Parallel Training of Recurrent Neural Networks

Bojian Han, Esha Uboweja

15-418/618 Final Project Report,
Carnegie Mellon University

1 Summary

We accelerated recurrent neural network (RNN) training on multi-core CPUs using Halide. Our implementation used data parallelism and parallel matrix multiplication operations (Multi-threads + SIMD Vectorization), and achieved approx. 39x speedup against sequential training of the RNN in Halide on a 61 core Xeon Phi CPU. We also compared our optimized Halide implementation with a optimized NumPy implementation of the same algorithm with batching on the Xeon Phi, and report our results.

2 Background

A recurrent neural network trains on input containing sequences of data, as it learns about time dependent relations between different parts of the input. For example, if we send as input a sequence of words, i.e. a sentence, an RNN can learn about the relations between different words, and hence learn rules of English grammar such as relationships between verbs and adverbs, etc.

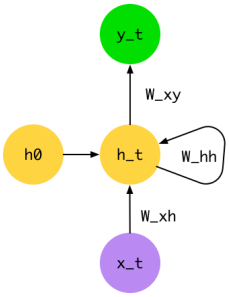

Fig. 1. Recurrent Neural Network Architecture

2.1 RNN Architecture

A basic RNN contains an input layer, hidden layer and output layer, containing n_x, n_h and n_y nodes respectively, as shown in Figure 1. A data point x_i contains n_x features at every time-step t, sent to the RNN. The connections between the input and hidden nodes are parametrized by a weight matrix W_{xh} of size $n_x \times n_h$. The weights in the hidden layer represent recurrent connections, where connections from hidden layer at time-step t to those at time-step $t+1$ (i.e. h_t to h_{t+1}) are parametrized by a weight matrix W_{hh} of size $n_h \times n_h$. Finally, weights from the hidden layer to the output layer at every time-step are parametrized by a weight matrix W_{hy}, of size $n_h \times n_y$. Note. We have decided not to include the bias layers b_h and b_y for the simplicity of this assignment.

There are two steps involved in training an RNN:

1. Forward Propagation:
   In this step, in general, given an input data point $\mathbf{x} = \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_T}$ of T time-steps, the RNN calculates the output $\mathbf{y} = \mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_T}$ of T time-steps (this can vary for different applications). All the updates are propagated from h_1 to h_T, sequentially, as highlighted by black arrows in an unrolled version of the RNN in Figure 2.
2. Backpropagation Through Time:
   In this step, the RNN updates the weight matrices using gradient descent. For this, gradients from time-step T have to be propagated all the way back to time-step 1, as highlighted by the back-ward black arrows in Figure 3.

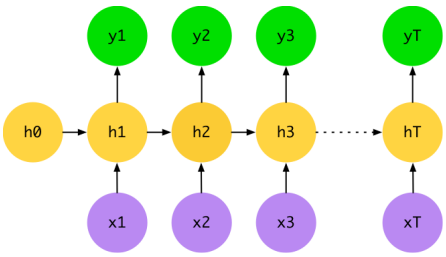

Fig. 2. Forward Propagation in an RNN

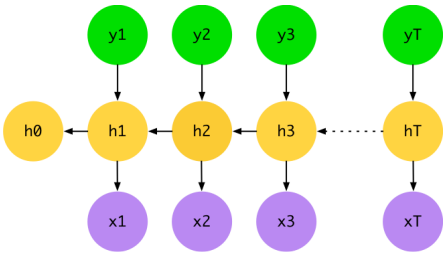

Fig. 3. Backpropagation Through Time in an RNN

The equations for the each time-step t of the RNNs are as follows:

The loss is defined as

$$E = \sum_t (\hat{y}_t - y_t)^2$$

where $\hat{y}_t$ is the output generated by the RNN at time-step t for input x_t, and y_t is the expected ground truth output for x_t.

The forward propagation equations for our RNN implementation are:

$$h_t = \tanh\left(W_{xh}x_t + W_{hh}h_{t-1}\right)$$
$$\hat{y}_t = \tanh\left(W_{hy}h_t\right)$$

The backward propagation through time propagates the gradients of the loss w.r.t. each of the weight matrices learnt by the RNN. These gradient equations are:

Gradient w.r.t. W_{hy}

$$\frac{\partial E}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial (W_{hy}h_t)} \frac{\partial (W_{hy}h_t)}{\partial W_{hy}} = \sum_{t=1}^{T} 2(\hat{y}_t - y_t) \times (1 - \hat{y}_t{}^2) \times h_t$$

