# A High-Level Design Framework for the Automatic Generation of High-throughput Systolic Binomial-Tree Solvers

Aryan Tavakkoli and David B. Thomas

***Abstract*—The binomial-tree model is a numerical method widely used in finance with a computational complexity which is quadratic with respect to the solution accuracy. Existing research has employed reconfigurable computing to provide faster solutions compared to general-purpose processors, but they require low-level manual design by a hardware engineer, and can only solve American options. This paper presents a formal mathematical framework that captures a large class of binomial-tree problems, and provides a systolic data-movement template that maps the framework into digital hardware. The article also presents a fully-automated design flow, which takes C-level user descriptions of binomial trees, with custom data types and tree operations, and automatically generates fully-pipelined reconfigurable hardware solutions in FPGA bit-stream files. On a Xilinx Virtex-7 xc7vx980t FPGA at a 100-MHz clock frequency, we require 54-$\mu$s latency to solve three 876-step 32-bit fixed-point American option binomial trees, with a pricing rate of 114k trees/s. From the same device and in comparison to existing solutions with equivalent FPGA technology, we always achieve better throughput. This ranges from 1.4× throughput compared to a hand-tuned register-transfer level systolic design, to 9.1× and 5.6× improvement with respect to scalar and vector architectures, respectively.**

***Index Terms*—binomial-tree numerical method, hardware design automation, field-programmable gate arrays (FPGA), reconfigurable hardware accelerators, high-level synthesis (HLS), option pricing, systolic arrays**

## I. Introduction

THE binomial-tree numerical method is used to solve option pricing problems which have no closed-form analytical solutions. The model is widely employed in finance for pricing many types of options [1], such as American and Bermudan options. Workloads range from making a single option evaluation for a trader, to performing millions of evaluations in order to obtain the implied volatility of underlying assets. The method models the price of underlying assets in discretised time and space domains, where in an $n$-step binomial tree, $O(n^2)$ node evaluations must be performed. Problems with higher $n$ produce more accurate results, but also require more computation, making the classic CPU-based solvers very slow when dealing with thousands of options [2].

In the past few years, the inherent parallelism of option valuations has been exploited using reconfigurable computing platforms. Some binomial-tree FPGA implementations use small degrees of spatial parallelism [3]–[6], so although they offer considerable acceleration over general purpose processing approaches, they still require latencies in $O(n^2)$. In contrast, the systolic approach [7] achieves $O(n)$ latency and gives orders of magnitude acceleration over previous FPGA solutions. However, in this design [7], hand-coding the entire solution in register-transfer level (RTL) not only provides a low-level interface to the application user, but also makes it difficult to reuse the design. Besides, all of the aforementioned methods only work for American options, which limits the scope of pricing problems they can cover. Automated frameworks [8] have been created which generate FPGA accelerators for different options with different asset price models, but they only use Monte Carlo methods and do not provide the widely-used binomial-tree solvers.

This paper presents a novel high-level design framework for creating binomial-tree solvers. The framework lets the end-user describe any problem that can be discretised to a binomial tree in a high-level form (like C) and provides automatically, with the press of a button, a massively parallel systolic FPGA solution, ready to be loaded onto reconfigurable hardware.

The main contributions of this paper are:

1) A framework for precisely defining binomial trees using a mathematical description which allows a large and important class of lattice-based problems to be captured.
2) A parametrisable data-movement template which represents the framework in digital hardware, which is fully pipelined using a systolic architecture.
3) A fully-automated design flow which converts user-defined problems described using the framework into high performance systolic reconfigurable hardware solvers, using the parametrisable data-movement template.
4) An evaluation of the approach showing it can maintain high performance with up to 90% of FPGA resource utilisation, with a rate of more than 114k options/s for 876-step trees on a Virtex-7 device. We also show the breadth of the problems that can be solved with the framework, including multiple financial option types.
5) A comparison to existing reconfigurable hardware approaches, showing that the improved design productivity does not limit performance, with throughput improved by orders of magnitude on modern 28-nm FPGAs and achieving better latency compared to previous hand-tuned systolic designs.

In this article, Section II describes the binomial-tree numer-

Manuscript received March 30, 2017; revised September 2, 2017; accepted October 5, 2017. *(Corresponding author: Aryan Tavakkoli.)*

The authors are with the Circuits and Systems Research Group, Department of Electrical and Electronic Engineering, Imperial College London, London, SW7 2AZ, U.K. (e-mail: aryan.tavakkoli10@imperial.ac.uk, d.thomas1@imperial.ac.uk).

ical method and the motivation behind this research. Section III suggests a formal framework that captures the binomial-tree problems. Section IV presents the hardware architecture we designed to implement the framework on reconfigurable hardware. Section V explains the automated design flow used to generate hardware solutions. Section VI contains performance and accuracy results for different problems and data types. Section VII explains how the framework compares with existing FPGA solutions in terms of performance and productivity measures. Section VIII concludes the paper.

## II. Background

### A. The Binomial-Tree Option Pricing Model

An option is a contract between a *writer* and an investor (option *holder*), which gives the investor the right, but not the obligation, to buy from or sell to the writer some underlying asset(s) at an agreed-upon price (*strike price*) by a specific date (*expiry date* or *maturity* date). In case of buying, it is a *call* option, while with a *put* option the holder can sell the asset. The option is a valuable instrument due to the flexibility for the holder in whether or not to *exercise* the right to trade the asset. Therefore, options are traded for a price, which is derived from multiple parameters, including those specified by the sides of the contract, such as the expiry date and the strike price. Other parameters are taken from market conditions, such as the asset's current price (*spot price*) and the interest rate.

Options are classified into several categories with respect to different features that the contract can have, such as the allowed exercise times, or the way the option pay-off is calculated. For instance, *European* options can only be exercised at the expiry date, while an *American* option can be exercised at any time before the option matures. There are other types of options with exotic exercise features, such as *barrier* options, where the pay-off depends on whether the underlying reaches a predetermined price. Many types of options, such as the widely-traded American options, have no analytical solutions and are priced using numerical methods. For instance, when the contract has complicated exotic features or there are multiple sources of uncertainty in modelling the asset price, Monte Carlo methods are often used [9]–[12]. These methods have the main disadvantage of being slow compared to other numerical models for American-style options [13], while also early exercise features are either difficult or impossible to implement using Monte Carlo methods [14]. In such cases, other numerical methods such as tree-based models are used.

The binomial-tree model, as a tree-based method, discretise time and asset price into a regular lattice, then consider the possible paths that the asset price can take. While the binomial-tree model is applied to different types of options, we take the American option as the standard use-case in order to describe the method. The model is based on the assumption that at any time step the asset price can either increase by a factor $u$ or decrease by a factor $d$, with probabilities $p$ and 1-$p$, respectively. This is shown generically in Figure 1(a). To construct a tree, the three parameters $u$, $d$ and $p$ must be known, which requires three equations. Two of the equations are obtained by matching the expected return and the variance of the tree model with those in a risk-free world. For the

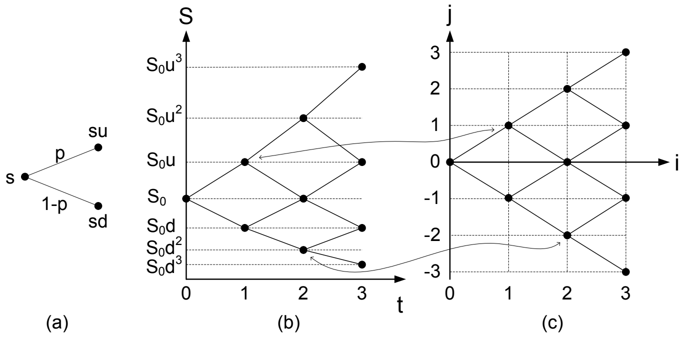

Fig. 1. Binomial tree of $n$=3 (a) One time step (b) Asset space (c) Coordinate space

third equation, a standard approach is to use the Cox-Ross-Rubinstein [15] formula where $ud = 1$. The tree parameters are therefore:

$$u = e^{\sigma\sqrt{\delta t}}, \quad d = 1/u, \quad p = \frac{e^{r\delta t} - d}{u - d} \tag{1}$$

where $\sigma$ is the volatility of the asset price, $r$ is the risk-free interest rate, $\delta t$ is the time step defined as $\delta t = T/n$ and $T$ is the time to expiration. To construct a tree, the starting point is the node at time 0 with a spot price value $S_0$, as demonstrated in Figure 1(b). Nodes in the next time steps are constructed according to the 1-step model in Figure 1(a) until it reaches the expiry date. After the tree is constructed, the pay-off of exercising the option at the expiry at any node is calculated, which is $\max(s - K, 0)$ for a call option, where $s$ and $K$ are the node and strike prices, respectively. The pay-offs are then discounted backwards until time 0 where the option price is returned. For an American option, the discounted value at every node is compared against the pay-off from an early exercise, and the higher of the two is taken as the value of that node. The node values are defined in Equation 2, where $V_{0,0}$ is the final option price.

$$V_{i,j} = \begin{cases} B_j, & \text{when } i = n \\ \max\begin{pmatrix} e^{-r\delta t} p V_{i+1,j+1} \\ + \\ e^{-r\delta t}(1-p) V_{i+1,j-1} \end{pmatrix}, B_j\Bigg) & \text{otherwise} \end{cases} \tag{2}$$

where

$$i \in \{0, 1, ..., n\}$$

$$\forall i : j \in \{-i, -i+2, ..., i-2, i\}$$

The pay-off values are:

$$B_j = \begin{cases} \max(K - S_0 u^j, 0), & \text{put options} \\ \max(S_0 u^j - K, 0), & \text{call options} \end{cases} \tag{3}$$

### B. Design Architecture Space

Many different hardware architectures have been used to implement the binomial-tree model in computing platforms, with task parallelism and data movement significantly impacting the performance of each solution. We will now briefly explore these architecturally distinct hardware solutions. We define a

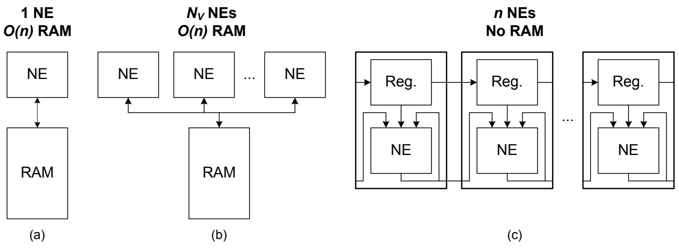

Fig. 2. Architectural design space for binomial-tree solvers. (a) Scalar Design (b) Vector Design (c) Systolic Design - *NE*: Node Evaluator. *Reg.*: registers

*node evaluator* (hereafter NE) as a hardware computing unit which can evaluate the value of a single node in the tree. At any time in the tree, there are up to $n$ *intermediate node values* which can be computed in parallel, but this decreases during backwards induction as $t$ goes to zero. Maximum task parallelism can therefore be achieved by using $n$ NEs. We refer to the $2n$+1 $B$ parameters as the *spatial constants* as their values depend only on their spatial location, shown in Figure 1 as the $j$ index. In solving binomial trees, we lay out the following architectural design space:

*1) Scalar Approach:* A *scalar* approach uses a single NE for the whole tree. Dedicated control logic is required to provide the evaluator with read/write access to an $O(n)$ array of memory elements, as illustrated in Figure 2(a). With a latency of $O(n^2)$, this approach is slow compared to modern parallel implementations. The use of single-core CPUs and FPGA solutions [3], [4] with one NE per pricing solver are examples of taking this classical approach.

