An FPGA based scalable architecture of a stochastic state point process filter (SSPPF) to track the nonlinear dynamics underlying neural spiking

Yao Xin, Will X.Y. Li, Ray C.C. Cheung, Rosa H.M. Chan, Hong Yan, Dong Song, Theodore W. Berger

1. Introduction

Cognitive neural prosthetic design is an emerging topic in neural engineering research. For many years, various studies have endeavored to develop a silicon-based prosthetic device that can be implanted into the mammalian brain, e.g., the hippocampal CA3 [1]. This device is expected to perform bi-directional communications between the intact brain regions while bypassing the degenerated CA3 region. Such a prosthetic device (if successfully developed) could provide fundamental treatment to diseases related to cognitive impairment, such as Alzheimers.

The generalized Laguerre–Volterra model (GLVM) proposed by Song et al. [2] is a data-driven model first applied to predict mammalian hippocampal CA1 neuronal spiking activity based on detected CA3 spike trains by which the expected neuroprosthetic function can be achieved. GLVM consists of five major components (Fig. 1). It uses a weighted sum of convolution products between the model inputs and the orthonormal Laguerre basis functions, passing through a threshold trigger to generate the predicted model outputs. A detailed calculation flow of the generalized Laguerre–Volterra (GLV) algorithm can be found in [2,3]. The prediction function of the GLV algorithm is very straightforward when implemented on different platforms. The Laguerre coefficients (weights of convolution products), however, must be estimated first using the recorded input/output data. This estimation process is often the most computationally intensive stage in the entire calculation flow. In this work, we focus on the estimation stage of the GLV algorithm and utilize an efficient adaptive method to estimate the coefficients dynamically and accurately.

The stochastic state point process filter (SSPPF) is a suitable choice for the aforementioned problem [4]. Derived by Eden et al. [4] based on Bayes’ rule Chapman–Kolmogorov paradigm and point process observation models, it has been verified to be effective for tracking dynamics of neural receptive fields under various conditions. In 2009, Chan et al. [5] applied the algorithm to realize the estimation function of the GLVM.

GLVM is only one successful application of SSPPF which is designed for point process and especially suitable to do neural encoding/decoding. This filter has been compared with SDPPF, extended Kalman filter (EKF), and pass-by-pass method [4] in the simulation of the response of pyramidal neurons in the CA1 region of the rat hippocampus to the movement of the animal. SSPPF

Article info
Article history:
Received 23 August 2013
Received in revised form
27 March 2014
Accepted 31 March 2014

Keywords:
SSPPF
Adaptive filter
Neural modeling
Field-programmable gate array

Abstract
Recent studies have verified the efficiency of stochastic state point process filter (SSPPF) in coefficients tracking in the modeling of the mammalian nervous system. In this study, a hardware architecture of SSPPF is both designed and implemented on a field-programmable gate array (FPGA). It provides a time-efficient method to investigate the nonlinear neural dynamics through coefficients tracking of a generalized Laguerre–Volterra model describing the spike train transformations of different brain sub-regions. The proposed architecture is able to process matrices and vectors with arbitrary sizes. It is designed to be scalable in parallel degree and to provide different customizable levels of parallelism, by exploring the intrinsic parallelism of the FPGA. Multiple architectures with different degrees of parallelism are explored. This design maintains numerical precision and the proposed parallel architectures for coefficients estimation are also much more power efficient.

© 2014 Elsevier Ltd. All rights reserved.
provided the most accurate tracking of both the linear (slow) and jump (rapid) evolution scenarios. In the adaptive decoding study [4], the feasibility of reconstructing a neural signal with SSPPF from an ensemble whose receptive fields are evolving is illustrated. Moreover, Salimpour et al. [6] have applied the SSPPF to the neural spiking activity of inferior temporal cortex of macaque monkey for the first time. The filter is used to estimate the conditional intensity of the point process observation as a time varying firing rate in cooperation with a particle filter. The results with a real data indicate that, based on the assessment of goodness-of-fit, the neural spiking activity and the biophysical property of neuron could be captured more accurately compared with conventional methods.

Although the SSPPF has been successfully and widely applied, due to the fact that this algorithm was implemented with commercial software and run on a desktop setup, the calculation process has certain limitations. Although conventional CPUs have experienced a significant improvement on computing power, there still exist many bottlenecks, such as the difficulty to further increase clock rates and a mismatch between memory bandwidth and ever-increasing CPU speed, which are preventing them from successfully keeping up with the demands from high-performance computing (HPC) applications. When the number of GLVM input and output grows, the number of Laguerre coefficients to estimate increases exponentially. Therefore, a high-performance platform is required for off-line (for model training) and on-line (for prediction) coefficient estimations.

In order to fill the enlarged gap between HPC requirement and limited CPU performance, other parallel platforms are attracting attentions from researchers. The FPGA is a competitive solution which is reconfigurable to satisfy various computational requirements, like large-scale data or real-time video processing. It is able to provide different levels of parallelism according to available hardware resource to maximize the performance. More importantly, the inherent low-power property of FPGAs makes it suitable for implantable neural prosthetic designs. Furthermore, the FPGA design is always demanded as a prototype of silicon-based prosthetic device for verification and testing.

