PolyBench/NN: Adding Neural Networks to PolyBench (Short Paper/Poster)

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

Neural Networks are currently being studied by the Polyhedral Compilation community so as to propose automatic parallelization techniques for latest architectures. In this short paper/poster, we discuss four varieties of neural network layers viz convolutional, pooling, recurrent and long short term memory. For benchmarking purposes, we implemented these programs in the PolyBench/C as well as in the PolyBench/Julia frameworks. We also evaluated our benchmarks on the PLuTo polyhedral compiler in order to study speedups obtained by automatic loop transformations.

1. INTRODUCTION

With the recent trend of using Machine Learning(ML) techniques extensively in solving real world problems, it has become important to parallelize and optimize these kernels. In computer vision and Natural Language Processing(NLP) applications, Neural Network(NN) models are trained in order to learn a pattern, after which the model can be used to classify unseen input. Due to extensive usage of high resolution graphics and audio, it is important to efficiently train such networks on real world inputs. This makes application of compiler optimizations, parallelization (and tuning) strategies vital. In this short paper, we discuss implementation of some varieties of NNs as a set of loop programs.

As PolyBench/C [6] is a widely used benchmark for Polyhedral tools, we implemented these kernels by following the PolyBench/C structure. While three of our implementations are perfectly polyhedral codes, whereas for others, presence of *stride parameters* makes them non-affine. However, these stride parameters typically are just constant integer literals for that particular layer, thereby allowing application of polyhedral optimizations onto them.

We evaluate our codes using PLuTo [2], a polyhedral compiler to study speedups obtained by loop transformations. The larger goal of this study is aimed at applying polyhedral techniques to widely used architectures and provide a basis to study further optimization opportunities.

Rest of the paper is organized as follows: In Section 2, we describe motivation. Then, in Sections 3—6, we describe some deep learning architectures widely used by ML researchers. We also present the computational kernels corresponding to forward and backward phase. In Section 7, we briefly discuss our re-implementation of NNs in PolyBench/Julia. In Section 8, we discuss the performance improvements obtained by transformations. Finally, In Section 9, we state conclusions and some future work.

2. MOTIVATION

Deep NNs have been successful at solving many tasks of utmost importance. However, deep learning workloads are computationally intensive, and manually optimizing their kernels is not only a challenging and time-consuming process, but is also an error-prone one.

Artificial Neural Networks (ANNs) are biologically inspired from interactions of neurons in the human brain. ANNs mimic the human brain activities by framing the neural structure as a computational model. ANNs consist of several layers with each layer containing nodes obtaining input from nodes in the previous layers via interconnections between layers. The activation of a node is determined by the input values, and the weights on the connections between the input and the nodes. The learning process occurs using backpropagation in which the weights on connections are modified based on the error generated at the output layer. The error derivative is back-propagated through the network, and the weight update is performed using gradient descent to obtain a minimum error value.

NN models are implemented as programs that perform various array operations (for instance multiplication, reductions etc.) possibly lying within a deep loop nest, and therefore are best candidates for polyhedral loop transformations.

We now briefly discuss some widely used classes of deep NNs and state their broad application domain.

Convolutional Neural Networks (CNNs) are an important and promising class of NNs largely employed for image and video data. In the recent past, CNNs have achieved tremendous improvement in accuracies for several computer vision tasks [5],[8]. Convolutional network consists of one or more convolutional layers often accompanied with a subsampling layer, and then followed by one or two fully connected layers, with convolutional layers accounting for roughly 80% of the computation time.

Pooling layer is a type of layer within a deep CNN, which summarizes the input features presented to it, reducing the spatial size of the representation and thereby requiring storage of lesser number of parameters. Since CNNs are compute intensive, pooling helps to compress the feature map as it flows through the deep net. There are many ways to implement pooling depending upon the type of reduction operator. Among the many alternatives like max, avg, MaxPool layer is the most used one.

Recurrent Neural Networks (RNNs) have become a de facto for modelling sequential dependencies in discrete time series, and hence widely useful in context driven tasks. RNNs are extensively used for applications in NLP and text-processing.