*2) Vector Approach:* The *vector* approach uses a small number $N_V$ of NEs operating in parallel to reduce the latency of the solver to $O(\frac{n^2}{N_V})$, as shown in Figure 2(b). Similar to the scalar form, the vectorised evaluators use a central memory to read and update intermediate node values. This introduces a high memory bandwidth requirement which degrades performance for large trees. For this reason, previous FPGA solutions [5], [6] have not used more than $N_V$=10 node evaluators in their vector approach.

*3) Systolic Approach:* The *systolic* approach provides significant acceleration [7] over the scalar and vector forms. While the systolic method also works on parallel NEs, the acceleration comes from two major sources: the large number of compute resources in modern FPGAs and the provision of custom data-movement structures. The first argument states that the large number of compute resources in modern FPGAs matches the size of those trees which provide sufficient accuracy for real applications. As a consequence, full-exploitation of the resources provide fully-parallel binomial solvers. The systolic design, is therefore, based on using exactly $n$ parallel NEs, as shown in Figure 2(c). Also, the systolic nature of data-movement provides zero-overhead in accessing the model constants, leaving a hardware solution with an $O(n)$ latency.

## C. Motivation: Combining Performance and Productivity

The scalar and vector approaches are slow as they require long execution runs to produce accurate results. The previous systolic approaches [7], on the other side, significantly lacked design productivity and user customisability, as they were only able to solve a single problem, namely American options, and only in fixed-point precision. The problem and the hardware solution were both defined using hardware description languages (HDLs). Therefore, adaptation of the design for other types of binomial trees (like barrier or Bermudan options) required hand-coded RTL in almost every part of the design, from arithmetic operations to data-movement architecture. In the following Sections III to V, we provide a design framework which automatically generates hardware solutions for any binomial tree defined in the new formal framework. The framework gives the user the flexibility to either define a single binomial problem, like a barrier option, or multiple trees (such as American and barrier options) to be priced simultaneously.

## III. Formal Framework

In this section, we propose a high-level *mathematical framework* in order to capture binomial-tree problems. The main feature of this framework is that it forms the problem in a way that can be easily mapped into parallel computing models. We will also propose a computing *design framework*, which enables representing the mathematical model in the computing world (using programming languages). The design framework can automatically turn user-defined computing abstraction of the problem into parallel reconfigurable hardware solutions. The end-user is not required to define or understand hardware implementation details. Such a decoupling of the user-defined problem specification from the implementation effort increases productivity, while also ensuring high performance.

### A. Mathematical Model

We define a binomial-tree problem as a quintuple $(\mathcal{P}, \mathcal{G}, \mathcal{S}, \mathcal{N}, \mathcal{R}, \mathfrak{g}, \mathfrak{s}, \mathfrak{f}_\mathfrak{o}, \mathfrak{f}, \mathfrak{r})$ with sets and functions defined in Table I. The tuple defines the problem regardless of the tree size $n$: a free variable which sets the level of discretisation. In other words, the binomial-tree problem represented by a tuple $x$, can be solved with one value of $n$ ($n$=$n_1$) or another ($n$=$n_2$). $n$ can be defined by the end-user or the hardware engineer according to accuracy versus execution time requirements. Indices $i$ and $j$ are also free parameters which represent a node's position in the tree and are dependent on $n$.

From the tuple, an instance of the input parameters set $\mathcal{P}$ defines the option(s). Some parameters are used everywhere in the model, such as $u$ in Equation 1. We call them the global constants set $\mathcal{G}$, and they are generated by the function $\mathfrak{g}$. The constants $B_j$ of an American option (Equation 3) imply a set of spatial constants $\mathcal{S}$ whose values depend only on the spatial tree index $j$, and are generated by the function $\mathfrak{s}$. The set of intermediate node values, shown by $\mathcal{N}$ (Equation 2), are produced by the node evaluator function $\mathfrak{f}$. Equation 4 gives the evaluation model of the mathematical framework:

$$\begin{gathered} V_{i,j} = \mathfrak{f}\Big(\mathfrak{g}(P), \mathfrak{s}\big(P, \mathfrak{g}(P), j\big), V_{i+1,j-1}, V_{i+1,j+1}\Big) \\ i \in \{0, 1, ..., n-1\} \\ V_{n,j} = \mathfrak{f}_\mathfrak{o}\Big(\mathfrak{s}\big(P, \mathfrak{g}(P), j\big)\Big) \\ R = \mathfrak{r}\big(\mathfrak{g}(P), V_{0,0}\big) \end{gathered} \tag{4}$$

TABLE I
VARIABLES, SETS AND FUNCTIONS OF THE FORMAL FRAMEWORK – CALC.: CALCULATE

| Symbol | Type | C Declaration | Description | Example: American Option |
|---|---|---|---|---|
| $n$ | Natural number | const int n | Tree size | |
| $i$ | $\{i \in \mathbb{Z} \wedge 0 \leq i \leq n\}$ | int i | Temporal indices | |
| $j$ | $\{j \in \mathbb{Z} \wedge -n \leq j \leq n\}$ | int j | Spatial indices | |
| $\mathcal{P}$ | Finite set | struct P_t | Input parameters | $\{S_0, K, r, \sigma, T \in \mathbb{R},\ type \in \{-1, 1\}\}$ |
| $\mathcal{G}$ | Finite set | struct G_t | Global constants | $\{\delta t, u, w_d, w_u \in \mathbb{R}\}$ |
| $\mathcal{S}$ | Finite set | struct S_t | Spatial constants | $\{B_j \in \mathbb{R}\}$ |
| $\mathcal{N}$ | Finite set | struct N_t | Node values | $\{V_{i,j} \in \mathbb{R}\}$ |
| $\mathcal{R}$ | Finite set | struct R_t | Results | $\{Q \in \mathbb{R}\}$ |
| $\mathfrak{g}$ | $\mathcal{P} \mapsto \mathcal{G}$ | G_t fGlobal(P_t) | Calc. global constants | $u = e^{\sigma\sqrt{\delta t}}, w_d = e^{-r\delta t}(1-p), w_u = e^{-r\delta t}p$ |
| $\mathfrak{s}$ | $\mathcal{P} \times \mathcal{G} \times j \mapsto \mathcal{S}$ | S_t fSpatial(P_t, G_t, j) | Calc. spatial constants | $\max\{type * (S_0 u^j - K), 0\}$ |
| $\mathfrak{f}_0$ | $\mathcal{S} \mapsto \mathcal{N}$ | N_t fNodeInit(S_t) | Give initial node values | $B$ |
| $\mathfrak{f}$ | $\mathcal{G} \times \mathcal{S} \times \mathcal{N} \times \mathcal{N} \mapsto \mathcal{N}$ | N_t fNode(G_t, S_t, N_t, N_t) | Evaluate nodes | $\max\{w_d V_d + w_u V_u, B\}$ |
| $\mathfrak{r}$ | $\mathcal{G} \times \mathcal{N} \mapsto \mathcal{R}$ | R_t fResult(G_t, N_t) | Extract results | $V$ |

where $V_{i,j} \in \mathcal{N}, P \in \mathcal{P}, R \in \mathcal{R}$. Node values at time $T$ are chosen from the set of spatial constants using the function $f_0$. In some cases the node value at time 0 is not the final result of the solver. With an American option, for instance, prices can be normalised by dividing initial node values by the asset price $S_0$ and multiplying back $V_{0,0}$ with $S_0$ [7]. Function $\mathfrak{r}$ is defined in such cases to generate the set of final results $\mathcal{R}$ by transforming the node value at time 0. Capturing binomial-tree problems by the model provided in Equation 4 provides an easy way of mapping the problem abstraction to systolic hardware architectures, as will be described in the next section.

## B. Minimal Executable Implementation

To clarify how the model is executed, we now give a minimal sequential realisation of the framework written in C++ shown in Figure 3, which we call as the *C Solver*. The actual definitions of the C data structures and C functions are provided by the user, and we refer to them as the *User C Model*. The User C Model defines the framework's sets and functions (Table I), and is opaque to the framework. When the User C Model is created, the C Solver can be compiled into different platforms, such as CPUs and GPUs, depending on the user's cost and performance requirements. When the user chooses an FPGA platform, our design framework does *not* use the C Solver to generate a hardware solution, but instead uses the User C Model to automatically create an FPGA solver. The C Solver in our framework is merely used for software modelling and verification purposes. This way, the problem representation is made easy by only asking the user to define a few C data structures and functions.

```
R_t BinomialTree_Solver (P_t P) {
  S_t S[2*n+1];         // Spatial constants
  N_t V[n+1], V_next[n+1], V_tmp; // Node Values
  G_t G = fGlobal(P); // Calculate global constants
  // Calculate spatial constants
  for (int j=-n; j<=n; j++) {
    S[j+n] = fSpatial(P, G, j);}
  // Give initial node values
  for (int j=-n; j<=n; j+=2) {
    V[(j+n)/2] = fNodeInit(S[j+n]);}
  // Node evaluations
  for (int i=n-1; i>=0; i--) {  // Time loop
    for (int j=-i; j<=i; j+=2) {  // Space loop
      V_tmp=fNode(V[(j+i)/2],V[(j+i)/2+1],G,S[j+n]);
      V_next[(j+i)/2] = V_tmp;}
    // Update current node values
    memcpy(V, V_next, sizeof(V));}
  // Extract final results
  return fResult(G, V[0]);}
```

Fig. 3. The C Solver: A C++ Implementation of the Formal Framework

## IV. HARDWARE ARCHITECTURE

To realise our framework in digital hardware, we propose an architecture in this section based on the systolic approach. We describe the customisability of the hardware architecture in terms of user-defined functionality and data-movement channels. To explain the parallelism behind the systolic architecture, Algorithm 1 is presented. It is an alternative form of the model in Figure 3, but is defined using the framework notations of sets and functions, and uses parallel constructs. Stepping the node values backwards occur sequentially in $n$ time steps, as shown in function *NodeEvaluation* (line 12) with a parallel inner loop (line 15). The parameters used throughout the node evaluations (sets $G$, $S$ and $V_n$) are calculated in $n$+1 steps with a single *for* loop in function *ConstantCalculation* (line 2). The sequential calls (line 21) to *ConstantCalculation*, *NodeEvaluation* and the result extraction function $\mathfrak{r}$ completes the pricing model.

