When Polyhedral Optimizations Meet Deep Learning Kernels

Abhishek A. Patwardhan
Indian Institute of Technology Hyderabad
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Akilesh Badrinaaraayanan
Indian Institute of Technology Hyderabad
6 PUBLICATIONS **6** CITATIONS
SEE PROFILE

When Polyhedral Optimizations Meet Deep Learning Kernels

*Hrishikesh Vaidya[1], *Akilesh B[2], *Abhishek A Patwardhan[3], Ramakrishna Upadrasta[4]
[1,2,3,4] Dept of Computer Science and Engineering, IIT Hyderabad, India
{cs13b10[35[1],42[2]], cs15mtech11015[3], ramakrishna[4]}@iith.ac.in

***Abstract*—Deep Neural Networks (DNN) are well understood to be one of the largest consumers of HPC resources and efficiently running their training and inference phases on modern heterogeneous architectures (and accelerators) poses an important challenge for the compilation community. Currently, DNNs are actively being studied by the automatic parallelization and polyhedral compilation communities for the same purpose. In this (initial) paper, we study the kernels of four varieties of DNN layers with the goal of applying automatic parallelization techniques for latest architectures. We show the *affine (Polyhedral) nature* of these kernels thereby showing that they are amenable to well known polyhedral compilation techniques. For benchmarking purposes, we implemented forward and backward kernels for four varieties of layers namely convolutional, pooling, recurrent and long short term memory in `PolyBench/C`, A well known polyhedral benchmarking suite. We also evaluated our kernels on the state-of-art `Pluto` polyhedral compiler in order to highlight the speedups obtained by automatic loop transformations.**

I. Introduction

Machine Learning (ML) techniques are being extensively used for solving real world problems in various domains. In applications from Computer Vision and Natural Language Processing (NLP), Neural Network (NN) models are trained in order to learn a pattern, after which the model can be used for an unseen input. Due to extensive usage of high resolution graphics and large textual datasets, the real-world HPC requirements of DNNs is quite large. This makes the application of compiler optimizations, parallelization (and tuning) strategies to the training and inference phase as a vital key to effectively parallelize and optimize these computations.
In this paper, we discuss the issues that arise from a *per-layer* implementation of the main classes of DNNs as a *specific variety* of (affine) Polyhedral loop programs. Our implementation focuses on following the PolyBench/C [7] structure, a widely used benchmark for Polyhedral compilation tools. While two of DNN implementations are perfectly polyhedral codes, presence of *stride parameters* makes the remaining two non-polyhedral. We describe a practical way in which they can be turned into polyhedral programs. As a proof of concept of our implementation, as well as to show the potential of polyhedral compilation framework on DNNs, we optimize our kernels using a well known polyhedral compiler Pluto [2], to study the speedups obtained by applying a complex sequence of loop transformations. We see our work as the first step in automatically generating accelerator specific kernels using advanced polyhedral compilation techniques. The larger goal of this study is aimed at applying polyhedral techniques to automatically generate accelerator specific efficient programs on various architectures.

II. Motivation

Artificial Neural Networks (ANNs) are biologically inspired from interactions of neurons in the human brain. ANNs consist of several layers with nodes in each layer obtaining input from nodes in the previous layer via interconnections between layers. The activation of a node is determined by the input values and the weights on the connections between the inputs and the nodes. The training phase involves updating the weights on interconnections so that expected results are obtained at the output layer. The forward pass involves executing the network on new training inputs, while backward pass updates the weights to reduce the deviation from the expected output. Deep NNs typically comprise of a large number of layers, and their design is crucial to improve the accuracy of the network.
We now briefly discuss some widely used classes of deep NNs and state their broad application domain. Convolutional Neural Networks (CNNs) are a class of NNs widely employed for image and video data. In the recent past, CNNs have achieved tremendous improvement in accuracies for several computer vision tasks [6, 8].