In this work, we overcome the limitations of the previous works and for the first time, implement the SSPPF on the FPGA-based hardware platform to achieve more efficient and effective model coefficients estimation.

The major contributions of this paper are as follows: (1) The first hardware architecture of SSPPF has been designed and implemented, which is practical to be applied to the general framework of future cognitive prosthetic device. (2) This design is capable of processing arbitrary size matrices/vectors without any pre-configurations; the maximal size supported is limited only by the available on-chip memory resource. (3) In order to achieve high performance computing, the architecture is made scalable through parallelism. Multiple designs in different parallelism degrees are explored, implemented, and tested. (4) Our design can be generalized to readily accommodate other adaptive filters which require extensive matrix/vector operations, since the essential matrix operations (multiplication and inversion) are realized in current work.

The rest of the paper is organized as follows: Section 2 describes related studies. Section 3 introduces the SSPPF algorithm. Section 4 describes both the overall hardware architecture and its components. Section 5 provides the implementation results and performance evaluation. Section 6 summarizes the paper.

## 2. Related works

The performance of a platform established for conducting the estimation function can be measured with three standards: effectiveness, efficiency, and realisticity of application. A silicon-based implementation of the GLVM dates back to 2005, when Berger et al. [7] first completed initial prototyping work utilizing a field-programmable gate array (FPGA). In 2006, Hsiao et al. [8] fabricated such an architecture using the 18 μm process of Application Specific Integrated Circuit (ASIC) technology. Their study was based on the single-input, single-output (SISO) GLVM, the simplest form of the model. However, in a real-world situation, model output is often affected by the spiking activity of multiple inputs. This activity weakens the realisticity of the design application into a practical neuroprosthetic device.

In 2011, Li et al. [9] successfully implemented the multi-input, multi-output (MIMO) GLVM on the FPGA-based hardware platform. They achieved a remarkable speedup in model coefficient estimation when compared to a traditional software based approach. They adopted the steepest decent point process filter (SDPPF) as the tracking method. This method is simpler in the mathematical representation but sacrifices certain levels of accuracy when compared to other well-established methods, i.e., the SSPPF or the Kalman Filter. Thus, this method is less effective.

A major improvement of SSPPF over SDPPF is the introduction of the adaptive learning rate. Learning rate is important for
adaptive filters [10] as it controls the trade-off between convergence speed and divergence. It also modulates the innovation vector which is the basic error signal between the instantaneous estimation and the current output. Previously, only a constant learning rate was adopted [9], which can simplify the iterative computation procedure. The brain activities of behaving animals, however, can be time-variant and subject to stochastic variations, such as environmental changes. Thus, the model needs to be more realistic. The learning rate itself should be updated in an iterative way using the firing probability calculated both in previous time and the detected model output in present time [9]. The block diagrams of two filters with our customized configurations are shown in Fig. 2.

The hardware architectures for various kinds of adaptive filters have been intensively explored [11–15]. Robinson et al. [11] realized the fixed-point and the floating-point FPGA implementations for both a recursive-least square (RLS) filter and a Kalman adaptive filter, which are applied in predicting physiological hand tremors. The tradeoff between hardware utilization and different data representing precisions is elaborated and discussed in this work. A parallel extended Kalman filter (EKF) architecture for mobile robots was presented to address 2-D maps with 1.8k features in real time [12]. This design allows multiple processing elements to be achieved with independent memory bank. Another EKF implementation on FPGA was presented by Guo et al. [14]. This implementation was developed in the C/C++ program; and it was achieved using the System on Programmable Chip (SoPC). Although this implementation has certain reconfigurability, the efficiency is more compromised when compared with pure hardware designs. Zhu et al. [13] proposed an FPGA implementation of the Kalman filter algorithm in a neural ensemble activity application. It contains optimized parallel processing elements to accelerate matrix computations. However, the maximum number of rows or columns supported in a matrix is only constrained to 32, which is a major limitation for the entire architecture. These studies are enumerated in Table 1 and compared with our proposed architecture.

3. Stochastic state point process filter

SSPPF was proposed by Eden et al. to adaptively estimate neuron parameters for point process observations when analyzing neural plasticity [4]. In SSPPF, the system model of neural firing is defined as a set of parameter vectors $C_k$. It is formulated with linear evolution processes and Gaussian errors [4]:

$$ C_{k+1} = F_k C_k + \eta_k $$

where $F_k$ is the system transition matrix, while $\eta_k$ is the zero-mean white-noise process with covariance matrix $Q_k$.

