our module into operations and perform a resource balancing algorithm on each individual operation.

III. HAM AND ODOST

A. Biomedical Model Overview

Biomedical models are often represented by a set of ODEs describing time varying variables and parameters. In our research, we use selected biomedical models from the CellML model repository[1] which contains 300+ models. Each CellML model is component based and components are represented by one or more mathematical equations or expressions. For example:

$$alpha\_m = \frac{-(V+47)}{e^{-\frac{V+47}{10}} - 1} \tag{1}$$

$$beta\_m = 40 \times e^{-0.056 \times (V+72)} \tag{2}$$

$$\frac{dm}{dt} = alpha\_m \times (1-m) - (beta\_m \times m) \tag{3}$$

The above equations represent the component of sodium m gate current in the Beeler-Reuter model[2], a model that describes the mammalian ventricular action potential. The model contains a total of 13 components with 26 mathematical equations/expressions. Each CellML model contains a list of state variables (V and m) that are time dependent, a list of rate constants and intermediate variables ($alpha\_m$ and $beta\_m$) and the rates of the states ($\frac{dm}{dt}$) at time t. For a single time step of a model integration, the values of the intermediate variables are computed first based on the state variables (and rate constants if they are required). The rate of change for the state variable is then computed which is dependent on the intermediate variables. Once the value of rate is available, a numerical integration method is used to approximate the state value at the next time step. A variety of such numerical integration algorithms exist and, in this paper, we use a forward Euler's method [28], which is a simple and fast numerical method that is widely used. According to Euler's method, the computation of the state variable m at time $t + \triangle t$ is represented in Eq. (4).

$$m_{t+\triangle t} = m_t + \triangle t \times \frac{dm}{dt} \tag{4}$$

In order to achieve accurate and stable results, the above process is performed using fine time steps. For example, to integrate $1\,ms$ of the model at one point, we divide the time interval into 1000 time steps with each time step taking $1\,\mu s$. At each time step for the "sodium_current_m_gate", the computations in Eqs. (1 - 3) are performed first to obtain the rates of change and then numerical integration is performed to find the new states after $1\,\mu s$. The new state variables are then passed to the next step for the next time integration and so on. During this integration process, each point is integrated individually and independent of other points. According to the fine integration process, the time requirement of I/O data communication is far less than the compution requirement. This is because each cell is integrated 1000 times individually and independently before sending the data back to the host for spatial solving. For example, in the human ventricular tissue model, the data to computation ratio is 588 bytes : 527,000 FLOPs. Therefore, pipelining and concurrency features of FPGAs can be largely exploited.

B. Hardware Accelerator Module

We have developed the Hardware Accelerator Module (HAM) architecture to accelerate biomedical models using pipelined floating point operations. The pipelined structure allows a multiple number of independent cells to be accessed cycle by cycle and hence to achieve one cell operation per cycle throughput. The hardware/software co-design architecture is shown in Figure 1, and is composed of a host computer and an FPGA board connected through a PCIe interface. The arrows indicate the data communication flows throughout the system. The software module is used as a bridge application from the biomedical simulator such as OpenCMISS [2] and interacts with the FPGA by sending and receiving data through the PCIe interconnects.

On the FPGA side, there is a PCIe IP core that interacts with the PCIe connector and maps to the on-chip memory directly for the control signals, and through the DMA (Direct Memory Access) controller for data transfer. The on-chip memory is used as an intermediate data buffer and because of the low I/O data requirement (e.g.,

[1] http://www.cellml.org/model

[2] models.physiomeproject.org/e/9a

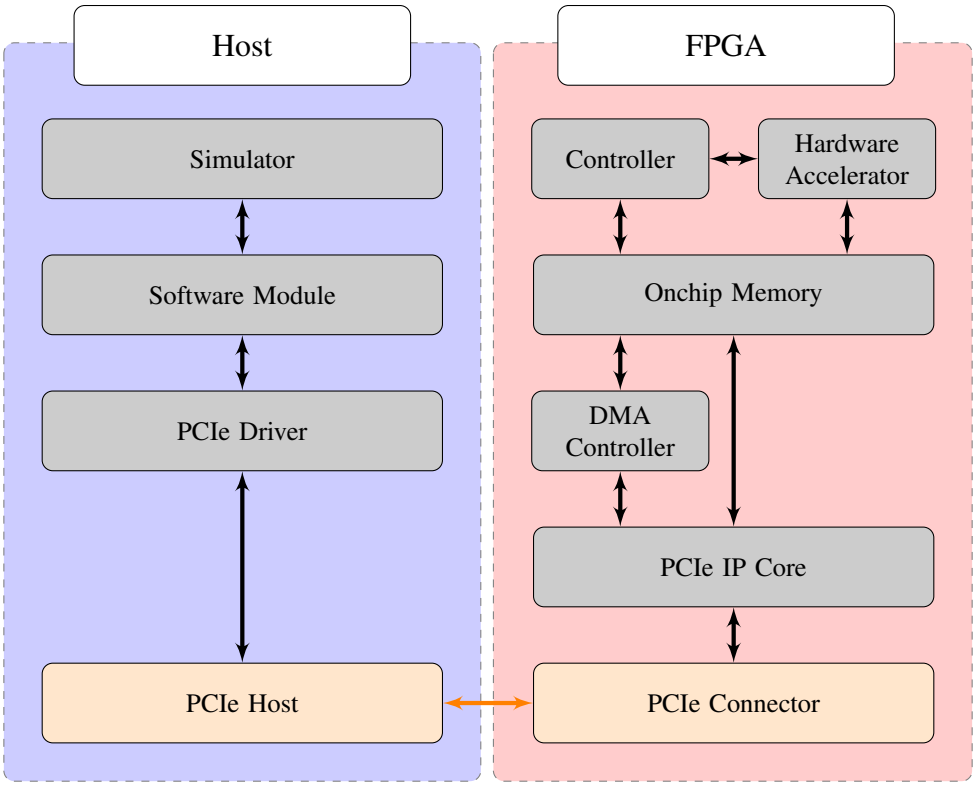


Figure 1: Hardware Accelerator Module System Architecture

complex models like the human ventricular tissue only requires 200 KB at a time), the capacity of on-chip memory is adequate for this usage. The data received from the host computer is written into on-chip memory through the DMA controller. An FPGA data flow controller is interfaced with the on-chip memory through the memory mapped interfaces and to send/receive signals to/from the host computer and interact with the CellML hardware model to control the data transfer flow.

The data control is implemented by a state machine that comprises of six states: *Idle*, *Read-ToFIFO*, *Read-FromFIFO*, *Compute*, *Write-ToFIFO* and *Write-FromFIFO*. In short, the state machine works as follows, for more details refer to [5].

At the *Idle* state, the host is in control. The FPGA continues checking the relevant sector in on-chip memory to obtain the data control signal. After the host finishes loading data to the on-chip memory, it grants the control to the FPGA, the state moves to *Read-ToFIFO*. At the *Read-ToFIFO* state, data is read from the on-chip memory to an input FIFO buffer. The FIFO buffer is used to balance the computation in the pipeline datapath. Once all the inputs are in the FIFO buffer, the state changes to *Read-FromFIFO*. At this state, the input data is read from the FIFO buffer into the hardware accelerator pipeline cycle by cycle and the model computation starts. The *Compute* state starts when all the input data sets are passed into the hardware accelerator. At each micro time step, the states variables computed from the previous micro time step enter the pipeline for the next integration. The length of pipeline and total number of micro time steps are predetermined and a shift register counter is used to count the micro time steps. Similar to the read, at the *Write-ToFIFO* state, the hardware accelerator is doing the final micro time step computation. The output data immediately enters a FIFO buffer. Once the computation finishes, the state machine moves to *Write-FromFIFO*. At *Write-FromFIFO* state, data is written into the on-chip memory from the FIFO buffer. Once all the output data is available on the on-chip memory, the FPGA passes the data control to the host by updating the control signal on the on-chip memory and the state machine moves to the *Idle* state waiting for next set of input data. Host captures the control signal and activates the DMA to read the output data from the on-chip memory.

C. ODE-based Domain Specific Synthesis Tool

Although it is generally agreed that hardware accelerators using FPGAs have the potential for performance improvement with efficient energy use, implementing such an accelerator for a given biomedical model requires an enormous effort which might offset the advantages of using FPGA. Therefore, we have developed ODoST [5], a domain-specific High-Level Synthesis (HLS) tool, for ODE-based biomedical simulations. The tool is aimed at biomedical scientists and engineers, who often have little knowledge of designing hardware, to create accelerators targeting FPGAs.

ODoST stands for **O**DE-based **Do**main-specific **S**ynthesis **T**ool. It generates the HAM with both software and hardware modules from an ODE-based biomedical model. ODoST contains three phases: analysis phase, generation phase and system integration phase. In the analysis phase, ODoST reads a biomedical input model in C99 format, converts equations from infix format to an HDL favourable data structure through postfix processing. In the generation phase, ODoST uses the Jinja2 [29] template engine to generate HDL codes and configuration scripts based on the data from the analysis phase. In the system integration phase, the configuration files are used to produce the entire hardware module based on the HDL files from the generation phase. The automatic generation process is complete as the produced software and hardware modules are ready to run on the hardware.

Evaluation on real hardware of the HAMs generated by ODoST from selected CellML models was performed and compared against CPU and GPU implementations. The results show that FPGAs can provide a highly energy efficient solution and remarkable processing performance compared to both multi-core processors and GPUs. Although the HAM design already showed a significant speed up, there is still potential for further optimisation. Three optimisation strategies–compiler optimisation, resource fitting and balancing, and multiple pipelines–are proposed and discussed in the rest of this paper.

IV. Compiler Optimisation

Compiler optimisation is a process to minimise or maximise some attributes of a computer program. Most modern software compilers provide automatic optimisation techniques that are not provided by ODoST. Therefore, the previous evaluation is not an entirely fair comparison until C-based CellML models are optimised for hardware processing (in the similar way as software compilers do) before being processed by ODoST.

The most common goals for compiler optimisations are either to improve a program's execution speed on a software programmable processor, or to reduce its code size and thereby its memory footprint. These goals are intertwined, as a program executing less instructions is both faster and smaller, and a smaller program may execute faster due to cache effects. Architecture-independent transformations therefore share the basic idea of eliminating redundant computations. This can be achieved by reusing the results of equivalent computations, or by evaluating constant operations at compile time.

In the targeted code optimisation for ODoST, we have a similar but not identical goal, as we aim to optimise for lower hardware resource consumption after the high-level synthesis process. Arguably this can be similar to optimising for code size and we investigate the usefulness of well-known optimisations, for which we rely on LLVM's implementations. LLVM [30] is a compiler infrastructure written in C++ and is designed for compile-time, link-time, run-time and "idle-time" optimisation of a program written in arbitrary programming languages.