## A. Base Systolic Approach

The parallel form of the binomial-tree method shown in Algorithm 1 demonstrates the systolic nature of the model in terms of generation and consumption of the constants, and through their ordered access pattern. A systolic binomial-tree solver [7] uses an array of $n$ parallel *systolic cells* with one NE in each cell. In the first phase of a coarse-grained pipeline, the cells receive streams of problem constants (*propagation phase*) over $n$+1 time steps, as shown in Figure 4. In the

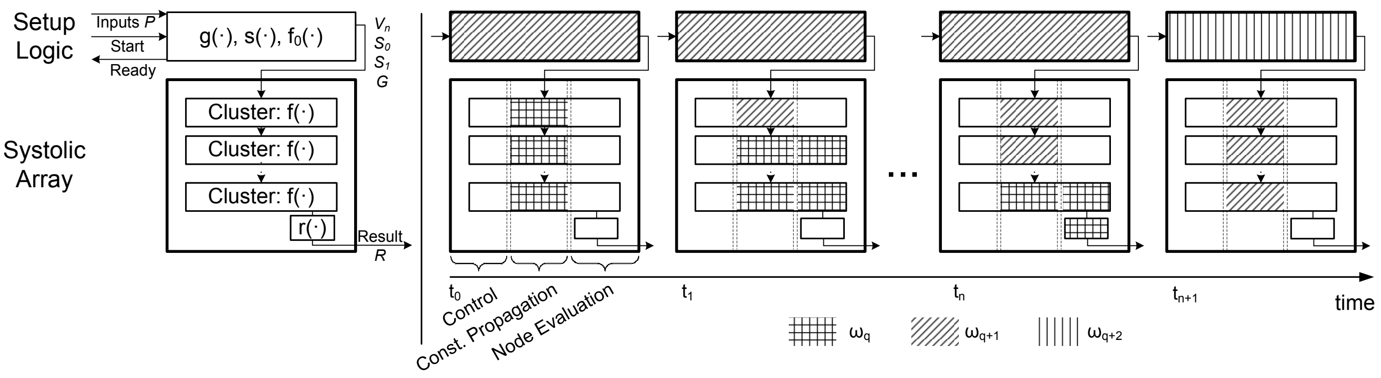

Fig. 4. Systolic approach applied to our framework: Hardware blocks (left) Timing diagram of a systolic cycle (right). Cluster contains $\mathfrak{f}$. $\omega_q$ denotes data or active processes related to problem $q$.

**Algorithm 1** Parallel Binomial-tree Solver

```
1:  input $n, P$
2:  function CONSTANTCALCULATION($P$)
3:      $G = \mathfrak{g}(P)$
4:      for $k = 0$ to $n$
5:          $j \leftarrow -n + 2k$
6:          $S_j \leftarrow \mathfrak{s}(P, G, j)$
7:          $S_{j+1} \leftarrow \mathfrak{s}(P, G, j+1)$
8:          $V_{n,j} \leftarrow \mathfrak{f}_0(S_j)$
9:      end
10:     return $(G, S, V_n)$
11: end function
12: function NODEEVALUATION($G, S, V_n$)
13:     for $k = 0$ to $n - 1$
14:         $i \leftarrow n - 1 - k$
15:         parfor $j = -i$ to $i$, $j = j + 2$
16:             $V_{i,j} \leftarrow \mathfrak{f}(G, S_j, V_{i+1,j-1}, V_{i+1,j+1})$
17:         end
18:     end
19:     return $V_{0,0}$
20: end function
21: main
22:     $(G, S, V_n) \leftarrow$ CONSTANTCALCULATION($P$)
23:     $V_{0,0} \leftarrow$ NODEEVALUATION($G, S, V_n$)
24:     return $\mathfrak{r}(G, V_{0,0})$
25: end
```

second pipeline phase, the systolic array then performs node evaluations in exactly $n$ time steps (*evaluation phase*), after which the solver result is generated. With separate propagation and evaluation phases for a problem in the coarse-grained pipeline, while one problem is in its evaluation phase, the constants of a second problem are propagated through the array. The periodic occurrence of the two phases is called a *systolic cycle*, where each cycle takes $n$+1 time steps.

The block on the top in Figure 4, called *setup logic*, provides streams of constants (function *ConstantCalculation* in Algorithm 1). While the latency of the node evaluation phase in the systolic array is $n$ time steps (line 13), the setup logic divides the $2n$+1 spatial constants into two streams of lengths $n$ and $n$+1 words. Using two streams means that the latency of the setup logic is matched with that of the systolic array so that the coarse-grained pipeline (lines 22 & 23) has almost no bubbles, providing higher throughput. Control in the systolic array is localised in order to avoid global fan-outs to the large number of cells, enabling decent clock rates to be achieved. Hence, a small number of cells, known as the parameter $b$, are grouped into *clusters*, and are controlled by a local controller (Figures 4 & 5). The variable $b$ must be limited to avoid long combinational delays due to large local fan-outs. It also must not be very low so that unnecessary area is dedicated to local controllers. In practical cases [7], $b$ is less than 10. The value of $b$ and the size of target reconfigurable platform define how many clusters are implemented.

### B. Systolic Hardware Realisation of the Framework

To realise our framework in systolic hardware we propose an architecture based on the systolic model laid out in Algorithm 1 and Figure 4. The architecture must be a generic template which is customisable in terms of the user-defined functionality and data-movement channels. In simpler terms, the systolic architecture is a data-movement template with functionality holes (user-defined functions) and variable-width and length systolic communication channels between the holes, and between the I/O and the holes. Our design framework must take the User C Model, fill in the functionality holes and customise the data-movement channels. The main challenges in designing such a data-movement architecture are linked to the implementation of the following: (a) *hardware blocks* representing the user-defined C functions (we call them *user-defined fn. blocks*), (b) *for loops* in Algorithm 1, (c) the *mechanism* which provides streaming constants to the systolic array at the right time of a systolic cycle. To gain sufficient accuracy, the hardware resources must be employed for the systolic array in an efficient way so that maximum number of NEs are implemented. This means that the setup logic must take a relatively small area compared to the systolic array. We will now describe these challenges in more detail and explain how we have overcome them.

*1) User-defined Function Blocks:* To implement the five user-defined fn. blocks in the framework with acceptable clock rates there are two options. One is to synthesise the blocks into finite state machines, which adds handshaking interface

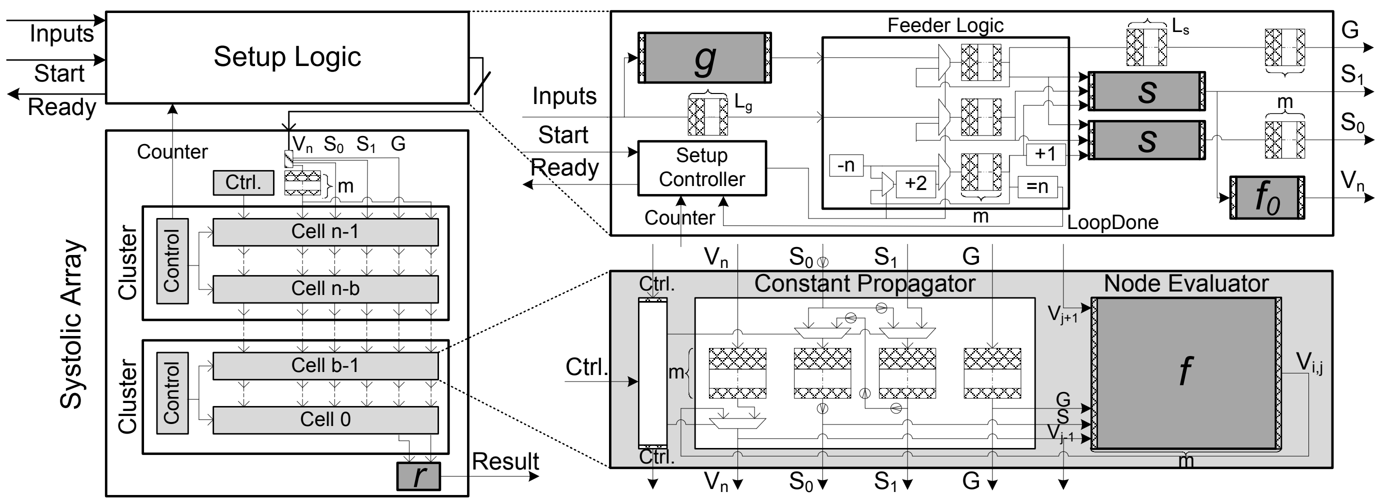

Fig. 5. Hardware Architecture of the Systolic Binomial-Tree Solver. Shaded boxes: registers. Dark grey boxes: user-defined fn. blocks.

logic to the blocks. The second approach, is to turn the blocks into pipelines which increases the efficiency by using the resources in every clock cycle. We chose the second approach, as it simplifies the design and increases the throughput in a C-slow [16] fashion. This way, $m$ problems can be priced concurrently, which we refer as one *problem set*.

Existing HLS tools can be used to generate the user-defined fn. blocks from the User C Model, but we must ensure they can be pipelined. To direct the HLS tool to make pipelines, the user C functions must be in the Static Single Assignment (SSA) form. SSA means each variable must be assigned exactly once and defined before it is used. If the code is not in the SSA form, loops are synthesised to blocks with initiation intervals of more than one, and hence multiple trees cannot be handled this way. Higher $m$ increases the throughput and allows for higher clock rates. When there is a clock frequency target defined by the user, the HLS tool chooses the best $m$ to meet the timing constraint. A great feature of our framework is that the user is not required to define any form of task scheduling.

*2) Systolic Array:* The outputs of the *ConstantCalculation* function (Algorithm 1) are fed to the systolic array in four streams: $G \in \mathcal{G}$, $S_0, S_1 \in \mathcal{S}$ and $V_n \in \mathcal{N}$. The streams pass through every cell, as shown in Figure 5. Intermediate node values are transferred systolically between the cells. To account for the concurrent evaluation of $m$ trees, the function block $\mathfrak{f}$ and the constant propagator are each C-slow pipelined [16] with $m$ levels of registers (Figure 5).

