Noname manuscript No.
(will be inserted by the editor)

Performance Modeling of the Sparse Matrix–Vector Product via Convolutional Neural Networks

**Maria Barreda · Manuel F. Dolz ·
M. Asunción Castaño ·
Pedro Alonso-Jordá ·
Enrique S. Quintana-Ortí**

Received: date / Accepted: date

Abstract Modeling the execution time of the Sparse Matrix-Vector multiplication (SpMV) on a current CPU architecture is especially complex due to *i)* irregular memory accesses; *ii)* indirect memory referencing; and *iii)* low arithmetic intensity. While analytical models may yield accurate estimates for the total number of cache hits/misses, they often fail to predict accurately the total execution time. In this paper, we depart from the analytic approach to instead leverage Convolutional Neural Networks (CNNs) in order to provide an effective estimation of the performance of the SpMV operation. For this purpose, we present a high-level abstraction of the sparsity pattern of the problem matrix and propose a blockwise strategy to feed the CNN models by blocks of non-zero elements. The experimental evaluation on a representative subset of the matrices from the SuiteSparse Matrix collection demonstrates the robustness of the CNN models for predicting the SpMV performance on an Intel Haswell core. Furthermore, we show how to generalize the network models to other target architectures to estimate the performance of SpMV on an ARM A57 core.

Keywords Sparse Matrix-Vector Multiplication (SpMV), Performance Modeling, Supervised Learning, Convolutional Neural Networks (CNN).

1 Introduction

The Sparse Matrix-Vector multiplication (SpMV) is an operation of utmost importance in many scientific and engineering applications. Indeed, it is a fundamental

Maria Barreda, Manuel F. Dolz, M. Asunción Castaño
Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I de Castelló, Spain
E-mail: {mvaya,dolzm,castano}@uji.es

Pedro Alonso-Jordá
Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
E-mail: palonso@upv.es

Enrique S. Quintana-Ortí
Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Spain
E-mail: quintana@disca.upv.es

kernel, for instance, in the iterative solution of sparse linear systems associated with partial differential equations (PDEs) arising in the analysis of the resistance of concrete structures, the estimation of the electron orbits, or the evaluation of the Earth's gravitational field, among many others [1]. The SpMV kernel also plays a fundamental role in various data-analytical processes, such as web search engines, information retrieval, or the creation of economic models [2].

In many of these applications, the SpMV kernel is one of the most time-consuming components, in part due to the memory-bound nature of this operation. Estimating the execution time of SpMV is an important yet challenging problem due to the combined effects of a number of properties intrinsic to the kernel: *i)* irregular memory accesses (low temporal locality); *ii)* indirect memory referencing (low spatial locality); and *iii)* low arithmetic intensity.[1] For instance, the authors in [3,4] found that typical sequential SpMV implementations generally achieve no more than 10% of the machine peak floating-point rate on commodity microprocessors. In the best case, the arithmetic intensity of the SpMV in FP32 is 0.5, meaning that the performance is generally bounded from above by the peak memory bandwidth of the target platform. Other key elements that dictate the performance of SpMV on a current computer architecture are the non-zero sparsity pattern and row-density in the sparse matrix. These elements, together with the algorithm, determine the sequence of memory accesses and, consequently, time-costly cache misses. Over the last few years, a significant research effort has been conducted with the purpose of modeling the performance of SpMV (see [5–7] and the references therein). These analytical models, however, often rely on simplified cache replacement policies and algorithmic costs which, in general, only provide theoretical estimations of the number of the memory-stalls and arithmetic operations, respectively. Futhermore, they require both a deep understanding of the processor architecture as well as a detailed analysis of the SpMV implementation [8].

On the other hand, Machine Learning (ML) is an alternative approach to analytical models for the derivation of mathematical models from sample data. Neural Networks (NNs) in particular have the ability to learn from a set of training data and approximate linear and non-linear functions. Specifically, Convolutional Neural Networks (CNNs) may provide a powerful means to capture spatial and temporal dependencies using abstract representations of the sparse matrices involved in the SpMV through a set of convolution filters that capture sophisticated relations.

In this paper, we leverage CNNs as a tool to visualize and identify complex, intricate interaction patterns and features present in the sparse matrix involved in the SpMV operation with the goal of providing an accurate estimate of the execution time on a CPU core. In particular, our paper makes the following contributions:

- We leverage CNNs to model the execution time of the SpMV on a recent Intel Xeon core using CSR [9] as the storage format. The approach though carries over to any other specialized format for sparse matrices.
- We propose a blockwise realization to make the CNN model architecture-independent of the dimensions of the sparse matrix, which helps at increasing the amount of training/validation data.

[1] The arithmetic intensity is defined as the ratio of total floating-point operations to total data movement (in bytes).

- We evaluate the accuracy and demonstrate the robustness of the CNN-based models using a representative subset of cases arising from real applications collected in the SuiteSparse Matrix collection.
- We migrate the CNNs to an ARM architecture to assess the generality of the proposed models.

The most obvious application for our offline cost estimator is that, given trained models for a variety of processor architectures, choosing the best option among them does not require direct access to the processors. In particular, inference can be run offline on the (trained) models, on a single architecture different from that which the networks reflect. Also, being able to estimate the execution cost of an irregular and challenging operation such as SpMV paves the road toward applying similar ML-driven techniques to modeling the cost of memory accesses for more complex numerical kernels or even general-purpose applications.

