Single-Pass Covariance Matrix Calculation on a Hybrid FPGA/CPU Platform

Lukas On Arnold[1,*] and *Muhsen* Owaida[2]

[1]Columbia University, *formerly ETH Zurich*
[2]ETH Zurich

Abstract. Covariance matrices are used for a wide range of applications in particle physics, including Kálmán filter for tracking purposes or Primary Component Analysis for dimensionality reduction. Based on a novel decomposition of the covariance matrix, a design that requires only one pass of data for calculating the covariance matrix is presented. Two computation engines are used depending on parallelizability of the necessary computation steps. The design is implemented onto a hybrid FPGA/CPU system and yields speed-up of up to 5 orders of magnitude compared to previous FPGA implementation.

1 Introduction

1.1 Covariance matrix

Covariance is a measure of variability between two different data sets X and Y defined as:

$$\mathrm{cov}(X, Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]. \tag{1}$$

We will use the understanding of the expected value as the sample mean:

$$E(X) = \overline{X} = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{2}$$

The **covariance matrix** K represents the covariances between all the permutations of the different data sets. We use the convention that m denotes dimensionality, i.e. the number of different data sets, and n the number of samples. Data are represented by an $m \times n$ matrix X. Then the covariance matrix is defined as:

$$\mathrm{K}_{ij} = \mathrm{cov}(X_{i.}, X_{j.}). \tag{3}$$

Note: subscripts denote indices and subscript $\mathrm{A}_{z.}$ is referring to the subvector with all the elements in row z from matrix A. The covariance matrix will be a symmetrical $m \times m$ matrix, with the diagonal K_{ii} being the variance on data set $\mathrm{X}_{i.}$:

$$\mathrm{K} = \begin{bmatrix} \mathrm{var}(\mathrm{X}_{1.}) & \mathrm{cov}(\mathrm{X}_{1.}, \mathrm{X}_{2.}) & \dots & \mathrm{cov}(\mathrm{X}_{1.}, \mathrm{X}_{m.}) \\ \mathrm{cov}(\mathrm{X}_{1.}, \mathrm{X}_{2.}) & \mathrm{var}(\mathrm{X}_{2.}) & & \\ \vdots & & \ddots & \\ \mathrm{cov}(\mathrm{X}_{1.}, \mathrm{X}_{m.}) & \mathrm{cov}(\mathrm{X}_{2.}, \mathrm{X}_{m.}) & & \mathrm{var}(\mathrm{X}_{m.}) \end{bmatrix}. \tag{4}$$

*e-mail: lukas.arnold@cern.ch

1.2 Applications and challenges

Covariance matrices serve a variety of purposes within particle physics, particularly as dimensionality reduction and filtering techniques. They are used within **Primary Component Analysis** (PCA) to decompose the eigenvalues from [1], and form the base of **Whitening transformation** [2]. Its most prevalent use in particle physics is its employment in the **Kálmán filter**. Kálmán filters are linear estimators that rely on covariance matrices of state vectors. They are widely applied as particle tracking algorithms, and their implementation onto FPGA forms one of the approaches to increase low-lever trigger capabilities for future luminosity upgrades. [3] [4]

Covariance matrices face the challenge of the *curse of dimensionality* – denoting the exponential increase in computation complexity with increasing dimensionality – leading to unbearable memory usage and computation times. To mitigate this obstacle, estimators such as the maximum likelihood estimator can often approximate the covariance matrix up to sufficient accuracy [5], but might be vulnerable to non-normally distributed random variables.

1.3 Related work

As the implementation base line we use the work of Perera and Li (2011) [6]. They implemented an accurate covariance matrix calculator on a Xilinx Virtex-6 FPGA clocked at 100MHz. Their design was tested on 8.56MB of integer input data, representing a fixed amount of 64 dimensions. The design is built upon two passes of data: Firstly, the mean is calculated according to eq. 2, and after the means $\overline{\mathrm{X}}_{i.}$ and $\overline{\mathrm{X}}_{j.}$ have been calculated, the corresponding covariance is calculated according to:

$$\mathrm{K}_{ij} = \frac{\sum_{k=1}^{n} \left(\mathrm{X}_{ik} - \overline{\mathrm{X}}_{i.}\right)\left(\mathrm{X}_{jk} - \overline{\mathrm{X}}_{j.}\right)}{n^2}. \tag{5}$$

This approach requires two passes of data: data has to be passed a first time to calculate the means – including a division – and then a second time to calculate the actual covariance value – with the number of mn^2 multiplications, as eq. 5 is performed for each covariance matrix value.