The recursive expression, to estimate time-varying $C_k$ at each discretized point in time, is based on its posterior density, given past observations. The first step is the one-step prediction in which the prior mean $C_{k|k-1}$ and prior covariance $R_{k|k-1}$ at time $t_k$ are predicted respectively:

$$ C_{k|k-1} = F_k C_{k-1|k-1} $$

$$ R_{k|k-1} = F_k R_{k-1|k-1} F_k^T + Q_k $$

Setting $F_k = I$ for simplicity [4], the simplified denotation $R(k)$ representing posterior variance matrix $R_{k|k}$, and $C(k)$ representing the posterior mean vector $C_{k|k}$ which is the estimated coefficient that can be updated by the following recursive equations:

$$ R(k)^{-1} = [R(k-1) + Q(k)]^{-1} $$

$$ + \left[ \left( \frac{\partial \log P(k)}{\partial C(k)} \right)^T P(k) \left( \frac{\partial \log P(k)}{\partial C(k)} \right) - (y(k) - P(k))^T \frac{\partial^2 \log P(k)}{\partial C(k) \partial C(k)^T} \right] $$

$$ C(k) = C(k-1) + R(k) \left( \frac{\partial \log P(k)}{\partial C(k)} \right)^T (y(k) - P(k)) $$

where $P(k)$, $y(k)$, and $k$ are the firing probability of spike firing, spike observed, and the discrete time steps respectively.

The diagram of SSPPF is shown in Fig. 2, together with that of SDPPF. In SDPPF, $R$ is set as a constant matrix. While applying the SSPPF, this covariance matrix $R(k)$ is updated iteratively at each time step employing the predicted firing probability and observed neuronal firing output, thus being adaptive to system operations and changing dynamics. $R(k)$ acts as the learning rate for the coefficient estimations in (5). The trade-off for adopting this adaptive learning rate is an increased complexity of computation.

Applying (4) and (5) to our previously established generalized Laguerre–Volterra model [5], the two equations can be rewritten in the following forms:

$$ R(k) = [(R(k-1) + Q(k))^{-1} + k_1 \times M(k)^T \times M(k)]^{-1} $$

$$ C(k) = C(k-1) + k_2 \times M(k) \times M(k)^T $$

where the vector $M(k)$ is the convolution product between the model inputs/output and the Laguerre basis functions. The $k_1$ and $k_2$ can be derived as follows:

$$ k_1 = \alpha^2 P(k) + \beta \sigma_y (k - P(k)) $$

$$ k_2 = \alpha \sigma_y (k - P(k)) $$

$$ ak = \frac{1}{\sqrt{2\pi P(k)}} \exp[-w^2(k)] $$

$$ \beta = \frac{w(k)}{\sqrt{2\pi P(k)}} \exp[-w^2(k)] + \frac{1}{2\pi P(k)} \exp[-2w^2(k)] $$

The GLVM functions for the calculation of $M(k)$, $y(k)$, $P(k)$, and $w(k)$ using the time-varying coefficients were introduced in [16,5], hence they are not elaborated here.

The SSPPF algorithm is reformed into two major parts. The first part involves a large amount of matrix computation, which is the most time consuming and principal aspect of the computation. The second part involves intensive arithmetic operations that are difficult for the hardware to implement. Therefore, our architecture of SSPPF is designed to lay a special focus on the calculation

<table>
<thead>
<tr>
<th>Related works</th>
<th>Numerical format</th>
<th>Max. matrix size</th>
<th>Random matrix size</th>
<th>Parallel degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robinson et al. [11]</td>
<td>Fixed-point SPFP DPFP</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Bonato et al. [12]</td>
<td>SPFP</td>
<td>Limited by off-chip RAM</td>
<td>No support</td>
<td>4</td>
</tr>
<tr>
<td>Zhu et al. [13]</td>
<td>SPFP</td>
<td>32</td>
<td>No support</td>
<td>32</td>
</tr>
<tr>
<td>Guo et al. [14]</td>
<td>SPFP</td>
<td>19</td>
<td>No support</td>
<td>1</td>
</tr>
<tr>
<td>Proposed</td>
<td>SPFP</td>
<td>Limited by on-chip RAM</td>
<td>Support</td>
<td>Scalable</td>
</tr>
</tbody>
</table>
stages revealed by (6) and (7), which have more potential for parallelization. A host PC device is responsible for the second part and M(k) update, communicating with FPGA in real-time. Required initial parameters as well as the matrices are loaded from the external input to the hardware. In practice, if the number of parameters in C(k) to estimate is N, R(k) and Q would be a N × N matrix, M be a 1 × N vector, while C be a N × 1 vector.

4. Parallel architecture design

The FPGA parallel architecture is constructed for the following reasons: (1) Traditional CPU platforms have certain performance limitations [17]: they are operating fairly near the speed limit (in the order of GHz for basic frequency), due to the bottleneck of semiconductor technology; high power consumption and limited memory bandwidth also impose restrictions on the increase of processing cores. (2) Possible parallelism for accurate coefficient estimation can be explored efficiently with SSPPF. (3) Low-power embedded platforms for real-time parameter estimation may be required in the future. (4) The former architecture for parameter estimation for GLVM is based only on the SDPPF, which is compromised in accuracy (as previously mentioned).

Our hardware architecture is designed according to the reformed SSPPF algorithm. Unlike most hardware designs, the size of the vector and the matrix is not fixed. Instead, it is rather dynamically changed on-the-fly without pre-configurations. The maximal size is limited only by the on-chip memory. Moreover, this architecture could achieve different degrees of parallelism because it scales with the number of computing units and requires little change in code (i.e., data-width and control conditions). These modifications are facilitated by our highly parameterized design. Multiple architectures with different parallel degrees are thus explored to verify this capability.