Table 1: 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

Table 2: 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

Long Short Term Memory(LSTM) network is another variant of RNN specialized so as to improve accuracy during learning phase. They were proposed to get rid of the vanishing gradient problem observed in RNNs [4].
LSTMs employ memory cells to store long term dependence information. A set of gates is used to control the context being stored when an input is presented to the memory cell. The gating mechanism decides what information to send as output and what to forget. This architecture mitigates the problem of vanishing gradient and helps the LSTMs to learn long term dependencies.

Algorithm 1 Convolutional layer: Forward pass

Require: N, K, C, H, W, R, S, U, V
Require: inp[N][C][H][W]: Input data
Require: W[K][C][R][S]: Weight matrix for a given layer
1: P ← (H - R)/U + 1
2: Q ← (W - S)/V + 1
3: $\forall(n \in N, k \in K, p \in P, q \in Q, c \in C, r \in R, s \in S$) **do** {
4: out[n][k][p][q] += W[k][c][r][s] * inp[n][c][u*p+R-r-1][u*q+S-s-1]
5: }

In subsequent sections, we discuss each class of NN model, and also show computational kernel for a single layer within a network.

3. CNN

The program parameters of CNN are described in Table 1. For parallelizing purposes, the CNN program 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.

Algorithm 2 Convolutional layer: Backward pass

Require: N, K, C, H, W, R, S, U, V
Require: inp[N][C][H][W]: Input data
Require: W[K][C][R][S]: Weight matrix for a given layer
1: P ← (H - R)/U + 1
2: Q ← (W - S)/V + 1
3: $\forall(n \in N, c \in C, h \in H, w \in W, k \in K, r \in R, s \in S, p \in P, q \in Q$) **do** {
4: **if**((u*p-(h-R+r+1)=0) **and** (u*q-(w-S+s+1)=0))
5: err_in[n][c][h][w] += W[k][c][r][s] * err_out[n][k][p][q]
6: }

The array index expression for *inp* array in Algorithm 1 accesses the appropriate location in the input feature map accounting for inverting and striding. It should be noted that our current implementation assumes absence of padding within input data set. The array access is clearly non-affine (due to the multiplication of the stride parameter with the corresponding indices). The reader is referred to Section 4.3 for a note on Affinity of CNN and MaxPool.
In the backward pass 2, 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.

4. POOLING

Pooling is a form of layer usually added after convolutional layer in CNN to reduce the dimensions of input space. Two variants namely, Max-Pooling and Sum-Pooling, have been implemented. In both cases, a fixed sized window slides over the input feature map which maps the values in the window to a single scalar value. Depending on the stride value, overlapping of adjacent windows may occur as well. The program parameters for pooling operation are provided in Table 2.

4.1 MaxPooling

Algorithm 3 Max pooling layer: Forward pass

Require: N, D, IH, IW, DH, DW, SH, SW
Require: inp[N][D][IH][IW]: Input data
1: OH ← (IH - DH)/SH + 1
2: OW ← (IW - DW)/SW + 1
3: $\forall$ ($n \in N, d \in D, r \in OH, c \in OW$) **do** {
4: val ← MIN_INT
5: **for each** $h \in [SH * r, min(SH * r + dh, ih))$ **do**
6: **for each** $w \in [SW * c, min(SW * c + dw, iw))$ **do**
7: val ← MAX(val, inp[n][d][h][w])
8: **end for**
9: **end for**
10: out[n][d][r][c] ← val
11: }

In MaxPooling, the maximum input value within the window is termed as the output of the operation as shown in Algo. 3. Only the maximum value of input window contributes to the output value. During backpropagation phase (shown in Algo. 4)), the error derivative with respect to output is added only to the input pixels which have contributed to the output value.

4.2 SumPooling

In SumPooling, the input values lying within window are summed up to get the final output value. During backward pass, the error derivative with respect to output is added to all input pixels which contributed to the output pixel value during forward pass. Please refer to 3 for code listing.