Convolutional network consists of one or more (convolutional) layers often accompanied with a subsampling layer and fully connected layers; the convolutional layers account for roughly 80% of the computation time. Pooling layer is a type of layer within a deep CNN, which summarizes the input presented to it. Since CNNs are compute intensive, pooling helps to compress the data as it flows through the deep net. Recurrent Neural Networks (RNNs) have become a de facto for modeling sequential dependencies in discrete time series, useful in context driven tasks (NLPs). Long Short Term Memory (LSTM) network is another variant of RNN specialized for improving accuracy during learning phase.

Deep learning workloads are computationally intensive and manually optimizing their kernels on a variety of modern parallel architectures (and accelerators) is a challenging task. In this paper, we try to explore potential of automatic parallelization for deep learning kernels.

The rest of this paper is organized as follows: In Section III, we give a quick overview of polyhedral compilation. Then, in Sections IV–VII, we present the computational kernels corresponding to forward and backward phases of various deep NNs. In Sec. VIII, we discuss the performance improvements obtained by loop transformations. Finally, in Sec. IX, we state conclusions and some future work.

III. Polyhedral Compilation

The Polyhedral model focuses on optimizing and parallelizing the loop nests. It is a powerful formalism to analyze and transform the input affine programs so as to run them on varieties of modern heterogeneous architectures. It can statically analyze programs which involve affine loop bounds and affine array access functions. Typically, a polyhedral compiler first creates a model of input loop nest. A statement nested within a d deep loop nest is represented as d-dimensional polyhedron where each integral point represents dynamic instance of that statement. After extracting such a representation, data dependence analysis—a well studied problem that boils down to solving an integer linear programming problem [3]—is performed. This analysis finds the (dynamic) instances of two possibly different statements which access the same array location, and at least one of the accesses is a write. Such an analysis is important to preserve the semantics of original program. The second step of polyhedral compilation is the affine scheduling problem [4], that involves finding a complex sequence of classical loop transformations (such as loop tiling, permutation, skewing) which expose the parallelism and improve the data locality. A number of approaches exist to find a good program transformations from a large search space, and one practical approach involves scheduling two dependent statement instances as much closer (in time space) as possible. The above approach was first implemented in the Pluto source to source compiler [2] which we use in this work. The final step of polyhedral compilation involves generating a loop nest which scans all valid integer points in polyhedra [1]. The parallel loops is be marked with apt pragmas (like OpenMP, OpenACC) during code-generation.

Deep Neural Network layers	**BLAS / HPC kernels**
Convolutional layer	Stencils, tensor multiplication
Recurrent layer	Stencil with varying time steps
LSTM	Set of Matrix vector products
Max, sum Pooling	Max/Sum reductions

TABLE I: Correspondance among DNN layers and HPC kernels

TABLE II: Program Parameters for CNN

N	Number of Input Images in batch
C	Number of Input feature maps
K	Number of Output feature maps
P × Q	Size of output feature map
R × S	Size of filter kernel
U,V	Stride parameters

In recent past, polyhedral compilation has shown to be effective in accelerating various linear algebra kernels, tensor contractions, stencils, image processing applications etc. We make the *crucial observation* that many of the layers used in deep learning pipelines perform computations that are similar to the ones polyhedral compilation has been successful in optimizing. It is known that entries in the column 2 of above table are well optimized by polyhedral compilers. In this paper, we try to explore how polyhedral model optimizes various deep learning layers given the close correspondence as depicted in table I. We also release the NN kernels. Earlier researchers who worked only on CNN [9] did not release their code (Neither HDL nor HLS/C code) to open-source. To the best of our knowledge, there is no known implementation of DNNs (**as (affine) Polyhedral programs**) for benchmarking purposes. With this paper we overcome the above limitation.

```
for (n = 0; n < _PB_NN; n++)
  for (k = 0; k < _PB_NK; k++)
    for (p = 0; p < _PB_NP; p++)
      for (q = 0; q < _PB_NQ; q++)
        for (c = 0; c < _PB_NC; c++)
          for (r = 0; r < _PB_NR; r++)
            for (s = 0; s < _PB_NS; s++)
              out_F[n][k][p][q] += W[k][c][r][s] *
                 inp_F[n][c][u*p+NR-r-1][u*q+NS-s-1];
```