Data representation format should be considered seriously as it affects the accuracy of filter results. The effect of both fixed-point and floating-point with various precision had been previously studied [11,18]. Although fixed-point is easier to implement, it always suffers from not only limited dynamic range but also fine precision. FPGAs, however, provide increasing support for high-speed floating-point computation within the hardware. Therefore, most recent architecture designs for filters adopt a floating-point format in the IEEE 754 standard. Similarly, the data in our design is represented in single precision floating-point notation. The computation units take advantage of the Xilinx Floating-point Operator intellectual property (IP) core to perform floating-point operations.

Each basic computing unit is designed to consist of either four adders or four multiplier-adders in the current design. The intrinsic FPGA parallelism can be explored by scaling up computing units in parallel, with data-width for operation and storage increased accordingly. Operations are performed in a horizontal sweep fashion to process multiple elements of a row in parallel until the entire row is scanned.

4.1. Overall architecture

The top layer module implements two sub-modules according to the major steps in SSPPF calculation (illustrated in Fig. 3). Both $k_1 \times M^T \times M$ and $k_2 \times R(k) \times M^T$ are computed in the matrix multiplication module, while the matrix inversion module performs two rounds of inversion for matrix R(k) in each iteration. Vectors and matrices involved in the algorithm are stored separately in Block RAMs, many of which are configured to true dual-port mode. In aspects of control logic, the top module utilizes a finite state machine (FSM) to issue address and data signals in different states in accordance with the calculation order in the SSPPF algorithm. Fig. 4 shows the time–space diagram of operation scheme for the top-level architecture. The states, dataflow and corresponding modules being executed are presented. Since there exists data dependency between most modules, most modules are activated step by step.

Both single and multiple floating-point computing units are set up in different versions to achieve scalable parallelism. The data-width of RAM is correspondingly configured equal to that of the computing unit input. For instance, the data-width is 128-bits comprising four floating point data if the computing-unit number is one.

In addition to the scalability in parallel degree of computation, the size of the supported matrix is also scalable. The data from each row in a matrix is organized within an independent space in memory. Meanwhile, the addresses are allocated dynamically, determined only by the matrix size. Such a feature empowers our architecture to address random size matrices in real-time. The module input port matrix_size can be altered in different rounds of computation (see Fig. 3). Both the internal parameters and the address range for storing each row in memory adjust automatically. With this flexibility, our design is versatile without re-configuration to estimate either different types of parameters or vectors containing unfixed numbers of parameters.

Algorithm 1. Gauss–Jordan algorithm [19].

```
Input [A]
Output [A⁻¹]
1 B = [A]
2 row_pivot = 1
3 for i = 1 to i = N do
   // phase 1: locate pivot
   4 for j = i to j = N do
      if |B(j, i)| > |B(i, i)| then
         row_pivot = j
5 end
6 end
7 // phase 2: switch pivot row and current row
8 for j = i to j = N do
   9 temp = B(i, j)
10 B(i, j) = B(row_pivot, j)
11 B(row_pivot, j) = temp
12 end
13 // phase 3: normalization
14 for j = i to j = N do
   15 B(i, j) = B(i, j) / B(i, i)
16 end
17 // phase 4: row elimination
18 for k = 1 to k = N do
   19 if k ≠ i then
      for j = 1 to j = 2N do
         20 B(k, j) = B(k, j) – B(k, i) × B(i, j)
21 end
22 end
23 end
24 return B = [A⁻¹]
```

4.2. Matrix inversion module

The matrix inversion is based on the Gauss–Jordan elimination algorithm with partial pivoting (see Algorithm 1). Given an N × N square matrix A, the inversion process begins by building an augmented matrix with the size N × 2N: [A | I]. The identity matrix I is on the right side of A. Row operations are then performed.
Fig. 3. Top-level structure for SSPPF with three major sections included: (1) two secondary modules: matrix multiplication and matrix inversion module; (2) the computing units which is scalable with parallel degree; (3) control unit which is a FSM controller.

Fig. 4. Time–space diagram of data scheduling scheme.

Fig. 5. Detailed structure of matrix inversion module. There are two major parts: (1) three secondary modules implemented according to matrix inversion flow in which the elimination module plays the most important role in parallelization; (2) two block RAMs caching original and identity matrix.
iteratively to transform $[A | I]$ into $[ I | A^{-1}]$. There are several related works focusing on the FPGA architecture for Gauss-Jordan elimination [20–22], most of which are not as scalable as our design.

The architecture of the matrix inversion module is illustrated in Fig. 5. Two dual-port Block RAMs are implemented, because the inversion needs both the original and identity matrix to be cached. Three sub-modules correspond to different iterative phases in the inversion algorithm. The initial-pivot finder module locates the first pivot row when the matrix data is written to the RAMs. The row switch module interchanges the pivot row and the current row. The elimination module deals with both the normalization and the elimination phase. The elimination module (the principal part of the matrix inversion architecture) contains the majority of the floating-point operators (presented in Fig. 6).

The normalization and elimination operations are completely pipelined. The major arithmetic operations include division, multiplication and addition, which are performed in floating-point operators. The pivot row normalization step is realized by multiplying the pivot’s reciprocal calculated in advance by each row element. The benefit here is that multipliers consume fewer resources and require less latencies than dividers. Multipliers can be reused in the consequent elimination phase.