The *parfor* loop in Algorithm 1 (line 15) is represented by the parallel $n$ systolic cells. The sequential outer loop (line 13) is executed during the evaluation phase, when one less NE is active as the number of nodes in the tree decreases by one in every time step. When the node evaluation is active the cell is in *evaluation mode*, and when it is only propagating constants for the next problem set it is in *propagation mode*. In evaluation mode, the multiplexers create a sequential path between the shift registers of the two spatial constant channels, as shown by small arrowed circles in Figure 5. During the propagation mode, in contrast, spatial constants are carried in parallel through the shift registers. With this higher bandwidth provided, the $2n$+1 spatial constants can be present in $n$ cells in $n$+1 clock cycles.

At the beginning of a systolic cycle, all cells make a simultaneous transition from propagation to evaluation mode, using signals generated by the Control block in every cluster (Figure 5). During the systolic cycle, the gradual transition from evaluation to propagation modes of the cells is managed by a single signal, which is generated by the Ctrl. block at the beginning of the systolic cycle. This signal shifts through cells one by one, letting a new cell make a transition every $m$ clock cycles. The array initially starts operation with a power reset and then follows periodic systolic cycles forever, regardless of whether constants are streamed into it. The challenge is to know when the setup logic should start processing, so that constants are streamed at exactly the right clock cycle. For this purpose, the Counter of a local Control block in the array is monitored by the setup logic (Figure 5).

*3) Setup Logic:* The setup logic is responsible for transforming the input parameters into $2n$+1 streaming constants fed to the systolic array. This is done by applying the user-defined fn. blocks $\mathfrak{g}$, $\mathfrak{s}$, $\mathfrak{f}_0$, and by using dedicated logic for running the *for* loop in the *ConstantCalculation* function (Algorithm 1 line 4). The setup logic also provides I/O flags to inform the user logic when to enter the input parameters, which are usually in the orders of kilo-bytes and are often read from a host processor software application or from a network port. The variables of the *for* loop are stored in the registers of the Feeder Logic, which is controlled by the Setup Controller. The spatial indices $j$ start from $-n$ and a comparator detects $+n$ when the loop ends. At this time, the feedback signal *LoopDone* is triggered indicating to the Setup Controller that the parameters for all the loop iterations has been fed to the $\mathfrak{s}$ blocks. Pipeline registers with lengths $L_s$ and $L_g$ in the setup logic ensure routes have the same latency. The $m$-levels deep registers on the streams $G$ and $S_0$ make them reach the systolic array in the correct time, compared to the streams $S_1$ and $V_n$.

*4) Performance Model:* We define the performance of the binomial-tree solver based on three metrics as defined in Table II: *tree latency*, *tree throughput* and *node throughput*. The

TABLE II
PERFORMANCE MODEL OF THE HARDWARE ARCHITECTURE - CC: CLOCK CYCLES. LTY.: LATENCY

| Parameter | Symbol | Unit | Expression | Definition |
|---|---|---|---|---|
| Problem size | $n$ | $time\ steps$ | | Size of binomial tree |
| Concurrency parameter | $m$ | $trees$ | | Number of trees solved concurrently |
| Implementation frequency | $f$ | $MHz$ | | Clock frequency of the implemented design |
| Lty. of setup logic | $L_{SL}$ | $CC$ | $m + L_g + L_s$ | Lty. of setup logic: port Inputs to port $S_1$ (Fig. 5) |
| Lty. of array constant transfer | $L_{CT}$ | $CC$ | $m(n+1)$ | Lty. of loading constants into systolic array (Fig. 4) |
| Lty. of array node evaluation | $L_{NE}$ | $CC$ | $m.n$ | Lty. of NEs producing a result (Fig. 4) |
| Tree latency | $L$ | $s$ | $\big(L_g + L_s + 2m(n+1)\big)/f$ | Input-output lty. of solving a single binomial tree |
| Node throughput | $S$ | $nodes/s\ (n/s)$ | $n.f/2$ | Avg. number of node evaluations per unit of time |
| Tree throughput | $Thp$ | $trees/s\ (t/s)$ | $f/(n+1)$ | Sustained avg. no. of solved trees per unit of time |

table contains other parameters such as $L_g$ and $L_s$ for the latencies of the user-defined fn. blocks $\mathfrak{g}$ and $\mathfrak{s}$, respectively. The tree latency $L$ is the sum of the setup logic latency and the latencies of the two phases in the systolic array: constants transfer and node evaluation, i.e. $L = (L_{SL}+L_{CT}+L_{NE})/f$, which simplifies to the expression given in Table II. As shown in Figure 4, the NEs on average are utilised during half of the systolic cycle, assuming a full work-load of incoming binomial-tree parameters; the node throughput is hence $nf$ divided by 2. The tree throughput $Thp$ is the rate of trees solved per second. In this case, we can feed $m$ problems in every iteration of the coarse-grained pipeline which has a latency of $L_{CT}$ (Figure 4) with a clock frequency $f$. Therefore, $Thp = fm/L_{CT}$ which simplifies to $Thp = f/(n+1)$.

## V. DESIGN FLOW

We now propose a methodology which takes (a) user-defined problem instances (User C Model), with (b) hardware representations of the data-movement architecture (Figure 5), and then automatically produces an FPGA bit stream.

### A. User-Defined Problem Instance

Based on the formal framework presented in Section III, the end-user needs to define three categories of design inputs in the User C Model: *problem size*, *data structures* and *framework functions*, as listed in Table I. An example User C Model is given in Figure 6 for an American option whose mathematical model was presented in Section II. This example demonstrates the simplicity of defining problem instances in our design framework. In choosing the problem size, the user has two options. With the first option, the user can instruct the design framework to choose the maximum $n$ that fits the FPGA target. For this option, the framework first automatically implements the setup logic and a small systolic array (e.g. $n$=100) as two separate designs. Following that, based on the resource usage of these two implementations, and the device's available resources, the design framework estimates a maximum $n$ for the target, and implements a design with that maximum $n$, all of which is automated. With this feature, the design framework provides a solution with the best accuracy to the user. For the second option, if the framework is not instructed to estimate the maximum $n$, the user has the flexibility to choose a desired $n$ based on the accuracy and performance requirements.

```
const int n = TreeSize;
// Data Structures:
struct P_t {
  float So, K, r, sg, T;
  bool OT; /*Option type; call: true, put: false*/};
struct G_t {
  float u, w1, w2;};
struct S_t {
  float Csk;};
struct N_t {
  float v;};
struct R_t {
  float result;};
// Functions:
G_t fGlobal (P_t P) {
  float u = exp((P.sg) * sqrt((P.T)/n));
  float a = exp((P.r) * ((P.T)/n));
  float p = (a-(1/u)) / (u-(1/u));
  return G_t{u, (1/a)*(1-p), (1/a)*(p)};}
S_t fSpatial (P_t P, G_t G, int j) {
  float S = (P.So) * pow(G.u,j); // Pipelined power
  return S_t{fmax(0,(P.OT?S:P.K)-(P.OT?P.K:S))};}
N_t fNode(N_t N1, N_t N2, G_t G, S_t S) {
  return N_t{fmax(S.Csk, G.w1*N1.v + G.w2*N2.v)};}
N_t fNodeInit(S_t S) {
  return N_t{S.Csk};}
R_t fResult(G_t G, N_t N) {
  return R_t{N.v};}
```

Fig. 6. An example User C Model: Standard American option defined in C++

### B. Automation Methodology

Our proposed automation methodology is shown in Figure 7. The flow uses a data-movement hardware description of the systolic architecture (Figure 5) which we built in RTL, called the *Design Template Files* in Figure 7. The User C Model consists of header and source files (C++), and is used by the *HLS Tool*, along with *Run Script* and project setting files, to generate user-defined fn. blocks (*RTL Function Blocks*). We observed in our experiments that some HLS tools do *not* provide the I/O port connections of their output RTL file in a structured form. They rather expand the I/O connections across the individual variables (or signals in RTL) of every structure used for that function. This is shown in Figure 8 for the NE function $\mathfrak{f}$ in the *HLS-Generated Fn. Block*. On the other side, the Design Template Files (Figures 7 and 8) are problem independent and have implicit references to the user-defined fn. blocks. Therefore, one challenge is to apply the explicit user-dependent RTL function blocks, to the generic

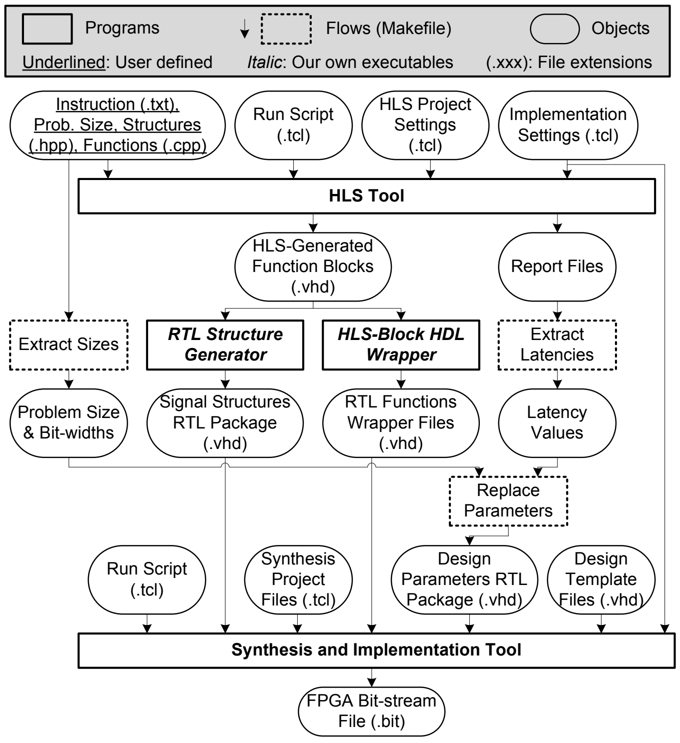
Programs
Flows (Makefile)
Objects
Underlined: User defined
Italic: Our own executables
(.xxx): File extensions
Instruction (.txt), Prob. Size, Structures (.hpp), Functions (.cpp)
Run Script (.tcl)
HLS Project Settings (.tcl)
Implementation Settings (.tcl)
HLS Tool
HLS-Generated Function Blocks (.vhd)
Report Files
Extract Sizes
RTL Structure Generator
HLS-Block HDL Wrapper
Extract Latencies
Problem Size & Bit-widths
Signal Structures RTL Package (.vhd)
RTL Functions Wrapper Files (.vhd)
Latency Values
Replace Parameters
Run Script (.tcl)
Synthesis Project Files (.tcl)
Design Parameters RTL Package (.vhd)
Design Template Files (.vhd)
Synthesis and Implementation Tool
FPGA Bit-stream File (.bit)

Fig. 7. Fully-automated flow, producing FPGA bit stream from user code

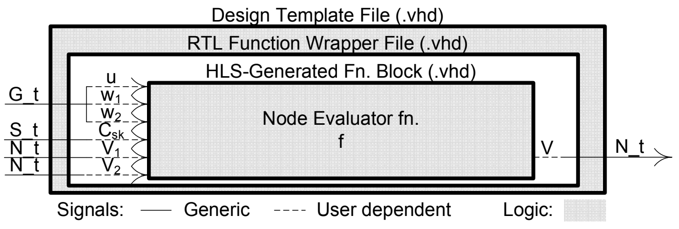
Design Template File (.vhd)
RTL Function Wrapper File (.vhd)
HLS-Generated Fn. Block (.vhd)
Node Evaluator fn.
f
G_t
S_t
N_t
N_t
u
w1
w2
Csk
V1
V2
V
N_t
Signals: Generic User dependent
Logic:

Fig. 8. The HDL Wrapper file acting as an intermediate layer

problem-independent data-movement template.