A. Local Optimisations

The LLVM facilitates local algebraic simplifications through pattern matching and replacement, as well as constant folding, and a simple form of dead code elimination.

The local transformations either reduce the number of instructions, or normalise instructions into a canonical form. These normalisations are important, because they allow the other transformations to match for fewer patterns, and create more constant folding opportunities.

Figure 2 shows examples for patterns that are applicable to the CellML equation systems. Eq. (5) moves constants to the right-hand side of commutative operations. Operations involving their respective neutral element are eliminated in Eq. (6). Subtractions and divisions are replaced in favour of commutative and simpler additions and multiplications with the inverse constant in Eqs. (7, 8). Eq. (9) transforms a multiplication with -1 into a negation, which is encoded as $0 - x$, because the LLVM-IR (intermediate representation) does not contain a dedicated negation instruction. Eqs. (10, 11) are used to eliminate the extra subtraction where possible. The square operation is transformed from a call to the generic power function to a single multiplication in Eq. (12). Eqs. (13-16) show the application of

$$c \oplus x \Rightarrow x \oplus c \tag{5}$$
$$x + 0, x \cdot 1 \Rightarrow x \tag{6}$$
$$x - c_1 \Rightarrow x + c_2 \quad \text{with } c_2 = -c_1 \tag{7}$$
$$x/c_1 \Rightarrow x \cdot c_2 \quad \text{with } c_2 = 1/c_1 \tag{8}$$
$$x \cdot (-1) \Rightarrow 0 - x \tag{9}$$
$$(0 - x) + c_1 \Rightarrow c_1 - x \tag{10}$$
$$(0 - x) \cdot c_1 \Rightarrow x \cdot c_2 \quad \text{with } c_2 = -c_1 \tag{11}$$
$$x^2 \Rightarrow x \cdot x \tag{12}$$
$$c_1 \oplus c_2 \Rightarrow c_3 \quad \text{with } c_3 = c_1 \oplus c_2 \tag{13}$$
$$(x \oplus c_1) \oplus c_2 \Rightarrow x \oplus c_3 \quad \text{with } c_3 = c_1 \oplus c_2 \tag{14}$$
$$(x \cdot c_1 + c_2) \cdot c_3 \Rightarrow x \cdot c_4 + c_5 \quad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{15}$$
$$(c_1 - x \cdot c_2) \cdot c_3 \Rightarrow c_4 - x \cdot c_5 \quad \text{with } c_4 = c_1 \cdot c_3 \text{ and } c_5 = c_2 \cdot c_3 \tag{16}$$

Figure 2: Exemplary transformations done by LLVM [30]. x denotes an unknown value, the c_i are constants, $\oplus$ is either an addition or a multiplication, $\pm$ is either an addition or subtraction.

constant folding where Eq. (15, 16) are even performed on distributive expressions if the number of operations can be reduced thereby.

These equations hold for floating-point arithmetic, with the exception of Eq. (8), which is only safe to do if the reciprocal is accurate, and Eqs. (14-16).

Along these transformations, a simple store-to-load forwarding is performed. This connects consumers of a model variable to the defining expression, circumventing the effect that the underlying pairs of array stores and loads normally break the def-use chain in the intermediate representation.

B. Common Subexpression Elimination

The common subexpression elimination is a compiler optimisation strategy that eliminates commonly used subexpressions. It can be applied either locally in an equation, or globally, among a set of equations. For example:

$$\begin{aligned} y &= x \cdot a + b \\ z &= x \cdot a + c \end{aligned} \tag{17}$$

can be transformed to:

$$\begin{aligned} tmp &= x \cdot a \\ y &= tmp + b \\ z &= tmp + c \end{aligned} \tag{18}$$

which eliminates one multiplication.

C. Higher-order powers

Higher-order positive integer powers can be represented by a number of multiplications which is also referred as addition-chain exponentiation. In this paper, we use the binary exponentiation method [31] in Algorithm 1 to transform V^P for an LLVM Value V and an integer power P into a sequence of multiplications.

First, we construct V^{2^k} with $2^k \leq P \Leftrightarrow k = \lfloor ld(P) \rfloor$, and store it alongside the intermediate powers $\mathrm{V}^{2^0}, \ldots, \mathrm{V}^{2^{k-1}}$ in the map *Powers*. This requires k multiplications. Then, we define R to satisfy $V^P = V^{2^k} \cdot V^R$, and construct V^R by reusing the pre-calculated values from the map *Powers* corresponding to the bits set in the binary representation of R, which requires as many multiplications as non-zero bits in *R* minus 1. One additional multiplication is used to calculate (e.g. $V^{30} = V^{16} \cdot V^{14}$, 4 multiplications are required by V^{16}, and 2 multiplications are required by

Algorithm 1 Algorithm to turn V^P into a minimal series of multiplications

Require: V : Value and P : int
1: X : Value
2: Powers : map(int->Value)
3: k,R : int
4: $k \leftarrow \text{LOAD}(P)$
5: $X \leftarrow Powers[0] \leftarrow V$
6: **for** $i = 1$ *to* k **do**
7: $\quad X \leftarrow Power[i] \leftarrow new\ mul\ X, X$
8: **end for**
9: $R = \{r_m, ..., r_0\} \leftarrow P - 2^k$
10: **for all** $r_j \in R$ with $r_j = 1$ **do**
11: $\quad$ X $\leftarrow$ new mul X, Powers[j]
12: **end for**
13: **return** X

$V^{14} = V^8 \cdot V^4 \cdot V^2$ (reusing the V^8,V^4,V^2 results) plus 1 multiplication for the final product). The resource consumption of a generic power function on FPGA is equivalent to around 8 multipliers. According to the resource usages of the multiplier and the power function, the power to multiplications transformation is benefited upto eight multiplications, which is sufficient to transform powers as large as 46, and thereby covers all practical cases. The results of the binary exponentiation algorithm are not necessary optimal and the related problem of finding a shortest addition chain for a given set of exponents has been proven NP-complete [32].

D. Exponential Function Simplification

Exponential relations are common in the biological processes modelled by CellML descriptions. This justifies an extra effort towards the optimisation of expressions involving the exponential function. We focus on expressions of the form

$$\exp(x \cdot \underline{a} + \underline{b}) \cdot \underline{c} \tag{19}$$

where constant subexpressions are underlined. We can fold the second multiplication into the existing addition by using the power laws:

$$\exp(x \cdot \underline{a} + \underline{b} + ln\underline{c}) \tag{20}$$

Applying this pattern without an existing addition has replaced one multiplication by an adder. This results in an overall saving of resource usage since a multiplication is generally more expensive than an addition.

Additional operations can be saved by separating the variable and constant parts of the expressions in sets of n expressions of the form:

$$\begin{array}{c} \exp(x \cdot \underline{a_1} + \underline{b_1}) \\ \vdots \\ \exp(x \cdot \underline{a_n} + \underline{b_n}) \end{array} \tag{21}$$

In the special case $a_1 = \cdots = a_n$, the multiplication can be reused across the expressions which reduces the number of multiplications by $n - 1$. Alternatively, splitting and reusing the variable exponentiation leads to

$$\begin{array}{c} t_1 \leftarrow exp(x \cdot \underline{a_1}) \\ t_1 \cdot \underline{\exp(b_1)} \\ \vdots \\ t_1 \cdot \underline{\exp(b_n)} \end{array} \tag{22}$$

which adds one multiplication, but eliminates $n - 1$ exponentiations and n additions.

E. Source-to-source Optimiser

According to the optimisation strategies discussed above, an LLVM-based source-to-source optimiser, *cellml-opt*, was developed. The optimiser generates optimised C code from the original CellML model which uses standard C that follows a fixed scheme. LLVM's C frontend clang parses the C code of the CellML model and constructs the LLVM intermediate representation (IR). *cellml-opt* reconstructs the model's C representation from the LLVM IR.

Program equations of the model map to the LLVM functions. Both C-built-in arithmetic operators (e.g., *,+) and functions defined in the C math library, such as power and exponential functions, are mapped one-to-one to the respective LLVM instructions. Program variables are accessed via pointer values constructed from the function's arguments. The particular variable and the index in an array, e.g., $STATES[2]$, can be easily determined from LLVM's special getelementptr instruction used by a load or a store instruction. The input variables for each equation are treated as independent variables, made possible by the fact that the input code follows a fixed scheme where this is guaranteed. This enables the alias analysis framework of LLVM to detect that all variable accesses, excluding the ones with the same base and index, are independent. On the other hand, all variable accesses with the same base and index can be identified to a single value, greatly helping later optimisations.

The optimised output C code looks almost the same as the input code in terms of style and structure. A simple intermediate representation, "EqIR", is introduced in *cellml-opt* so that the models original equations are transformed into a list of pairs, where a pair consists of a left-hand side expression and a right-hand side expression. The EqIR is then mapped into LLVM Intermediate Representation (IR). Before the resulting C code is finally emitted from the EqIR, temporary variables are added to the equation system to represent LLVM values that are reused across different expressions.

V. Resource Fitting and Balancing

In our previous research, we used ODoST to generate HAMs using the FloPoCo generated floating point cores with DSPs. Generally, DSPs provide an order of magnitude higher performance with lower power consumption. However, the DSPs are a limited resource within one FPGA and they can become the bottleneck compared to the other resources in the device. In order to solve the problem of exhaustive use of DSPs, FloPoCo also provides floating point cores employing only logic elements. Apart from FloPoCo, there are numerous existing floating point cores provided by the vendors of FPGAs and other third party floating point platforms. Some floating point cores employ DSP blocks and others use pure logic. There is no right answer to the question which floating point core is the best, since it depends on the FPGA resource capacity and the particular biomedical model. With the given resources and model, an effective resource allocation algorithm can provide better resource utilisation and hence increase the computational throughput. Before we propose such an algorithm, we briefly discuss the resources of an FPGA and the resource usage of selected floating point cores.

A. FPGA Resource Capacity

The heterogeneous nature of modern reconfigurable devices means it is complicated to determine the capacity of an FPGA. The evaluation of the generated HAMs from our previous work shows that the logic, registers, memory and DSPs are the four key resources consumed in the implementation, so we focus on the capacities of those resources. Table I lists four selected high-end FPGAs from two leading FPGA vendors. The resources of different FPGA generations and vendors are organised differently, however, they follow the similar principle that the main resources are formed by logic, registers, memory and DSPs.