Gradient w.r.t. W_{hh}

$$\frac{\partial E}{\partial W_{hh}} = \frac{\partial^+ E_T}{\partial h_T}\frac{\partial h_T}{\partial W_{hh}} + (\frac{\partial^+ E_T}{\partial h_T}\frac{\partial h_T}{\partial W_{hh}} + \frac{\partial^+ E_{T-1}}{\partial h_{T-1}})\frac{\partial h_{T-1}}{\partial W_{hh}} + \ldots + (\frac{\partial^+ E_1}{\partial h_1}\frac{\partial h_1}{\partial W_{hh}} + \frac{\partial^+ E_0}{\partial h_0})\frac{\partial h_1}{\partial W_{hh}}$$
$$\frac{\partial^+ E_t}{\partial h_t} = \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial (W_{hy}h_t)}\frac{\partial (W_{hy}h_t)}{\partial h_t} = 2(\hat{y}_t - y_t) \times (1 - \hat{y}_t{}^2) \times W_{hy}$$
$$\frac{\partial h_t}{\partial W_{hh}} = \frac{\partial \tanh\left(x_t W_{xh} + h_{t-1}W_{hh}\right)}{\partial W_{hh}} = h_{t-1}(1 - h_t^2)$$

where "+" indicates immediate derivative.

The calculation of the gradient w.r.t. W_{xh} is similar to that of gradient w.r.t. W_{hh}, except that we have

$$\frac{\partial h_t}{\partial W_{xh}} = \frac{\partial \tanh\left(x_t W_{xh} + h_{t-1}W_{hh}\right)}{\partial W_{xh}} = x_t(1 - h_t^2)$$

2.2 Challenges

There are 2 important aspects to training an RNN, which are computationally expensive:

1. The forward and backward propagation steps are inherently sequential. In forward propagation, information is propagated from time-step 1 all the way to time-step T. In backward propagation, gradients from time-step T are propagated to all previous time-steps, $T-1, T-2, \ldots, 1$.

2. In back-propagation step, gradients are computed from future time-steps and current time-steps. That is, at time-step t, gradients from time-steps $t+1, t+2, \ldots, T$ are added to the gradient at time-step t. This requires that we store the sequential updates at every time-step from the forward propagation step, and hence has a high memory footprint.

Accelerating each of these steps can decrease the training time for RNNs, which is desirable. Both of these steps can benefit from parallelization.

2.3 Workload

As described above, there are inherent sequential dependencies between consecutive time-steps, and throughout the network. However, we can use data parallelism to speedup the network training (both steps), so that forward propagation produces outputs for multiple inputs at a time, and the backward-propagation uses all these outputs to propagate gradients from time-step T to 1. This is called mini-batch training of neural networks.

Since these steps involve several matrix multiplications, there is spatial locality in the problem. It is better to perform matrix multiplication on chunks of matrices at a time, so that the cache is hot with the concerned set of rows and columns required in the multiplication, and there are fewer cache misses. Further, SIMD can speed up these multiplications so that more elements are multiplied in parallel.

2.4 Problem Setup

For the final results for our project, we use the CIFAR-10 image dataset to train an RNN to rotate an image by 10 degrees anti-clockwise. Figure 4 shows a description of the set of inputs and outputs.