The rest of this paper is organized as follows. Section 2 reviews some basic concepts about SpMV and CNNs. Section 3 describes the strategy to accommodate the CSR format as a valid input for the CNN models and details the selected CNN regression-based architecture. Section 4 evaluates the training process of the proposed CNNs tuned with hyperparameter optimization; analyzes the accuracy attained by the networks for SpMV; and migrates the models to a different architecture. Section 5 revisits a few other works related to performance modeling and/or linear algebra operations using NNs. Finally, Section 6 offers a few concluding remarks and summarizes future research lines.

2 Background

In this section, we briefly review the SpMV kernel and some basic aspects of CNNs. These concepts are the basis for the CNN models that are introduced in this work to estimate the time cost of SpMV.

2.1 The sparse matrix-vector product

Consider the SpMV operation $y = Ax$, where A is a sparse (input) matrix of size $m \times n$, consisting of nnz non-zero elements; x is a dense input vector, of size n; and y is the dense output vector, of size m. In this operation, the elements in the matrix A are usually stored using a compressed format, such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate (COO) or Ellpack (ELL) [9]. In this work, we target CSR as this variant provides a flexible, memory-efficient, and architecture-agnostic solution.

The CSR format stores the matrix using three arrays (vectors) that contain the non-zero values, the row pointers, and the column index of each element, making efficient use of the memory and permitting fast row accesses. Figure 1 provides a simple example of a 4×4 matrix stored in this format. There:

- The array $vval$, of length nnz, holds the non-zero entries of A in row-major order.
- In the array $vptr$, of length $n + 1$, the difference between the elements $i + 1$ and i specifies the number of non-zero elements in the i^{th} row of A. $(2 - 0 = 2$ in the first row, $3 - 2 = 1$ in the second row, etc.)

Algorithm 1 Realization of the SpMV algorithm using the CSR format.

```
Require: $A \rightarrow m \times n, x \rightarrow n, y \rightarrow m$
1: for $i = 1, 2, \ldots, n$ do
2:     for $j = A.vptr[i], A.vptr[i] + 1, \ldots, A.vptr[i + 1] - 1$ do
3:         $y[i] := y[i] + A.vval[j] \cdot x[A.vpos[j]]$
4:     end for
5: end for
```

– The entries of the array *vpos* specify the column index of each matrix entry of A, and hence, it is of length *nnz* as well. (We assume 0-based indexing.)

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 6 \end{pmatrix} \qquad \begin{aligned} vval[6] &= \{1,\ 2,\ 3,\ 4,\ 5,\ 6\} \\ vptr[5] &= \{0,\ 2,\ 3,\ 5,\ 6\} \\ vpos[6] &= \{0,\ 3,\ 2,\ 1,\ 2,\ 3\} \end{aligned}$$

Fig. 1: Example of a matrix (A) stored in CSR format.

Algorithm 1 shows the implementation of the SpMV kernel where the matrix A is stored using such CSR format. There, the outer loop (indexed by i) iterates over the matrix rows, while the inner loop (indexed by j) moves through the entries of each row, using the *vptr* and *vpos* arrays to retrieve the proper index to access x. Finally, *vval* is used to retrieve the value at the coordinate (i, j) of A. As previously mentioned, the irregular memory accesses of this implementation occur while retrieving the elements of the x vector, which are indirectly accessed through the *vpos* array.

2.2 Convolutional Neural Networks

CNNs are a class of neural networks which are very efficient at identifying patterns in data classification problems [10]. Indeed, CNNs are mainly used in pattern recognition tasks with the purpose of mimicking the functionality of the receptive fields in the human visual cortex. In general, a CNN of L layers consists of a collection of C convolutional (CONV) layers, usually arranged in the first layers of the network, followed by a reduced number of $F = L - C$ fully-connected (FC) layers in the final stages of the network. The neurons in a CONV layer l are connected to a small subset of neurons in layer $l - 1$, and they are activated according to the result of the convolution operation on n-dimensional filter or kernel. The dimensions of this kernel in a 1D convolution, for instance, are $k \times c_{l-1}$, with k being the filter size and c the number of channels in layer $l-1$. A CONV layer can combine multiple filters, each responsible for detecting a complex nonlinear feature and producing a single feature map (or channel) at layer l. The resulting activations in such feature maps are then passed through a nonlinear function, such as the Rectified Linear Unit (ReLU), which has reported to achieve fast training in supervised learning of deep neural networks [11].

Many CNN architectures exhibit a common organization. For instance, the first CONV layers aim at detecting basic patterns, while the subsequent ones try to identify high-level complex features. To reduce the input data dimensionality, as it is processed through the network, pooling layers are placed in-between successive

CONV layers. Pooling layers semantically merge similar information by creating downsampled and summarized versions of the features detected by the CONV layers. Consequently, they descrease the number of parameters and computation in the network. Typical pooling operations are the maximum and average functions.