Each basic computing unit consists of four multipliers and four adders which are cascade connected. As aforementioned, the operations are performed in a horizontal sweep fashion. The number of operators utilized determines the degree of parallelism. Computing units can easily concatenate in parallel to achieve higher parallelism (Fig. 7).

Two Block RAMs are set to synchronize multiple operand flows in the computing units: one to cache a normalized pivot row and another to cache all elements in the column where the pivot is located. This method allows two sets of data to be read, from RAMs to multipliers, simultaneously in the elimination stage. A comparator is the last node in the computing pipeline, directly receiving data flow from the adder outputs, to achieve efficient pivot locating. Therefore, the pivot search process is performed, in general, with only one extra clock. The found pivot row address would be output for the next round of row interchange when the current elimination process is complete.

4.3. Matrix multiplication module

This module combines two calculations ($k_1 \times M^T \times M$ and $k_2 \times R(k) \times M^T$) into one module, the memory of vector $M$, together with the floating-point operators, is thus shared (see Fig. 8). The time consumption for this module is trivial compared with that for

![Fig. 6. Elimination module: parallelism is exploited mostly in this module because the elimination phase is most time-consuming throughout the overall process. Indeed, two sets of computing units are implemented for original and identity matrices, but only one set is presented in this figure.](image-url)

![Fig. 7. Pipelined and parallel structures of the elimination operation (not including normalization and column cache phase). There exist some delays in memory reading and floating point computing units which are skipped in this figure to make it more compact. The actual pipeline stages are more than presented.](image-url)
the inversion process, thus only two multipliers and one accumulator are implemented. Calculating $k_1 \times M^T \times M$ requires two continuous multiplications and $k_2 \times R(k) \times M^T$ requires an extra accumulation. Fig. 9 shows the pipelined structure for this module.

4.4. Scalability for degree of parallelism

Parallelism is exploited efficiently in current architecture. Multiple operations are performed in parallel, in order to achieve high-performance computing. Unlike the work in [23] which implements multiple Processing Elements (PEs) each working independently, the data structure is difficult to be partitioned equally in our design. The methods adopted in this study are pipelining, implementing multiple computing units, increasing I/O datawidth and throughput. Specifically, the adders in top module and multiply-adders in elimination submodule are implemented in parallel, where the latter units are our major focus because the elimination phase is most time-consuming. The design is highly parameterized to facilitate the modification of the datawidth and control logic according to a different parallelism. By doing this, it is convenient to customize the design with our preferred degree of parallelism.

4.5. Computational complexity

The calculation steps for the update of vector C(k) in each round are listed in Table 2 associated with approximated clock cycles in each step. The $D_P$ denotes the degree of parallelism (this degree would be 4 if only one computing unit was implemented). In this table, both the latency of the floating-point operators and the control signal overheads are overlooked; each would be fairly trivial when the matrix size $N$ increases. The matrix inversion computation dominates the total computing time whose complexity is $O(N^3)$ so the time complexity of the entire system is also $O(N^3)$. However, it is straightforward to see that, the complexity is reduced by $D_P$ times. Higher parallelism implies less computational complexity.

5. Experimental results

5.1. Implementation results

Four architectures with different parallelism degrees are synthesized, placed, and routed separately on a single Xilinx Virtex-6 (XC6VLX240T-1) FPGA chip. Degrees ($D_P$) of 4, 8, 16, 32 are explored separately. Table 3 summarizes the resources occupied by various architectures and corresponding major modules. The limitation of

Fig. 8. Matrix multiplication module: it realizes the vector/matrix multiplications. Three RAMs are implemented to store vector M, matrices $k_1 M^T M$ and $k_2 R M^T$. Two start signals indicate which module function needs to be performed.

Fig. 9. Pipelined and parallel structures of matrix multiplication module. There exist some delays in memory reading and floating point computing units which are skipped in this figure to make it more compact. The actual pipeline stages are more than presented.

Table 2
Approximate number of clock cycles required ($N$: vector/matrix size, $D_P$: degree of parallelism).

<table>
<thead>
<tr>
<th>Step no.</th>
<th>Operation</th>
<th>Number of clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$r_1(k) = R(k - 1) + Q$</td>
<td>$N \frac{N}{D_P}$</td>
</tr>
<tr>
<td>2</td>
<td>$r_2(k) = r_1^{-1}(k)$</td>
<td>$(N^2 + 13N) \frac{N}{D_P} + N^2$</td>
</tr>
<tr>
<td>3</td>
<td>$r_3(k) = k_1 \times M^T \times M$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>4</td>
<td>$r_4(k) = r_2(k) + r_3(k)$</td>
<td>$N \frac{N}{D_P}$</td>
</tr>
<tr>
<td>5</td>
<td>$R(k) = r_2^{-1}(k)$</td>
<td>$(N^2 + 13N) \frac{N}{D_P} + N^2$</td>
</tr>
<tr>
<td>6</td>
<td>$r_5(k) = k_2 \times R(k) \times M^T$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>7</td>
<td>$C(k) = (C(k - 1) + r_5(k)$</td>
<td>$N$</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>$(2N^2 + 28N) \frac{N}{D_P} + 4N^2 + N$</td>
</tr>
</tbody>
</table>
matrix (vector) size $N$ lies in the on-chip memory resources. The entire complexity is much high when $D_p$ exceeds 32, resulting in general performance degradation.