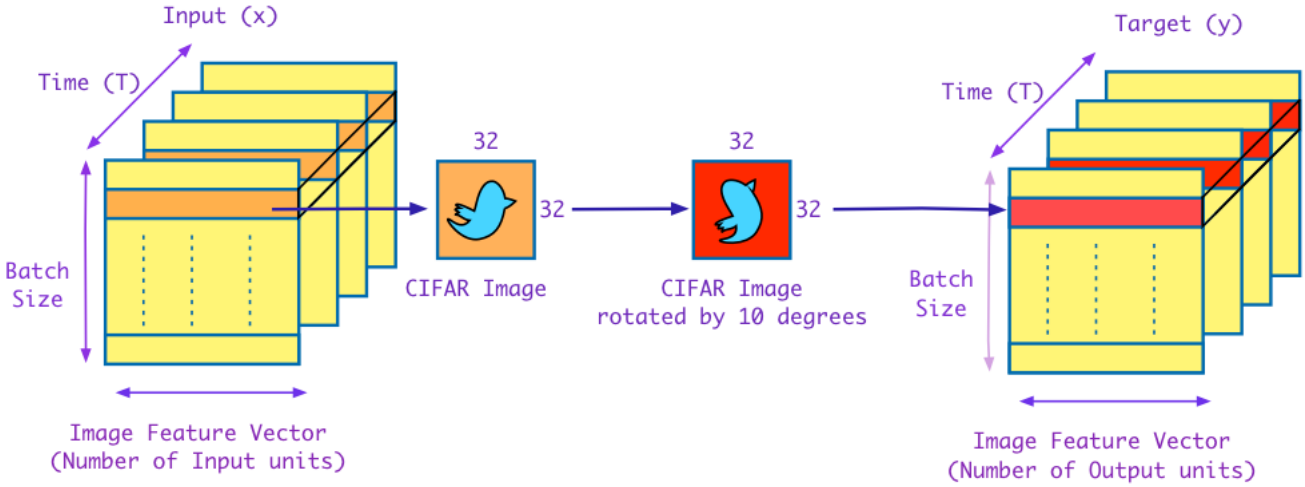

Fig. 4. Training an RNN to learn image rotation by 10 degrees anti-clockwise

As shown in the figure, Figure 4, our data parallel input now has 3 dimensions, time-steps T, batch size (multiple images stacked up together), and n_x, the number of features in 1 input image at time-step t. For our application, we took grayscale images from the dataset, and flattened them into a $1x1024$ vector, so that every row shown in the matrix as 1024 elements and corresponds to one image. If we take a cross-section of the $3D$ matrix along the time axis, at time-step t, the input is 1×1024 image result of rotating the image at time-step $(t-1)$, 10 degrees anti-clockwise. The corresponding outputs are shifted accordingly, so that $\mathbf{y_t} = \mathbf{x_{t+1}}$, i.e. the output at time-step t is the image at time-step t ($\mathbf{x_t}$) rotated 10 degrees anti-clockwise. For actual performance testing, we have decided to truncate the image feature vector to 1000 instead so as to avoid testing matrix multiplications of powers of 2 which leads to cache evictions in diagonal blocks.

2.5 Initial Analysis

Initial analysis of the problem reveals several methods for parallelism. The first method is to perform data parallelism by batching the inputs and targets and utilize the parallelism by parallelizing over matrix multiplication operations as well as element-wise matrix multiplication. The amount of parallelism using this method depends heavily on the size of the input and size of batch.

Another method is to perform parallelization using the diagram. Notice that the arrows into the hidden layers for both forward propagation and backward propagation can be potential target for parallelization since they do not depend of each other. Also notice the arrows out of the hidden layers can also be potential target for parallelization if the hidden layers are computed already for that step. The amount of parallelization in this depends on the time step T.

3 Approach

We implemented the RNN to train quickly on a multi-core CPU. Specifically, we tested our Halide implementation and NumPy implementation on the `latedays` cluster, on the Intel Xeon Phi CPU, which has 61 cores and 512-bit SIMD vector registers.

3.1 Improving the BPTT algorithm

We found a "WildML" blog tutorial[1] that presented an $O(N^2)$ algorithm to implement the back-propagation through time step of training an RNN.

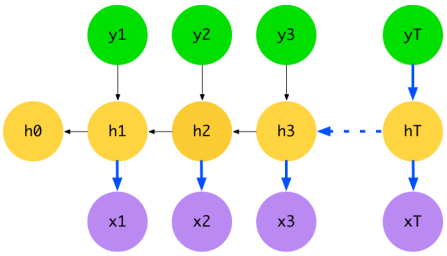

Fig. 5. $O(N^2)$ approach for implementing BPTT

Figure 5 shows representation of 1 iteration of the $O(N^2)$ approach for BPTT, where gradients from time-step T are propagated all the way back up to time-step 1, before calculating the gradients at time-step $(T-1)$. This is quadratic in the number of time-steps and hence slow.

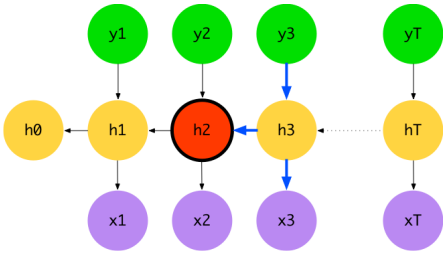

Fig. 6. $O(N)$ approach for implementing BPTT

We optimized BPTT and implemented an $O(N)$ algorithm linear in the number of time-steps. As shown in Figure 6, this implementation calculates the gradients at time-step t, and these are then stored and re-used at time-step $(t-1)$. Therefore, now we don't go all the way back to the first time-step in every iteration. This optimization is useful for both serial and parallel implementations of the RNN.

3.2 Technologies

We developed an RNN implementation in 3 different ways:

1. Theano: This is a neural network library commonly used in Python. Theano is an API that compiles the code setup of an RNN dynamically and generates code for forward and backward propagation steps for training the RNN.
2. NumPy: This is a library that is useful for implementing fast linear algebra operations in Python (using BLAS). We wrote both the forward and backward propagation steps from scratch in NumPy.

3. Halide: This is a domain specific language used with C++ for fast image processing operations. Since images can be represented as matrices, Halide is useful for fast matrix operations. We therefore developed an RNN from scratch in Halide, and optimized our implementation.