CNNs may also contain dropout layers, which are inserted to improve the learning process and to counter the network overfitting. The idea underlying this type of layers is to ignore a set of neurons that is chosen with a given probability in a certain pass of the trainig process. This permits the co-dependency among neurons which in turn leads to a better generalization on new input data. CNNs may also include transformation layers, also known as "batch normalization" [12]. The purpose of these layers is to normalize the activations of a layer at each mini-batch so that the mean activation and the standard deviation are close to 0 and 1, respectively. This approach usually permits the use of higher learning rates while it makes the training less sensitive to weight initialization.

Once all feature maps are processed through the set of C CONV layers, the set of F FC layers generates a result for the CNN. An FC layer l connects each of its neurons to all neurons in layer $l-1$, following the same principles as those of traditional multilayer perceptrons (MLPs). Depending on whether the CNN is a realization of a classification problem or a regression model, layer L may contain as many neurons as classes or a single one. In the former case, the neurons are activated via non-linear functions (e.g. sigmoid, softmax or ReLU), while in the latter a linear function activates the single neuron.

In the subsequent sections, we describe in detail the architecture of our CNNs for modeling the execution time of SpMV based on regression.

3 Modeling SpMV using CNNs

In this section, we present our methodology to estimate the performance of SpMV using CNNs. The memory-bound nature of SpMV is due to the low non-zero densities and the irregular sparsity patterns in matrix A which, in general, dictate a considerable volume of cache misses and DRAM memory accesses to retrieve the entries of the x vector during the operation. Taking this into account, the *vpos* array can be regarded as a key element to understand the distinct arithmetic-to-memory access intensities and predict the global execution time of the operation. With this idea in mind, we design a CNN where the inputs are the values of the array *vpos*, from the CSR format. This vector represents a one-dimensional image of the sparse matrix A that captures the order in which the entries of x are retrieved from memory and the distances between consecutive accesses to this vector. Once trained, the filters in the convolutional layers should be capable of capturing meaningful features in the *vpos* array, such as patterns of distances between non-zero entries, that yield useful estimations of the SpMV performance via the relation between flops and cache hits/misses.

3.1 Methodology

Figure 2 depicts the methodology proposed to tackle the SpMV modeling problem. As mentioned at the beginning of this section, the goal is to design a CNN that

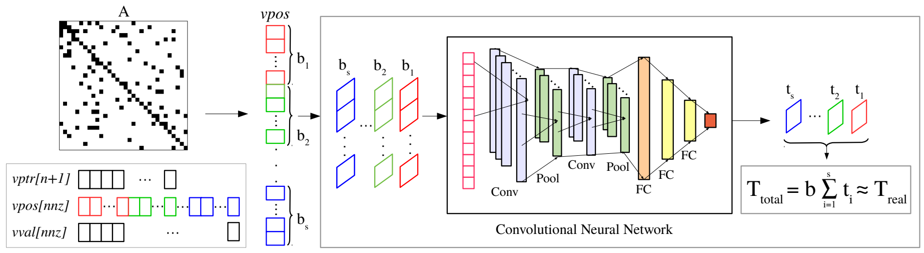


Fig. 2: Workflow for modeling the SpMV performance.

receives the *vpos* array as an input. However, given that sparse matrices may present large variations on their size and number of nonzero entries (nnz), we propose to split the *vpos* array into chunks (or blocks) of size b so as to obtain a CNN design with a constant number of inputs. The main advantage of this approach is that the CNNs can be used uniformly to predict individual execution times of equal-sized blocks, which may belong to any sparse matrix, regardless of its size and nnz value. Thus, considering that t_i is the execution time per non-zero element of the i-th block of A, the estimated total execution time for this matrix can be calculated as the aggregation of the time-per-element for the $\lceil nnz/b \rceil = s$ blocks multiplied by b; that is, $T_{total} \approx b \sum_{i=1}^{s} t_i$. Our design iteratively "feeds" each block of *vpos* to the trained CNN so that the outputs provided by the network correspond to estimates of the partial execution time (per element) of this block, which can then be summed up in order to obtain the total execution time of the SpMV for the matrix A. Note that if the number of nnz entries is not multiple of b, the remaining block of size $b' < b$ is discarded. In our case, this is safe as the smallest value of nnz for the selected sparse matrices is always higher than 5 M. In consequence, with $b = 5{,}000$, only one block among 1,000 will be discarded.

The partitioning of *vpos* into blocks, though, forces us to implement a blockwise version of the classic CSR-based SpMV Algorithm 1. This modification is required to generate the training dataset for the CNNs, as each block of *vpos* has to be labeled with its corresponding execution time per number of non-zero elements. For instance, the input training data for a matrix A can be decomposed into a set of s blocks of the *vpos* array along with their corresponding execution time per non-zero element (t_{nnz}). Algorithm 2 presents the procedure for computing the SpMV by blocks of b non-zero elements. For each block, the algorithm computes the product of the non-zero elements (according to *vpos*) by the corresponding values of the input vector x. The result is then stored in the analogous position of the output vector y (lines 14 and 21). Note that the algorithm includes the timing instructions (lines 11 and 20) that gather the labels to later train the CNN.

3.2 Network architecture

Considering the proposed blockwise strategy, the next step is to design a CNN architecture that offers accurate estimates of the execution time for SpMV. For that purpose, we design a CNN-based regression model, where the activation of the single output neuron is the estimated value for t_{nnz}.