The basic building blocks of the Altera's Stratix series are the Adaptive Logic Modules (ALMs) that provide logic and dedicated registers. However, Stratix V devices use enhanced ALMs that contain 6% more logic and double the number of registers compared to Stratix IV ALMs. In Xilinx's Virtex series, the Configurable Logic Blocks (CLBs) are the main logic resources for implementing circuits. The Altera and Xilinx FPGAs also provide DSP blocks that implement multiplication, multiply-add, multiply-accumulate (MAC), and dynamic shift functions efficiently. They can be effectively used by floating point multipliers, exponential functions, power functions and logarithms to reduce logic usage and achieve high performance. From Table I, we refer to the equivalent logic elements (LEs) for the Altera FPGAs and logic cells for the Xilinx FPGAs so that they can be compared.

The Altera Stratix EP4SGX530 FPGA built in the Terasic DE4 board [33] is the target FPGA device in our investigations and evaluations. Instead of using the equivalent LEs which is used for comparision between different FPGAs, we use the Adaptive Look-Up Tables (ALUTs) in the analysis. The Altera Stratix EP4SGX530 FPGA contains 212,480 ALMs. Each ALM is composed of two ALUTs, two registers and other logic and interconnects.

Family	Stratix IV	Stratix V
Device	EP4SGX530KH40C2	EP5SGXEA7N2F45C2
Equivalent LEs	531,200	622,000
Registers	424,960	938,880
Memory Bits	21,233,664	50,000,000
DSPs	1024	768

(a) Altera FPGAs

Family	Virtex-6	Virtex-7
Device	XC6VHX565T	XC7V485T
Logic Cells	566,784	485,760
Registers	708,480	607,200
Memory Bits	32,832,000	37,080,000
DSP Slices	864	2,800

(b) Xilinx FPGAs

Table I: Resource Capability for Selected Devices

Function	Output Latency	ALUTs	DLRs	ALMs	DSPs	Fmax
ALTFP_ADD_SUB	7	576	345	375	-	227
ALTFP_DIV	33	1646	2074	1441	-	308
	6	207	304	212	16	358
ALTFP_MULT	5	138	148	100	4	274
ALTFP_EXP	17	631	521	448	19	275
ALTFP_LOG	21	1950	1864	1378	8	385

Table II: Altera Single Precision Floating Point Megafunctions Resource Utilization and Performance for Stratix IV Devices

The ALUTs are used for either combinational or memory and the capacity of ALUTs in the EP4SGX530 FPGA is 424,960. Registers refers to the Dedicated Logic Registers (DLRs).

B. Floating Point Cores

As mentioned, there are numerous existing floating point cores provided by the vendors of FPGAs and other third party floating point platforms. These cores typically exploit the freedom of an FPGA by providing customisation of variable widths and of exponent and mantissa size to meet designers' specifications. They also offer IEEE standard single and double precision cores that are used in the proposed hardware accelerator. This section describes two floating point cores, Altera Floating Point Megafunctions and FloPoCo and the implementation of additional operations that are not supported by the cores.

Altera Floating Point Megafunctions: Altera provides a comprehensive set of IEEE 754-compliant floating point operations as IP modules for their FPGAs [34]. The Altera floating point megafunctions support single and double precision selection and single extended configurable precision and can be parameterised by balancing the frequency at which the operators run and the pipeline latency of the operator hardware to fine-tune its overall performance, power and area. The typical resource usages and latencies of the typical single precision Altera floating point cores are displayed in Table II for our experimental target FPGA. Altera Floating Point Megafunctions support round-to-nearest-even rounding mode, the default of IEEE-754-1985. They also support exception signals for underflow and overflow.

FloPoCo: FloPoCo, standing for Floating Point Cores, is an open source generator of arithmetic cores for FPGAs [35]. In difference to IEEE floating point representations, FloPoCo has a special floating point format with an additional two-bit prefix. The two bits are only used to signal special-case numbers, namely 00 for zero, 01 for normal numbers, 10 for infinities, and 11 for NaN. In IEEE, those exception signals are handled by exponent and mantissa. This saves quite a lot of decoding/encoding logic. The main drawback of this format is when results have to be stored in memory, where they consume two more bits. However, FPGA embedded memory can accommodate 36-bit data, so adding two bits to a 32-bit IEEE-754 format is harmless as long as data resides within the FPGA. Conversion only needs to take place when passing data to and from the host PC.

In the hardware acceleration design, floating point cores are generated individually. The resource usage and latencies of the generated single precision compatible FloPoCo Floating Point Cores for Stratix IV devices with

Function	Output Latency	ALUTs	DLRs	ALMs	DSPs	Fmax
FPAdd	12	269	622	395	-	523
FPDiv	17	1188	1407	1116	-	308
FPMult	4	73	219	132	4	835
	5	893	524	725	-	370
FPExp	17	436	878	507	2	195
	17	816	939	755	-	237
FPLog	21	831	1210	808	18	175
	22	1434	1885	1399	2	331
FPPow	45	1808	3307	2058	31	177
	50	3884	4359	3620	5	232

Table III: Resource utilization and performance of FloPoCo generated single precision floating point cores for Stratix IV Devices

Variation	ALUTs (%)	DLRs (%)	DSPs (%)
Altera (A)	0.0325	0.0348	0.389
FloPoCo-DSP (B)	0.0172	0.0515	0.389
FloPoCo-logic (C)	0.210	0.123	0

Table IV: Resources percentage usage of the three variations of floating point multiplication

DSPs and without DSPs are displayed in Table III. The resource usage depends on the configurations specified during the generation, especially whether to use the DSP blocks or not. $FPAdd$ and $FPDiv$ are only implemented with pure logic. $FPMult$, $FPExp$, $FPLog$ and $FPPow$ contain implementations that either favour the use of DSP blocks with hardware multipliers or pure logic. For devices with a large number of DSPs, but lack of logic, the floating point implementations with DSPs are favoured otherwise, the implementations with pure logic are preferred. Furthermore, multiple variants of a single operation can also be used together in a larger design, e.g. mixing pure logic implementations and DSP implementations, to achieve better resource utilisation and balance.

C. Resource Allocation Techniques

For biomedical models that do not fit on a given FPGA after equation optimisation, resource fitting techniques can be used to balance the logic, register, memory and DSP consumption. This process is called the resource planning process and is performed on the original or optimised C code of the CellML models. Memory in the HAM implementations is mainly used as the data buffer and RAM-type shift registers and it is unlikely to reach the memory capacity of an FPGA before the other resources. Therefore, in our resource allocation algorithm, we only consider the logic, registers and DSPs. Since the number of a certain operation within the original or optimised CellML model is fixed, we deal with each operation individually aiming at achieving the minimum usage of each resource, while the differences between the percentage resource usage are minimised.

Formulating the Problem: We use multipliers as a case study for the underlying resource allocation techniques, but apply the same technique to the optimisation of other operators. -According to the floating point multiplication cores illustrated in Table II and III, there are three variants of a multiplier. We define the three variants as A - the Altera implementation, B - the FloPoCo implementation with DSPs and C - the FloPoCo implementation with the pure logic. The percentage usage of the three variants are summarised in Table IV. Let PL_A, PL_B and PL_C denote the percentage usage of logic for each variant of multiplier. Let PR_A, PR_B and PR_C denote the percentage usage of registers for each variant and PD_A, PD_B and PD_C denote the percentage usage of DSPs for each variant, respectively. In an FPGA design, let N_A, N_B and N_C stand for the number of multipliers implemented as variant A, B, C, respectively for each variant in a CellML model and N stand for the total number of multipliers in the model. Therefore, the following condition must be satisfied.

$$N_A + N_B + N_C = N \tag{23}$$

with $N_A, N_B, N_C, N \in \mathbb{N}^0$.The total usage of each resource is

$$PL = PL_A \cdot N_A + PL_B \cdot N_B + PL_C \cdot N_C \tag{24}$$

Scheme	Action	Condition
$PL > PD \geqslant PR$	N_B++; N_A–	$N_A > 0$
$PR \geqslant PL > PD$	N_A++; N_C–	$N_C > 0$
$PR > PD \geqslant PL$	N_A++; N_B–	$N_B > 0$
$PD \geqslant PL > PR$	N_A–; N_C++	$N_A > 0$
$PL > PR > PD$	N_B++; N_C–	$N_C > 0$
$PD \geqslant PR \geqslant PL$	N_B–; N_C++	$N_B > 0$

Table V: Schemes for reducing PT used in the greedy algorithm

$$PR = PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \tag{25}$$

$$PD = PD_A \cdot N_A + PD_B \cdot N_B + PD_C \cdot N_C \tag{26}$$

For the FPGA resource balancing problem, we are going to determine the best values for N_A, N_B and N_C to minimise the maximum resource usage, P_{max}, which is the potential bottleneck. The maximum resource usage is usually minimised by increasing the usage of other resources and hence the pair-wise gap between each resource usage is minimised to achieve the resource balance purpose. The problem can be expressed as minimising P_{max} in the following expression.

$$P_{max} = \max(PL, PR, PD) \tag{27}$$

Exhaustive Algorithm: The above problem can be naively solved by an exhaustive algorithm. The algorithm enumerates all possible value combinations of N_A, N_B and N_C that adhere to (23), calculates P_{max} for each combination and keeps track of the values that make P_{max} the smallest. This naturally leads to the optimal solution of the problem. The complexity of the algorithm is high when the number of implementation choices, k, is high. It is the number of compositions of N into exactly k parts, which is the following binomial coefficient:

$$O(exhaustive) = \begin{pmatrix} N-1 \\ k-1 \end{pmatrix}$$

For the three alternative implementations considered here it is $\frac{(N-1)!}{(k-1)!((N-1)-(k-1))!} = \frac{(N-1)!}{2(N-3)!} = \frac{(N-1)(N-2)}{2} = O(N^2)$. Due to the size of typical problems (e.g., in the optimised TNNP model used in the evaluation, $N = 166$) and the limited number of choices, in many cases, the exhaustive algorithm is still adequate for the resource balancing problem.

Multivariate Equations: We are looking for the minimised value of P_{max}. As said, one way is to minimise the pair-wise difference between each resource usage. Eqs. (24-26) can be reformulated into

$$PL_A \cdot N_A + PL_B \cdot N_B + PL_C \cdot N_C \approx PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \tag{28}$$

$$PR_A \cdot N_A + PR_B \cdot N_B + PR_C \cdot N_C \approx PD_A \cdot N_A + PD_B \cdot N_B + PD_C \cdot N_C \tag{29}$$

Eqs. (28, 29) together with Eq. (23) are a simply ternary linear equation set that can be solved directly. These multivariate equations are easy to solve by hand and there are also many existing computational tools/libraries that can be used to solve these equations, e.g., Matlab [36]. However, one constraint of the equations is that N_A, N_B and N_C should be natural numbers, but the results for the equation set may end up with negative or non-integer values which are not acceptable in our analysis. If any result is negative, it can be replaced with zero and we solve the remaining linear equations. Non-integer results can be simply replaced with the nearest integers.