2 Theory

For our implementation, focus is lead on having a *single-pass* design that requires just one pass of data. The reasoning is that we expect to face a huge number of data samples within a lower ($\leq$ 160) range of dimensions in potential applications. Thus, the computation task is an *I/O-bound problem* limited by data transfer via cache line from CPU to FPGA up to the point where dimensionality exceeds parallelizability. Assuming that computation time is proportional to sample size, a single-pass design will be able to speed up every dual-pass design by the factor of two at least and it can potentially be used in real-time applications.

We use a novel decomposition of the covariance matrix:

$$\mathrm{K} = \frac{1}{n}\left(\mathrm{X}\mathrm{X}^{\mathrm{T}} - \frac{1}{n}\mathrm{X}\mathbb{1}\mathrm{X}^{\mathrm{T}}\right), \tag{6}$$

where $\mathbb{1}$ refers to an $n \times n$ matrix of ones.

2.1 Proof

The first part of the proof is commonly known: Taking the definition of covariance from eq. 1 with our understanding of the expected value given in eq. 2 and the definition of the covariance matrix in eq. 3 we come from

$$\mathrm{cov}(\mathrm{X}_{i.}, \mathrm{X}_{j.}) = \frac{1}{n}\left(\sum_{k=1}^{n}\left(\mathrm{X}_{ik}\mathrm{X}_{jk}\right) - \sum_{k=1}^{n}\left(\overline{\mathrm{X}}_{i.}\mathrm{X}_{jk}\right) - \sum_{k=1}^{n}\left(\overline{\mathrm{X}}_{j.}\mathrm{X}_{ik}\right) + \sum_{k=1}^{n}\left(\overline{\mathrm{X}}_{i.}\overline{\mathrm{X}}_{j.}\right)\right) \tag{7}$$

to

$$\underbrace{\mathrm{cov}(\mathrm{X}_{i.}, \mathrm{X}_{j.})}_{=\mathrm{K}_{ij}} = \frac{1}{n}\left(\underbrace{\sum_{k=1}^{n}\left(\mathrm{X}_{ik}\mathrm{X}_{jk}\right)}_{=\mathrm{A}_{ij}} - \underbrace{\frac{1}{n}\sum_{k=1}^{n}\mathrm{X}_{ik}\sum_{k=1}^{n}\mathrm{X}_{jk}}_{=\mathrm{B}_{ij}}\right). \tag{8}$$

For the second part, we use individual equivalencies to the terms A_{ij} and B_{ij} defined in the previous eq. 8. In the first part, we can use the general definition of matrix multiplications to derive

$$\mathrm{A}_{ij} = \sum_{k=1}^{n}\left(\mathrm{X}_{ik}\mathrm{X}_{jk}\right) = \mathrm{X}_{i.}\mathrm{X}_{j.}^{\mathrm{T}} = \left(\mathrm{X}\mathrm{X}^{\mathrm{T}}\right)_{ij}. \tag{9}$$

and for the second part, we use an n-dimensional *vector of ones* $\vec{\mathbb{1}}$:

$$\mathrm{B}_{ij} = \frac{1}{n}\sum_{k=1}^{n}\mathrm{X}_{ik}\sum_{k=1}^{n}\mathrm{X}_{jk} = \frac{1}{n}\sum_{k=1}^{n}\left(\mathrm{X}_{ik}\cdot\underbrace{1}_{=\vec{\mathbb{1}}_k}\right)\sum_{k=1}^{n}\left(\underbrace{1}_{=\left(\vec{\mathbb{1}}\right)_k^{\mathrm{T}}}\cdot\underbrace{\mathrm{X}_{jk}}_{=\mathrm{X}_{kj}^{\mathrm{T}}}\right) = \frac{1}{n}\left(\left(\mathrm{X}\vec{\mathbb{1}}\right)\left(\vec{\mathbb{1}}^{\mathrm{T}}\mathrm{X}^{\mathrm{T}}\right)\right)_{ij} \tag{10}$$

and – building an $n \times n$ *matrix of ones* $\mathbb{1}$ and using the definition of matrix multiplication again – we get:

$$\mathrm{B}_{ij} = \frac{1}{n}\left(\mathrm{X}\underbrace{\vec{\mathbb{1}}\vec{\mathbb{1}}^{\mathrm{T}}}_{=\mathbb{1}}\mathrm{X}^{\mathrm{T}}\right)_{ij} = \frac{1}{n}\left(\mathrm{X}\mathbb{1}\mathrm{X}^{\mathrm{T}}\right)_{ij}. \tag{11}$$

Eq. 9 and eq. 11 correspond to the term definitions in eq. 8, thus correspond to the decomposition in eq. 6.