In current version, the upper limit of $N$ is fixed at 256 which is sufficient in practical applications. Although the number of Laguerre coefficient grows rapidly with the number of model inputs and the model order, the effective inputs and significant cross-terms should be selected instead of estimating too many parameters, which would reduce the number of coefficients dramatically [3]. Moreover, we can also choose other powerful FPGA chips equipped with more memory elements to further increase $N$ in future implementation.

It is observed that the amount of RAM resource stays almost constant in Table 3, which is because the matrix/vector data stored on-chip remains the same in different parallelism degrees, and only the data structure in Block RAMs is needed to be reconfigured.

### 5.2. Function verification

Two sets of synthetic experimental data are collected as input into the hardware to verify the functionality of the SSPPF architecture. This data includes initial Laguerre coefficients $C(0)$ in GLVM, the covariance matrix $R(0)$, and the estimated firing probability $P(k)$ together with convolution product $M(k)$ in each iteration. The FPGA receives both $k_1$ and $k_2$ calculated with a PC, performs an SSPPF computation, and transfers the updated $C(k)$ to the PC in each iteration. A total of 100 iterative calculations have been conducted during each experimental session. The size of vector $C(k)$ (number of coefficients to estimate) in two datasets is set at 49 and 139 which account for two specific conditions in the MISO model: (1) only self-kerne in second-order are considered; (2) self-kernel and cross kernel in second-order are considered. The input number of MISO is $5$, and number of Laguerre basis functions $L$ is $3$ in both situations. Meanwhile, a Matlab description of the SSPPF algorithm is realized with a double floating-point format. This program utilizes the same synthetic datasets to estimate the Laguerre coefficients as a reference. The computed vector estimation results for the two datasets in 100 iterative numbers are all depicted in Fig. 10.

### 5.3. Error analysis

Since single precision floating-point is adopted, the analysis of the error caused by the limit length of data representation is essential [24], especially for coefficient estimation. Certain accuracy is required in practical applications. An error could be produced by rounding in operators and propagated in subsequent operations. Therefore, an accumulated error would increase when the amount of operations grows.

In our error analysis, the Matlab realization with double precision is employed as a comparator against our FPGA design. Both an initial input coefficient vector and a covariance matrix are randomly generated. The values of $k_1$ and $k_2$ play an important role; each can rescale the error accumulated during computation. The curves for $k_1$ and $k_2$ with an input set of $P(k), w(k)$, and $y(k)$, are depicted in Fig. 11. In this figure, the value of $P(k)$ is defined to be within the range of $[0.01,1]$. We select an absolute maximum of $k_1$ and $k_2$ to evaluate a possible maximal error.

We perform 100 independent experiments to update $C(k)$ and thus calculate the average maximal Mean Error (ME) and Mean Square Error (MSE) of FPGA implementation results in each SSPPF iteration. ME and MSE are computed with (12) and (13) (the $i$ is the element index in $C(k)$) respectively (Eq. 12). Although both increase as size $N$ grows, the MSE for $N=250$ remains under $2 \times 10^{-5}$. The error estimated is under worst conditions. Its normal value, however, would be much less in most situations. The practical requirement of MSE is $2 \times 10^{-4}$ for the coefficient estimation of GLVM. Therefore, it is sufficient for current application. Since single precision floating-point is relatively resource saving, tradeoff has been made between area and performance with current design to achieve an optimized result suitable to our application. Furthermore, we are able to use the curves in Fig. 12 to predict the accuracy intuitively in further experiments:

$$ ME = \frac{1}{100} \sum_{k=1}^{100} \left( \sum_{i=1}^{N} \left( C_{FGPA}(k,i) - C_{Matlab}(k,i) \right) \right) $$ (12)

$$ MSE = \frac{1}{100} \sum_{k=1}^{100} \left( \sum_{i=1}^{N} \left( C_{FGPA}(k,i)^2 - C_{Matlab}(k,i)^2 \right) \right) $$ (13)

### 5.4. Performance evaluation

We first utilize the peak Giga Floating-point Operations per second (GFLOPS) to quantify and evaluate the computational efficiency of hardware architecture designs. Table 4 lists the peak GFLOPS of different parallel architectures at maximum frequencies. The peak GFLOPS is achieved in the matrix inversion module, during the matrix elimination phase, which dominates the entire filter calculation time. The matrix size $N$ is its maximum value, the latency of floating-point computing units and pipelining length is relatively small compared to the entire elimination procedure. Moreover, in order to process both the original and the identity matrix simultaneously in this module, there are two sets of fully pipelined structure implemented working in maximum clock frequency.