To address the above challenge, an intermediate RTL layer is required to map the signal structures to their corresponding signal members: an *RTL Function Wrapper File* (Figure 8). To generate such files automatically, we made a program called *HLS-Block HDL Wrapper* (Figure 7). This program, for each user-defined fn. block, parses through the HLS-generated top-level RTL text file, finds all I/O signals, and creates an RTL wrapper file. The wrapper file has an instantiation of the HLS-generated RTL function, and maps the signal structures to individual signal elements (Figure 8). On the other side, the RTL signal structures used in the data-movement template must be defined. But the HLS tool we used does not provide such definitions. To address this challenge, we made a second program called *RTL Structure Generator*, which parses through the HLS-generated top-level RTL files and creates signal structures in a separate RTL file (*Signal Structures RTL Package* in Figure 7).

## C. Automated Tool Chain

In order to implement our proposed automation methodology, we used the following tools and methods. The tool flow was scripted in a *Makefile* and the GNU *Make* tool was used to run the framework. We used Vivado HLS 2015.4 for high-level synthesis and Vivado 2015.4 for logic synthesis and implementation. To work with the Vivado tools, Tool Command Language (TCL) scripts were used to build projects and run synthesis automatically. We coded our executable programmes in C++, and compiled them using the GNU g++ compiler with the *c++11* switch. The hardware RTL model was described in VHDL. The Design Template Files (Figure 7) have entity declarations of the user-defined fn. blocks, and the actual definitions reside in the RTL Functions Wrapper Files. VHDL *record* types were used to define signal structures. The Design Parameters RTL Package contains parameters for the width and latencies of the data communication channels.

Our tool chain is fully automated. The design framework is executed by running a single command line which asks the Make tool to run the Makefile. The framework then reads all the user-defined and other design files, performs all the steps mentioned in the translation methodology (Figure 7), and produces the FPGA hardware solution in a bit stream.

## VI. Results

In this section we explore the performance and productivity of our framework for contemporary FPGAs. We will implement six different problem instances to evaluate the accuracy and performance characteristics, and to explore the trade-off between accuracy and performance.

## A. Applied Real-world Problem Instances

We have applied our framework to six problem instances of commonly used option types, given in Table III. Although European options are priced analytically, we used it for FPGA area comparison as it requires the least amount of logic compared to other types of options. For the American barrier option problem instance, we used the down-and-out type, which means it cannot be exercised if the asset price falls below a certain barrier. The user can price all four types of barrier options using our framework. In Bermudan options, the value of a tree node is a function of the node's time step, unlike other options with no time dependencies. Pricing of Bermudan options demonstrates that our framework is capable of not only providing spatial constants, but also constants whose values depend on their temporal location in the binomial tree.

We also implemented problem instances with two options defined on a shared asset: *American or barrier*; and *American and barrier*. The former allows the user to choose which option to price at run-time, with the benefit that most of the logic is shared between the two options. The latter problem instance is suitable for users who require pricing of multiple types of options at the same time. To our knowledge, this is the first design framework with the flexibility to price different types of options concurrently using the binomial-tree model.

Inefficient use of the resources in the setup logic can limit the size of systolic array and hence the solution accuracy. For example, the single-precision pipelined power function in Figure 6 (line 21) on its own takes 23 DSP blocks in a modern FPGA running at 100 MHz. For this reason, the arithmetic

TABLE III
APPLIED PROBLEM INSTANCES — $\mathbb{B} = \{0,1\}$, $\mathbb{B}' = \{-1,1\}$

| **Problem Instance** | $\mathcal{P}$ | $\mathcal{G}$ | $\mathcal{S}$ | $\mathcal{N}$ | $\mathfrak{f}$ | $\mathfrak{f}_o$ |
|---|---|---|---|---|---|---|
| American Option | $S_0, K, r, \sigma, T \in \mathbb{R}, ot \in \mathbb{B}'$ | $u, w_1, w_2 \in \mathbb{R}$ | $C_{sk} = \max(0, ot *(S_0.u^j - K)) \in \mathbb{R}$ | $V_{i,j} \in \mathbb{R}$ | $V_{i,j} \leftarrow \max(C_{sk}, w_1.V_{i+1,j-1} + w_2.V_{i+1,j+1})$ | $V_{n,j} \leftarrow C_{sk}(j)$ |
| European Option | $S_0, K, r, \sigma, T \in \mathbb{R}, ot \in \mathbb{B}'$ | $u, w_1, w_2 \in \mathbb{R}$ | $C_{sk} = \max(0, ot *(S_0.u^j - K)) \in \mathbb{R}$ | $V_{i,j} \in \mathbb{R}$ | $V_{i,j} \leftarrow \{w_1.V_{i+1,j-1} + w_2.V_{i+1,j+1})$ | $V_{n,j} \leftarrow C_{sk}(j)$ |
| Bermudan Option | $S_0, K, r, \sigma,$ $T, T_B \in \mathbb{R}$ $ot \in \mathbb{B}'$ | $u, w_1, w_2,$ $\in \mathbb{R}$ $B = n\times$ $T_B/T \in \mathbb{Z}$ | $C_{sk} = \max(0, ot *(S_0.u^j - K)) \in \mathbb{R}$ $t_0 = n \in \mathbb{Z}$ | $V_{i,j} \in \mathbb{R}$ $t \in \mathbb{Z}$ | $s = w_1.V_{i+1,j-1} + w_2.V_{i+1,j+1}$ $m = \max(C_{sk}, s)$ $V_{i,j} \leftarrow (t > B\ ?\ m : s)$ $t \leftarrow t - 1$ | $V_{n,j} \leftarrow C_{sk}(j)$ $t \leftarrow t_0$ |
| American Barrier Option | $S_0, K, r, \sigma,$ $T, B \in \mathbb{R}$ $ot \in \mathbb{B}'$ | $u, w_1, w_2,$ $B \in \mathbb{R}$ | $b = (S_0.u^j \leq B) \in \mathbb{B}$ $C_{sk} = b\ ?\ 0\ :\ \max(0, ot * (S_0.u^j - K)) \in \mathbb{R}$ | $V_{i,j} \in \mathbb{R}$ | $V_{i,j} \leftarrow !b * \max(C_{sk}, w_1.V_{i+1,j-1} + w_2.V_{i+1,j+1})$ | $V_{n,j} \leftarrow C_{sk}(j)$ |
| American or Barrier (Amer.) Option (Shared Asset) | $S_0, K, r, \sigma,$ $T, B \in \mathbb{R}$ $ot \in \mathbb{B}'$ $s \in \mathbb{B}$ | $u, w_1, w_2,$ $B \in \mathbb{R}$ $s \in \mathbb{B}$ | $c = \max(0, ot *(S_0.u^j - K))$ $b = (S_0.u^j \leq B) \in \mathbb{B}$ $C_{sk} = (s\ ?\ c\ :\ !b * c) \in \mathbb{R}$ | $V_{i,j} \in \mathbb{R}$ | $V_{temp} \leftarrow \max(C_{sk}, w_1.V_{i+1,j-1} + w_2.V_{i+1,j+1})$ $V_{i,j} \leftarrow s\ ?\ V_{temp} : !b * V_{temp}$ | $V_{n,j} \leftarrow C_{sk}(j)$ |
| American and Barrier (Amer.) Options (Shared Asset) | $S_0, K, r, \sigma,$ $T, B \in \mathbb{R}$ $ot \in \mathbb{B}'$ | $u, w_1, w_2,$ $B \in \mathbb{R}$ | $C_{sk\|A} = \max(0, ot *(S_0.u^j - K)) \in \mathbb{R}$ $b = (S_0.u^j \leq B) \in \mathbb{B}$ $C_{sk\|B} = (!b * C_{sk\|A}) \in \mathbb{R}$ | $V_{A\|i,j} \in \mathbb{R}$ $V_{B\|i,j} \in \mathbb{R}$ | $V_{A\|i,j} \leftarrow \max(C_{sk\|A}, w_1.V_{A\|i+1,j-1} + w_2.V_{A\|i+1,j+1})$ $V_{B\|i,j} \leftarrow !b * \max(C_{sk\|B}, w_1.V_{B\|i+1,j-1} + w_2.V_{B\|i+1,j+1})$ | $V_{A\|n,j} \leftarrow C_{sk\|A}(j)$ $V_{B\|n,j} \leftarrow C_{sk\|B}(j)$ |
| Adder (Fig. 12) | $p \in \mathbb{Z}$ | $g \in \mathbb{Z}$ | $s \in \mathbb{Z}$ | $V_{i,j} \in \mathbb{Z}$ | $V_{i,j} \leftarrow \{s + V_{i+1,j+1}\}$ | $V_{n,j} \leftarrow s$ |

in the setup logic must be using a data type which finds a good balance between resource usage and accuracy. As a user of the framework, we did all arithmetic in the setup logic using single-precision data types, and converted the constants to their appropriate data types before streaming them. With this approach, we never had more than 3% of resource usage in the setup logic, and in the remaining area we obtained sufficient number of NEs to get good accuracy.

We targeted the modern Xilinx Virtex-7 xc7vx980t FPGA as our default platform, as it is the largest across all 7 Series families with 3600 DSPs, in order to have as many NEs as possible. We verified the functionality of our hardware design using the Vivado Simulator. To reduce compile time, we set the synthesis option *flatten_hierarchy* to *none* which keeps the design hierarchy during synthesis.

We also applied some platform-specific optimisations to the user-level source code to increase $n$ in our implementations. Since the node prices of an option can never have negative values, we used unsigned data types for the fixed-point solvers with an equal number of integer and fractional bits. Also, as the weight constants are always positive fractions, we omitted their integer parts from the data type. For instance, a 32-bit fixed-point implementation has unsigned 16.16 price variables and unsigned 0.16 weight constants.

