PCSE Final Project

Taka Koutsomitopoulos

January 11, 2023

Code can be found at https://github.com/takakoutso/PCSE-Final-Project/

1 Abstract

Machine Learning(ML) is a vast field with many potential interdisciplinary applications, like for self-driving cars, cancer detection, and infinitely more. This is due to the core definition of ML and NNs, which is in using data to converge and approximate a function to solve any problem. In this project, I sought to apply the concepts of HPC and parallel computation to the exhaustive training process common to all ML problems. With one method, parallel computation applied itself easily and lended expected positive results, with another, more complicated method, I got some positive, but mostly negative, results

2 Introduction & Objective

Producing a usable NN that performs beyond human expectation is expensive - models require immense stores of data to learn from and easily employ NNs on the scale of 20 million neurons to train. HPC and Parallel Programming seek to distribute the load to multiple cores, so that training times can be lowered, models can be iterated through faster, and research and convergence can be performed easier. As NN architectures and their datasets continue to grow in size and complexity, Parallel methods will continue to scale well alongside them. In my own research, I deal with ML models that have potentially hundreds of layers - I sought to explore a parallelized technique to specifically optimize for these types of situations.

This project seeks to parallelize an existing serial ML library(EduDL-public). There are two methods that will be tried, to see the effectiveness and applicability of each technique.

1. The first, and more straightforward approach, will simply be to parallelize individual operations and time-wasting loops. As the majority of execution speed is spent in matrix-vector products, matrix-matrix products, and vector iterations, this will certainly cut down on execution time

with little difficulty. The ability of OpenMP to perform domain decomposition amongst threads will be leveraged in these expensive, repetitive operations.

2. The second approach will seek to parallelize the stages of the model by layer. This will in effect create a pipeline that will allow the model to feed-forward values at a faster throughput, although actual computation speed for an individual layer may remain the same. This will ideally scale well towards models with large amounts of consecutive layers, such as ResNet models. Since the other DL group is working on the backpropagate, I tried to focus my efforts on the feedforward.

3 Implementation

3.1 Approach 1

3.1.1 Relevant Code

```
void VectorBatch::v2mp(const Matrix &m, VectorBatch &y) const {
  const int
    xr = item_size(),   xc = batch_size(),   // column storage
    mr = m.rowsize(),   mc = m.colsize(),    // row storage
    yr = y.item_size(), yc = y.batch_size(); // column
  const auto& mmat = m.values();
  const auto& xvals = vals_vector();
  auto& yvals = y.vals_vector();
  int xi{0},yi{0},mi{0};
  #pragma omp parallel for lastprivate(xi, yi, mi) schedule(static)
  for (int i = 0; i < yr; i++) { // row index
    for (int j = 0; j < yc; j++) { // batch index
      float sum = 0.0;
      for (int k = 0; k < mc; k++) { // column index
            sum += ELEMENTr( mmat,i,k,mr,mc ) * ELEMENTc( xvals,k,j,xr,xc ) ;
     //both are in row-major order
            mi = INDEXr( i,k,mr,mc ); xi = INDEXc( k,j,xr,xc ) ;
      }
      ELEMENTc( yvals,i,j,yr,yc ) = sum;
      yi = INDEXc( i,j,yr,yc );
    }
  }
  assert( xi==xvals.size()-1 );
  assert( yi==yvals.size()-1 );
  assert( mi==mmat.size()-1 );
}
```

3.1.2 Discussion

As mentioned in Section 12.3 in the HPC textbook, the partioning of the matrix is important - splitting the layers' weights by column has worse performance than splitting by row, specifically for C++. This is due to consecutive elements in arrays being stored in row-major order, allowing us to use the #pragma omp parallel command on the outermost loop. Thus, fewer time-consuming memory accesses and more cache accesses are likely to be occur. Static scheduling was used, as each unit of work for given index i is roughly equivalent.

The same techniques used the code above were applied to various operations across vector2.cpp and vectorbatch_impl_reference.cpp. For timing data and analysis of the success of this approach, go to the results section.