Algorithm 4 Max pooling layer: Backward pass

Require: N, D, IH, IW, DH, DW, SH, SW
Require: inp[N][D][IH][IW]: Input data
Require: err_out[N][D][OH][OW]: Input data
OH ← (IH - DH)/SH + 1
OW ← (IW - DW)/SW + 1
$\forall\ (n \in N, d \in D, r \in OH, c \in OW\ )$ **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
}

Table 3: 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

4.3 Affinity of CNN and MaxPool

A central operation in CNN is convolution which involves non-affine array accesses, due to stride parameters. Stride parameter gets multiplied with loop dimension, to get the required offset in the input image, making the array index expression to be non-affine.
However these stride parameters are constant integer literals for each individual layer within a deep neural net and are pre-decided while designing the network. Hence the program depicting functionality of entire neural network can be fitted into the polyhedral model. A similar strategy was used by Zang et al. [9] who used Polyhedral techniques for FPGA code generation. The same argument applies to affinity condition of MaxPooling as well.

Algorithm 5 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): Vector of size S
Require: input(t): Vector of size P
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

5. RNN

RNNs are recurrent in time dimension. The unique aspect of RNNs is the feedback loop where the output of the neuron is passed as input to the same neuron. 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 and 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 3.
As described in Algorithm 5, 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. $state(t)$.

Table 4: Program Parameters for LSTM

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

Algorithm 6 RNN layer: Backward pass

Require: BT, T, P, Q, S
Require: U[S][P], W[S][S], V[Q][S]: Weight matrices
Require: err_{out}(t): Vector of size Q, state(t): Vector of size S
Require: input(t): Vector of size P,
for each $t \in [T-1, 1]$ **do**
  $err_V += err_{out}(t) * state(t)$
  $err_S^A[1:r] = \text{V} * err_{out}(t)$
  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

In Algo. 6, describing the backward pass, the error derivatives are summed up for each time step t to get the error derivative with respect to a given parameter. These gradients are calculated using chain-rule during backpropagation. The backward pass computes the error derivative at each time step which essentially depends on 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.

6. LSTM

LSTMs are a special kind of RNNs, capable of learning long-term dependences over the time dimension. Introduced by Hochreiter et al., LSTMs were designed to combat vanishing gradients [4] through a gating mechanism. A typical LSTM layer is comprised of forget gate, input gate and output gate. Each gate masks some information (from the stream of data flowing through the network) propagating through the same or its previous layers depending on type of gate. The parameters required to describe LSTM are given in Table 4
In Algo. 7 , $input_{gate}, forget_{gate}, output_{gate}$ represent the input, forget and output gates respectively, and work like masks. The input gate $input_{gate}$ decides to what extent the current input contributes to the newly computed state(memory(t)).

The forget gate $forget_{gate}$, defines the factor of the previous state which is retained in the current state. Finally, the output gate $output_{gate}$, 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 7 LSTM Neural Network layer: Forward pass

Require: T, P, Q, S
Require: input(t): Vector of size P
Require: state(t): Vector of size S
Require: W_i[S][S], W_f[S][S], W_o[S][S], W_g[S][S]: Weight matrices for hidden state.Suffix represent type of gate among input, forget, output, hidden candidate state.
Require: U_i[S][P], U_f[S][P], U_o[S][P], U_g[S][P]: Weight matrices for input state.Suffix follows same interpretation.
1: **for each** $t \in [1, T)$ **do**
2: $input_{gate}$[1:S]←input(t)*U_i+state(t-1)*W_i
3: $forget_{gate}$[1:S]←input(t)*U_f+state(t-1)*W_f
4: $output_{gate}$[1:S]←input(t)*U_o+state(t-1)*W_o
5: $cand_{state}$[1:S]←input(t)*U_g+state(t-1)*W_g
6: memory(t)←memory(t-1)*$forget_{gate}$+ $cand_{state}$*input(t)
7: state(t)←memory(t)*$output_{gate}$
8: **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 backpropogation phase for LSTM(Algorithm. 8) 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.

Algorithm 8 LSTM Neural Network layer: Backward pass

Require: T, P, Q, S
Require: 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] : Same as seen for LSTM forward pass.
Require: memory(t): Vector of size S, input(t): Vector of size P
Require: $output_{gate}$[1:S], $input_{gate}$[1:S], $forget_{gate}$[1:S], $cand_{state}$[1:S]
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