We had some issues setting up Theano on `latedays` because the assembler could not recognize some of the optimized assembled instructions (same issue happened when compiling Halide code using -O3 flag.). Since on a 2 core Intel i5 processor, our NumPy implementation was slightly faster than the Theano framework implementation for a fixed set of parameters, we benchmarked our Halide implementation against the NumPy implementation.

Take note that both the NumPy and Theano framework uses the batching technique and same batching parameters in order for it to be a fair comparison against Halide. Additionally, both NumPy and Theano are specifically set to use `float32` instead of the default `float64` so as to match the Halide implementation of using a 4 byte float.

Furthermore, take note that all the implementations uses exactly the same set of training data and initial weights generated as an image in MATLAB. We have verified from the floating numbers that all three implementations obtain the same results except for small floating point differences.

We went through many steps in optimizing our serial Halide implementation. While we discuss these optimizations below, we present results in the next section and describe what worked well there.

1. Changing the batch size, i.e. number of images used in data parallel training per epoch. Having matrix-matrix multiplication is faster than a lot of small matrix-vector multiplication due to spatial locality and temporal locality.
2. Tiling of for loops in matrix multiplication, to improve spatial locality and decrease cache capacity misses.
3. SIMD Vectorization : Since the code for matrix multiplication is the same for all elements of the matrix, using SIMD to process many elements in parallel can improve the training speed of RNN.
4. `parallel()` halide optimization: This optimization uses OpenMP threads to process different rows/columns of the matrix in parallel.
5. AOT (Ahead of Time) compilation: Halide by default uses JIT (Just in Time) which is slow, as it dynamically re-compiles different sections of the code every time it processes that section. AOT avoids this by pre-compiling the code into re-usable binary files, so that we can just run the code with those binary files, and hence decrease program run-time.

4 Results

We present below a set of graphs explaining how these optimizations helped in speeding up our Halide implementation. For testing our RNN, we used a batch size of 1000, $n_x, n_h, n_y = 1000$, and $T = 10$ time-steps, unless mentioned otherwise. The benchmark utilized for testing is average time for a single epoch in seconds relative to NumPy's implementation on latedays. Note that since NumPy uses BLAS for matrix operations, it has access to SIMD vector instructions on Xeno Phi which gives it up to 16x speed up.

4.1 Theano v/s NumPy on Intel Core i5 (2 cores)

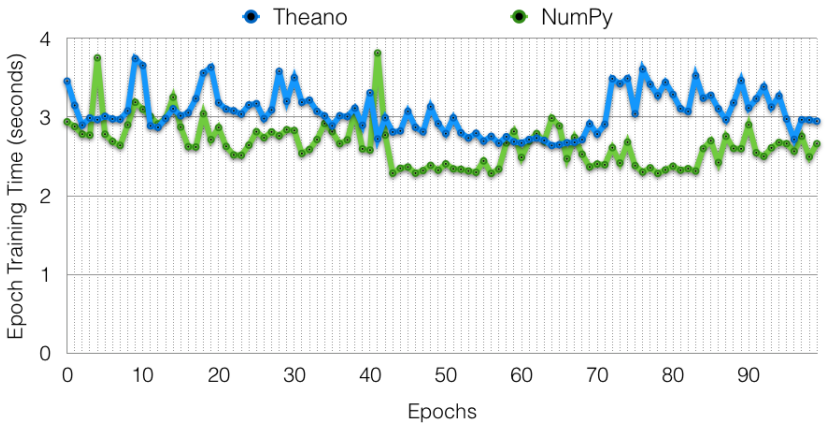

Fig. 7. Theano v/s NumPy on Intel Core i5 (2 cores)

As discussed previously and shown in Figure 7, our NumPy implementation of RNN from scratch is competitive with the RNN implemented using Theano on a 2-core CPU. An interesting note is that for much smaller data, the NumPy approach can reach 2x speedup compared to the Theano approach, primarily because of the overhead of calling a compiled Theano function.