Greedy Algorithm: An alternative for this problem is to use an effective greedy algorithm. We propose a greedy algorithm for the resource balancing problem as illustrated in Algorithm 2. In this greedy algorithm, we define six schemes illustrated in Table V to reduce the value of P_{max} in each execution. Table V, used in the greedy algorithm, is created following the rules that (i) variant combinations to increment/decrement are selected to reduce the gap between the resources with maximum and minimum percentage usage, (ii) each combination should occur only once, and (iii) looping situations are avoided, i.e. in one step, NA++; NB– and in the next step, NB++, NA–. The algorithm starts with the selection of the initial conditions by choosing the best values of N_A, N_B, N_C and

Algorithm 2 Greedy algorithm for resource balancing

Require: N
Ensure: NA + NB + NC = N and NA >= 0, NB >= 0, NC >= 0
1: **procedure** GREEDY(N)
2: $Pmax \leftarrow 100$
3: $newNA, newNB, newNC, newPmax \leftarrow$ GETSINITIALVALUES(N)
4: **while** $newPmax < Pmax$ **do**
5: $NA \leftarrow newNA$
6: $NB \leftarrow newNB$
7: $NC \leftarrow newNC$
8: $Pmax \leftarrow newPmax$
9: newNA, newNB, newNC = SELECTSCHEME(NA, NB, NC)
10: $newPmax \leftarrow$ CALCULATEPMAX$(newNA, newNB, newNC)$
11: **end whilereturn** NA, NB, NC, $Pmax$
12: **end procedure**

N	Techniques	N_A	N_B	N_C	$P_{max}(\%)$
50	Exhaustive Algorithm	0	18	32	7.03
	Multivariate Equations	-75/0	91/19	34/31	7.39
	Greedy Algorithm	0	18	32	7.03
100	Exhaustive Algorithm	0	36	64	14.06
	Multivariate Equations	-150/0	-182/37	68/63	14.39
	Greedy Algorithm	0	36	64	14.06
200	Exhaustive Algorithm	0	72	128	28.12
	Multivariate Equations	-300/0	-364/74	136/126	28.79
	Greedy Algorithm	0	72	128	28.12
400	Exhaustive Algorithm	0	144	256	56.24
	Multivariate Equations	-600/0	729/148	271/252	57.57
	Greedy Algorithm	0	144	256	56.24
500	Exhaustive Algorithm	0	180	320	70.30
	Multivariate Equations	-750/0	911/185	339/315	71.97
	Greedy Algorithm	172	13	315	71.97

Table VI: Evaluation results for the resource balancing example for a different numbers of multipliers (For multivariate equations method, the results for the first equations set Eq. (23), Eq. (28) and Eq. (29) contain negative values where the additional equations set is required).

P_{max} from a set of preselected or randomly generated conditions. In every iteration, it calls the SELECTSCHEME method to choose the scheme with $newN_A$, $newN_B$ and $newN_C$ according to the order of the PL, PR and PD, given that the condition of the scheme is satisfied. It then calculates $newP_{max}$. The $newP_{max}$ is compared with the value of the current P_{max}. If the $newP_{max}$ is smaller, it will continue to update the current P_{max} and N_A, N_B, N_C from new values and perform the next iteration check, otherwise, it reaches the local optimum and the results are returned.

The greedy algorithm is more efficient than the exhaustive algorithm. The worst case complexity is $(k-1) \cdot N$ iterations where initial condition is $N_A = N$ and then all is changed to the condition with $N_B = N$ and then to $N_C = N$ and so forth. The complexity is then at most $O(kN)$, which is much better than the exhaustive algorithm. However, the solution can stop at a local optimum and we cannot guarantee it is optimal solution. Careful selection of initial values can help the algorithm to find high quality solutions, as will be see in the evaluation of the next section.

There are further alternatives for the resource balancing. For example, a integer linear program (ILP), that could be solved with an ILP solver like CPLEX [37]. However, the proposed algorithms already achieve quite satisfiable performance and prove our concept.

Evaluation of the Resource Balancing Techniques: To test our techniques, we evaluated the three techniques/algorithms discussed above with 50, 100, 200, 400 and 500 multipliers to determine the proper values for N_A, N_B and N_C so that the maximum individual percentage resource usage is minimised. The results are shown in Table VI.

Of the three techniques, the exhaustive algorithm is easy to implement and accurate, but the execution complexity is the highest. The use of multivariate equations is low in complexity, but it is very likely that the results are negative

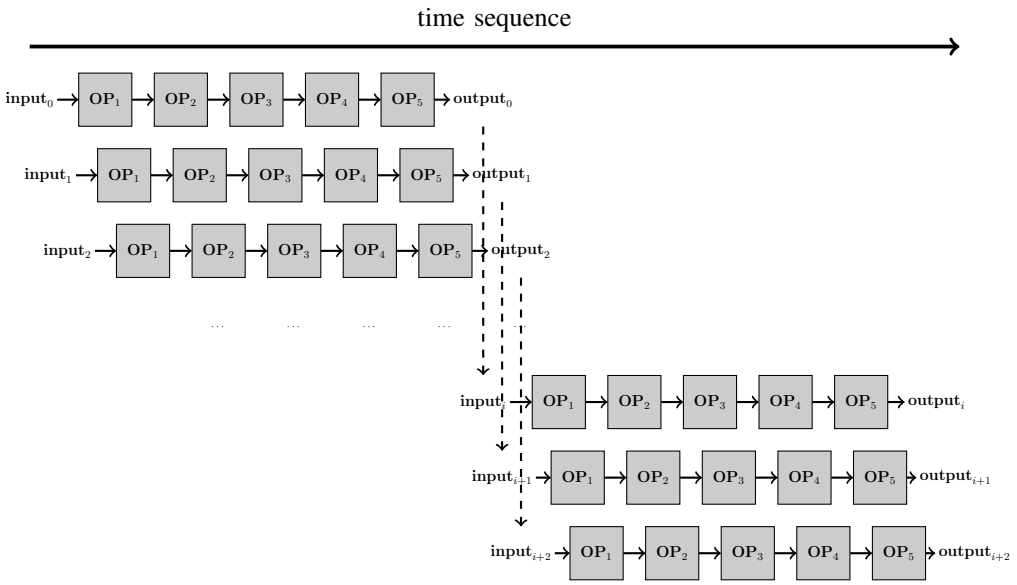


Figure 3: Single Pipeline Flow

or non-integer values where further equations formulation and solving are required. It is also not guaranteed to obtain optimised results- and accuracy of the results are highly dependent on the selection of equations. For example, choosing three equations from Eqs. (23), (28), (29) and $PL \approx PD$. The Greedy algorithm has low complexity as well, however, it depends on the initial values and may end up with a local optimum. In the evaluation, the results of our greedy algorithm are are often identical to the exact result ($N = 50$ to 400). For a large problem size ($N = 500$), the Greedy Algorithm does not work so well, because all the initial conditions chosen by the algorithm are larger than the board capacity (i.e., >100%) and therefore, a new random condition is chosen and the algorithm is likely to find a local optimum. Since the problem size in the resource fitting/balancing algorithm is often relatively small, we suggest to use the exhaustive algorithm to ensure the best results.

VI. Multiple Pipelines

The generated hardware accelerator module is implemented with a fully pipelined architecture. This architecture approach targets high performance applications, allowing new inputs to be applied with every clock cycle. For large biomedical models that use most of the resources on an FPGA, a single pipeline is sufficient. For small to medium sized biomedical models, the HAMs with single pipeline only use a fraction of the available resources and the remaining resources remain idle. With multiple pipelines, the performance of the HAM can be easily improved. Multiple pipelines can be implemented in two ways, either by expanding the temporal direction or by replicating in the spatial direction. We name the two methodologies Extended Pipeline and Parallel Pipeline, respectively.

A. Single Pipeline

The single pipeline flow is illustrated in Figure 3. As described in Section III, the operations of the complete pipeline correspond to the calculation of all the equations of the biomedical model. This calculation needs to be repeated many times for one data item, once for each micro time step. The set of pipelines shown in the figure illustrate that the same pipeline is started on each subsequent data input. Each block in a pipeline represents one operation and with every cycle a new data item enters. Once a data item has passed through the entire pipeline, the output of the last stage is fed back to the input of the pipeline for the next micro time step.

For t micro time steps of cell integration, each input data is iterated through a single pipeline for t times. To complete the entire macro time step integration of one input data set, with the p cycles/stages pipeline, it requires $p \cdot t$ cycles computation. In the pipeline structure, the maximum number of cells allowed for computation in one data chunk depends on the number of pipeline stages,p. After the first data item completes the entire macro time step integration, it requires another $p - 1$ cycles to finish the whole data chunk (i.e. to drain the pipeline) before the outputs are available to the host. Therefore, the total time used for the computation of p cells is $(p \cdot t + p - 1)$

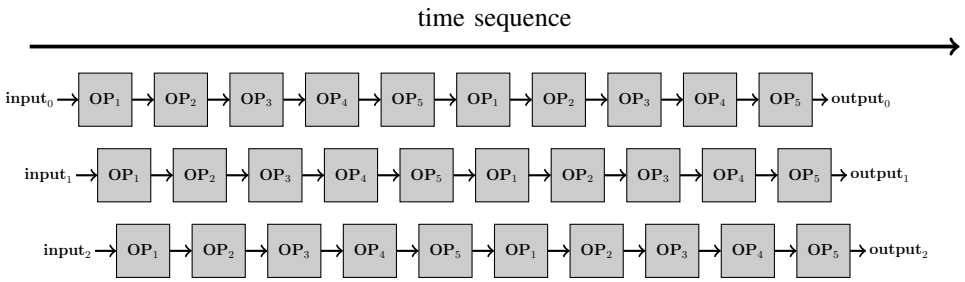


Figure 4: Extended Pipeline Architecture

cycles. Compared to the non-pipelined structure where each cell is computed sequentially $(p \cdot p \cdot t)$, the speedup of the single pipeline is:

$$SpeedUp_{pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{pipe}} = \frac{p^2 \cdot t}{p \cdot t + p - 1}$$

With a 100 stage pipeline of 1000 iterations, the speedup of a pipelined computation against a non-pipelined computation is 99.9. Consequently, filling and draining the pipeline is negligible for the speedup for typical values for the pipeline size and the number of iterations.