For the floating-point implementations, we applied the Vivado HLS source-code synthesis directives *FMul_meddsp* (single precision) and *DMul_meddsp* (double precision), to the results of multiplications in the NEs. This had the effect of balancing the use of DSPs and LUTs to implement the multipliers, so that we could increase $n$. With half-precision arithmetic, Virtex 7 can fit large trees up to a point when the pay-offs at the maturity reach infinity, causing the solution to fail. However, since those tree leaves carry small probabilities in the binomial distribution of the prices at the maturity, we can omit them with negligible damage on the accuracy, as will be demonstrated later. Therefore, for the half-precision solvers, we have set all the infinite-valued spatial constants to zero in $\mathfrak{s}$. This is a user-level modification, so can be tested in the C Solver, independent of hardware.

## B. Accuracy Analysis

To evaluate the accuracy of our framework applied to modern FPGAs, we first implemented American option solvers on the Virtex 7 device to find out the maximum achievable $n$ for each implementation. We then simulated the problem instances with the same $n$ values using Vivadlo HLS and its bit-accurate libraries to obtain the accuracy. The inputs were randomly sampled from a parameter space of $(1 \leq S_0 \leq 1000,\ 16^{-1} * So \leq k \leq 16{*}So,\ 0 < r \leq 40\%,\ 0 < \sigma \leq 60\%,\ 0 < T \leq 2)$. This space covers for most applications and is a superset of the spaces used by existing FPGA solvers [7], [13]. The accuracy was calculated based on the average relative error for N=1000 input samples. For each implementation, error is:

$$e = \frac{1}{N}\sum_{i=1}^{N} e_i \tag{5}$$

$$e_i = \begin{cases} 0, & \text{when } R_i, \hat{R}_i < 0.01 \text{ or } \dfrac{R_i}{S_0}, \dfrac{\hat{R}_i}{S_0} < 10^{-5} \\ \dfrac{|R_i - \hat{R}_i|}{R_i}, & \text{otherwise} \end{cases}$$
$$\forall i \in (1, ..., N):\ p_i \in \mathcal{P},\ R_i, \hat{R}_i \in \mathcal{R} \tag{6}$$

where $p_i$: input samples, $R_i$: results, $\hat{R}_i$: reference results. To make the error formula $e_i$ meaningful for financial applications, we forced it to zero when reference and solver results

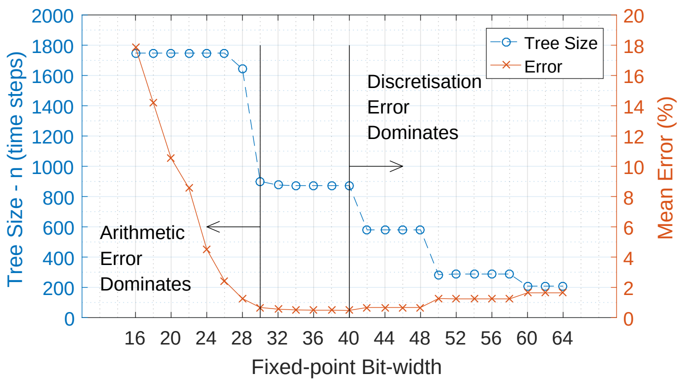

Fig. 9. Accuracy of custom-precision fixed-point American option results on Virtex-7 xc7vx980t, with 95MHz clock frequency for 34 & 50 bits and 100MHz for the rest

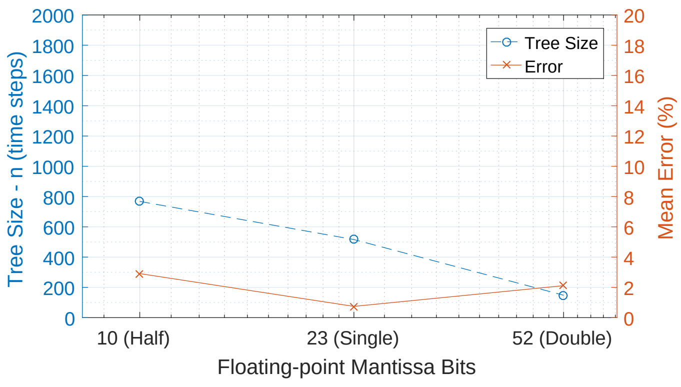

Fig. 10. Accuracy results for floating-point American option solvers on Virtex-7 xc7vx980t, with clock frequencies 100MHz for half & single, and 90MHz for double precisions

are either (a) below 0.01 (1 cent) for very cheap options, or (b) are small compared to the underlying asset price for relatively cheap options. For the reference solver, we used option pricing libraries from the Financial Toolbox in MATLAB R2016a. Tree sizes reported in the literature to provide highly accurate results included values of 10000 [17] and 15000 [2] time steps. We also used 15000 time-step trees in the reference program, using double-precision arithmetic. The accuracy of the solvers we implemented was affected by two major sources of error: discretisation error and arithmetic error. The discretisation error is due the degree on which the binomial tree is descritised in time and space, i.e. higher $n$ produces more accurate results. The arithmetic error comes from the precision chosen for the data variables. In the following, we will analyse our results and show how these types of error affects the accuracy.

*1) Fixed-Point:* We varied the fixed-point bit-width $w$ of the systolic array's data variables from 16 to 64 bits, as shown in Figure 9. For each $w$, we chose $n$ so that the utilisation is near 100%. The utilisation was dominated by DSP blocks, with more than 98% usage for $w$=16:56. For 60 to 64 bits, near-100% utilisation caused significant congestion and failure in routing. For these three cases, therefore, we relaxed the routing by targeting 95% DSP utilisation. Increasing $w$ (less arithmetic error), required more compute resources in NEs, and hence smaller trees were implemented, causing an increase in the discretisation error. Between 16 and 26 bits, although large trees of $n$=1748 were implemented, arithmetic error dominated the accuracy with more than 2% of mean error. The best balance of discretisation versus arithmetic errors was obtained for solvers with $w$=30:48 and $e$<0.7%, which includes the optimum fixed-point solver of $w_{opt}$=40 with $e_{opt}$=0.5%.

*2) Floating-Point:* We implemented floating-point solvers in half, single and double precisions with 96%, 95% and 89% utilisations, respectively, with the single-precision as the most accurate solver as shown in Figure 10. Assuming custom-precision floating-point logic is available, with larger FPGAs, the optimum accuracy point would move from the left of the plot to the right by getting closer to the double-precision point. This means better accuracy is achieved with wider data words as FPGAs grow in size. A strength of our framework is that, since it supports targeting different devices, it can take advantage of the growth in FPGA size to improve accuracy.

## C. Area–Performance Results

While we looked at how the accuracy changes when $w$ is varied, we now want to find out the performance for different problem instances. We also look at how the FPGA resources are shared between different components of the hardware architecture. Table IV demonstrates area and performance results for different problem instances and data types, with maximum $n$ for each implementation. The first four rows are standard American solvers with different data types, whose $n$ and accuracy values were presented earlier in Figures 9 and 10. The bottom group in Table IV with five problem instances demonstrates the breadth of the problems that our framework can generate solutions for. For four of these five solvers, the tree sizes are either close to $n$=500 or well above that. According to Figure 10, such values of $n$ produce accurate results in single precision. We therefore chose the single-precision data type for these implementations. Although the American & barrier problem instance only achieved $n$=289, it was also implemented using single precision so that comparison can be made with the other solvers.

The usage of slices and DSP blocks is balanced in the solvers, as shown in Table IV. We see good performance for all three performance measures (defined in Table II), with tree latencies of 40-200 $\mu$s, and tree throughput values above 100k trees/s. Higher values of $m$ increases latency, but the tree throughput is also enhanced.

## D. Performance–Accuracy Trade-off

We observed previously that changing $w$ affects the maximum $n$ that can fit on the device, and the combination of $w$ and $n$ determines the solver's accuracy. On the other hand, Table II showed that all three performance metrics are directly affected by $n$, so $w$ affects both accuracy and performance. We will now merge these observations, and show a fundamental trade-off between the accuracy and the tree throughput. We will consider the American option fixed-point solvers with different $w$, as shown in Figure 11. The lowest $w$ has the highest $n$ and hence the lowest tree throughput, while it also has the highest error. By increasing $w$, error decreases and as soon as $n$ decreases, the tree throughput is enhanced. For an error value of 0.5% ($n$=872), no higher throughput than 114k

TABLE IV
AREA-PERFORMANCE RESULTS – VIRTEX-7 XC7VX980T WITH 3600 DSP BLOCKS & 153K SLICES – CP: CONCURRENCY PARAMETER

| **Implementation** | | | **Setup Logic** (Single Prec.) | | | | **Systolic Cell** | | **Binomial-Tree Solver - Maximum n** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Data Type** | **Freq.** | **Prob. Instance** | **Latency** | | **Resources** | | **Resources** | | **CP** | **Size** | **Resources** | | **Performance** | | |
| | **f** (MHz) | | **L$_g$** $(CC)$ | **L$_s$** $(CC)$ | **Slice** (%) | **DSP** (%) | **Slice** | **DSP** | **m** | **n** | **Slice** (%) | **DSP** (%) | **L** $(\mu s)$ | **S** $(Gn/s)$ | **Thp** $(Kt/s)$ |
| Fixed 16.16 | 100 | American | 50 | 23 | 2.6 | 2.44 | 205 | 4 | 3 | 876 | 45 | 100 | 54 | 43.8 | 114 |
| Half | 100 | | 51 | 23 | 2.5 | 2.44 | 355 | 2 | 13 | 768 | 96 | 45 | 201 | 38.4 | 130 |
| Single | 100 | | 48 | 21 | 2.5 | 2.44 | 522 | 4 | 10 | 516 | 95 | 60 | 104 | 25.8 | 193 |
| Double | 90 | | 48 | 20 | 2.6 | 2.44 | 859 | 21 | 11 | 148 | 51 | 89 | 38 | 6.6 | 604 |
| Single | 100 | European | 48 | 21 | 2.5 | 2.44 | 434 | 4 | 8 | 708 | 97 | 81 | 114 | 35.4 | 141 |
| | | Bermudan | 48 | 21 | 2.7 | 2.53 | 542 | 4 | 10 | 500 | 95 | 58 | 101 | 25.0 | 199 |
| | | Barrier (Amer.) | 48 | 21 | 2.5 | 2.44 | 579 | 4 | 10 | 516 | 94 | 60 | 104 | 25.8 | 193 |
| | | Amer. or Barr. | 48 | 21 | 2.6 | 2.44 | 509 | 4 | 10 | 504 | 94 | 58 | 102 | 25.2 | 198 |
| | | Amer. & Barr. | 48 | 21 | 2.6 | 2.44 | 955 | 8 | 10 | 284 | 97 | 66 | 58 | 14.2 | 350 |

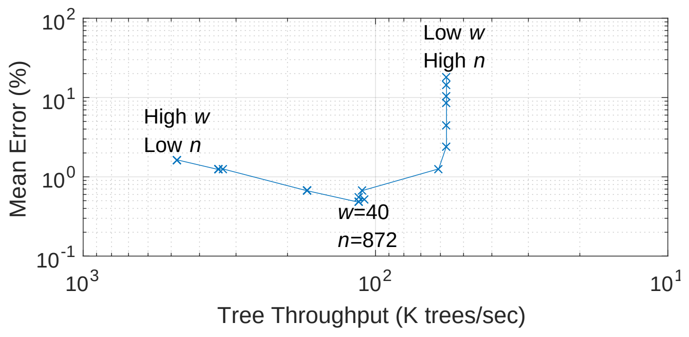

Fig. 11. Throughput-error trade-off for different fixed-point American option solvers with different bit-width on Virtex-7 xc7vx980t with >95% utilisation

trees/s can be achieved. While this point belongs to $w$ of 34 to 40, all the less-accurate and lower-throughput implementations become useless (corresponding to the right-hand side of the optimum point). The highest tree throughput is 478k trees/s for the left-most point in the figure, with a mean error of 1.6%. All solvers between these two points belong to the Pareto frontier which give the best accuracy-performance trade-off. Based on the accuracy-performance requirements, the user can choose the appropriate $w$ to give the best implementation.