4.2 Best Performance of Halide v/s NumPy on Latedays Xeon Phi

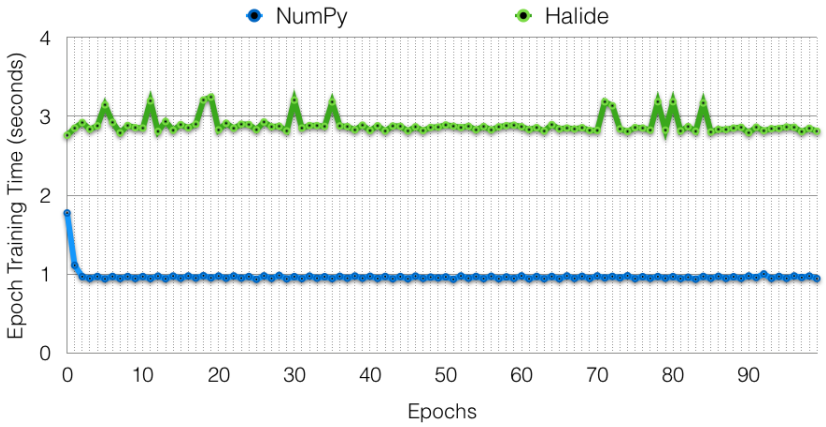

Fig. 8. Best Performance of Halide v/s NumPy on Latedays Xeon Phi

As shown in Figure 8, our best optimized version of Halide RNN is about 3 times slower than our NumPy implementation on a 61 core Xeon CPU, when we train the RNN for 100 epochs. Note that NumPy utilizes BLAS which contributes to highly optimized matrix multiplication and parallalism via SIMD optimizations.

4.3 Training time per epoch decreases with increasing batch size

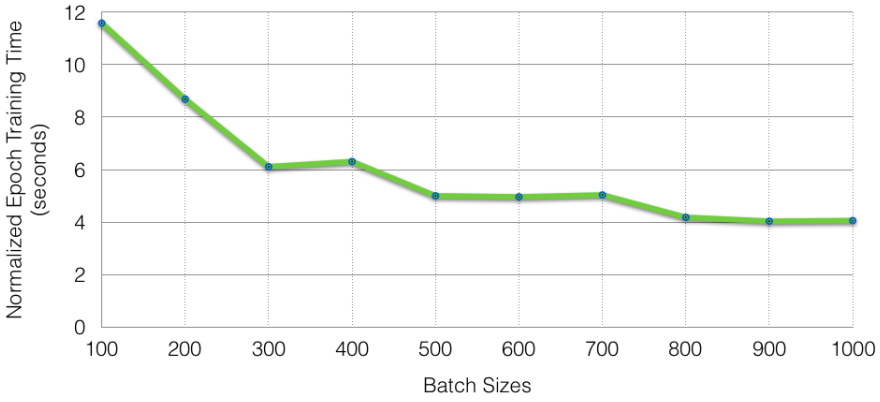

Fig. 9. Training time per epoch decreases with increasing batch size

We experimented with the batch size parameter for training the RNN, using batch sizes of 100 up to 1000, incrementally increasing by 100. The graph presented in Figure 9 shows the normalized time for training. For a particular batch size b, we trained the RNN with $b \times T$ dataset size, as we just sliced the original dataset along the batch size dimension. Each point in the graph is obtained by averaging the training time for 10 epochs for that batch size. To account for the fact that we train the network with different dataset sizes, we normalize the training times for each batch size by the dataset size.

As shown in the graph, increasing the batch size improves the training performance, and we believe that this is because larger matrices provide more data for the algorithm to parallelize and also increase spatial and temporal locality, which in turn can contribute to an increase in math operations, hence greater arithmetic intensity. Since the graph times are normalized, this graph also correctly shows that for a fixed dataset size,

using a smaller batch size for training increases the average training time per epoch, as now there are many small matrix multiplications. For a larger batch size, more images are stacked up together in a larger input matrix, which reduces the number of cache misses when we read many different smaller matrices.

We have also experimented with a variety of workloads such as dimensions of 32 for small workloads and 4096 for large workloads. For smaller workloads, we believe the effects of JIT compilation dominate the training time in Halide execution, preventing the application from gaining much speedup. For larger workloads, we only found some improvements in the speedup ratio between Numpy and our Halide implementation. However, we were limited by the amount of time one epoch will take and also limited by the size of generated training data in terms of feeding the training data to both NumPy and Halide. (Our current pipeline for NumPy does not support reading of images of large sizes, such as 16384×16384.)