B. Extended Pipeline

The numerical integration of differential equations involves repetitive calculations where the same operations are repeated with an integrated data set. The extended pipeline approach expands the pipeline in the temporal direction so that two or more identical single pipelines are joined sequentially to form a long pipeline. Figure 4 shows an extended pipeline that joins two single pipelines. The output of the last stage of the first pipeline is the input of the second pipeline, as can be seen in the figure, and so forth for multiple concatenated pipelines.

For a t micro time steps cell integration, each input data is iterated through a single pipeline for t times. Therefore, an extended pipeline that joins n single pipelines reduces the pipeline iterations to t/n (for simplicity assuming here that t is divisible by n). However, the length of the pipeline increases n times so it requires $p \cdot n$ cycles to complete the pipeline. The latency for one cell integration does not change. But since the pipeline length increases, the maximum number of input cells allowed for computation in one data chunk increases to $p \cdot n$. In other words, once the pipeline is full, the computation and integration for $p \cdot n$ cells is done in parallel. Therefore the total time used for the computation of $p \cdot n$ cells is $(p \cdot t + p \cdot n - 1)$ cycles. Compared to the non-pipelined structure, the speedup of the extended pipeline is then:

$$SpeedUp_{ext-pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{ext-pipe}} = \frac{p^2 \cdot t \cdot n}{p \cdot t + p \cdot n - 1}$$

For two 100 stage single pipelines with 1000 iterations that are joined into one 200 stage extended pipeline, the speedup against a non-pipelined computation is 199.6.

However, since one extended pipeline contains n cell iterations, it is required that the number of iterations to be divisible by n. In order to get the correct results, a more complicated state machine is needed in the controller to obtain the right output from the extended pipeline.

C. Parallel Pipelines

Alternatively, multiple pipelines can be implemented in parallel. Figure 5 shows such implementation with two identical parallel executing pipelines represented in different colour. The set of pipelines shown in the same colour indicates that the same pipeline is reused on subsequent input data sets. The two parallel executing pipelines are neither data dependent nor instruction dependent and can be treated as two completely isolated accelerators. They are repeatedly doing the same operations with different input data sets.

Each pipeline in the parallel pipeline structure is executing exactly the same operations as the single pipeline. Since n pipelines are executing in parallel, parallel pipelines can achieve $n\times$ speedup compared to single pipeline.

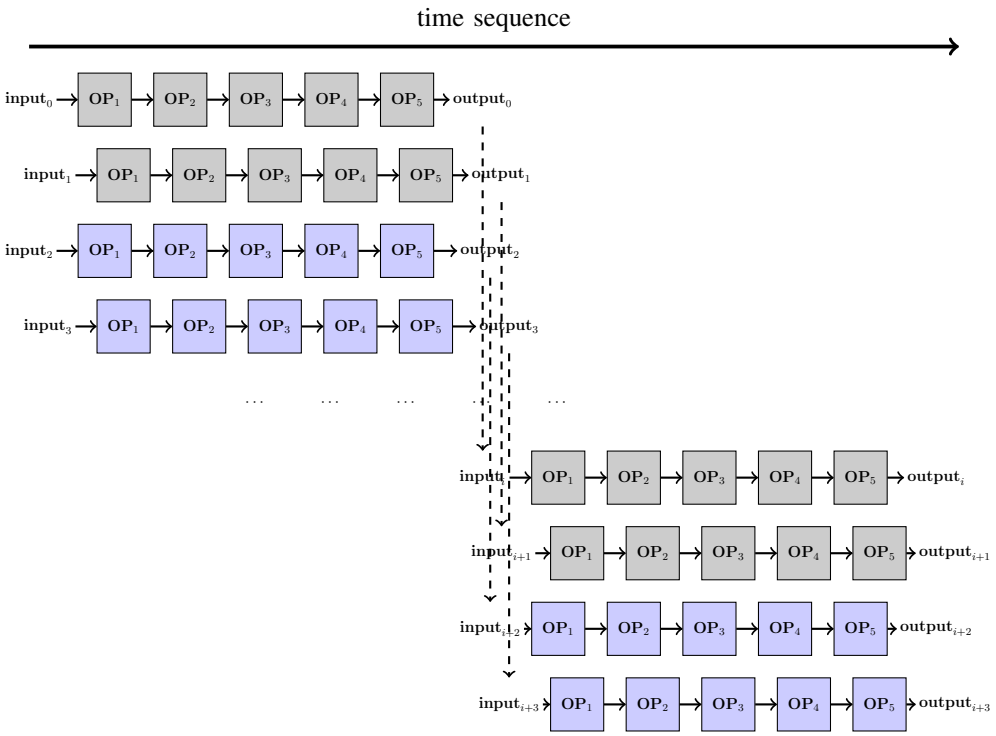


Figure 5: Parallel Pipeline Flow (Different colours represents different pipelines executing in parallel)

Therefore, its speedup compared to the non-pipelined structure is:

$$SpeedUp_{para-pipe} = \frac{ExecTime_{non-pipe}}{ExecTime_{para-pipe}} = \frac{p^2 \cdot t \cdot n}{p \cdot t + p - 1}$$

For two parallel pipelines each with 100 stages, the speedup with 1000 iterations against a non-pipelined computation is 199.8.

The advantages of parallel pipelines are that they are easy to implement and the same principle can be used across multiple FPGA boards. Therefore, the parallel pipeline is selected for implementation and evaluation.

D. Implementation

The implementation of the parallel pipeline is based on the basic HAM model that is discussed in Section III. The controller and the hardware accelerator are multiplied by n and they are interconnected with the onchip memory individually. Each controller and hardware accelerator are associated with an ID which determines the chunk of the cells in the onchip memory that the accelerator deals with and the bits of the control signal the controller corresponds to.

The HDL codes and configurations are generated by ODoST that are ready for Qsys [38] Integration. The Qsys configuration is then modified by increasing the on-chip memory size and creating multiple hardware accelerators mapped to the on-chip memory. Each controller/accelerator is operating individually and in parallel with no interaction with other controllers/accelerators. Therefore, the only change the software module needs to determine is when all the accelerators finish their work. To achieve this, individual controller signals are aggregated to a global signal and the software module reads the global signal to receive the computation completion indication.

To automate the process, the ODoST can be configured based on n, the number of pipelines. The templates including the Qsys configuration template and the software module template can be adjusted to suit the parallel pipelines as required.

VII. Evaluation

This section undertakes an experimental evaluation of the three proposed optimisation strategies. The experiments will use the strategies on two selected biomedical models. The optimisations are used in conjunction with the

Pipelines	1	2	3
Input (bytes)	80	160	240
Output (bytes)	104	208	312
Addition	49	98	147
Subtraction	34	78	102
Multiplication	60	120	180
Division	28	56	84
Exponential Function	25	50	75
Logarithm	1	2	3
Power Function	1	2	3

Table VII: Operations and I/O of Beeler-Reuter models show increasing linearly with the number of pipelines

Model	TNNP (original)	TNNP (optimised)
Input (bytes)	252	252
Output (bytes)	336	336
Addition	114	129
Subtraction	64	91
Multiplication	156	166
Division	129	84
Exponential Function	52	51
Logarithm	4	4
Power Function	26	2

Table VIII: Operations and I/O of an optimised TNNP model against the original model

previously proposed ODoST software that automatically creates the HAMs for the two models. The HAMs and optimisation technologies are assessed according to their resource usage, processing speed and power efficiency. The processing speed and power efficiency are also compared against CPU and GPU implementations of the models.

A. Experimental Setup

Two biomedical models are selected for the evaluation:

- Beeler-Reuter model developed by Beeler and Reuter [39] describing the membrane action potentials of mammalian ventricular myocardial fibres;
- TNNP model[3] developed by Tusscher et al. [40] describing action potentials in human ventricular tissue.

The Beeler-Reuter model was selected because it has GPU results available for comparison. The model has low to medium complexity and the auto generated HAM fits well on the DE4 board. In fact, according to previous results, only 33% of the resources (with DSP usage being the highest) are used and the other resources remain idle [5] . Given these available resources, we employ them to instantiate multiple pipelines as discussed in Section VI, using the parallel pipeline approach.

The Beeler-Reuter model with two parallel pipelines HAM and three parallel pipelines HAM are evaluated in the experiments and the required operations and I/O of the model are listed in Table VII.

The TNNP model was selected due to its high complexity. Indeed, the model is so large that its HAM with the initial ODoST generation does not fit onto the Stratix IV EP4SGX530 board used in the evaluation [5]. So in this evaluation, the C code equations of the TNNP model are first optimised using the equations optimisation (Section IV) and then reformulated with the proposed resource fitting approach (Section V).

Table VIII compares the operations and the I/O of the original C code of the model with the optimised C code. As expected, there is no change in the I/O, but there are noticable changes in the number of operations. The optimised code uses more additions, subtractions and multiplication, however it significantly reduced the much more resource hungry division and power function operations. Essentially, the latter has been replaced in most cases with multiplication, leading to the drop from 26 to only 2. This will significantly reduce the FPGA resource requirements as shown in the next section. As before, ODoST is used to generate the HAM from the optimised C code.

The CPU test platform is an Intel Xeon E5-4650 @2.7 GHz with eight cores and 16 hardware threads [41]. It is selected due to its higher core counts and multi-socket capability compared to desktop-grade CPUs. It is a faster CPU than the one used for the host machine in the FPGA test platform. In addition, this system has the Intel

[3]https://models.physiomeproject.org/exposure/140813f9584b1108c1e7decf0a6f8099

Model	TNNP (without balancing)	TNNP (with balancing)
ALUTs	42%	67%
DLRs	82%	88%
DSPs	88%	59%

Table IX: Estimated resource consumption of TNNP HAM before and after resource allocation optimisation (Estimates calculated as sums of resource usages for each operation).

compiler installed which is one of the faster compilers for x86 and supports comprehensive auto-vectorisation using Streaming SIMD Extensions. The pure software implementations are compiled with icc version 14.0.2 running on a Linux 2.6.32-358 64-bit kernel. For each biomedical model, two software test cases are measured for comparison with the relevant HAM: single thread with optimisation and sixteen threads with optimisation.

The results of the Beeler-Reuter Model are also compared to previous GPU results of Shubhranshu [42]. The GPU test platform that was used was an NVidia Tesla C2070 GPU with 448 Streaming processor cores and 6GB of GDDR5 memory [43] attached to a system with an Intel Xeon X5650 @2.67 GHZ with 6 cores and 12 GB of DDR3 RAM. Shubhranshu developed an unoptimised and automated GPU implementation and a hand optimised GPU implementation. The GPU device to host computer transfer rate configured in his experiment was 8Gb/s.