### *E. Scaling Limits of the Problem-independent Architecture*

The spatially-distributed independent controllers are meant to protect the systolic array's performance from frequency degradation for very large trees. To verify this, we defined a very small NE: a 4-bit signed adder in $\mathfrak{f}$, defined in the last row of Table III. This ensures that the critical path does not occur within the NE. We also defined a 600 MHz clock constraint, based on maximum achievable clock rate for $n$=1000. The generated NE is pure combinational, and with the registers added by the flow we have $m$=2. Figure 12 demonstrates the frequency and node throughput of the 4-bit adder solvers for very large trees of up to $n$=25000, showing no degradation in the overall behaviour of the clock frequency. The 5% fluctuation in the frequency, between 570 MHz and 600 MHz, is due to the varying value of $n$ which affects the maximum value of the counters used in the Setup Controller. The node throughput values confirm that the systolic infrastructure by its own and independent to the problem instance, provides linear scaling in the overall design performance.

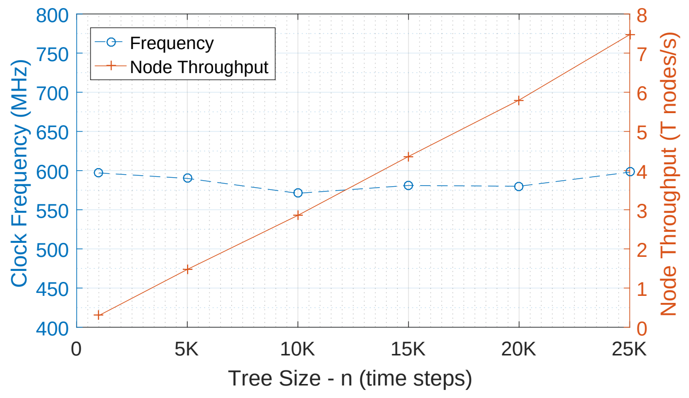

Fig. 12. Scaling the frequency and performance (node throughput) of the problem-independent architecture for designs as large as $n$=25000

## VII. RELATED WORKS

In Section II we classified the computational approaches of the existing binomial-tree solvers into three main categories; scalar, vector and systolic designs. In this section, we will describe the existing FPGA binomial-tree solutions in terms of their computational structures and the design productivity they offered. We will also make comparisons to our work in terms of both qualitative and quantitative measures.

### *A. Qualitative Analysis*

Table V provides measures of design productivity and area-performance complexities for the existing binomial-tree FPGA solvers. Jin [3], [4] presents a scalar FPGA design, based on $R$ replications of an American option pricer. Each pricer consists of one NE which is C-slowed by a fixed factor $C$, preventing them from trading off throughput for latency. They can price $R \times C$ independent options concurrently, with trees of $n \leq$ 1000. Wynnyk [5] presents a vector FPGA design which prices American options by using four parallel NEs ($N_V$=4). They use memory management techniques to solve single trees of up to 64000 time steps in double precision, but do not provide any argument for why such large trees must be evaluated in

TABLE V
COMPARING EXISTING FPGA BINOMIAL SOLVERS. SOM: SCOPE OF MODIFICATION, CLT: CONFIGURABLE $L$-$Thp$, HSTREAMS: HYPERSTREAMS

| | **Design Productivity** | | | | | | **Hardware Architecture** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Problem Coverage** | | **User Interface** | | **Impl. Flexibility** | | **Type** | **Performance** | | **Resource** | |
| | **Instances** | **Size** | **Prog. Lang.** | **SoM** | **Precision** | **CLT** | | **Latency** | **Tree Thrput.** | **Compute** | **Mem.** |
| Jin [3], [4] 2008, 09 | American | Med. | Handel-C, HStreams | global | Fx. 16.16, Sgl., Dbl. | No | Scalar | $O(C.n^2)$ | $O\left(\frac{1}{C.n^2}\right)$ | $O(1)$ | $O(C.n)$ |
| Wynnyk 2009 [5] | American | Lrg. | Verilog Altera IP | global | Double | No | Vector | $O\left(\frac{n^2}{N_V}\right)$ | $O\left(\frac{N_V}{n^2}\right)$ | $O(N_V)$ | $O(n)$ |
| Morales 2014 [6] | American | Med. | OpenCL | global | Single Double | No | Vector | $O\left(\frac{n^2}{N_V}\right)$ | $O\left(\frac{N_V}{n^2}\right)$ | $O(N_V)$ | $O(n)$ |
| Tavakkoli 2014 [7] | American | Med. | VHDL, FloPoCo | global | Fixed | Yes | Systolic | $O(n)$ | $O\left(\frac{1}{n}\right)$ | $O(n)$ | $O(1)$ |
| Ours | Any | Med. | C/C++ | local | Any | Yes | Systolic | $O(n)$ | $O\left(\frac{1}{n}\right)$ | $O(n)$ | $O(1)$ |

practice. While they solve large single trees, there is no way to increase their tree throughput. Their design is coded in Verilog HDL and it is not clear whether they can port their design to different FPGA targets.

The work in [6] by Morales is another vector solver ($N_V$=8) of American options with medium-sized binomial trees ($n$=1024) using GPUs and energy-efficient FPGA implementations. They use the OpenCL language and Altera's OpenCL compiler to synthesise their design. Our own previous American option pricer in [7] provides a systolic reconfigurable architecture with $n$ systolic cells. The work presented fixed-point FPGA solvers with hand-coded RTL blocks and FloPoCo [18] floating-point cores. It gained considerable accelerations, both in terms of latency and throughput, compared to scalar and vector approaches, and showed practical applications for medium-sized ($n$=768) fixed-point trees. However, relocation of the hand-coded RTL solution to other data precisions limited the productivity of this design.

In contrast to the works above, which can only solve American options, our current design framework can take any binomial-tree problem instance and is capable of pricing different types of options concurrently. The user is not required to define the problem using HDL languages and the scope of user modification is limited to certain parts of the framework (User C Model). The user is able to move the design in the latency-throughput space by changing the clock constraint.

## B. Quantitative Comparison

Table VI demonstrates the FPGA implementation details of existing binomial-tree solvers, which includes our American pricing results for comparison. From Jin's results in [3] and [4] with the same architecture, we chose those from [4] as they are better for all performance measures. The value of $C$ is not reported in [4], so we estimated $C$ based on the number of pipeline registers in their diagram ($C$=6), whose generated clock frequency matches what both we and [7] obtained for the same pipeline depth. We have also scaled up their utilisation results by the $R$ value they reported, to account for the maximum concurrency they can achieve. All other data in Table VI are directly copied from the published papers. The variable $m$ is a design parameter in [7] whose value affects the clock frequency. We have included the best-case $m$ in [7] in terms of the frequency.

*1) Accuracy:* Wynnyk [5] has not reported accuracy values of their results, which is acceptable as they price very large trees of up to $n$=64000. In [3]–[5] it has not been mentioned for what input samples they get their reported error values. In [7], the absolute error is reported to be under the application-level boundary of 1 cent, for their reported specific range of input parameters which is a subset of our input sample set. It is true that the values of $n$ in our results is less than of those in most of the related works. However, we showed in Section VI that a single-precision solver of $n$=516 provides enough accuracy for practical applications, particularly for low latency and high throughput use cases.

*2) Performance:* As shown in Table VI, while one previous work used the same FPGA target with a 28-nm technology [7], all other implementations were on FPGAs using older technologies. To compare our results against the latter group, we have taken two approaches. One is to scale down our Virtex-7 results to older Xilinx devices with the same technologies as listed by others. The other is to scale up the results of others (including Xilinx and Altera) to the corresponding 28-nm technologies for their same FPGA brand (Virtex-7 for Xilinx and Stratix V for Altera). The different FPGAs used in our comparison are the following: 90-nm: Virtex-4 xc4vsx55 (V4); 65-nm: Virtex-5 xc5vsx240t (V5), Stratix III ep3se260 (S3); 40-nm: Virtex-6 xc6vsx475t (V6), Stratix IV ep4sgx530 (S4); 28-nm: Virtex-7 xc7vx980t (V7), Stratix V 5sgsd8 (S5). The scaled results are shown in Table VII and were used to compare performance. Direct comparison (without scaling) was made for our previous work [7], shown as *Tav* in Figure 13 which demonstrates the speed-ups we achieved compared to existing works in terms of latency and throughput.