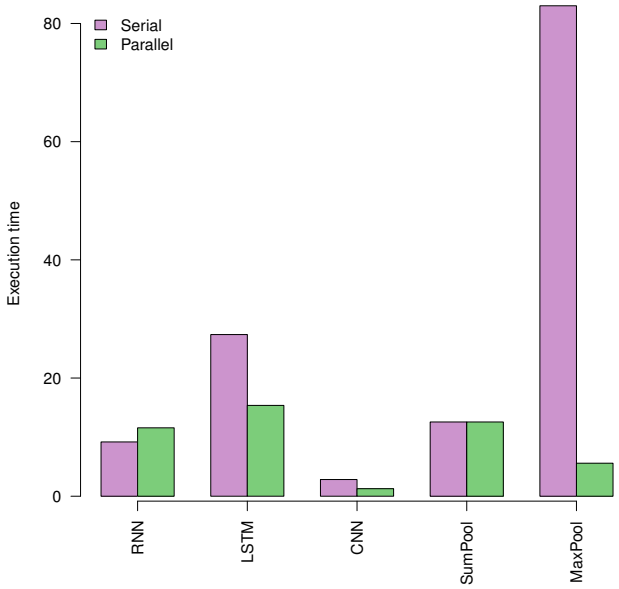

Figure 1: Execution times for forward pass

7. POLYBENCH/NN IN JULIA

Julia [1] is a high-level language suited for scientific applications. Julia code gets translated into LLVM-IR through its JIT compiler so as to facilitate varieties of compiler optimizations implemented in LLVM. Recently during Google Summer of Code-2016, polyhedral transformations were enabled into Julia via LLVM-Polly [3]. Hence, we also ported our PolyBench/NN programs into PolyBench.jl framework [7].

8. PERFORMANCE ANALYSIS

To study speedups obtained by applying polyhedral transformations, we compiled our set of programs through PLuTo [2], a widely used source-to-source polyhedral optimizer. The version of PLuTo used was 0.11.4. While using PLuTo, we provided the `- tile` and `- parallel` flags. All experiments were performed with data set sizes set to the PolyBench variable `EXTRA-LARGE`. We compiled our programs using GCC-7.0.0. We used OpenMP-4.5 to execute parallel codes provided by PLuTo. 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 these three trials as the ex-

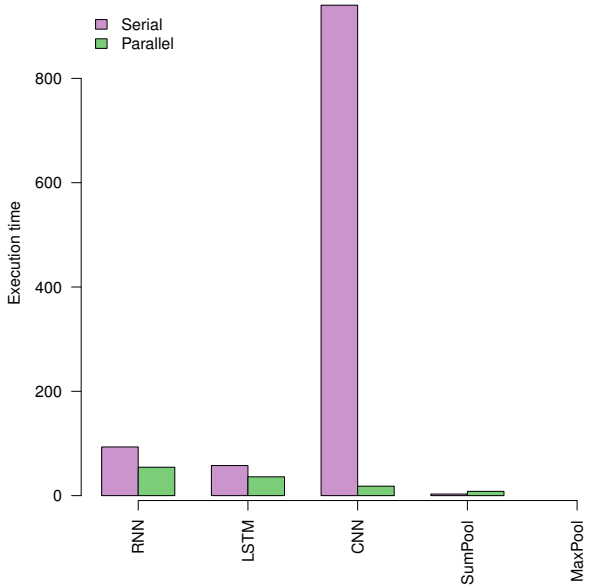

Figure 2: Execution times for backward pass

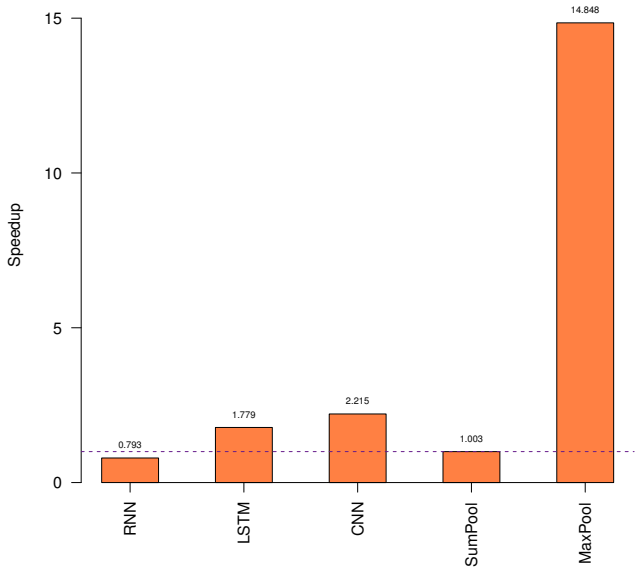

Figure 3: Speedups for forward pass

ecution time. We recorded the execution times for serial and parallel versions separately for both forward and backward phases. The plots showing execution times for forward and backward phases are given in 1, 2. Also, observed speedups can be found in 3, 4.
Here are some observations:

- Backward phase is more compute intensive than forward phase for all layers except SumPooling.
- RNN, LSTM consist of affine control loops and hence

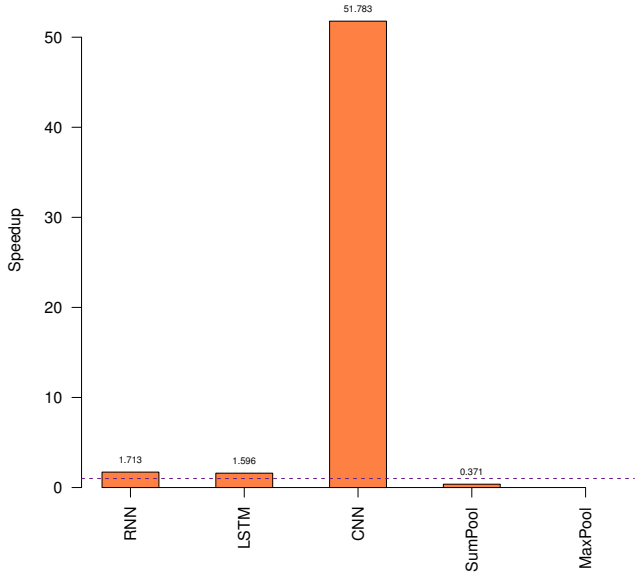

Figure 4: Speedups for backward pass

were successfully parallelized by PLuTo.

- For CNN and Maxpool, we were forced to replace the stride parameters with integer constants as defined in the header file of our implementation. This made CNN and Maxpool(forward pass) analyzable for PLuTo.
- The backward phase of MaxPooling kernel consists of a data dependent condition which PLuTo's dependence analysis was unable to analyze.
- No speedups were observed for forward phase of RNN and backward phase of SumPooling.
- Average speedups observed for forward and backward were 2.155003635, 2.69322549 respectively.

9. CONCLUSIONS AND FUTURE WORK

We have implemented 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. To handle these cases we had to fix the stride parameters of these codes manually, so that the benchmarks satisfy affinity conditions. The modified programs showed significant speedups after applying polyhedral program transformations. Our PolyBench/NN C implementation is available at: https://github.com/hrishikeshv/polybench/tree/master/polyNN, while the PolyBench/NN Julia version at: https://github.com/hrishikeshv/PolyBench.jl/tree/master/src/polyNN.
We believe that our work could form a basis to apply automatic loop transformations which improve data locality and expose parallelization opportunities for different neural network architectures.
Currently, we are implementing Alphabet/AlphaZ versions of PolyBench/NN programs. We plan on studying the application of well-known stencil optimizations like time-skewing/

diamond tiling for the CNN kernel as well as the SIMD vectorization opportunities for them.

10. ACKNOWLEDGMENTS

The authors would like to thank Prof. Sanjay Rajopadhye, Dr. Louis-Noel Pouchet and Matthias Reisinger for their encouragement and feedback.

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 29th ACM SIGPLAN Conference on PLDI*, pages 101–113, NY, USA, 2008. ACM. http://pluto-compiler.sourceforge.net/.

[3] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly âĂŤ performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[6] Louis-Noël Pouchet et al. Polybench Benchmarks. https://sourceforge.net/projects/polybench/.

[7] Matthias Reisinger. Polybench benchmarks in julia. https://github.com/MatthiasJReisinger/PolyBench.jl.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on FPGAs*, pages 161–170, New York, NY, USA, 2015. ACM.