Algorithm 2 Blockwise realization of the SpMV algorithm based on the CSR format.

```
Require: $A \to m \times n, x \to n, y \to m, A.rows \to nnz, b \to block\_size$
 1: for $i = 0, 1, \ldots, m-1$ do
 2:     for $j = A.vptr[i], A.vptr[i]+1, \ldots, A.vptr[i+1]-1$ do
 3:         $A.rows[j] := i$
 4:     end for
 5: end for
 6: $start := 0$
 7: $end := \min(b, nnz)$
 8: while $start < nnz$ do
 9:     $prv := m$
10:     $aux := 0.0$
11:     start_timer()
12:     for $i = start, start+1, \ldots, end-1$ do
13:         if $A.rows[i] > prv$ then
14:             $y[A.rows[i-1]] := y[A.rows[i-1]] + aux$
15:             $aux := 0.0$
16:         end if
17:         $aux := aux + A.vval[i] \cdot x[A.vpos[i]]$
18:         $prv := A.rows[i]$
19:     end for
20:     stop_timer()
21:     $y[prv] := y[prv] + aux$
22:     $start := end$
23:     if $(start + b) < nnz$ then
24:         $end := start + b$
25:     else
26:         $end := nnz$
27:     end if
28: end while
```

For the network architecture, we initially mimic the general structure of state-of-the-art neural networks (e.g., AlexNet, LeNet or VGG), in which the common trend is the stacking of blocks of convolutional layers combined with a pooling layer at the end; see Section 2.2. Specifically, we opted for a sequence of one or two convolutional layers (C), followed by a pooling layer (P), and repeated this sequence twice. We denote these two distinct models as CPCP and CCPCCP, respectively. We also considered the placement of dropout operators after the pooling layers to reduce overfitting. In some cases, normalization layers were also included between the last convolutional and pooling layers to adjust and scale the activations among batches. For the second part of the network, we appended some FC layers that combine the features detected in the previous convolutional stages. Finally, the last FC layer generates the estimated output. It is important to remark that some configuration parameters of the networks are set as hyperparameters, e.g., the number of stacked FC layers at the end of the net, the number of neurons per FC layer, the percentages of dropout in the corresponding layers, and the number of kernels in the convolutional layers.

Following the previous ideas, we designed two different CNN architectures that combine both arrangements of convolutional layers, we denote them as CNN-R1 and CNN-R2, depending on the CPCP and CCPCCP arrangement used, respectively.

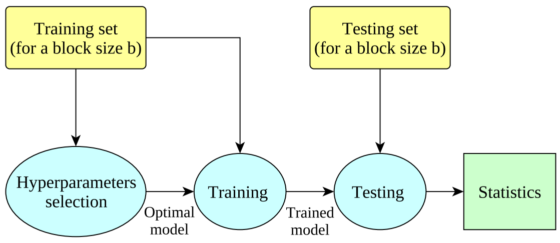


Fig. 3: Evaluation workflow.

4 Experimental evaluation

In this section, we describe *i)* the generation of the training and testing datasets; *ii)* the hyperparameter search for the proposed CNNs; *iii)* the training process and analysis of the models' performance in terms of validation accuracy and loss-related metrics; *iv)* the testing evaluation with a set of sparse matrices through relative mean errors of the SpMV execution time; and *v)* the migration and evaluation of the models for estimating the SpMV execution time when executed on a distinct target architecture. To carry out these tasks of the experimental evaluation, we have employed the following hardware and software components:

- *Hardware:*
  - The compute node where the networks were trained consisted of two Intel Xeon E5-2698, with a total of 40 cores clocked at 2.20 GHz, and four NVIDIA Tesla P100 GPUs with 16 GB of DRAM at 1.48 GHz interconnected via NVLink.
  - The execution times corresponding to the SpMV operation were obtained first on an Intel Xeon E5-2630 core at 2.40 GHz, hereafter referred to as Haswell. To test the model migration technique, the SpMV operation was then re-run on an ARM Cortex-A57 core at 2.00 GHz embedded on a NVIDIA Jetson TX2, hereafter A57.
- *Software:* The machine learning framework for building the networks was Keras v2.2.4 [13] on top of TensorFlow r1.10 [14]. Moreover, we employed Hyperas v0.4.1 [15], a wrapper around Hyperopt [16] that implements an hyperparameter optimization algorithm for Keras-based models that leverages Bayesian search algorithms such as the tree of Parzen estimators [17]. Finally, the ad-hoc SpMV benchmark was implemented in C and compiled with GCC 5.3.0 with the usual optimization flags.

The training and evaluation workflow is depicted in Figure 3. First, the training and testing datasets for a given block size of *b* are built. Note that the datasets are labeled by executing the SpMV operation on the Haswell core. Next, we obtain the optimized versions of the models by performing a hyperparameter search. Afterward, we train the models using the previous training dataset. Finally, we apply them repeatedly (i.e., to each block of the sparse matrix) to estimate the total execution times of the SpMV operation (inference) and compute the relative errors with respect to the real execution times for each sparse matrix in the testing dataset.

4.1 Obtaining the dataset

The training and testing datasets were obtained by realizing the blockwise SpMV operation, as detailed in Algorithm 2, for the selected sparse matrices while measuring the time-per-nonzero, t_{nnz}, for each of the $vpos$ blocks. For this purpose, we selected 173 sparse matrices with a number of nonzeros ranging between 1 M and 10 M, from the SuiteSparse Matrix Collection [18]. From these 173 matrices, 108 (63%) were selected for training, while the remaining 65 (37%) were reserved for testing. Similarly, 80% of the training dataset was employed only for training, and the remaining 20% was used for validation so as to prevent overfitting and guide the training process.