While on Virtex-7 we achieved $n$=876 for 32-bit fixed-point solvers, we used our $n$=764 results to compare against Tav [7], that is the largest tree the work could implement. All other $n$ values are the maximums implementable by us. For instance, the $n$ values in Table VII are the worst-case maximums that we can achieve in the old devices for the data type in any column. These were calculated by assuming the same DSP and slice count for the setup logic and the systolic cell in

TABLE VI
FPGA IMPLEMENTATIONS OF AMERICAN OPTION PRICERS WHICH ARE USED FOR PERFORMANCE COMPARISON. DATA WITH * ARE SCALED BY $R$.

| | **Jin [4]** | | | **Wynnyk [5]** | **Morales [6]** | **Tavakkoli [7]** | **Ours** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Precision | Fix 16.16 | Single | Double | Double | Double | Fix 16.16 | Fix 16.16 | | Single | Double |
| FPGA@MHz | V4@82.7 | V4@76 | V4@67.3 | S3@150 | S4@163 | V7@143 | V7@150 | V7@200 | V7@65 | V7@40 |
| Max Size (n) | 1000 | 1000 | 1000 | 64000 | 1024 | 768 | 876 | 764 | 516 | 148 |
| HW Specs. | $C$:6, $R$:14 | $C$:6, $R$:6 | $C$:6, $R$:3 | $N_V$:4 | $N_V$:8 | $m$:8 | $m$:6 | $m$:8 | $m$:6 | $m$:6 |
| Slice Usage | 99% * | 93% * | 87% * | 10% logic | 66% logic | 29% | 54% | 48% | 95% | 49% |
| DSP Usage | 112 (22%) * | 48 (9%) * | 96 (19%) * | 148 (19%) | 760 (76%) | 85% | 100% | 87% | 60% | 89% |
| BRAM Usage | 252 (79%) * | 108 (34%) * | 60 (19%) * | 84% bits | 39% bits | 1% | 0 | 0 | 0 | 0 |
| Perf. (Kt/s) | 1.16 Gn/s * | 0.46 Gn/s * | 0.2 Gn/s * | 1.152 | 2.4 | 4.4 | 171 | 261 | 125 | 268 |

TABLE VII
SCALED DATA USED FOR COMPARISON OF FIGURE 13. SU: SCALED UP, SD: SCALED DOWN. L: LOGIC, M: MULTIPLIER.

| | **Ours SD** | | | | | **Jin [4] SU** | | | **Wyn. [5] SU** | **Mor. [6] SU** |
|---|---|---|---|---|---|---|---|---|---|---|
| Precision | Fix 16.16 | Single | Double | Single | Single | Fix 16.16 | Single | Double | Double | Double |
| FPGA@MHz | V4@65 | V4@65 | V4@65 | V5@65 | V6@65 | V7@82.7 | V7@76 | V7@67.3 | S5@150 | S5@163 |
| Prob. Size (n) | 104 | 43 | 19 | 75 | 156 | – | – | – | – | – |
| HW Specs. | $m$:6 | $m$:6 | $m$:6 | $m$:6 | $m$:6 | $C$:6, $R$:87 | $C$:6, $R$:40 | $C$:6, $R$:21 | $N_V$:20 | $N_V$:30 |
| Utilisation | M'100% | L'100% | M'100% | L'100% | L'100% | L'100% | L'100% | L'100% | M'19% | M'76 % |

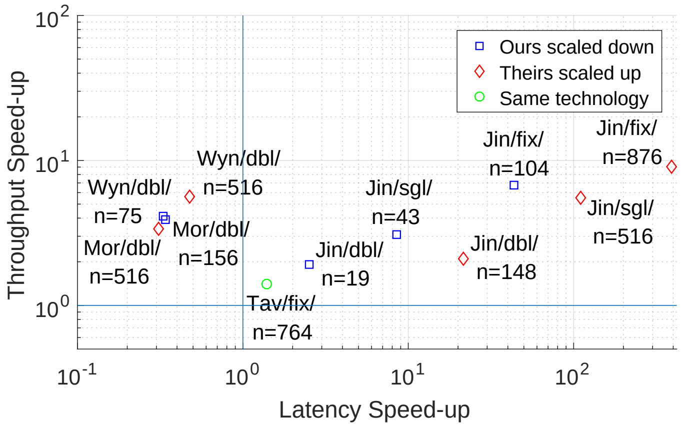

Fig. 13. Our throughput and latency gains compared to related FPGA works

the old device and estimating the number of replications of the systolic cell. The maximum $n$ in comparisons for floating-point solvers was $n$=516. For this $n$, the difference in accuracy of a single-precision solver compared to a double-precision is negligible (less than 0.01%). Therefore, we used our single-precision results to compare against the works of Wynnyk [5] and Morales [6] which were both in double precision.

We gained higher throughput in all cases, as shown in Figure 13. For the scalar implementations, although they also use C-slow pipelining to increase throughput, since their execution time is quadratic we achieve better overall throughput, and also latency speed-ups of up to 390×. The vector designs [5], [6], in contrast to our C-slowed NE, used their pipeline for a single option. Compared to them, we have orders of magnitude better throughput, but also lower latency. However, given that we can decrease $m$ by relaxing the clock constraint, we expect to get slightly better latency values. Compared to our old systolic design [7], we gain latency and throughput speed-ups (both at 1.4×) due to the increased clock frequency that we achieved for $m$=8 ($f$=200MHz) compared to [7] ($f$=143MHz) with the same $m$. This demonstrates that our automated design framework with a high-level user interface performs better than a hand-tuned RTL design with a manual flow. In moving from old to new FPGAs with larger trees, as shown in Figure 13, we achieve higher speed-ups, with the work of Morales [6] as the only exception. This is because their Stratix IV FPGA is not the largest device of that family of 40-nm technology. While we obtained 79% of extra DSP blocks moving from Virtex-6 to Virtex-7, the scaled results for Morales were based on 186% extra multiplier blocks.

## VIII. CONCLUSION

This paper presented a high-level design framework which captures binomial-tree problems in a highly-abstract form and converts them automatically into high-throughput systolic FPGA solvers. The user effort is limited to defining mathematical sets and functions of the binomial-tree formal framework in a high-level language like C. In this framework, the problem definition is decoupled from the hardware realisation, where the user is not required to define any task scheduling (like a *for* loop). This is a significant improvement in design productivity compared to the previous FPGA solutions, where users needed to define both the tasks and their scheduling, and sometimes in RTL. Also, to our knowledge, this is the first design framework for FPGA solutions which can take more than one type of binomial-tree problems (including American, barrier and Bermudan options). This is in contrast to the previous methods which could only solve American options.

With our fully pipelined systolic implementation on a Virtex-7 we achieve latencies in the range of 40 to 200 micro-seconds for different problems and data types with practical tree sizes. The same implementations give us higher throughput compared to all the previous FPGA solutions, ranging from 1.4× speed-up compared to a hand-coded RTL systolic design, to orders of magnitude gain (of up to 9.1×) with respect to scalar and vector approaches.

A weakness of the systolic architecture that we employed is that the maximum problem size is limited by the number of FPGA multipliers. However, we showed that medium-sized trees on a Virtex-7 can produce results with practical accuracies. Also, newer FPGAs with more multipliers, such as the Xilinx Virtex UltraScale+ [19] with 3× as many DSPs as a Virtex-7, would allow even more accurate results.

For future work, we plan to target large trees by using the extensive memory in the block RAMs which is completely untouched in our current implementations. This way $O(n)$ of memory resources will be sufficient to hold intermediate node values at any time step for large trees. Other plans include providing data-movement templates for trinomial trees [4] and explicit finite difference models [20]. To enhance efficiency, two-point Richardson extrapolation techniques [2] can be employed by very simple modifications to the framework. For instance, two solvers of $n$=50 and $n$=100 can be instantiated on the same chip to give a result as accurate as a 1000-step binomial tree [2] with 10× better throughput.

## References

[1] J. Hull, *Options, Futures, and Other Derivatives: Solutions Manual*. Pearson Education, Limited, 2012.

[2] M. Broadie and J. Detemple, “American Option Valuation: New Bounds, Approximations, and a Comparison of Existing Methods,” *The Review of Financial Studies*, vol. 9, pp. 1211–1250, 1996.

[3] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, “Exploring reconfigurable architectures for binomial-tree pricing models,” in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 245–255.

[4] ——, “Exploring reconfigurable architectures for tree-based option pricing models,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 4, p. 21, 2009.

[5] C. Wynnyk and M. Magdon-Ismail, “Pricing the american option using reconfigurable hardware,” in *Computational Science and Engineering, 2009. CSE’09. International Conference on*, vol. 2. IEEE, 2009, pp. 532–536.

[6] V. M. Morales, P.-H. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton, “Energy-efficient fpga implementation for binomial option pricing using opencl,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[7] A. Tavakkoli and D. B. Thomas, “Low-latency option pricing using systolic binomial trees,” in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 44–51.

[8] N. K. Pham, K. M. M. Aung, and A. Kumar, “Automatic framework to generate reconfigurable accelerators for option pricing applications,” in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Nov. 2016, pp. 1–8.

[9] A. H. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk, “Reconfigurable control variate monte-carlo designs for pricing exotic options,” in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 364–367.

[10] ——, “Efficient reconfigurable design for pricing asian options,” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 4, pp. 14–20, 2011.

[11] R. Sridharan, G. Cooke, K. Hill, H. Lam, and A. George, “FPGA-Based Reconfigurable Computing for Pricing Multi-asset Barrier Options,” in *2012 Symposium on Application Accelerators in High Performance Computing*. IEEE, Jul. 2012, pp. 34–43.

[12] G. W. Morris and M. Aubury, “Design Space Exploration of the European Option Benchmark using Hyperstreams,” in *2007 International Conference on Field Programmable Logic and Applications*. IEEE, Aug. 2007, pp. 5–10.

[13] Q. Jin, W. Luk, and D. B. Thomas, “On Comparing Financial Option Price Solvers on FPGA,” *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, no. Mc, pp. 89–92, May 2011.

[14] P. Wilmott, J. Dewynne, and S. Howison, *Option Pricing: Mathematical Models and Computation*. Oxford Financial Press, 1993.

[15] J. C. Cox, S. A. Ross, and M. Rubinstein, “Option pricing: A simplified approach,” *Journal of Financial Economics*, vol. 7, no. 3, pp. 229–263, Sep. 1979.

[16] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, “Post-placement c-slow retiming for the xilinx virtex fpga,” in *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*. ACM, 2003, pp. 185–194.

[17] L. Andersen, M. Lake, and D. Offengenden, “High-performance American option pricing,” *The Journal of Computational Finance*, pp. 39–87, Aug. 2016.

[18] F. De Dinechin and B. Pasca, “Designing custom arithmetic data paths with flopoco,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 0018–27, 2011.

[19] Xilinx Inc., “UltraScale Architecture and Product Overview (DS890 v2.7),” pp. 1–31, 2016.

[20] Q. Jin, D. B. Thomas, and W. Luk, “Exploring reconfigurable architectures for explicit finite difference option pricing models,” in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009, pp. 73–78.

**Aryan Tavakkoli** received the B.Eng. degree in Electronic and Electrical Engineering from Brunel University London. He also received the M.Sc. degree in Analogue and Digital Integrated Circuit Design from Imperial College London, where he is currently pursuing a Ph.D. degree in Electrical and Electronic Engineering. His research interests include reconfigurable hardware acceleration, application-specific design automation, financial computing.

**David B. Thomas** received the M.Eng. and Ph.D. degrees in computer science from Imperial College London. He is a Lecturer with the Electrical and Electronic Engineering Department, Imperial College London. His research interests include hardware-accelerated cluster computing, Monte Carlo simulation, random number generation, and financial computing.