4.4 Tiling works well with Vectorization

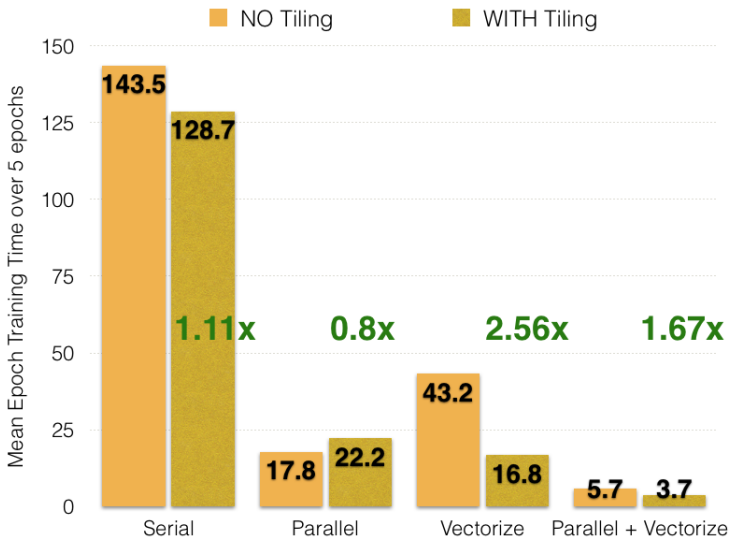

Fig. 10. Tiling works well with Vectorization

We ran 8 tests for testing all the different combinations of 3 Halide optimization techniques, namely, tiling, `parallel` using multiple OpenMP threads and SIMD vectorization for 5 epochs each. All these tests were performed with Ahead-of-time (AOT) compilation instead of Just-in-time compilation in Halide. The code with all these 3 techniques combined is also the one that generated the best performance against NumPy. Please note that there may be some variation in timing due to a small number of epochs in testing. Hence, the 3.7 seconds in this diagram for the best performing code does not contradict the 2.9 seconds performance of the same code in the previous diagram.

For matrix multiplication, it is useful to optimize for spatial locality by tiling, so that when we multiply matrices A and B, instead of reading 1 row for A and 1 column (many rows, hence many cache evictions/misses) for B, we divide the matrix into smaller chunks/tiles. Multiplying tiles at a time improves the spatial locality for matrix B. When we used tiling of 64 rows and 16 columns, we observed a significant speed up compared to the serial Halide implementation, and a positive speed up when we used tiling with SIMD vectorization. SIMD vectorization works well with tiling as we optimize for spatial locality in cache, and process 16 elements in parallel, reducing training time. These results are presented in the graph in Figure 10. Note that we utilized 16 for columns because this matched with the 512-bit SIMD of Xeon Phi and utilized 64 rows because this value generated the best performance.

One interesting observation is the behavior of tiling with Halide's `parallel()` optimization. As shown in the graph, tiling with `parallel()` performs worse than `parallel()` optimization (alone) of input rows compared to the serial Halide implementation. One hypothesis to explain this ambiguity is that there is both low spatial and low temporal locality when tiling is combined with `parallel()`. Halide implements `parallel()` using OpenMP threads, parallelizing over `for` loops. However, parallelizing across tiles implies that there can be scenario where thread th_t processes a tile indexed by rows $0:64$, columns $0:16$ and after this th_t processes the tile of rows $640:724$, columns $0:16$, which is very far away from the previous tile. This can cause issues of spatial locality as the same thread can swap in and out different rows of a matrix, and temporal locality as the same row will be read in and processed different at different times by different threads. This can therefore increase the program run time of training the RNN.

4.5 Vectorization gives large speedup on Xeon Phi

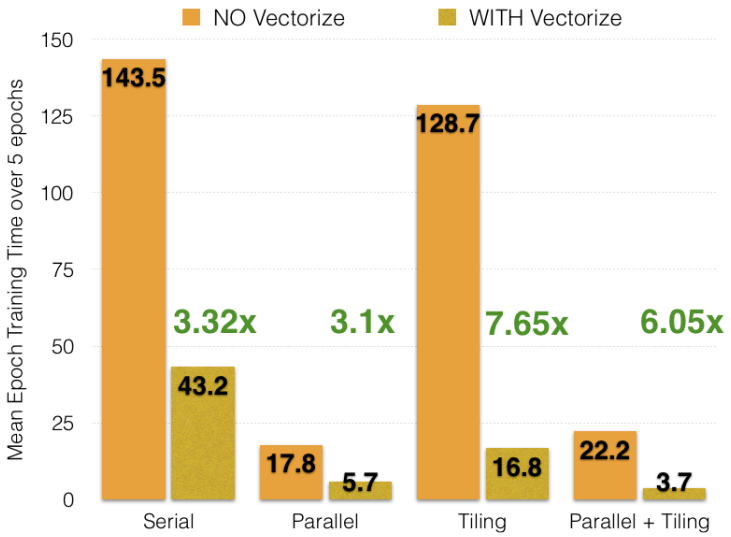

Fig. 11. Vectorization gives large speedup on Xeon Phi

SIMD vectorization of 16 elements for matrix multiplication is beneficial with all optimizations possible for this code, since the code for multiplication is same for all elements. We observe this effect in the graph shown in Figure 11.

Notice that our best speed up is only about 8x times which half of theoretical maximum which is 16x. However this might be because the whole RNN training involved many different steps of matrix operations as well as elementwise matrix operations. Hence, there is some latency in switching from one operation to the next operation. Also, we use a non-linear function, `tanh`, which does not vectorize well as stated on the Halide docs.