To analyze the impact of the block size, i.e., the number of non-zero elements per chunk, we also experiment with different values of $b \in \{250, 500, 750, 1{,}000, 3{,}000, 5{,}000\}$. For that purpose, we obtained different training, validation and testing datasets for each value of b. Note that a concrete sparse matrix with nnz nonzero entries yields $\lceil nnz/b \rceil$ blocks that are part of the respective dataset. In the end, each block is labeled with its corresponding t_{nnz}, together with the number of rows/columns and nnz elements of the associated sparse matrix. It is important to remark that we do not use block sizes below 250 given that their execution time may be biased by inherent cache data locality effects.

4.2 Building and tuning the models

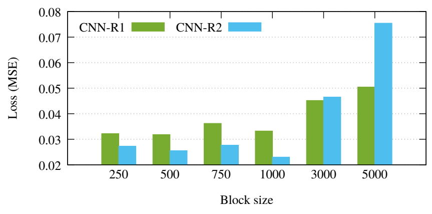
The proposed CNN models were implemented using Keras on top of TensorFlow, for each of the selected block sizes (i.e., inputs to the CNN). During the model building process, we identified a set of hyperparameters that have to be properly set prior to perform the training phase. These hyperparameters comprise the number of kernels in CONV layers, the kernel dimensions, the number of FC layers and number of neurons in each, the batch size, the learning rate, the dropout percentages, and the optimization algorithm.

Setting and testing each of the hyperparameters combinations manually is a cumbersome and prone-error process, as the search space is too large for a complete search within a manageable amount of time. To deal with this problem, we employed the Hyperas tool, a wrapper around Hyperopt for hyperparameter optimization of Keras models which partially search the parameter space. To use Hyperas, we define the range of values that the hyperparameters can take in order to allow the algorithm to find fair configurations.

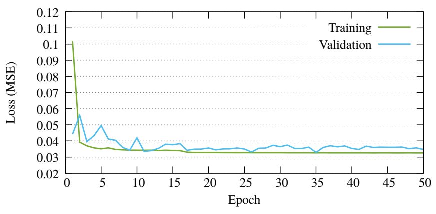
Table 1 shows the number of times, expressed in percentage, that a given hyperparameter was chosen by Hyperas within the proposed convolutional models and the selected block sizes. We denote this metric as *percentage of choice*. The values in the tuple (k_1, k_2) appearing in columns of Table 1 labeled as "*number of kernels in blocks of CONV layers*" stand, respectively, for the number of filters used in the first (k_1) and second (k_2) block of CONV layers in both CNN-R1 and CNN-R2 models. Note that the table cells are colored from green to red representing percentages from 0% to 100%, respectively. Regarding the number of kernels in the CONV layers, the preferred option consisted of 32 and 64 filters in the first and second block of the CONV layers, respectively. With respect to the kernel size, all tested values for the number of FC layers and the number of neurons per

Hyperparameter	# of kernels in blocks of CONV layers		Kernel size in CONV layers				# of FC layers			# of neurons in FC layers			SGD optimizer			Initial learning rate			Batch size			
	(16, 32)	(32, 64)	3 × 1	5 × 1	7 × 1	9 × 1	1	2	3	10	100	1,000	Adam	SGD	RMSprop	0.1	0.01	0.001	128	256	512	1,024
CNN-R1	50.0	50.0	20.0	20.0	33.3	26.7	50.0	33.3	16.7	10.0	60.0	30.0	100.0	–	–	–	83.3	16.7	–	50.0	16.7	33.3
CNN-R2	83.3	16.7	25.0	12.5	31.3	31.3	33.3	66.7	–	20.0	–	80.0	66.7	16.7	16.6	–	50.0	50.0	50.0	33.3	–	16.7

Table 1: Percentages of choice for the CNN models hyperparameters.



(a) Validation loss (MSE) for varying block sizes.



(b) Progress of the training and validation accuracy using the CNN-R2 model with $b = 250$.

Fig. 4: Validation and progress of the training for the CNN models.

FC layer are equally valid. In contrast, the preferred SGD optimizer is Adam with an initial learning rate value of 0.001. Note that in our experiments we use an adaptive learning rate that multiplies by 0.1 when the mean squared error (MSE) does not improve any longer. Regarding the batch size, a generally good choice is 256.

4.3 Training process

In supervised learning, the entire training dataset is commonly divided randomly into the training (in-sample) and validation (out-of-sample) (sub)sets. First is used for estimating the model weights (and biases), while the second is only leveraged to guide the training process. Although this practice is well established, the random division of a sample collection may bias the weight estimation, affecting the performance of the models. For this reason, an important step prior to the remaining experimentation is to use a cross-validation scheme to analyze the performance behavior among k data folds. Cross-validation is a technique to assess how the models generalize given an unseen independent dataset [19].

In this study, we use 5-fold cross-validation, where the dataset is divided into five equal-sized partitions, retaining a single partition as validation data and the remaining four as training data. The process is repeated five times, with each of the five subsamples selected exactly once as validation data. The results showed that the training-validation partitioning does not affect the models' performance, as there are enough representative data in the considered partitions. Therefore, from now on, we pick the first 20% samples of the entire training set for validation.