APPENDIX

PolyBench/NN C kernels

A. RNN

```
#pragma scop

  for (t = 0; t < _PB_NT; t++)
  {
    for(s1 = 0; s1 < _PB_NS; s1++)
      for(p = 0; p < _PB_NP; p++)
        s_F[t][s1] += U[s1][p]*inp_F[t][p];


    if(t > 0)
      for(s2 = 0; s2 < _PB_NS; s2++)
        s_F[t][s1] += W[s1][s2]*s_F[t-1][s2];
    for(q = 0; q < _PB_NQ; q++)
      for(s1 = 0; s1 < _PB_NS; s1++)
        out_F[t][q] += V[q][s1]*s_F[t][s1];
  }

#pragma endscop
```

Listing 1: Forward Pass

```
#pragma scop

  for (t = _PB_NT - 1; t > 0; t--)
  {
    for(q = 0; q < _PB_NQ; q++)
      for(s = 0; s < _PB_NS; s++)
        del_V[q][s]+=err_out[t][q]*s_F[t][s];

    for(s = 0; s < _PB_NS; s++)
    {
      del_TA[s] = (DATA_TYPE) 0;
      for(q = 0; q < _PB_NQ; q++)
        del_TA[s] +=V[q][s]*err_out[t][q];
    }

    for(step=t+1;step> MAX(0,t-bptt_trunc);step--)
    {
      if(step > 0)
        for(r = 0; r < _PB_NS; r++)
          for(s = 0; s < _PB_NS; s++)
            del_W[r][s]+=del_TA[r]*s_F[step-1][s];

      for(s = 0; s < _PB_NS; s++)
        for(p = 0; p < _PB_NP; p++)
          del_U[s][p]+=del_TA[s]*inp_F[step][p];

      for(r = 0; r < _PB_NS; r++)
      {
        del_TB[r] = (DATA_TYPE) 0;
        for(s = 0; s < _PB_NS; s++)
          del_TB[r] +=del_TA[s]*W[s][r];
      }

      for(r = 0; r < _PB_NS; r++)
        del_TA[r] = del_TB[r];
    }
  }
#pragma endscop
```

Listing 2: Backward Pass

B. POOLING

```
#pragma scop

  for(n = 0; n < _PB_NN; n++)
    for(d = 0; d < _PB_ND; d++)
      for(r = 0; r < _PB_NR; r++){
        for(c = 0; c < _PB_NC; c++){
          val = 0;
          for(h = 10*r;h < min(10*r+dh,ih);h++)
            for(w = 10*c;w < min(10*c+dw,iw);w++)
              val +=inp_F[n][d][h][w];
          out_F[n][d][r][c] = val;
        }
      }

#pragma endscop
```

Listing 3: Forward Pass

```
#pragma scop

  for(n = 0; n < _PB_NN; n++)
   for(d = 0; d < _PB_ND; d++)
```

```
  for(r = 0; r < _PB_NR; r++){
   for(c = 0; c < _PB_NC; c++){
    for(h = 10*r; h < min(10*r + dh, ih);h++)
     for(w = 10*c ; w < min(10*c+dw,iw);w++)
      err_in[n][d][h][w]+=err_out[n][d][r][c];

    }
   }

#pragma endscop
```

Listing 4: Backward Pass

C. LSTM

```
#pragma scop

  for (t = 0; t < _PB_T; t++)
  {

    for(s1 = 0; s1 < _PB_S; s1++)
    {
      i[s1] = (DATA_TYPE) SCALAR_VAL(0.0);
      for(p = 0; p < _PB_P; p++)
        i[s1] +=U_i[s1][p]*inp_F[t][p];


      if(t > 0)
        for(s2 = 0; s2 < _PB_S; s2++)
          i[s1] +=W_i[s1][s2]*s_F[t-1][s2];
    }


    for(s1 = 0; s1 < _PB_S; s1++)
    {
      f[s1] = (DATA_TYPE) SCALAR_VAL(0.0);
      for(p = 0; p < _PB_P; p++)
        f[s1] +=U_f[s1][p]*inp_F[t][p];


      if(t > 0)
        for(s2 = 0; s2 < _PB_S; s2++)
          f[s1] +=W_f[s1][s2]*s_F[t-1][s2];


    }
    for(s1 = 0; s1 < _PB_S; s1++)
    {
      o[s1] = (DATA_TYPE) SCALAR_VAL(0.0);
      for(p = 0; p < _PB_P; p++)
        o[s1] +=U_o[s1][p]*inp_F[t][p];


      if(t > 0)
        for(s2 = 0; s2 < _PB_S; s2++)
          o[s1] += W_o[s1][s2]*s_F[t-1][s2];
    }
    for(s1 = 0; s1 < _PB_S; s1++)
    {
      g[s1] = (DATA_TYPE) SCALAR_VAL(0.0);
      for(p = 0; p < _PB_P; p++)
        g[s1] += U_g[s1][p]*inp_F[t][p];



      if(t > 0)
        for(s2 = 0; s2 < _PB_S; s2++)
          g[s1] +=W_g[s1][s2]*s_F[t-1][s2];
    }

    if(t > 0)
      for (b = 0; b < ns; b++)
        c_F[t][b] =c_F[t-1][b]*f[b]+g[b]*i[b];


    for (b = 0; b < ns; b++)
      s_F[t][b] =c_F[t][b]*o[b];

  }
#pragma endscop
```