4.6 Halide `parallel()` using OpenMP threads works better with vectorization than with tiling

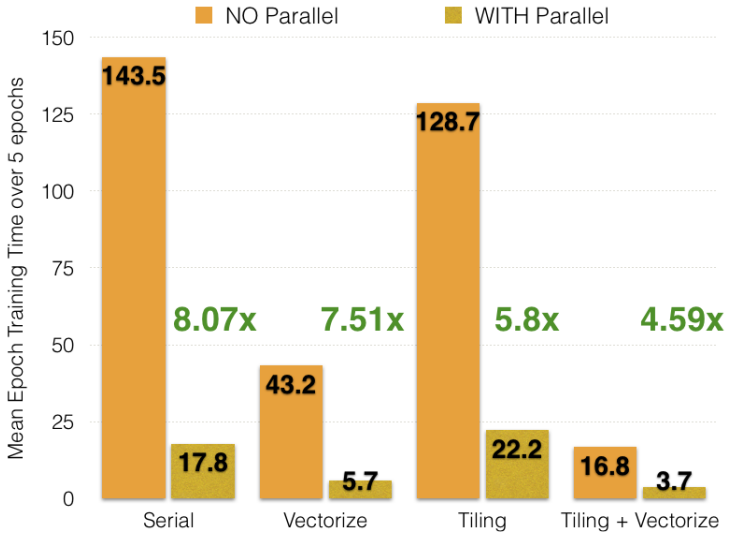

Fig. 12. Halide `parallel()` using OpenMP threads works better with vectorization than with tiling

As discussed previously, the Halide `parallel()` optimization doesn't work well with tiling. However, this gives a good speedup against a serial implementation, as it parallelizes the multiplication across different rows. Further, it works well with SIMD vectorization because now each thread processes 16 elements in parallel for a given row, decreasing the running time further.

However, note that this is still nowhere near the optimum of 61x speed up. There are 2 possible reasons for this. One reason is that with vectorization and tiling, there isn't much computation for each thread to perform, compared to the amount of overhead for the thread to execute that work, which essentially implies low arithmetic intensity for each thread. Another reason is that Halide might have just put a simple `#pragma openmp parallel for` without adding specific scheduling parameters in to the compiled code. The evidence for this is that we have performed a `compile to c` on the Halide `Func` and examined the resulting code and observed this issue.

4.7 Specialization of non-matrix multiplication operations improves speedup slightly

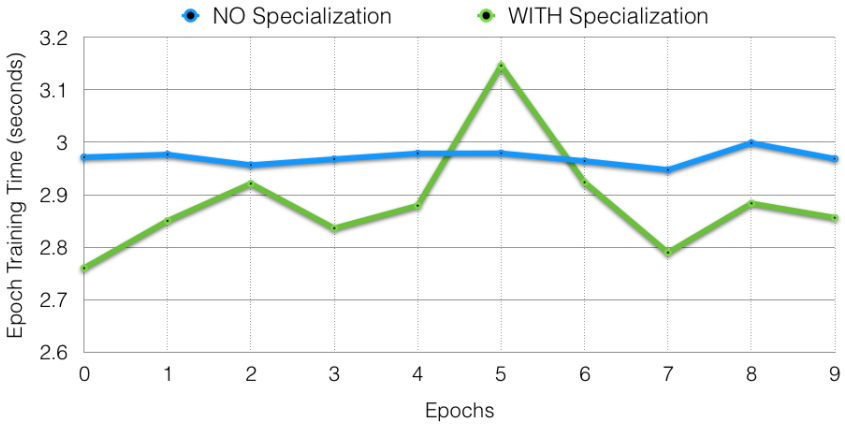

Fig. 13. Specialization of non-matrix multiplication operations improves speedup slightly

We have also performed other additional optimizations on top of the 3 main optimizations mentioned earlier on. We have attempted to specialize the non-matrix multiplication operations by using a different set of scheduling parameters by parallelizing the rows instead in order to exploit the speed-up seen in Halide `parallel()` without tiling for only basic element-wise arithmetic operations. However, since most of the execution time depends on the matrix multiplication operations, this only provided marginal speedup in training time per epoch as shown in Figure 13.