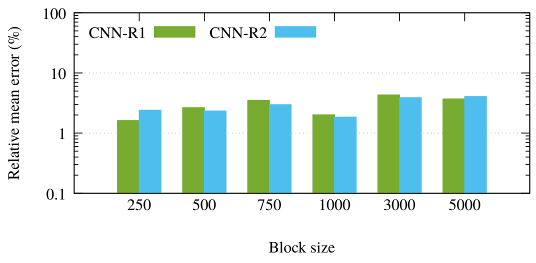


Fig. 5: Relative mean error in the execution time using different block sizes on Haswell.

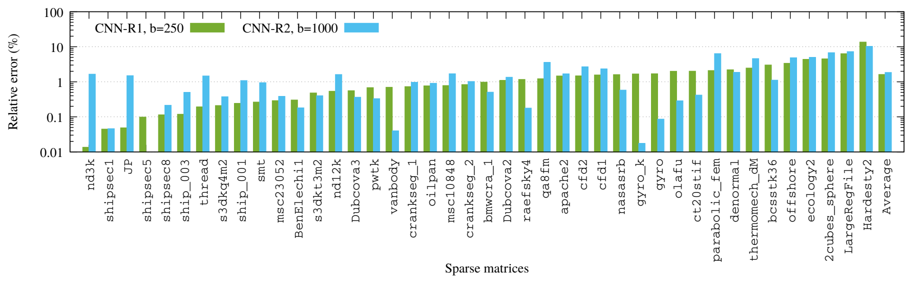


Fig. 6: Relative error in the execution time using the optimal block size on Haswell.

The training of the proposed CNNs aims to minimize the loss function, given the regression based nature of the model. For that, we train the optimized versions of the models according to the selected hyperparameters with the previous datasets. Figure 4a shows the MSE validation loss metric, expressed in ns^2, used to measure the model quality for the CNN-R1 and CNN-R2 models using the collection of block sizes. In the plot, we observe a clear trend for both models showing that the MSE increases with the block size. Also, the MSE for CNN-R2 is lower than that for CNN-R1 when small block sizes are adopted, which indicates that a Conv-Conv-Pool pattern with small block size delivers higher accuracy. From this analysis, we can conclude that the best choice is to employ a small block size and the CNN-R2 model.

To gain further insights into the evolution of the MSE for the training and validation data, Figure 4b displays the learning curves for the CNN-R2 model and $b = 250$. In this case, both the training and validation MSE losses stabilize after 15 epochs, where the validation accuracy has a slightly superior loss during the 50 training epochs.

4.4 Testing the models

Once the models are trained, the next step is to evaluate them using the testing dataset, which is composed of 65 unseen sparse matrices. For that, we use the relative mean error (RME) as a metric to account for the average relative error

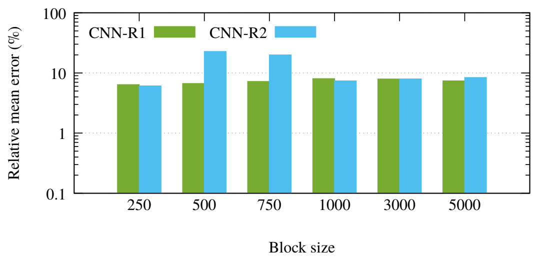


Fig. 7: Relative mean error in the execution time using different block size on A57.

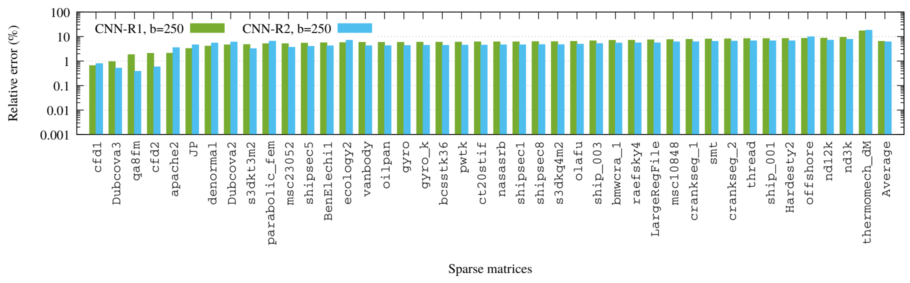


Fig. 8: Relative error in the execution time using the optimal block size on A57.

between the predicted and measured total execution time for all test matrices, i.e.,

$$RME = \frac{1}{p}\sum_{i=1}^{p}\frac{|predicted_i - measured_i|}{measured_i},$$

where p is the total number of matrices in the testing dataset.

Figure 5 reports the RME for the selected block sizes in the testing dataset. In general, the RME metric ranges between 1% and 7%, which indicates that the models provide fairly good estimations. Regarding the models, we do not observe a significant difference between the alternative convolutional configurations. We also notice that a small block size yields a lower RME. In addition, given that the error is below 10%, the strategy of discarding the last block of *vpos* array which is not entirely filled with values does not significantly affect the prediction performance.

