CS492 Project 3 Report

20170847 Mirali Ahmadli
20160857 Duman Kuandyk

Introduction

For project 3, we had to rewrite the code of the previous project by offloading most of the computation work to C using BLAS libraries. While the main code was still implemented in Python, it was connected to C by using `ctypes` — a library with C compatible data types. In general, `ctypes` is a very convenient library to use for integrating with Python code, but there were some issues, but it may cause crashes to the Python compiler, and also there exist alternative libraries like `swig`, which is more robust.

We used `OpenBLAS and cuBLAS` to offload the main heavy calculation in the YoloV2 model which is the convolution layer. There exist other alternatives such as `ATLAS` and `MKL`, but comparing their performance on our project specifically is out of the scope of this work. To use parallelism, we used `AVX`, `pthread`, and `CUDA` libraries. `AVX` was used to optimize the heaviest parts of the code in CPU, especially for dealing with intensive floating-point calculations. The first two were used for parallel programming in CPU, and the latter was targeted for parallelizing GPU.

The main focus of this homework was to optimize the code and when we looked at our previous implementation, we saw that there are a lot of redundant calculations in every layer and decided to optimize it first. Main optimizations have been done to `Conv2d` and `Maxpool` layers We used the `im2col` method [1]. It reduced our work to just matrix multiplication in the `Conv2D` layer and choosing maximum element from every column in the `Maxpool` layer.

Key functions:

OpenBLAS:

For OpenBLAS, we just do matrix multiplication for the Conv2d layer which is the major computation in our model.

cblas_dgemm: Computes a matrix-matrix product with general matrices.

Dgemm routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation is defined as C = alpha*A*B + beta*C, which in our case alpha is 1, and beta is 0.

cuBLAS:

For cuBLAS, we just do matrix multiplication for the Conv2d layer which is the major computation in our model.

cublasDgemm: Computes a matrix-matrix product with general matrices.

Dgemm routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation is defined as C = alpha*A*B +

beta*C, which in our case alpha is 1, and beta is 0. It is the same as the previous one but uses GPU, therefore it is faster. (GPU is good with large amount of data (high bandwidth))

The number of Memory Fetches and Floating-Point Operations are 2 * m * n * k. For every row in A and column in B, it caches Ai and Bj to reduces memory fetches. C matrices usually do not fit into GPU memory, therefore it divides matrix into blocks. It uses one block per thread and therefore, more registers are required for larger blocking and larger blocking reduces memory fetches.

We just pass C, A, B and their respective shapes, and it calculates everything.

AVX/pthread:

For AVX and pthread, we do matrix multiplication for Conv2d layer which is the major computation in our model and we try to optimize every part of it with choosing the appropriate number of threads and blocks per thread while utilizing as minimum memory as we can since we can have very huge matrices and it can result in a segmentation fault.

We do dynamic memory allocations as we are limited by memory space. We pass given A and B as global variable to be used by each thread. We calculate vector by vector calculations in pthread function using AVX instructions.

We have tested code using several setups and we found out that maximum number of threads we can use is 30000. Therefore if result have less than this size, we use 1 block at a time. Since the dimension is small, it performs faster, because we are only allocating 3 values in each thread call (2 for specifying column and row indices, and 1 for block). If the dimension is bigger, we choose 169 as the block size. Large block sizes result in failure because they require more memory allocation. 169 works perfectly as every column size of result matrices is divisible by it. We cannot have less than 100 since maximum number of threads is 30k and the first layer will fail.

We also do maxpooling by taking max element from every column of our im2col matrix.

_mm256_loadu_pd: This function loads 4 unaligned double-precision numbers from the given memory address, therefore we transposed the second matrix (B = KxN -> NxK) and padded K in both A and B to be divisible by 4. And then we loaded 4 double at a time.

_mm256_mul_pd: This function pointwise multiplies 2 given vectors which are 4 double from A and four double from B in our case.

_mm256_hadd_pd: This function adds the lowest located 2 double and highest located 2 double. We apply it to our result from mul_pd. Example:

Vector = x, y, z, w -> after hadd(vector, vector) -> x+y, x+y, z+w, z+w

pthread_create: creates a thread with a given thread pointer and applies function with shared data.

pthread_join: suspends execution of the calling thread until the target thread terminates unless the target thread has already terminated. Returns pointer to result if specified which is passed by **pthread_exit.**

We get an array with the size of number of blocks from each thread after calling this function and place it appropriately.

pthread_exit: terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread.