Also, another approach is given. From eq. 8, it can be seen that

$$\mathrm{K}_{ij} = \overline{\mathrm{X}_{i.}\mathrm{X}_{j.}} - \overline{\mathrm{X}_{i.}}\cdot\overline{\mathrm{X}_{j.}}. \tag{12}$$

Equivalency of $\frac{1}{n}\mathrm{X}\mathrm{X}^{\mathrm{T}}$ to $\overline{\mathrm{X}_{i.}\mathrm{X}_{j.}}$ can be directly derived from eq. 9. The second part also holds:

$$\left(\frac{1}{n^2}\mathrm{X}\mathbb{1}\mathrm{X}^{\mathrm{T}}\right)_{ij} = \frac{1}{n^2}\sum_{k=1}^{n}\mathrm{X}_{ik}\cdot\left(\mathbb{1}\cdot\mathrm{X}^{\mathrm{T}}\right)_{kj} = \frac{1}{n}\sum_{k=1}^{n}\mathrm{X}_{ik}\cdot\frac{1}{n}\sum_{k=1}^{n}\left(1\cdot\mathrm{X}_{jk}\right) = \overline{\mathrm{X}_{i.}}\cdot\overline{\mathrm{X}_{j.}}. \tag{13}$$

3 Design

3.1 Algorithmic complexity

In order that our assumption on speed-up holds, the computation task has to be I/O-bound. As it would be the case in real-time applications, data will be sent sample by sample, or – in terms of the above-defined data matrix X – column by column. We will split our calculations

Table 1. Specifications for the Intel HARP v2 system [8].

CPU Family	Intel Xeon Broadwell E5
Number of CPU Cores	14
CPU speed	2.4GB/s
FPGA Family	Intel Arria 10
FPGA-CPU bandwidth	20GB/s

described in eq. 6: Firstly, we will calculate the values $\mathrm{X}_{.k}^{\mathrm{T}} \otimes \mathrm{X}_{.k}$ for each data sample k via tensor product, whose outputs then are added up. This exploits the equivalency

$$\mathrm{X}\mathrm{X}^{\mathrm{T}} = \sum_{k=1}^{n} \mathrm{X}_{.k}^{\mathrm{T}} \otimes \mathrm{X}_{.k} \left(\Leftarrow \text{eq. } 9 \wedge \left(\left(\mathrm{X}_{.k}^{\mathrm{T}} \otimes \mathrm{X}_{.k} \right)_{ij} = \mathrm{X}_{ik}\mathrm{X}_{jk} \right) \right). \tag{14}$$

Secondly, we will compute the sums $\sum_{k=0}^{n} \mathrm{X}_{ik}$ in parallel, as they lie base for calculating the second term B_{ik}, as can be derived from eq. 13.

The computationally most complex step is the tensor product that requires $O(m^2 + m)$ multiplications to be performed in parallel (m denotes the number of dimensions), in order that the calculation is a strictly I/O-bound problem, meaning that it is cache transfer and not compute that limits throughput. By omitting this necessity, we can increase dimensionality.

3.2 Hardware and framework

The covariance matrix accelerator is implemented onto a hybrid FPGA/CPU device: the Intel HARP. Intel HARP v2 contains both an FPGA and a CPU; the FPGA and CPU both have access to a shared memory region [7]. The specifications are listed in table 1. Software-wise, it was implemented using doppioDB, a hardware-accelerated database framework [9].

3.3 Implementation

A strictly I/O-bound solution only works to the number of dimensions where the tensor operation is fully parallelizable. For higher dimensions, a compromise has to be found. Thus, our design is split into two cases: The first case holds, if the number of dimensions is below