As a complementary experiment, Figure 6 exposes the relative error (RE) per test matrix for the CNNs and the best block sizes (as determined in the previous analysis). There, the matrices in the plot are sorted according to the RE value delivered by CNN-R1. The results show that the RE is below 1% for roughly half of the tested matrices for both model architectures.

4.5 Cross-architecture model migration

To validate the generalization of the CNNs, we migrate the models to estimate the execution time of the SpMV operation on a different architecture. This migration is performed in order to store previously-gained knowledge and apply it to a related problem. In our case, we preserve the model features (i.e., CNN architecture and chosen hyperparameters), and re-train all their weights. The adopted training and testing datasets contain the same matrix used for the Haswell architecture, but the corresponding t_{nnz} values included in these new datasets are those obtained when the SpMV operation is realized on an ARM A57 core. It is important to remark that both A57 and Haswell core architectures are quite different in terms of cache hierarchy, a key feature in a memory-bound operation performance. For instance, the Haswell core has a 32-KiB L1 and a 256-KiB L2 associative caches with a 20-MiB L3 cache, while the A57 core is furnished with a 32-KiB L1 cache and a 2-MiB shared L2 cache. Moreover, the degree of associativity and replacement policies of the caches are also different. In general, thanks to this technique, we avoid designing the CNN and calculating the hyperparameters from scratch when the model has to predict the SpMV execution time on a different architecture.

Figure 7 presents the RME obtained by the models and the set of block sizes using the new dataset and the re-trained models. In general, the RME values are higher for the CNNs than those observed for the Haswell core. This is partly due to having inherited the optimal hyperparameters for the Haswell core. Although the weights can converge to optimal values during the re-training process, the hyperparameters are fixed and may not be the best options for the new target architecture. However, the RME results, ranging between 5% and 15% in most cases, still provide fairly good estimations. Thus, for example, the RME for the CNN-R1 model is always below 10%. Despite the increased RME values for all models when the target core is A57 instead of Haswell, we can observe that the model migration approach is an appealing technique as it avoids the cost of hyperparameter search.

Figure 8 displays the RE per sparse matrix for both models using the optimal block sizes (according to the previous experiment) while estimating the execution time on the A57 core. As illustrated there, the CNN-R1 and CNN-R2 attain similar RE for all sparse matrices.

In a separated experiment, we also used a transfer learning approach [20], i.e., preserving all the weights except those of the last FC layer. However, the RME values obtained by the models were much higher than those achieved re-training all the weights.

5 Related work

CNNs are key tools in supervised learning that date back to the 1960s and 1970s, though deep learning (i.e., machine learning via deep neural networks) has recently become very popular due to the adoption of modern accelerator architectures and the data deluge. In addition, tools like Keras [13] and TensorFlow [14] have contributed to making neural networks and deep learning more accessible. Deep learning techniques via CNNs outperform other mechanisms due to the integration of data feature extraction within the training process and the ability to deal with large data sets. Consequently, CNNs have been successfully applied in many machine learning-related areas [10, 21, 22]. However, the potential of deep learning is still largely unexplored in linear algebra. In particular, only a few works, briefly described next, have previously addressed problems related to performance modeling and/or the SpMV operation using the deep learning paradigm.

Götz and Anzt [23] view the matrix sparsity pattern as an image to train a CNN that is capable of detecting strongly connected blocks in order to derive block-Jacobi preconditioners. These preconditioners are then used to accelerate the iterative solution of the corresponding linear system. Similarly, Zhao et al. [24] leverage CNNs to select the most adequate storage format to store the sparse matrix involved in the SpMV operation. They also employ transfer learning to alleviate cross-architecture migration for CNN-based models to select the optimal matrix storage format. Cui et al. [25] also apply CNNs to determine the best-performing implementation of the SpMV operation from a given input sparse matrix. Nisa et al. [26] predict the best format for SpMV by feeding the networks with different sparsity features of the processed matrices, instead of using them as 2D images, as it was the case in the three previous works. The neural models adopted in this work were MLPs and had the same topology for all the experiments.

With respect to performance modeling, Tiwari et al. [27] employ MLPs to estimate the performance, power and energy usage of certain computational kernels. Benatia el al. [28] train a variety of MLPs to predict the GPU performance of the SpMV kernel for different types of matrix storage formats. The inputs of the MLPs, in this case, are a few sparsity features, which depend on the corresponding sparse matrix format to be evaluated. Nisa et al. [26] carry out a similar work assuming two different GPU architectures with simple and double-precision formats and different sparsity features. For each of these combinations, they train an MLP to predict the execution time of the SpMV operation for all the adopted sparse formats and another MLP for all the formats.

All in all, the afore-mentioned performance modeling works are different to the technique proposed in this paper in three main aspects: *i)* we leverage CNNs instead of MLPs, so that our approach permits capturing spatial features of the sparse matrices; *ii)* we feed the CNNs directly with the sparse matrix structure instead of using sparse matrix-related metrics that summarize specific features; and *iii)* our blockwise methodology allows creating large datasets with a reduced number of sparse matrices.