Fig. 1: C Code : CNN forward pass

IV. CNN

The program parameters of a CNN are described in Table II. For parallelizing purposes, the CNN program

TABLE III: Program Parameters for Pooling

N	Number of Input images
D	Number of feature maps
(IH,IW)	Size of input feature map
(OH,OW)	Size of output feature map
(DH,DW)	Size of Pooling kernel
(SH,SW)	Horizontal and vertical stride values

can be thought of as a stencil (with uniform dependences) defined over a loop nest of depth seven, with the loop body computing convolution. A quick study of the dependences of the code shows that all four outer dimensions, namely n, k, p, q, are completely parallel. The array index expression for *inp* array accesses the appropriate location in the input feature map accounting for inverting and striding. Though our current implementation assumes absence of padding in the input filters, though it can be added at a later time. The array access is

```
for(n = 0; n < _PB_NN; n++)
  for (c = 0; c < _PB_NC; c++)
    for (h = 0; h < _PB_NH; h++)
      for (w = 0; w < _PB_NW; w++)
        for (k = 0; k < _PB_NK; k++)
          for (r = 0; r < _PB_NR; r++)
            for (s = 0; s < _PB_NS; s++)
              for (p = 0; p < _PB_NP; p++)
                for (q = 0; q < _PB_NQ; q++)
                  if((u*p - (h - NR + r + 1) ==0) && (u*q - (w - NS + s + 1) ==0))
                    err_in[n][c][h][w] += W[k][c][r][s] * err_out[n][k][p][q];
```

Fig. 2: C Code : CNN backward pass

clearly non-affine (due to the multiplication of the stride parameter with the corresponding indices). The reader is referred to Section V for a note on affinity of CNN and MaxPool. In the backward pass, the error information is propagated from output of a layer to its input. *err_out* contains the error derivative with respect to the output of the layer. To compute the error derivative with respect to the input, *err_out* is multiplied with values from weight matrix W to accumulate values into *err_in* matrix.

V. Max Pooling

Pooling is a form of layer usually added after convolutional layer in CNN to reduce the spatial size of the representation in the network. The program parameters for max pooling operation are provided in Table III. In MaxPooling, the maximum input value within the window is termed as the output of the operation as shown in Algo. 1. Only the maximum value of input window contributes to the output value. During backpropagation phase (shown in Algo. 2), the error derivative with respect to output is added only to the input pixels which have contributed to the output value.

Affinity of CNN and MaxPool: A central operation in CNN is *convolution* which accesses the array index expression by multiplying the *stride parameter* with the loop dimension to get the required offset in the input image. Though this makes the array access function non-affine, as these stride parameters are constant integer literals for each individual layer within a DNN, and are fixed while designing the network, we fix them statically. A similar strategy was used by Zang et al. [9] who used Polyhedral techniques for FPGA code generation. The same argument applies MaxPool layer as well.

Algorithm 1 Max pooling layer: Forward pass

```
Require: N, D, IH, IW, DH, DW, SH, SW, inp[N][D][IH][IW]: Input
OH ← (IH - DH)/SH + 1
OW ← (IW - DW)/SW + 1
∀ $(n \in N, d \in D, r \in OH, c \in OW\ )$ do {
  val ← MIN_INT
  for each $h \in [SH * r, min(SH * r + dh, ih))$ do
    for each $w \in [SW * c, min(SW * c + dw, iw))$ do
      val ← MAX(val, inp[n][d][h][w])
    end for
  end for
  out[n][d][r][c] ← val
}
```

Algorithm 2 Max pooling layer: Backward pass

```
Require: N, D, IH, IW, DH, DW, SH, SW
Require: inp[N][D][IH][IW], err_out[N][D][OH][OW]: Input data
OH ← (IH - DH)/SH + 1
OW ← (IW - DW)/SW + 1
∀ $(n \in N, d \in D, r \in OH, c \in OW\ )$ do {
for each $h \in [SH * r, min(SH * r + dh, ih))$ do
  for each $w \in [SW * c, min(SW * c + dw, iw))$ do
    if out[n][d][r][c] == inp[n][d][h][w] then
      err_in[n][d][h][w] += err_out[n][d][r][c]
    end if
  end for
end for
}
```