B. Synthesis Results

In these experiments, we use the Quartus compiler to convert the synthesizable hardware modules, generated by ODoST, into output files for device programming. As before, a script generated by ODoST is used to automate the compilation processes using the Analysis & Synthesis, the Fitter, the Assembler, and the TimeQuest Timing Analyzer modules. The synthesis results are used here to estimate the resource consumption and clock frequency of the HAMs. For completeness, the usage of the Altera FPGA specific ALMs are also included in the resource analysis (see the resource discuss in Section V).

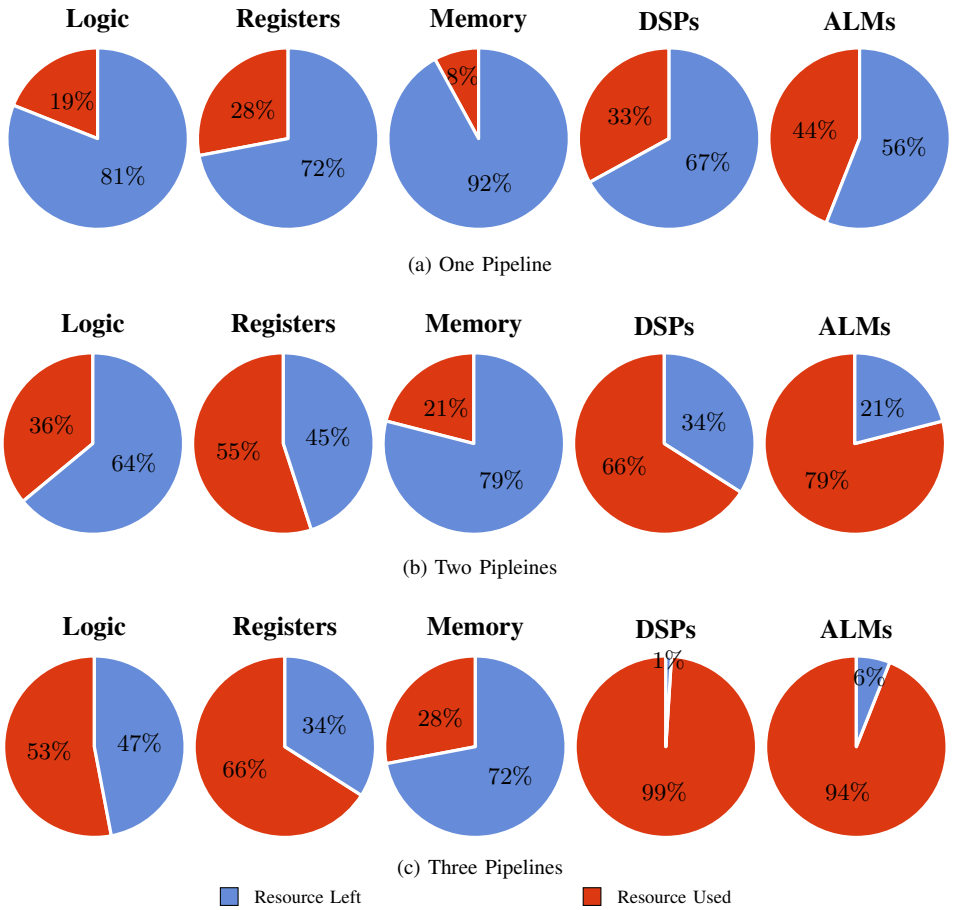
Resource Consumption: The estimated resource consumptions are obtained from the Quartus Fitter. The resources are divided into categories of Logic, Registers, Memory, DSPs and ALMs. The total device capacities are listed in Table I. “Logic” refers to the Combinational ALUTs, “Registers” refers to the Dedicated Registers, “DSPs” refers to the DSP blocks implemented by 18x18 hardware multipliers, “Memory” refers to the memory bits and ALMs refers to the Adaptive Logic Modules.

For the Beeler-Reuter model, the resource consumptions of the HAMs with one, two and three pipelines are represented as a percentage of the total device capacity in Figure 6. According to the results, the resource usage grows with the number of pipelines, but not in a strictly linear fashion. Logic, Registers and Memory usage grows slower and only the DSP usage grows in a strictly proportionally manner. This can be explained by the complex relation between ALMs and the other resource categories. Some minor equation reformulation was applied for the three pipelines implementation since ALM usage did overflow without it, which is very difficult to predict due to the multiple roles of ALMs.

For the TNNP model, the resource consumption of the non-optimised and optimised HAMs is illustrated in Figure 7. The non-optimised HAM does not fit onto the DE4 board since it uses up all the available DSPs and has an overflow for the ALMs. The optimised HAM for the TNNP model using the proposed equations optimisation, however, fits onto the DE4 board well.

Since the TNNP model with equation optimisation is already sufficient to execute the hardware accelerator on the DE4 board, it is not necessary to perform a further resource balancing optimisation on the model, unless the resource usage could be reduced to under half of the total resource capacity so that parallel pipeline optimisation, implementing two pipelines, can be used. We analysed the optimised TNNP model (i.e. after equations optimisation) with the exhaustive resource allocation algorithm for multipliers, divisions, exponential functions, logarithms and power functions. As shown by the resource estimation in Table IX, although the optimisation with resource allocation algorithm can achieve better resource balance, especially between the logic and DSPs, it is still not possible for two models to fit on the same FPGA.Therefore, the resource allocation optimisation is not adopted for the TNNP model.

Interestingly, there are differences between the estimated resource usage and the resource usage in the synthesis reports of the Quartus Synthesiser. The consumption of DLRs and DSPs after synthesis are less than the estimated usage, since the Quartus Synthesiser performs a further resource optimisation step through register and memory packing [44] on the overall HAM. The ALUT consumption after synthesis is more than the estimate,because the



(a) One Pipeline

(b) Two Pipleines

(c) Three Pipelines

Figure 6: Synthesis resource usage results of the HAMs for the Beeler-Reuter model

Number of Pipelines	F_{max}(MHz)
1	134.59
2	128.9
3	127.86

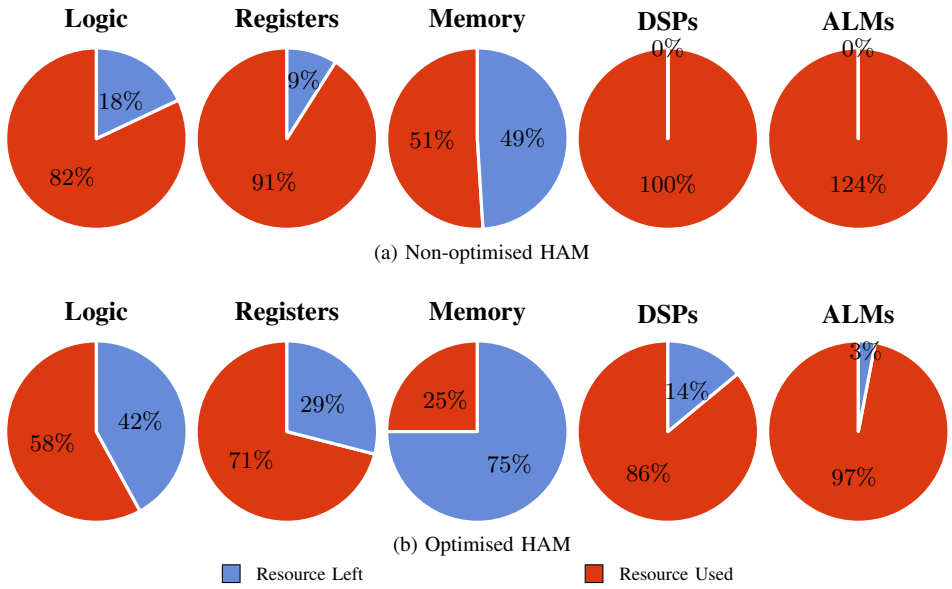
Table X: Predicted clock frequencies for the HAMs of the Beeler-Reuter model.

estimate does not include other logic components such as the controller, the onchip memory, the DMA and the PCIe IP core.

Predicted Clock Frequency: The predicted maximum clock frequency F_{max} is obtained from the synthesis results performed by Quartus TimeQuest Timing Analyzer. For the design, the operating conditions are set to the slow timing model, with a voltage of 900 mV, and temperature of 85°. Table X displays the frequency values for the different number of pipelines for the Beeler-Reuter model.

The HAMs show good scalability with respect to frequency versus number of pipelines. The frequency used in the implementation is 125 MHz. The predicted F_{max} reaches acceptable frequencies between 125 MHz and 135 MHz with reasonable fall-off for more pipelines. The maximum frequency drop-off is around 5%. This is a very small drop compared to the increase in complexity and hence performance. With a fully pipelined and parallel design in the HAMs, the throughput for the three pipeline implementation approximates 3 $cell/cycle$ during the entire computation. This is a significant performance advantage compared to the throughput for the single pipeline implementation which is approximately 1 $cell/cyle$.

The predicted maximum clock frequency for the optimised HAM of the TNNP model is 126.31MHz. Comparing to the models previously evaluated [5], there is a slight fall in F_{max}. It is reasonable and reflects the complexity of the design compared to the others. Using an FPGA at the upper limit of its capacity usually leads to a drop in



(a) Non-optimised HAM

(b) Optimised HAM

Figure 7: Synthesis resource usage results of the non-optimised and optimised HAMs for the TNNP model.

frequency as the placement of components cannot be fully optimised for speed by the fitter.