3.2 Approach 2

3.2.1 Relevant Code

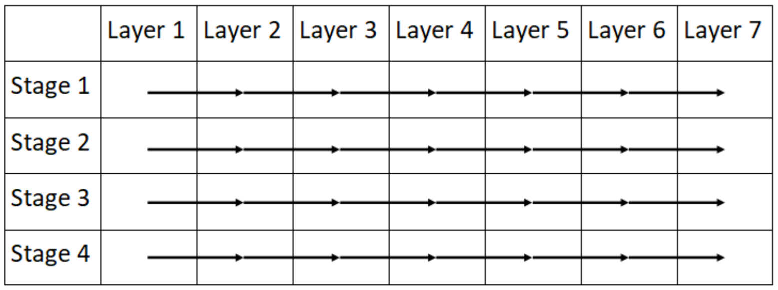
```
void Net::ffhelper(int stage, int layer, const VectorBatch &input, std::vector
     <bool> &completions) {

  assert(completions[(layer-1)*numStages+stage]); // make sure our input data
     is valid
  this->layers.at(layer).forward(this->layers.at(layer-1).activated_batch,
     stage, numStages);
  completions[layer*numStages+stage] = true;
  if (layer+1 < layers.size()) {
    #pragma omp task shared(completions, input)
    {
      ffhelper(stage, layer+1, input, completions); // schedule next task
    }
  }
}

//codesnippet netforward
void Net::feedForward(const VectorBatch &input) {
  if (trace_progress())
    cout << "Feed forward batch of size " << input.batch_size() << endl;
  allocate_batch_specific_temporaries(input.batch_size());
  std::vector<bool> completions(numStages*layers.size(), false);
  for (int stage = 0; stage < numStages; stage++) {
    #pragma omp task priority(2) shared(completions)
      {
        this->layers.front().forward(input, stage, numStages); // Forwarding
     the input
        completions[stage] = true;
        #pragma omp task shared(completions, input)
        {
          ffhelper(stage, 1, input, completions);
        }
      }
  }
}
void Layer::forward(const VectorBatch &prevVals, int stage, int numStages, std
     ::vector<int> &mvproducts, int level) {
    unsigned batch_idx_start = (prevVals.batch_size() * (stage)) / numStages;
    unsigned batch_idx_end = (prevVals.batch_size() * (stage+1)) / numStages;
    prevVals.v2mp( weights, activated_batch, batch_idx_start, batch_idx_end );
    int b = 0;
    #pragma omp single
    {
      b = ++mvproducts[level]; // last stage will perform addition and
    activation
    }
    if (b == numStages) {
      activated_batch.addh(biases); // Add the bias to entire batch
      apply_activation_batch(activated_batch, activated_batch); // Activate
    entire batch
    }
}
```

3.2.2 Discussion

For this section of code, I attempted to use OpenMP tasks to separate and distribute the computation across different cores. The first thing that was done was to separate each layer's computation of a batch into p different ”stages”. Each ”stage” performs the layer's matrix multiplication, bias addition, and activation on a ”minibatch”, which contains $1/p$ of the vectors in the batch. This allows each layer's work to be distributed across up to p processors, but requires the previous layer's equivalent stage to be completed by this point.



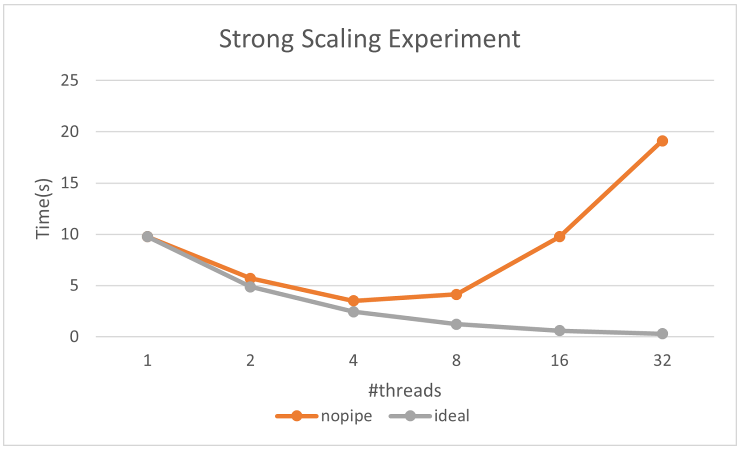
Depiction of a 4 stage, 7 layer ”pipeline”. Arrows depict both preconditions for each layer's computational effort, as well as the next task that a given task will create