VI. RNN

The unique aspect of RNNs is the feedback loop where the output of the neuron is passed as input to the same neuron leading to a recurrence in time dimension. The presence of feedback loop introduces a set of dependences during both forward and backward phases of the network. The layer has three weight matrices namely U, V, W which are *learnt* during back-propagation phase. During back-propagation, the error of output neuron is propagated T steps back in time. The kernel parameters for a typical RNN is shown in Table IV.

Algorithm 3 RNN layer: Forward pass

```
Require: BT, T, P, Q, S
Require: U[S][P], W[S][S], V[Q][S]: Weight matrices
Require: state(t), input(t): Vector of size S, P resp.
state(0) ← U * input(0)
output(0) ← V * state(0)
for each $t \in [1, T)$ do
  state(t) ← U * input(t) + W * state(t-1)
  output(t) ← V * state(t)
end for
```

TABLE IV: Program Parameters for RNN

T	Number of time steps
P	Size of input vector
Q	Size of output vector
S	Size of hidden vector
BPTT	Truncated Unroll factor

TABLE V: Program Parameters for LSTM

T	Number of time steps
P	Size of input vector
Q	Size of output vector
S	Size of hidden vector

As described in Algo. 3, in the forward pass of RNN, $state(t)$ denotes hidden state vector at each time step and similarly $output(t)$ is the output at timestep t. The hidden state ($state(t)$) computation at time step t uses information of current input vector and hidden state vector of previous time step. U and W are multiplied with $input(t)$ and $state(t-1)$ respectively and the quantities are added to get the final result. The output vector is obtained by computing an inner product of V and current hidden state i.e. $state(t)$.

Algorithm 4 RNN layer: Backward pass

```
Require: BT, T, P, Q, S, (U[S][P], W[S][S], V[Q][S]): Weights
Require: $err_{out}$(t), state(t), input(t): Vector of size Q, S, P resp.
for each $t \in [T-1, 1]$ do
    $err_V += err_{out}(t) * state(t)$
    $err_S^A[1:r]$ = V * $err_{out}(t)$
    for each $step \in [t+1, max(0, t-BT))$ do
        if $step > 0$ $err_W += err_S^A[1:r] * state(step-1)$
        $err_U += err_S^A[1:r] * input(step)$
        $err_S^B += err_S^A[1:r] * W$
        $err_S^A[1:r] = err_S^B[1:r]$
    end for
end for
```

We describe the backward pass in Algo. 4, where the error derivatives are summed up for each time step t. This computes the error accumulation of gradient using chain rule. The err_S^B acts as an intermediate vector during the back-propagation step to store the error derivative with respect to the hidden state vector $state(t)$ represented as err_S^A in the Algorithm.

VII. LSTM

LSTM is a special kind of RNN, designed to combat vanishing gradients [5] through a gating mechanism. A typical LSTM layer is comprised of a forget gate, an input gate and an output gate. Each gate masks some information (from the stream of data flowing through the network) propagating through itself, or its previous layers depending on the type of gate. The parameters required to describe LSTM are given in Table V.

In Algo. 5, $input_{gate}, forget_{gate}, output_{gate}$ represent the input, forget and output gates respectively, and work like masks. The $input_{gate}$ decides to what extent the current input contributes to the newly computed state $memory(t)$. The $forget_{gate}$, defines the factor of the previous state which is retained in the current state. Finally, the $output_{gate}$, defines how much of the internal state is exposed to the external network (that is, to the subsequent layers and to the next time step as well).