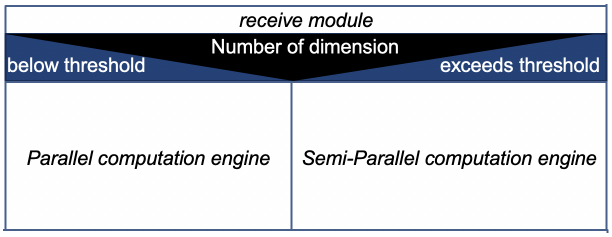
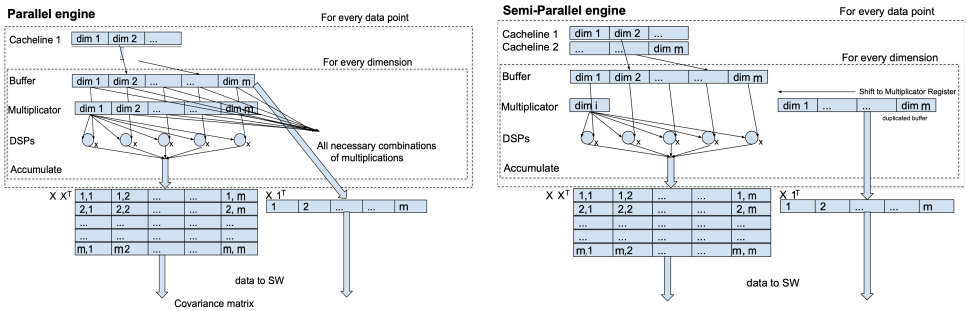


Figure 1. Which computation engine is used depends on dimensionality.

this threshold (which is by coincidence about the same as one cache line); in the second case where dimensionality exceeds parallelizability, an adjusted design will be used. The scheme is outlined in fig. 1 and a depiction of the data flow is illustrated for both cases in fig. 2. The computation engine is synthesized onto FPGA where it can take advantage of parallelizability of tensor operations using Digital Signal Processors (DSPs):



(a) Data flow in fully parallel mode.

(b) Data flow in case dimensionality exceeds parallelizability.

Figure 2. Data flow of the covariance matrix calculation within the FPGA. On the left side, the case is given where all the samples fit into the cache line in one go; on the right side, the case is illustrated where the samples have to be shifted into the multiplier DSP registers.

The engine receives data sample by sample. Then, the tensor operation is performed by Digital Signal Processors (DSPs). Depending on dimensionality, this is either done fully in parallel (fig. 2(a)) or batched by shifting the multipliers consecutively into the DSP registers. An accumulator than adds the DSP outputs to the XX^T matrix and the sample values to the $X\mathbb{1}X^T$ (sum) matrix.

The values are stored in RAM blocks that represent rows of the XX^T matrix. As this matrix is symmetrical, only the upper diagonal is stored. Data is sent sample per sample and cache line per cache line; thus, for lower dimensions, not the whole cache line is filled. Therefore, whereas the algorithm theoretically is I/O-bound, throughput is below the theoretical I/O-bound value. For values that require semi-parallel computation, computation time is bound by the time of shifting values to the DSP multiplication registers which is proportional to the number of dimensions; therefore, for higher dimensions, computation time will increase linearly with dimensionality.

After these operations are completed, the $m \times m$ output matrices are transferred to the CPU to perform the divisons. As dimensionality is assumed to be much lower than sample size, their computational costs are low. As these are mostly floating point operations, it would be disadvantageous to perform these on the FPGA, because this would either result in high hardware resource usage or high latency. The current design exploits advantages of both FPGA and CPU.

4 Results

4.1 Experimental setup

The test set-up uses homogeneously distributed random integer values. The sample size is set from 200k to 3G values, and the number of dimensions ranges from 2 to 160. These correspond to data sizes of 3MB to 10GB. The maximum number of dimensions is limited by the design; whereas there is no formal limit for sample size, the latter is constrained by accumulator overflow. Two different baselines are used:

- as the FPGA base line, the design by Perera and Li (2011) [6]. This design has – unlike the presented design – fixed dimensionality of 64.

- as the CPU base line, the NumPy covariance estimator, an estimation function run on an Intel i5 with 4 cores at 2.3GHz clock speed [10].

For both base lines, results are measured from the beginning of the compute until the results have been obtained by the host machine.

4.2 Comparison to FPGA designs

Throughput for the covariance matrix algorithm has been measured for data samples containing 2 to 160 dimensions, as presented in fig. 3. The threshold up to which the fully parallel data flow is used, is 16 dimensions. Up to this threshold, memory throughput is steadily increasing from 450MB/s to 3.9GB/s. For dimensions beyond that, throughput will drop sharply to – and stay constant at – 850MB/s, as the semi-parallel data flow is used and throughput is bound by computation time rather than I/O. The result shows a speed-up