Listing 5: Forward Pass

```
#pragma scop

  for (t = _PB_T - 1; t > 0; t--)
  {
    for(s = 0; s < _PB_S; s++)
      del_o[s]=c_F[t][s]*del_S[t][s];


    for(s = 0; s < _PB_S; s++)
      del_C[t][s]+= o[s]*del_S[t][s];


    if(t > 0){
      for(s = 0; s < _PB_S; s++)
        del_f[s] =c_F[t-1][s]*del_C[t][s];


      for(s = 0; s < _PB_S; s++)
        del_C[t-1][s] =f[s]*del_C[t][s];
    }


    for(s = 0; s < _PB_S; s++)
      del_i[s] =g[s]*del_C[t][s];


    for(s = 0; s < _PB_S; s++)
      del_g[s] = i[s]*del_C[t][s];


    for(s = 0; s < _PB_S; s++){
      for(p = 0; p < _PB_P; p++){
        del_Ui[s][p]=inp_F[t][p]*del_i[s];
      }
    }
    for(s = 0; s < _PB_S; s++){
      for(p = 0; p < _PB_P; p++){
        del_Uf[s][p]=inp_F[t][p]*del_f[s];
      }
    }
    for(s = 0; s < _PB_S; s++){
      for(p = 0; p < _PB_P; p++){
        del_Uo[s][p]=inp_F[t][p]*del_o[s];
      }
    }

    for(s = 0; s < _PB_S; s++){
      for(p = 0; p < _PB_P; p++){
        del_Ug[s][p]=inp_F[t][p]*del_g[s];
      }
    }

        if(t > 0){
    for(s = 0; s < _PB_S; s++){
      for(r = 0; r < _PB_S; r++){
       del_Wi[s][r]=s_F[t-1][r]*del_i[s];
      }
```

```
        }
        for(s = 0; s < _PB_S; s++){
          for(r = 0; r < _PB_S; r++){
            del_Wf[s][r]=s_F[t-1][r]*del_f[s];
          }
        }
        for(s = 0; s < _PB_S; s++){
          for(r = 0; r < _PB_S; r++){
            del_Wo[s][r]=s_F[t-1][r]*del_o[s];
          }
        }
        for(s = 0; s < _PB_S; s++){
          for(r = 0; r < _PB_S; r++){
            del_Wg[s][r]=s_F[t-1][r]*del_g[s];
          }
        }
        for(s = 0; s < _PB_S; s++){
          for(r = 0; r < _PB_S; r++){
            del_S[t-1][r]=W_i[r][s]*del_i[s]+
                W_f[r][s]*del_f[s]+ W_o[r][s]*
                del_o[s]+W_g[r][s]*del_g[s];
          }
        }
      }
    }
#pragma endscop
```

Listing 6: Backward Pass

D. CNN

```
#pragma scop
  for (n = 0; n < _PB_N; n++)
    for (k = 0; k < _PB_K; k++)
      for (p = 0; p < _PB_P; p++)
        for (q = 0; q < _PB_Q; q++)
          for (c = 0; c < _PB_C; c++)
            for (r = 0; r < _PB_R; r++)
              for (s = 0; s < _PB_S; s++)
                out_F[n][k][p][q]+= W[k][c][r][s]*inp_F[n][c][p*u+NR-r-1][q*v+NS-s-1];

#pragma endscop
```

Listing 7: Forward Pass

```
#pragma scop
  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];

#pragma endscop
```

Listing 8: Backward Pass