Algorithm 5 LSTM Neural Network layer: Forward pass

```
Require: T, P, Q, S, (input(t), state(t)): Vectors of size P, S resp.
Require: $W_i$[S][S], $W_f$[S][S], $W_o$[S][S], $W_g$[S][S]: Weight matrices for hidden state.Suffix
    represent gate type (input, forget, output, hidden)
Require: $U_i$[S][P], $U_f$[S][P], $U_o$[S][P], $U_g$[S][P]: Weight matrices for input state.
for each $t \in [1, T)$ do
    $input_{gate}$[1:S]←input(t)*$U_i$+state(t-1)*$W_i$
    $forget_{gate}$[1:S]←input(t)*$U_f$+state(t-1)*$W_f$
    $output_{gate}$[1:S]←input(t)*$U_o$+state(t-1)*$W_o$
    $cand_{state}$[1:S]←input(t)*$U_g$+state(t-1)*$W_g$
    memory(t)←memory(t-1)*$forget_{gate}$+ $cand_{state}$*input(t)
    state(t)←memory(t)*$output_{gate}$
end for
```

$cand_{state}$ is a *candidate* hidden state that is computed based on the current input and the previous hidden states. The method of computing $cand_{state}$ is the same as that of computing $state(t)$ in a RNN, except that the parameters U and W are replaced with U_g and W_g. $memory(t)$ can be considered as internal memory of the unit, which is a sum of two components: a) $memory(t-1)$ multiplied by the forget gate $forget_{gate}$, b) newly computed candidate hidden state $cand_{state}$ multiplied by the input gate $input_{gate}$. In other words, it is a combination of how we want to combine the new input with previous memory. Given the $memory(t)$, the output $state(t)$ is computed by multiplying the $memory(t)$ with the output gate $output_{gate}$.

The back-propagation phase for LSTM(Alg. 6) consists of computing errors for vectors representing various gates(input/output/forget/cand_state). Using these, errors for weight matrices are computed. Notice that, while computing errors for U_i, U_g, U_f, U_o, `input(t)` gets multiplied with the error values for a gate. While, error computation of W_i, W_g, W_f, W_o requires `state(t)`. This is so because during forward phase U_i, U_f, U_o, U_g represents weight matrices for `input(t)` and W_i, W_f, W_o, W_g represents weight matrices for candidate hidden state.

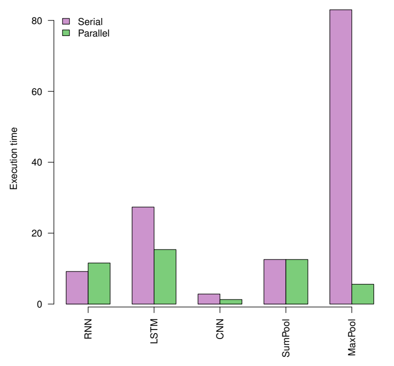

Fig. 3: Execution times for forward pass

VIII. Performance analysis

To study speedups obtained by applying polyhedral transformations, we use Pluto [2] (version 0.11.4), a widely used source-to-source polyhedral optimizer with `--tile --parallel` flags. All experiments were performed with the data set sizes set to the PolyBench variable `EXTRA-LARGE`. We compiled the parallel codes generated by Pluto using GCC-7.0.0, and OpenMP-4.5 for execution. The experiments were performed on Intel(R) Xeon(R) CPU E5-2630 v3@2.40GHz cluster having two processors with each processor having 8 hardware cores. We ran each program three times by using benchmarking script bundled within PolyBench, which internally runs it five times. We selected median of three trials as the execution time. We separately recorded the execution times for serial and parallel versions for both forward and backward phases.