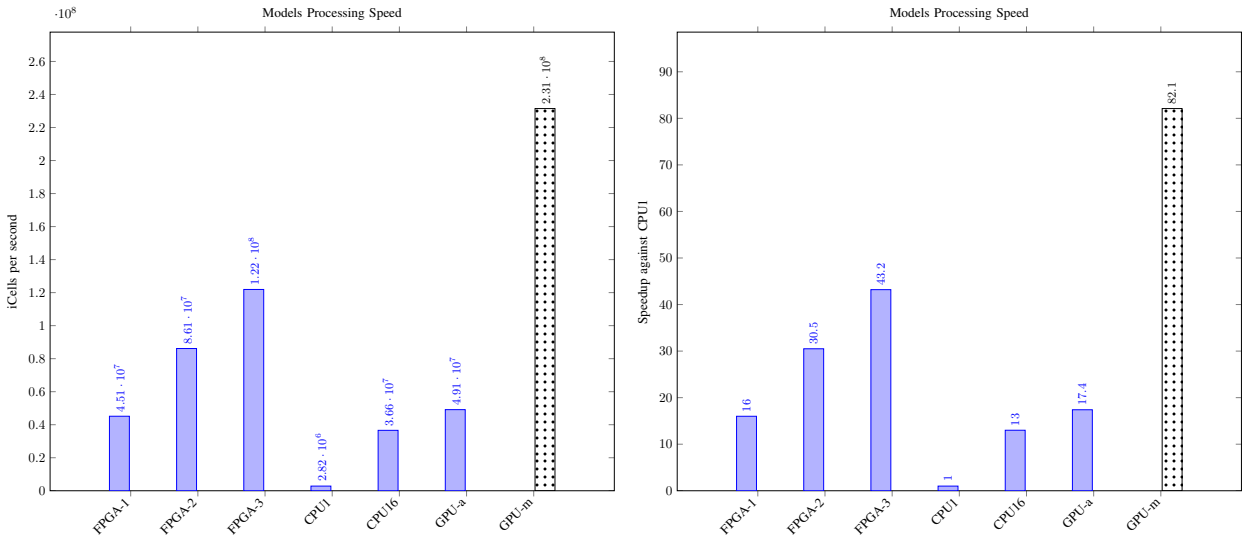
C. *Performance Results*

The performance of the HAMs are presented as their processing speed. For both the Beeler-Reuter model and the TNNP model, the processing speed measures throughput as the number of micro time step cell integrations per second. To simplify, we define the unit $iCells/s$ which stands for iteration cells per second. The results are compared to the CPU implementations of the two models. For the Beeler-Reuter model, the results are also compared against a GPU implementation [42].

Figure 8 presents the processing speed of the Beeler-Reuter model across the different implementations. Figure 8a presents the throughputs across the implementations in the unit of iCells per second. Figure 8b displays speedup against the CPU1 implementation. Each test case measures a biomedical simulation of 1 ms duration with 1 μs micro time step integration for 537,000 cells (number of pipeline stages (179) times maximum capable number of pipelines (3) times 1000). The hand optimised GPU implementation is only used here for a general comparison as all the other implementations in the evaluation are fully automated (or can be fully automated).

As shown in the figures, the two pipeline implementation achieves 1.91 speedup and the three pipeline implementation achieves 2.71 speedup compared to the single pipeline implementation, hence the results are within 10% of the theoretical optimal value. This is a reflection of the increase in the communication overhead since although the pipelines are executing in parallel, the data transfer is still in serial and a n-pipelines computation requires n times of data to be transferred. The three pipeline HAM implementation with most resource utilisation displays a great performance advantage compared to the CPU implementations (43.2x speedup) and automated GPU implementation (2.5x speedup). It even reaches just over half of the processing speed compared to the hand optimised GPU implementation.

Figure 9 presents the processing speed of the TNNP model on the FPGA and CPU platforms. Each test case measures a biomedical simulation of 1 ms duration with 1 μs micro time step integration for 364,000 cells (number of pipeline stages (364) times 1000). FPGA denotes the results for the HAM implementation. The other notations are as before. Figure 9a presents the throughput across the implementations in the unit of iCells per second. Figure 9b displays speedup against the CPU1 implementation. The HAM implementation has significant performance advantage over all the CPU implementations with nearly a 26x speedup compared to the single threaded implementation with SSE optimisation and 2.4x speedup compared to the sixteen threaded implementation with SSE optimisation.



(a) iCells per second

(b) Speedup against CPU1

FPGA-n: HAM implementation, where n represents the number of pipelines

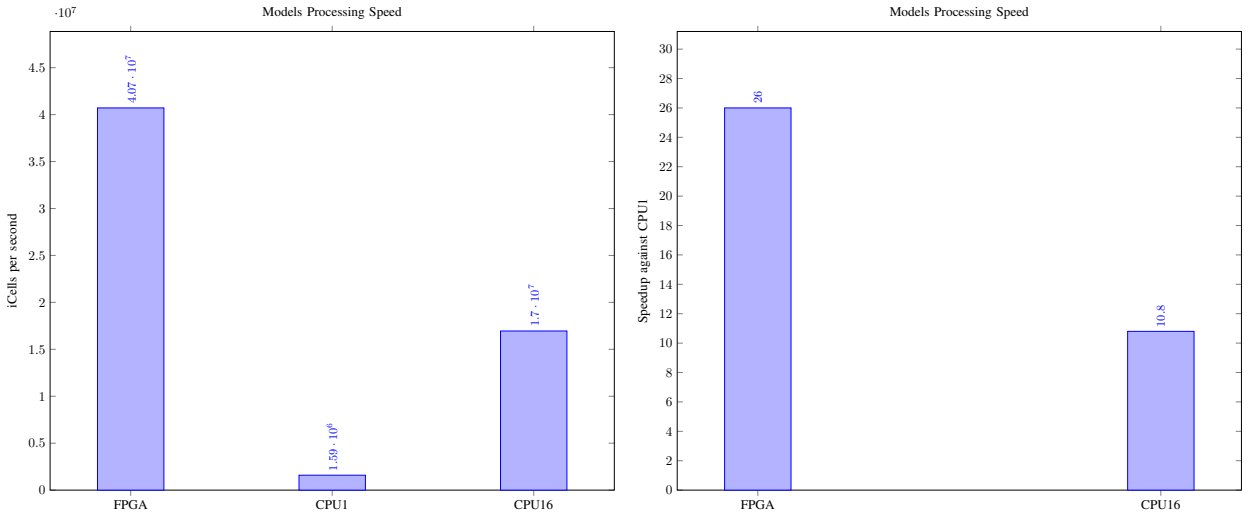
CPU1: SSE4.2 optimised, one thread

CPU16: SSE4.2 optimised, sixteen threads, OpenMP

GPU-a: Auto generated GPU impelmentation

GPU-a: Manual GPU impelmentation with hand optimisation

Figure 8: Processing speed of the HAMs compare to the CPU and GPU implementations for the Beeler-Reuter model (the bar with dotted pattern represents the hand optimised GPU implementation where all the other implementations in the evaluation are fully automated or can be fully automated).



(a) iCells per second

(b) Speedups against CPU1

Figure 9: Processing speed of the HAMs compare to the CPU implementations for the TNNP model.

Testing Platform		Power Measurement (W)	Measurement Basis
Stratix IV EP4SGX530	One Pipeline	15	PowerPlay Power Analyzer
	Two Pipelines	19.2	
	Three Pipelines	24.1	
Xeon E5-4650		130	Thermal Design Power
Tesla C2070		238	Thermal Design Power

Table XI: Power requirement for the Beeler-Reuter model on the three testing platforms.

D. Power Efficiency

For the Beeler-Reuter model power efficiency is compared between the HAMs, the best performing CPU implementation (CPU16) and the CUDA-based GPU implementations. The power requirement for the three testing platforms is shown in Table XI. Since resources are not fully consumed in the FPGA, the FPGA power usage is estimated by Altera's PowerPlay Power Analyser. The PowerPlay Power Analyser supports accurate power estimations and is executed at the post-fit phase of the design cycle. The estimated power requirement for the three HAM implementations are 15 W, 19.2 W and 24.1 W respectively. The power usage increases with increased resource consumption. For the triple pipelined HAM implementation, both the ALMs and the DSPs are approaching the resource capacity. The power estimation of 24.1 W is close to 25 W, the maximum power consumption allowed for a x8 PCI Express card [45]. To allow a fair comparison with the CPU and GPU, we assume the worst case for the FPGA and specify the device power characteristics to maximum and junction temperature to the maximum. The CPU power usage is estimated at 130 W and the GPU power usage is estimated at 238 W, both using the Thermal Design Power (TDP). The TDP of a device is the maximum amount of heat generated by the device that the cooling system is required to dissipate in a typical operation [46]. The TDP should be a good estimate for power consumption for the CPU during cell computation and integration, because the repeated use of SIMD instructions usually employs the CPU the TDP limit [47]. For the GPU, the hand optimised implementation is likely to work at the TDP limit. For auto generated GPU implementation the power consumption estimate might be less accurate. For that reason, we also include in our comparison the hand optimisted GPU version, even though all other implementations are mostly generated automatically.

The power efficiency of the Beeler-Reuter model on each platform is measured by the processing speed obtained from Figure 8 divided by the power requirement for each type of implementation from Table XI. The resulting values in iCells per watt second are converted to iCells per kWh and are presented in Figure 10. The results show that across the three FPGA implementations, the triple pipeline HAM implementation is the most power efficient. Although Table XI shows that the power usage increases with increased resource consumption, its growth rate does not compete with the performance increase. Therefore, there is still a trend of improved power efficiency with an increasing number of pipelines, but obviously not as much as the improvement of processing speed. Compared to the CPU16 and GPU implementations, the FPGA implementations show significantly better power efficiency. The triple pipelined HAM implementation is 18x more power efficient than the CPU16 implementation and is still 5.2x more power efficient than the hand optimised GPU implementation, despite the non-automatic nature of this implementation.

For the TNNP model, power efficiency is compared between the HAM and the best performing CPU implementation (CPU16). The power requirement for the two testing platforms is shown in Table XII. Since both the DSPs and ALMs are approaching the resource capacity of the FPGA, the FPGA power usage is specified to 25 W, the maximum power consumption allowed through a x8 PCI Express card [45]. The CPU power usage is estimated at 130 W using the TDP.

Again, the power efficiency of the TNNP model on the FPGA and CPU platforms is measured from the processing speed obtained from Figure 9 divided by the power requirement for each device/model from Table XII. The resulting values in iCells per kWh are presented in Figure 11. The results demonstrate that the HAM implementation is 12.5x more power efficient than the CPU16- implementation, while it also outperformed the CPU16 implementation by more than a factor of two.