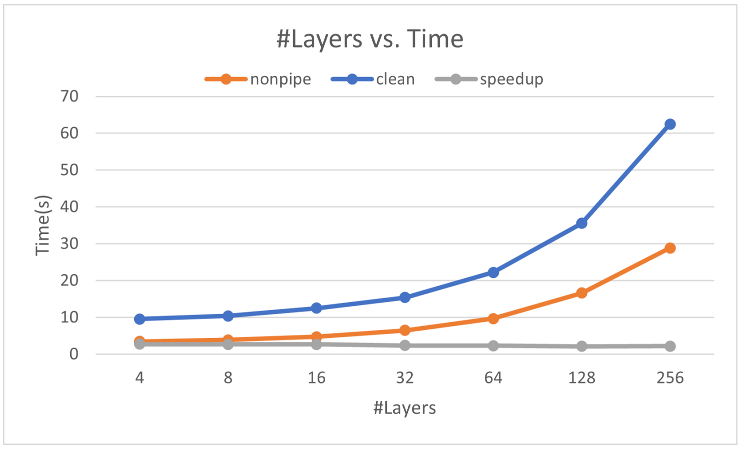
Feedforward tasks the OMP kernel with distributing the stages for the input layer, then calls a recursive helper function that schedules the remaining stages for the remaining layers. I use vectors mvproducts and completitions to prevent race conditions and prevent stages from making incorrect assumptions about initialized values. Layer::forward makes use of a SINGLE clause to ensure that only the last thread to finish will activate the output. All this combined in effect creates a sort of pipeline, though it differs from a pipeline in the idea that a pipeline will not decrease latency, while this paradigm can, for a given layer. This technique seeks to allow concurrent execution of the code, as well as flexibility for the user to allow the feedforward to prioritize for faster minibatch latency and throughput, as opposed to simple concurrent feedforward of the entire batch. Doing so would aid in real-time data analysis, where there are time constraints on computing entire batches of data points at the same time.

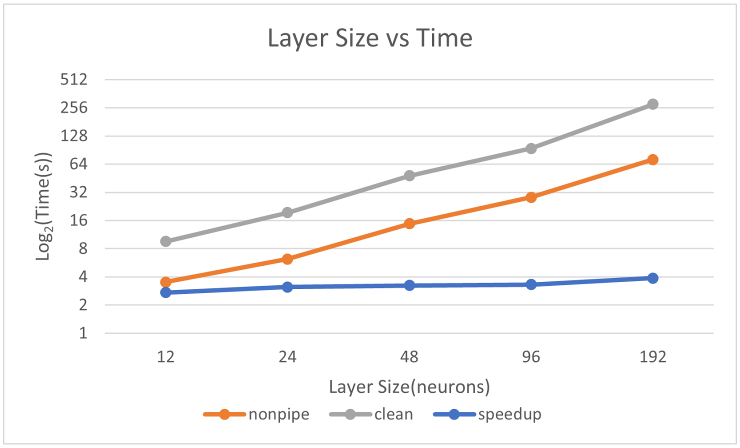
4 Results

Unless specified otherwise, code timings are generated with 4 epochs, 256 samples per batch, 4 levels of default sizes(12 as according to test_mnist.cpp), and 2 chunks per layer in the pipeline(for "pipe" data) on 4 threads. All timings were generated on the UT CS lab machines, as Frontera does not have a clang compiler with a recent enough version for me to get it to work there.