Algorithm 6 LSTM Backward pass

```
Require: T, P, Q, S, $W_i$[S][S], $W_f$[S][S], $W_o$[S][S], $W_g$[S][S], $U_i$[S][P], $U_f$[S][P], $U_o$[S][P],
         $U_g$[S][P], $output_{gate}$[1:S],$input_{gate}$[1:S],$forget_{gate}$[1:S], $cand_{state}$[1:S]
Require: memory(t), input(t) Vector of size S, P resp
 1: for each $t \in [T-1, 1)$ do
 2:     $err^g_{output}$ = memory(t)*$err_{state(t)}$
 3:     $err_{memory(t)}$ += $output_{gate}$[1:S]*$err_{state(t)}$
 4:     $err^g_{forget}$ = memory(t-1)*$err_{memory(t)}$
 5:     $err_{memory(t-1)}$ += $forget_{gate}$[1:S] * $err_{memory(t)}$
 6:     $err^g_{input}$ = $cand_{state}$[1:S] * $err_{memory(t)}$
 7:     $err^g_{cand\_state}$ = $input_{gate}$[1:S] * $err_{memory(t)}$
 8:     $err\_U_{i/g/f/o}$+=$input(t)$*($err^g_{input/cand\_state/forget/output}$)
 9:     $err\_W_{i/g/f/o}$+=$state(t)$*($err^g_{input/cand\_state/forget/output}$)
10:     $err_{state(t-1)}$ += $W_i$*$err^g_{input}$ + $W_f$*$err^g_{forget}$ + $W_o$*$err^g_{output}$ +
    $W_g$*$err^g_{cand\_state}$
11: end for
12: ▷ Note: Line 8,9 defines 4 statements, with one to one correspondence between LHS and RHS
    alternatives
```

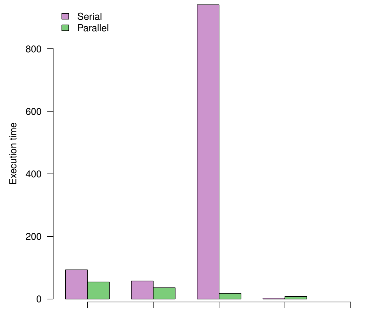

Fig. 4: Execution times for backward pass

The plots showing execution times for forward and backward phases are given in 3 and 4 respectively. We make following observations from plots: 1) Backward phase is more compute intensive than forward phase for all layers except SumPooling. 2) RNN, LSTM consist of Polyhedral loops that are successfully parallelized by Pluto. 3) For CNN and Maxpool, we were forced to replace the stride parameters with integer constants defined in our header file. This made CNN and Maxpool (forward pass) analyzable for Pluto. 4) The backward phase of MaxPooling kernel consists of a data dependent condition which Pluto's dependence analysis is unable to analyze. 5) No speedups were observed for forward phase of RNN and backward phase of SumPooling. 6) Average speedups observed for forward and backward phases are 2.15 and 2.69 respectively.

Type	Forward	Backward
CNN	2.21x	51.78x
RNN	0.79x	1.71x
LSTM	1.77x	1.59x
MaxPool	14.84x	NA

TABLE VI: Speed-up over serial execution

IX. Conclusions And Future Work

We implemented the four varieties of neural network layers as loop-programs in the PolyBench framework. While RNN and LSTM strictly adhere to Polyhedral framework's affinity conditions, CNN and MaxPool do not and we had to fix the stride parameters of their codes manually. The programs show *significant speedups* after applying polyhedral transformations. We released our PolyBench/NN C implementation https://github.com/hrishikeshv/polybench/tree/master/polyNN so that other researchers can work on advanced optimizations on these kernels. Though our work is *preliminary*, we believe it will form a basis to apply automatic loop transformations that expose parallelization as well as data locality optimization opportunities for different DNN architectures on various heterogeneous architectures and accelerators.

Acknowledgments: The authors are thankful to Prof. Sanjay Rajopadhye for motivation and encouragement.

References

[1] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on PACT*, 2004.
[2] Uday Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*. ACM, 2008. http://pluto-compiler.sourceforge.net/.
[3] Paul Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 1991.
[4] Paul Feautrier. Efficient solutions to the affine scheduling problem. *IJPP*, 92.
[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
[6] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 2012.
[7] Louis-Noël Pouchet et al. Polybench Benchmarks. https://sourceforge.net/projects/polybench/.
[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[9] Chen Zhang et al. Optimizing fpga-based accelerator design for deep cnns. In *ACM/SIGDA International Symposium on FPGAs*, 2015.