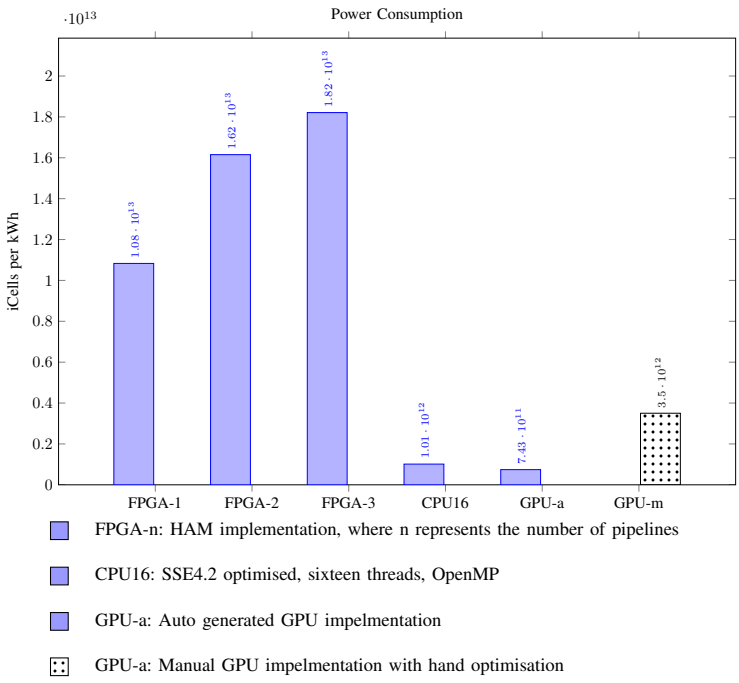


Figure 10: Power consumption of the HAM, CPU and GPU implementations for the Beeler-Reuter model (the bar with dotted pattern represents the hand optimised GPU implementation where all the other implementations in the evaluation are fully automated or can be fully automated).

Testing Platform	Power Measurement (W)	Measurement Basis
Stratix IV EP4SGX530	25	Maximum Power Consumption through x8 PCIe
Xeon E5-4650	130	Thermal Design Power

Table XII: Power requirement for the TNNP model on the two testing platforms.

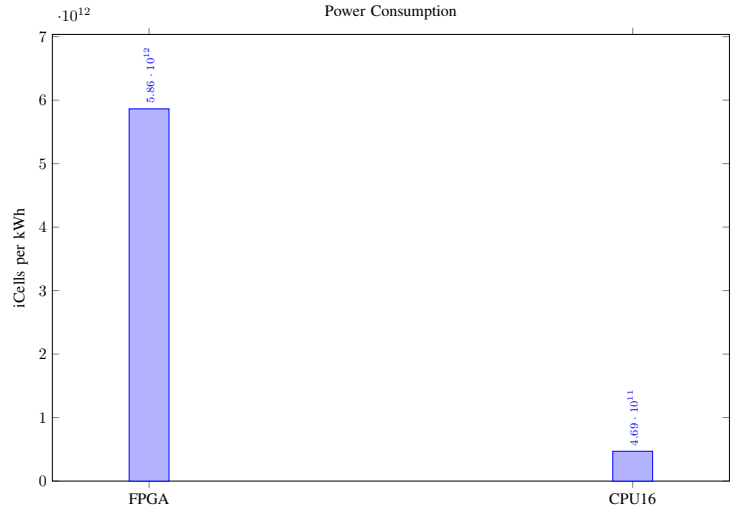


Figure 11: Power consumption of the HAM and CPU implementations for the TNNP model.

VIII. Conclusions

This paper proposes a set of optimisation strategies aimed at reducing resource consumption and increasing performance for the hardware acceleration modules that are generated by ODoST. The strategies are diverse and address the high-level synthesis process at different points: optimising the input, optimising the resource consumption and replicating modules for a better utilisation of the FPGAs. Those strategies are all suitable for automatic high-level synthesis and integrate well into ODoST. After studying the various optimisation approaches, this paper evaluates the optimised hardware accelerator modules for two biomedical CellML models. The results demonstrate that the optimised HAMs with parallel pipelines can provide significant improvements in processing performance and energy efficiency. Apart from the performance improvements, the optimisations are also useful to fit larger CellML models onto an FPGA device. In the future, further optimisations have the potential to improve the performance, e.g., overlapping communication with communication and advancing the compiler and resource fitting optimisations.

References

[1] A. A. Cuellar, C. M. Lloyd, P. F. Nielsen, D. P. Bullivant, D. P. Nickerson, and P. J. Hunter, "An overview of CellML 1.1, a biological model description language," *Simulation*, vol. 79, no. 12, pp. 740–747, 2003.

[2] C. Bradley, A. Bowery, R. Britten, V. Budelmann, O. Camara, R. Christie, A. Cookson, A. F. Frangi, T. B. Gamage, T. Heidlauf, and others, "OpenCMISS: a multi-physics & multi-scale computational infrastructure for the VPH/physiome project," *Progress in biophysics and molecular biology*, vol. 107, no. 1, pp. 32–47, 2011.

[3] TOP500, *TOP500 Supercomputer*, 2014. [Online]. Available: http://www.top500.org/project/

[4] T. Yu, C. Bradley, and O. Sinnen, "Hardware acceleration of biomedical models with opencmiss and cellml," in *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013, pp. 370–373.

[5] Yu, Ting and Bradley, Chris and Sinnen, Oliver, "Odost: Automatic hardware acceleration for biomedical model integration," Tech. Rep., 2014.

[6] NSR Physiome Project, *Mathematical markup language*, 2012. [Online]. Available: http://nsr.bioeng.washington.edu/jsim/docs/MML_Intro.html

[7] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[8] J. Pitt-Francis, P. Pathmanathan, M. O. Bernabeu, R. Bordas, J. Cooper, A. G. Fletcher, G. R. Mirams, P. Murray, J. M. Osborne, A. Walter *et al.*, "Chaste: a test-driven approach to software development for biological modelling," *Computer Physics Communications*, vol. 180, no. 12, pp. 2452–2471, 2009.

[9] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano, "Stochastic simulation for biochemical reactions on FPGA," in *Field Programmable Logic and Application*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds. Springer Berlin Heidelberg, Jan. 2004, no. 3203, pp. 105–114.

[10] Y. Osana, M. Yoshimi, Y. Iwaoka, T. Kojima, Y. Nishikawa, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Kitano, and H. Amano, "ReCSiP: An FPGA-based general-purpose biochemical simulator," *Electronics and Communications in Japan (Part II: Electronics)*, vol. 90, no. 7, pp. 1–10, Jul. 2007.

[11] D. Thomas and H. Amano, "A fully pipelined FPGA architecture for stochastic simulation of chemical systems," in *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2013, pp. 1–7.

[12] J. de Pimentel and Y. Tirat-Gefen, "Hardware acceleration for real time simulation of physiological systems," in *28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2006. EMBS '06*, Aug. 2006, pp. 218–223.

[13] H. Chen, S. Sun, D. Aliprantis, and J. Zambreno, "Dynamic simulation of electric machines on FPGA boards," in *Electric Machines and Drives Conference, 2009. IEMDC '09. IEEE International*, May 2009, pp. 1523–1528.

[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings*, Jan. 2003, pp. 461–466.

[15] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *SOC Conference, 2006 IEEE International*, Sep. 2006, pp. 199–202.

[16] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A high-level synthesis tool for DSP applications," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, Jan. 2008, pp. 147–169.

[17] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[18] Ravikesh Chandra, *Novel Approaches to Automatic Hardware Acceleration of High-Level Software*, 2013.

[19] Altera, "Altera SDK for OpenCL." [Online]. Available: http://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[20] T. Feist, "Vivado design suite," *White Paper*, 2012.

[21] Maxeler Technologies, "MaxCompiler White Paper," Tech. Rep. [Online]. Available: https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

[22] J. Cocke, "Global common subexpression elimination," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 20–24, 1970.

[23] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *Computers, IEEE Transactions on*, vol. 45, no. 3, pp. 294–306, 1996.

[24] T. Reiter, "Optimising code generation with haggies," *Computer Physics Communications*, vol. 181, no. 7, pp. 1301–1331, 2010.

[25] R. Tessier and H. Giza, "Balancing logic utilization and area efficiency in fpgas," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*. Springer, 2000, pp. 535–544.

[26] X. Liang, J. S. Vetter, M. C. Smith, and A. S. Bland, "Balancing fpga resource utilities." in *ERSA*, 2005, pp. 156–162.

[27] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% lut utilization)," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM, 1999, pp. 69–78.

[28] K. E. Atkinson, *An introduction to numerical analysis*. John Wiley & Sons, 2008.

[29] A. Ronacher, *Jinja2 (The Python Template Engine)*, 2011.

[30] LLVM, "LLVM: The -instcombine pass." [Online]. Available: http://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions

[31] A. C.-C. Yao, "On the evaluation of powers," *SIAM Journal on computing*, vol. 5, no. 1, pp. 100–103, 1976.

[32] P. Downey, B. Leong, and R. Sethi, "Computing sequences with addition chains," *SIAM Journal on Computing*, vol. 10, no. 3, pp. 638–646, 1981.

[33] Altera, *DE4 Development and Education Board*, 2011. [Online]. Available: http://www.altera.com/education/univ/materials/boards/de4/unv-de4-board.html

[34] ——, "Altera floating point megafunctions," 2014. [Online]. Available: http://www.altera.com/products/ip/dsp/arithmetic/m-alt-float-point.html

[35] F. de Dinechin and others, *FloPoCo, a generator of arithmetic cores for FPGAs*, 2010.

[36] H. Moore, *MATLAB for Engineers*. Prentice Hall Press, 2014.

[37] I. I. CPLEX, "V12. 1: UserâĂŹs manual for cplex," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.

[38] Altera, *Qsys - Altera's System Integration Tool*, 2014. [Online]. Available: http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html

[39] G. W. Beeler and H. Reuter, "Reconstruction of the action potential of ventricular myocardial fibres," *The Journal of Physiology*, vol. 268, no. 1, pp. 177–210, Jun. 1977.

[40] K. H. W. J. t. Tusscher, D. Noble, P. J. Noble, and A. V. Panfilov, "A model for human ventricular tissue," *American Journal of Physiology - Heart and Circulatory Physiology*, vol. 286, no. 4, pp. H1573–H1589, Apr. 2004.

[41] *Intel® Xeon® Processor E5-4650 (20M Cache, 2.70 GHz, 8.00 GT/s Intel® QPI)*, 2012. [Online]. Available: http://ark.intel.com/products/64622/Intel-Xeon-Processor-E5-4650-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI

[42] Shubhranshu, "Integrating SED-ML and CellML models on the GPU," Tech. Rep., 2011.

[43] *Tesla C2050 / C2070 GPU Computing Processor \textbar NVIDIA UK*, 2011. [Online]. Available: http://www.nvidia.co.uk/object/product_tesla_C2050_C2070_uk.html

[44] Altera, “Differences in logic utilization between quartus ii & synplify report files,” 2002.
[45] Z. Schoenborn, “Board design guidelines for pci express architecture,” in *PCI-SIG APAC Developers Conference*, 2004.
[46] D. Processor, “Power and thermal management in the intel® core tm,” *Intel® Centrino® Duo Mobile Technology*, vol. 10, no. 2, p. 109, 2006.
[47] Intel®, “Optimizing performance with intel advanced vector extensions,” http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf, 9 2014.