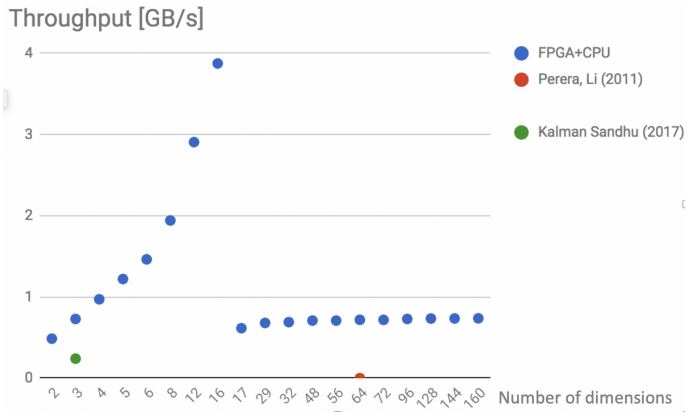


Figure 3. Throughput of the presented design, and the baselines [6], [11]. Note that the current implementation has variable input dimensionality, whereas the baseline designs have fixed dimensionality.

(increase of throughput) of up to 183000× compared to the base line (21.2kB/s [6]). The speed-up for dimensions above 16 is 40000×. For comparison, also a real-time covariance updating algorithm for the Kálmán filter has been included [11] for which the presented implementation achieves 3× higher throughput; but this yields limited significance due to the different nature of the algorithms. Compared to a standard CPU implementation of maximum likelihood covariance matrix estimation [10], significant speed-up is only achieved for number of dimensions using the fully parallel engine ($\leq$ 16 dimensions, up to 5×), but remains within the same order of magnitude for number of dimensions using the the semi-parallel engine.

5 Conclusion and outlook

A single-pass covariance algorithm design based on a novel decomposition of the covariance matrix has been implemented and successfully tested. It shows speed-up that is significant compared to previous FPGA implementation as well as compared to standard covariance estimation on lower number of dimensions. An outlook of a possible real-time application would be to use a novel covariance matrix: update: XX^T would be updated according to eq. 15 (k denotes the index of the new sample), the sums would be accumulated in parallel, and then

the covariance values would be calculated based on these updated values. This could provide an alternative to the currently used most common covariance updating algorithm [12]:

$$\mathrm{XX}^{\mathrm{T}}_{\mathrm{new}} = \mathrm{XX}^{\mathrm{T}}_{\mathrm{old}} + \mathrm{X}^{\mathrm{T}}_{.k} \otimes \mathrm{X}_{.k}. \tag{15}$$

Acknowledgments

The research has been performed within and has been supported by the Systems group, Department of Computer Science, at ETH Zurich. We express out gratitude to Gustavo Alonso, David Sidler and Kaan Kara for advice and support. We also thank Nevis Laboratories, Columbia University, for generous support to present the conducted research.

References

[1] D. Fernandez, C. Gonzalez, D. Mozos and S. Lopez. J Real-time Image Pr. **16(5)**, 1395-1406 (2019).

[2] W. Cho, S. Choi, D. Park, I. Shin and J. Choo. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* (IEEE, 2019), pp. 10639-10647.

[3] G. Cerati, P. Elmer, S. Krutelyov, S. Lantz, M. Lefebvre, K. McDermott, D. Riley, M. Tadel, P. Wittich, F. Wurthwein and A. Yagil, J Phys Conf Ser. **898(4)**, 042051 (2017).

[4] S. Summers and A. Rose, EPJ Web Conf. **214**, 01003 (2019).

[5] H. Li, S. Stoica and J. Li. IEEE Trans Signal Process. **47(5)**, 1314-1323 (1999).

[6] D. Perera and K. Li. *2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (IEEE, 2011), pp. 100-108.

[7] G. Alonso, Z. István, K. Kara, M. Owaida and D. Sidler. IEEE DE Bull. **42(2)** (2019).

[8] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon and P. Melet. Proceedings VLDB Endowment. **13(1)**, 71-85 (2019).

[9] D. Sidler, Z. István, M. Owaida, K. Kara and G. Alonso. *Proceedings of the 2017 ACM International Conference on Management of Data* (Association for Computing Machinery, 2017), pp. 1659-1662,

[10] S. van der Walt, S. Colbert and G. Varoquaux. Comput Sci Eng. **13(2)**, 22-30 (2011).

[11] F. Sandhu, S. Hazlina, S. Alavi and V. Mahalleh. IEEE Sens J. **17(17)**, 5749-5758 (2017).

[12] R. Frühwirth. Nucl Instrum Meth A. **262(2-3)**, 444-450 (1987).