6 Conclusions and future work

We have proposed a collection of CNNs to estimate the execution time of the SpMV operation, a memory-bound kernel with important applications in many scientific and engineering problems. Moreover, modeling the execution of the SpMV operation plays an important role when there is no direct access to the target platform. In this sense, the CNN models capture the complex patterns and features of the sparse matrix (stored in CSR format), which basically dictate the irregular accesses to the dense input vector. In order to make the CNN architecture independent of the sparse matrix size, we leverage a blockwise strategy. Furthermore, we tackle the performance estimation problem via regression realizations, resulting in different alternatives for the CNN architecture. With these CNN designs, we performed a hyperparameter optimization search on the number of filters in CONV layers, the number of FC layers and neurons per layer, optimization algorithm, and batch size, among others.

During the experimental evaluation, we trained the networks on a set blocks of sparse matrices from the SuiteSparse matrix collection labeled with the corresponding SpMV execution time on a Haswell core. The relative mean error for the set of testing matrices for all proposed models ranges between 1% and 7%. The results also reveal that network architectures deliver accurate results for small block sizes. This is because, in general, small blocks reflect a small set of sparsity patterns which, in turn, can be better captured by the CNN filters. Consequently, having small block sizes increases the execution time variability among blocks, so the predictions of t_{nnz} vary in a wider interval. In contrast, working with large block sizes leads to homogeneous execution time labels, preventing the CNN to learn important sparsity features comprised in a single block.

Finally, we re-trained the models obtained for the Haswell architecture using a second dataset with the block labels corresponding to the execution time of SpMV on an A57 core. In this case, while the relative mean error of the re-trained models increases to 15%, the use of this technique avoids the hyperparameter search, demonstrating that the models can be re-trained and re-used to estimate the execution time of the same operation on a distinct target architecture. With this study, we also demonstrate that a simple network architecture, inspired by the structure of well-known CNNs, such as AlexNet, LeNet or VGG, can be powerful enough to deliver accurate execution time estimations of the SpMV kernel.

As future work, we plan to extend the CNNs to estimate the execution time and energy consumption of the parallel implementation of the SpMV operation. We will also analyze network architectures which can accept hardware-dependent information, such as unchangeable constants of the target platform, e.g., cache sizes, CPU frequency, memory bandwidth, etc. An ultimate goal is to leverage this methodology to model the execution time and energy consumption of more complex linear algebra operations or even building blocks employed in neural network frameworks such as CONV layers.

Acknowledgments

This work was supported by project TIN2017-82972-R from the *MINECO*, Spain. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 from the *Generalitat Valenciana*, Spain. Maria Barreda was also supported by the POSDOC-A/2017/11 project from the *Universitat Jaume I*.

References

1. Ahmad Abdelfattah, Hatem Ltaief, and David Keyes. High performance multi-gpu SpMV for multi-component pde-based applications. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 601–612, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
2. William E Schiesser. *Computational mathematics in engineering and applied science: ODEs, DAEs, and PDEs.* CRC press, 2014.
3. Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, jan 2005.
4. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, Nov 2007.

5. A. Elafrou, G. Goumas, and N. Koziris. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 292–301, Aug 2017.
6. ShiGang Li, ChangJun Hu, JunChao Zhang, and YunQuan Zhang. Automatic tuning of sparse matrix-vector multiplication on multicore clusters. *Science China Information Sciences*, 58(9):1–14, Sep 2015.
7. Ping Guo and Liqiang Wang. Accurate crossarchitecture performance modeling for sparse matrixvector multiplication (SpMV) on GPUs. *Concurrency and Computation: Practice and Experience*, 27(13):3281–3294, 2015.
8. K. Li, W. Yang, and K. Li. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):196–205, Jan 2015.
9. Victor Eijkhout and Roldan Pozo. Data structures and algorithms for distributed sparse matrix operations. Technical report, 1994.
10. Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77(C):354–377, May 2018.
11. Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.
12. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pages 448–456, 2015.
13. Keras: The Python Deep Learning library. `https://keras.io/`.
14. TensorFlow, an open source machine learning library for research and production. `https://www.tensorflow.org/`.
15. Keras + Hyperopt: A very simple wrapper for convenient hyperparameter optimization. `http://maxpumperla.com/hyperas/`.
16. James Bergstra, Dan Yamins, and David D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms.
17. J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages I–115–I–123. JMLR.org, 2013.
18. SuiteSparse Matrix Collection. `https://sparse.tamu.edu/`.
19. Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.
20. Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22(10):1345–1359, October 2010.
21. Jrgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.
22. Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.
23. Markus Götz and Hartwig Anzt. Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation. In *Procs of ScalA18: 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, WS at Supercomputing 2018*, 11 2018.
24. Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the gap between deep learning and sparse matrix format selection. *SIGPLAN Not.*, 53(1):94–108, Feb. 2018.
25. Hang Cui, Shoichi Hirasawa, Hiroaki Kobayashi, and Hiroyuki Takizawa. A machine learning-based approach for selecting SpMV kernels and matrix storage formats. *IEICE Transactions on Information and Systems*, E101.D(9):2307–2314, 2018.
26. Israt Nisa, Charles Siegel, Aravind Sukumaran Rajam, Abhinav Vishnu, and P Sadayappan. Effective machine learning based format selection and performance modeling for SpMV on GPUs. EasyChair Preprint no. 388, EasyChair, 2018.
27. A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely. Modeling power and energy usage of HPC kernels. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 990–998, May 2012.
28. A. Benatia, W. Ji, Y. Wang, and F. Shi. Machine learning approach for the predicting performance of SpMV on GPU. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 894–901, Dec 2016.