4.8 AOT Halide v/s JIT Halide on Latedays Xeon Phi, AOT implementation is faster

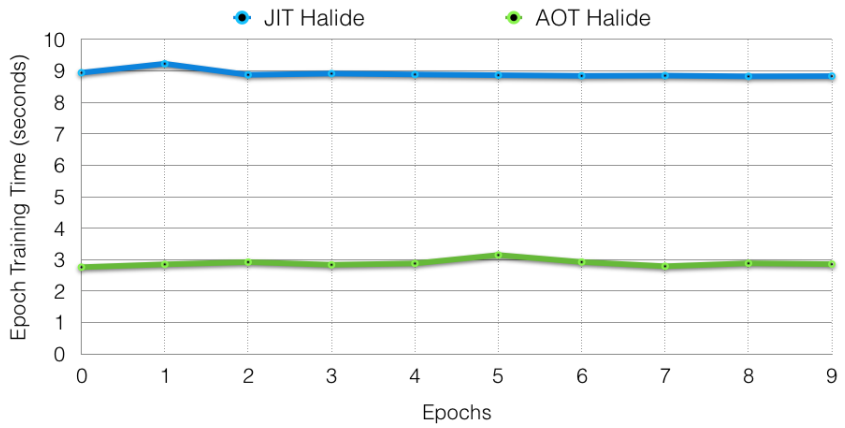

Fig. 14. AOT Halide v/s JIT Halide on Latedays Xeon Phi, AOT implementation is faster

We also compared the performance of Ahead-of-time(AOT) compilation v/s Just-in-time(JIT) compilation because we developed the RNN code using JIT before moving on to AOT to speed up the performance. The reason we switched to AOT is because we need to re-compile intermediate functions and run the code at every single step throughout both forward and backward propagation when the code could just be compiled once

before the file runs. This essentially broke the Halide interface because we need to obtain the `raw_buffer` pointer from the image, but it allowed us to achieve a 3x speed up as shown in Figure 14. Note that we still use JIT in some part of the AOT code, mainly to save the intermediate variables.

4.9 Other optimizations

We have also explored the option of loop fusion either within the Halide interface or outside the Halide interface. We can identify parallelism in the partial calculations of the hidden layers using just the inputs, calculations of the output layers using the hidden layer in parallel and calculations of error with respect to the output layers in parallel from our DAG diagram of the neural network. However, we realize performing parallelism over these domains is less ideal because this gives up the cache locality effect from being able to immediately compute the necessary output at a certain time-step. We have observed an almost 2x speed up when we performed loop fusion of all these calculations instead of separating them out into various steps and parallelizing them across the steps.

The loop fusion within the Halide interface is shown by creating a pure definition using a single long expression instead of creating a two stage pure definition and update definition in Halide. This also provided minor speedup in our Halide implementation, primarily because these expressions are inlined into the compiled code and it manage to exploit cache locality.

Additionally, we also initially focused on image feature size of 1024 and batch size of 1024 instead of 1000. However, we discovered that 1024 is a power of 2 and hence matrix multiplication using this as a dimension results in cache conflict misses on diagonal blocks as evident from 15-213 cache lab. As a result, there is a significant increase in per epoch speed as the dimensions increase from 1000 to 1024 (0.5 seconds increase per epoch).

There are also a number of small hacks utilized by using pointers in Halide to point to images in an attempt to avoid recopying them for multiple usage through different time step.

4.10 Deeper analysis by Workload breakdown

1. [Initialize Forward Propagation] 0.05%
2. [Calculate h_t] 22.16%
3. [Calculate $\hat{y}_t$] 5.78%
4. [Saving all intermediate variables] 22.55%
5. [Calculating loss] 0.79%
6. [Initialize Back Propagation] 0.16%
7. [Calculating $\frac{\partial E}{\partial (W_{hy} h_t)}$]0.54%
8. [Calculating $\frac{\partial E}{\partial (W_{xh} x_t + W_{hh} h_{t-1})}$] 23.33%
9. [Gradient Updates] 24.60%
10. [Gradient Descent] 0.06%

We have done a breakdown of the training time per epoch in each step in training the RNN. We observe that most of the execution time in training is primarily at the matrix multiplication steps (Step 2,8,9) instead of element-wise matrix multiplication. We have also conducted a test using an array of Image pointers to decrease the time for saving all the intermediate variables (Step 4) but that resulted in poorer cache locality for the Gradient Update steps and hence resulted in similar timing for epoch training time (using an array of 2D Image pointer instead a single 3D Image).

4.11 Choice of Machine

For our work, we believe that CPU is still a good choice for speedup because we do not need to be bandwidth-bound by the PCI interconnection and we are still able to utilize multi-threading and vectorization to increase parallelism for matrix multiplication. However, it would be interesting to observe the GPU performance on matrix multiplication since GPU supports more SIMD units and have cudaThreads which has lower overhead of setting up. However, there might not be sufficient work to distribute among the cudaThreads to effectively use the GPU resource for this task.

Division of Work

Sent via email to course staff.

References

1. Wild ML tutorials: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns
2. Halide Tutorials, http://halide-lang.org/tutorials/tutorial_introduction.html
3. On the difficulty of training recurrent neural networks, Pascanu, Mikolov, Bengio
4. Lecture on "Intro to parallel machines and models" by David Bindel http://www.cs.cornell.edu/ bindel/class/cs5220-f11/slides/lec03.pdf