In next evaluation, the FPGA architecture is compared to the software running on a commercial CPU platform. A software implementation of the SSPPF algorithm in C code is benchmarked. The implementation for both platforms is for Eqs. (6) and (7) instead of the entire SSPPF computation, and the reasons are as follows: (1) The calculation of Eqs. (8)–(11) and the data transfer between FPGA and CPU is less time-consuming, which can be negligible compared with the calculation of Eqs. (6) and (7). Only $k_1, k_2, M(k)$ and $C(k)$ are needed to be transmitted. (2) The PC platform is only a temporary choice but not the optimized one to calculate Eqs. (8)–(11). If available, it could be replaced by other low-power chips or just FPGAs, which would be the future plan. (3) Our objective is to only present the FPGA’s parallel advantage and the computation enhancement for matrix/vector operations.

### Table 3

<table>
<thead>
<tr>
<th>$D_p$</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Matrix multiplication module</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>2344 (1%)</td>
<td>3003 (1%)</td>
<td>2762 (1%)</td>
<td>3265 (2%)</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>2905 (1%)</td>
<td>4018 (1%)</td>
<td>5547 (1%)</td>
<td>8615 (2%)</td>
</tr>
<tr>
<td>RAM36E1</td>
<td>57 (13%)</td>
<td>58 (13%)</td>
<td>58 (13%)</td>
<td>58 (13%)</td>
</tr>
<tr>
<td>RAMB18E1</td>
<td>2 (1%)</td>
<td>2 (1%)</td>
<td>2 (1%)</td>
<td>2 (1%)</td>
</tr>
<tr>
<td>Max Freq</td>
<td>2616 MHz</td>
<td>2721 MHz</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Matrix inversion module</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>12,068 (8%)</td>
<td>22,299 (14%)</td>
<td>45,373 (30%)</td>
<td>89,142 (59%)</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>14,139 (4%)</td>
<td>26,900 (12%)</td>
<td>52,466 (22%)</td>
<td>102,791 (42%)</td>
</tr>
<tr>
<td><strong>Overall architecture</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>16,667 (11%)</td>
<td>29,608 (19%)</td>
<td>52,294 (34%)</td>
<td>100,945 (66%)</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>19,035 (6%)</td>
<td>36,250 (12%)</td>
<td>67,082 (22%)</td>
<td>129,312 (42%)</td>
</tr>
<tr>
<td>RAM36E1</td>
<td>176 (42%)</td>
<td>180 (43%)</td>
<td>186 (44%)</td>
<td>200 (48%)</td>
</tr>
<tr>
<td>RAMB18E1</td>
<td>3 (1%)</td>
<td>3 (1%)</td>
<td>5 (1%)</td>
<td>5 (1%)</td>
</tr>
<tr>
<td>Max Freq</td>
<td>208.2 MHz</td>
<td>222.5 MHz</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Y. Xin et al. / Microelectronics Journal 45 (2014) 690–701
so we shield the other part to make the comparison more straightforward.

Different datasets, with the number of coefficients ranging from 50 to 250, are executed. Other irrelevant applications are turned off during each test to avoid CPU utilization. Configuration of the software platform and the execution time are given in Tables 5 and 6 respectively. We have two versions of software implementation: direct one without deep optimization and deep speed-optimized one, to make comparison comprehensive. Only one core is utilized because we treat single-core as measurement criteria. We try to show the performance advantage of our design to most commonly used software execution mode.

The FPGA architecture of four versions in different parallel degrees is executed using the same datasets in the software, allowing us to compare the running speed of each. Each architecture is driven by its possible maximum frequency clock during the test. Fig. 13 illustrates the execution time of hardware for various matrix sizes, accompanied with speedup over corresponding software runtime. Although the speed enhancement with $D_p=4$ is insignificant, the resource occupation is quite limited in hardware. This property could thus be explored for embedded applications. When $D_p=32$, the speedup has reached $16 \times$ maximally over deep-optimized software. The parallel degree could be increased further with further architectural optimizations and more powerful chips. This intrinsic parallel capability would empower SSPPF architecture to perform better with more computing units implemented.

In next step, we estimate the sustained GFLOPS through the entire computation for hardware architectures and CPU. The GFLOPS is usually a fixed parameter for a certain CPU, but we can calculate the sustained GFLOPS for a certain implementation. This value is obtained by counting the number of all necessary floating-point operations in Fig. 4 and dividing this amount by real execution time. It is not fixed as it differs with various matrix ranges. We take maximum $D_p$ of 32 as an instance and present the sustained average GFLOPS for our architecture and CPU in Table 7 in which we use the run time of CPU with the deeply optimized code.

The average use ratio of the implemented computing units is also estimated. As shown in Fig. 4, different modules operate in different states while not all units are executing all the time. However, we calculate the weighted average percent of utilized computing units by using the expression

$$\text{Percent}_{\text{ratio}} = \sum_{i=1}^{n_{\text{max}}} \frac{N_i \times T_i}{T_{\text{total}} \times N_{\text{total}}}$$

(14)
where the $n_{state}$ is the number of states, $N_i$ is the computing units being executed in state $i$, $T_i$ is the duration time of state $i$, $T_{total}$ is the executing time of entire procedure. This value differs with matrix range and degree of parallelism $D_p$. We take $D_p$ of 4 as instance and calculate the weighted average percentage of computing-unit utilization with different matrix sizes and various $D_p$s. The result is in Table 8.