4.1 Approach 1



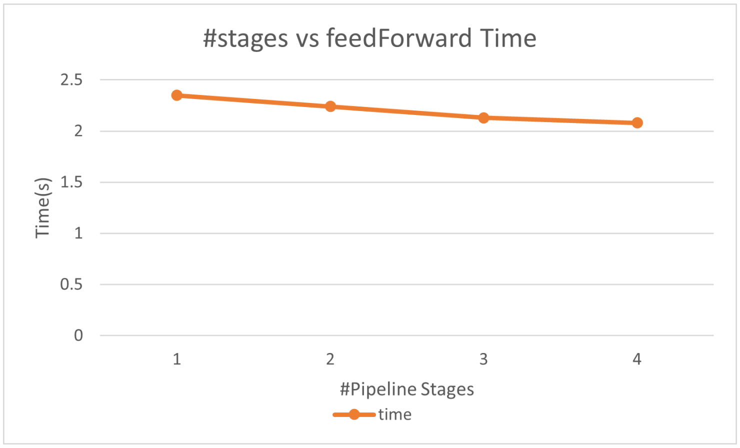


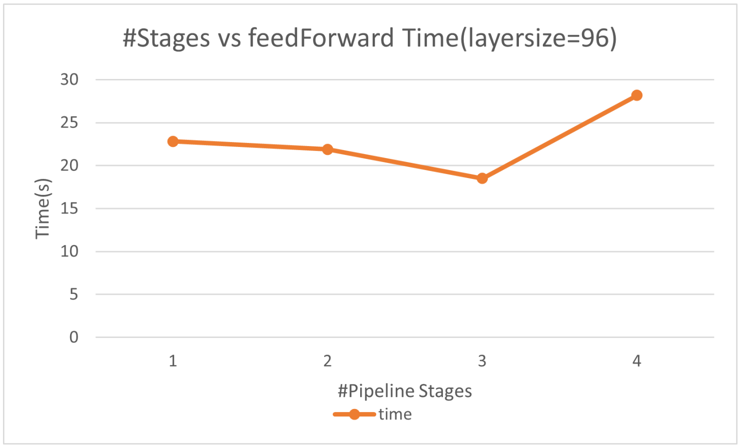


As the number of threads grows, the timing of the parallelized non-pipelined code(shown as nopipe in the graph on the left), decreases. However, after 4 threads, we hit diminishing returns and overheard on synchronizing and dividing work starts to become too high, and the timings start to diverge from the ideal timings(shown as ideal).

As the number of layers grows, the timing of the parallelized non-pipelined code also increases(shown as nonpipe in the graph on the right. It scales nearly linearly, alongside the base code(shown as clean). The speedup decreases as more layers are added, starting from 2.75 for 4 layers and ending at 2.16 for 256, as opposed to the ideal speedup of 4 for 4 threads. This is likely due to the small sizes of the layers, which introduce more overhead as compared to larger parallelizable operations. This is corroborated by the 3rd figure on the bottom, which shows the increase in speedup alongside the increase in layer size, from 2.73 with size 12 to 3.89 with size 192, which is almost equivalent to the ideal speedup of 4.

4.2 Approach 2





The second approach was not as successful, largely due to overhead concerns from the large amounts of tasks that are generated. From the chart on the left, with a small layer size, there is slight improvement as a function of the number of pipeline stages, but it is almost negligible. In addition, as tasks are used, there is a large possibility(that grows with the number of stages/number of layers) that different cores will be assigned to different stages. This means that cache utilization can be too low to have good performance. This explains why the figure on the right has diminished performance as a 4th stage is added, despite noticeable improvement in the first 3 stages. This could be avoided by not using the OpenMP tasks, but this would cut out the flexibility the user could have in prioritizing the task scheduling that I was hoping would occur.

5 Summary

Overall, while there are overhead concerns, individually parallelizing the various matrix multiplication subroutines and various expensive for loops was successful. With the small, toy 4 layer model, a speedup of 2.7 was possible, and this speedup carried into larger models with many more layers

Regaring the second approach, in future attempts to parallelize and pipeline the feedforward, more attention to memory locality and reusability would help. In addition, working on a machine that supports **numactl** to guarantee different scheduling options would also help, so as to distribute the memory for the models' weights, the dataset, and computation tasks more effectively.