Portable embedded applications may also be required in the future. In this situation, power consumption would be a major concern. The power dissipation of parallel architectures is estimated by Xilinx XPower Analyzer in the ISE tool. The power of the CPU is measured actual processor power while the software is being executed by Intel Power Gadget 3.0 [25] which is a software-based power usage monitoring tool enabled for second (or later) generation Intel Core processors. This comparison is made chip-to-chip with same computation (calculation of Eqs. (6) and (7)) performed in two platforms.

Fig. 14 depicts a power consumption comparison between different hardware architectures and the commercial CPU. The growth of design complexity increases FPGA dynamic power dissipation, while the quiescent power remains still throughout. In general, the hardware architecture is much more power efficient than the CPU.

**Table 4**  
Peak GFLOPS evaluation of parallel architectures.

<table>
<thead>
<tr>
<th>Parallel architecture</th>
<th>$D_p=4$</th>
<th>$D_p=8$</th>
<th>$D_p=16$</th>
<th>$D_p=32$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak GFLOPS</td>
<td>2.4</td>
<td>3.2</td>
<td>5.6</td>
<td>8.2</td>
</tr>
</tbody>
</table>

**Table 5**  
Software execution platform.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core (TM) i5-250 M @ 2.50 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>4 GB</td>
</tr>
<tr>
<td>Compiler</td>
<td>Visual Studio 2010</td>
</tr>
</tbody>
</table>

**Table 6**  
Software runtime for different matrix sizes.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Without deep optimization (ms)</th>
<th>With deep optimization (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>3.43</td>
<td>1.65</td>
</tr>
<tr>
<td>100</td>
<td>15.35</td>
<td>10.59</td>
</tr>
<tr>
<td>150</td>
<td>45.96</td>
<td>32.23</td>
</tr>
<tr>
<td>200</td>
<td>104.37</td>
<td>65.55</td>
</tr>
<tr>
<td>250</td>
<td>203.66</td>
<td>125.18</td>
</tr>
</tbody>
</table>

Fig. 11. Curves for $k_1$ and $k_2$ with $P(k), w(k)$ and $y(k)$.

Fig. 12. Mean error and mean square error of the FPGA results with a different matrix size $N$.  

1) Mean Error (ME)

2) Mean Squared Error (MSE)
than the commercial CPU. Although a PC is required for the other section of calculation and data transfer in current design, it is not our only choice to perform the computation of (8)–(11). Therefore, we only focus on the comparison for the section of matrix/vector computations on two platforms, where the difference of power consumption is straightforward to see.

6. Conclusions

In this study, we propose the first architecture of stochastic state point process filter (SSPPF) hardware architecture implemented on an FPGA. Our architecture is able to handle random sizes of vectors and matrices without the need of pre-configurations. The upper bound of a matrix's size is limited only by on-chip memory resources. This architecture is also scalable in degree of parallelism. The processing units are capable of concatenating in parallel efficiently. Different architectures with different parallelism degrees have been explored, providing multiple choices for coefficient

![Fig. 13. Hardware runtime and acceleration compared to software with different matrix sizes and degrees of parallelism \( D_P \).](image)

### Table 7
Average GFLOPS evaluation.

<table>
<thead>
<tr>
<th>Matrix size ( N )</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sustained GFLOPS of FPGA</td>
<td>3.48</td>
<td>4.85</td>
<td>6.03</td>
<td>6.75</td>
<td>7.68</td>
</tr>
<tr>
<td>Sustained GFLOPS of CPU</td>
<td>0.32</td>
<td>0.39</td>
<td>0.43</td>
<td>0.49</td>
<td>0.50</td>
</tr>
</tbody>
</table>

### Table 8
Average percent of utilized computing units.

<table>
<thead>
<tr>
<th>Matrix size ( N )</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D_P = 4 ) (%)</td>
<td>49.1</td>
<td>50.8</td>
<td>51.5</td>
<td>51.9</td>
<td>52.2</td>
</tr>
<tr>
<td>( D_P = 8 ) (%)</td>
<td>50.4</td>
<td>53.5</td>
<td>55.0</td>
<td>55.9</td>
<td>56.6</td>
</tr>
<tr>
<td>( D_P = 16 ) (%)</td>
<td>47.8</td>
<td>52.1</td>
<td>54.5</td>
<td>56.1</td>
<td>57.1</td>
</tr>
<tr>
<td>( D_P = 32 ) (%)</td>
<td>40.1</td>
<td>47.6</td>
<td>49.7</td>
<td>52.9</td>
<td>54.0</td>
</tr>
</tbody>
</table>

![Fig. 14. Comparison of power consumption.](image)
estimation, according to resource availability and throughput requirements.

Extensive evaluations are performed with synthetic experimental datasets together with simulation datasets. The functionality of our architecture is verified, while the maximal Mean Error and Mean Square Error of the FPGA implementation are analyzed according to Matlab implementation. Our results indicate that this architecture can effectively maintain accuracy. Finally, we compare the processing speeds of different parallel architectures with the software platform. The acceleration is significant when a high degree of parallelism is employed. When power consumption is compared, our architectures are able to conserve more energy than a commercial CPU.

In the future, the SSPPF hardware IP should be incorporated into our previously established, reconfigurable platform for more efficient, accurate GLVM coefficient estimations.

References