CUDA:

We implement each layer except Batch Norm (there was an issue with the result which we could not fix, but python code is fast enough too, but implementation is still in .cu file) because linear algebra computations associated with training large deep neural networks are commonly performed on GPU. In cases like these, the vectors and matrices are so large that the parallelization offered by GPUs allows them to outperform linear algebra libraries like NumPy. The benefits of parallelization will appear when there is enough data to take advantage of it because of the GPU factor. Otherwise, the benefits are canceled out by the overhead associated thread organization and data transfer.

Implementation of L_RELU, Bias_add, and BatchNorm is simple as we iterate through every element. It can be done easily using NumPy code as well, therefore we are not expecting performance gain. For the maxpool, it is also the same case. We just iterate num of columns time and every time we pick the maximum value in that column.

For Conv2d, we do matrix multiplication with a block size of 16x16 (32x32 did not improve performance). And the grid size will be the division of output sizes to 16.

dim3: dim3 is an integer vector type based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1. We use it to dim3 to specify grid size and block size while passing it to function that will run on GPU device.

cudaDeviceSynchronize: Waits for compute device to finish.

__global__: Global functions are also called "kernels". It's the functions that you may call from the host side using CUDA kernel call semantics (<<<...>>>) and we pass grid size and block size. Device functions can only be called from other devices or global functions. __device__ functions cannot be called from the host code. Grid size specified in <<< >>> is for using parallel blocks, block size is for multithreading (number of parallelisms). By passing the appropriate gird size and block size, we can know in which column and row we are in the function.

Parallelization strategies:

OpenBLAS, cuBLAS, and CUDA handle parallelization by themselves. For manual parallelization, we identified that we can do vector multiplication in parallel since the multiplication of ith row of A and jth column of B gives the result for Cij. For maxpool, we did the same strategy. We identified that all kernel sizes are very small and there is no need to use AVX instructions. We just applied multithreading to maxpool layers since each column gives the result of one index of output and they are independent of each other.

For the Conv2d layer, we have a block size of 169 (explained above) and the number thread is M*N/169. Then we share A and B as global arrays using dynamic memory allocation. We also pass 2 variables at a time to our parallelization function and we get an array of doubles with size BLOCKSIZE from this function for every function call.

Results:

Initial provided code and our previous implementation had running time over 1400 seconds. We then vectorized every layer. The main performance gain came from im2col algorithm as it reduced our problem to a basic matrix multiplication problem and getting max element from every column problem.

Remote server using isolated Docker containers:
- Previous code: 1400+seconds
- Modified python code:
  - DNN inference elapsed time: 0.984
  - End-to-end elapsed time: 1.018
- OpenBLAS:
  - DNN inference elapsed time: 1.491
  - End-to-end elapsed time: 1.518
- cuBLAS:
  - DNN inference elapsed time: 1.995
  - End-to-end elapsed time: 2.023
- CUDA:
  - DNN inference elapsed time: 1.539
  - End-to-end elapsed time: 1.566
- AVX/Pthreads:
  - DNN inference elapsed time: 5.883
  - End-to-end elapsed time: 5.910

AWS EC2 Instance:
- Previous code: 1000+seconds
- Modified python code:
  - DNN inference elapsed time: 1.286
  - End-to-end elapsed time: 1.376
- OpenBLAS:
  - DNN inference elapsed time: 1.743
  - End-to-end elapsed time: 1.815
- cuBLAS:
  - DNN inference elapsed time: 1.683
  - End-to-end elapsed time: 1.724
- CUDA:
  - DNN inference elapsed time: 1.455
  - End-to-end elapsed time: 1.493
- AVX/Pthreads:
  - DNN inference elapsed time: 9.658
  - End-to-end elapsed time: 9.700

The main performance gain was the im2col algorithm and it proves that no matter how parallel or distributed your code is, if it does not use the optimal algorithms, it will suffer. From the previous projects' code, we can see that multiprocessing on the previous project has lost against NumPy code since it does not use the best possible algorithm.

One interesting result is that on AWS, CPU modules slowed down while GPU libraries run faster. It is probably because of EC2 instance has more GPUs than we use in our remote server.

CUDA code might fail to NumPy code because GPUs are good for handling large amounts of data and towards the end of our model, layers will have very small sizes which in those cases as I have mentioned before, CPU will outperform GPU since it is mostly fetching data than actual computation.

AVX/Pthread code is not good either because for the first few layers, it faces huge matrices and as we know, CPU is not good for such cases.

OpenBLAS and cuBLAS use a very efficient GEMM algorithm that handles matrices with any size very well. And we can see that their performance is quite close to CUDA performance.

As a result, we achieved 1000x, 750x, 800x, 930x, and 140x performance speedup against plain and scalar python code while using NumPy, OpenBLAS, cuBLAS, CUDA, and AVX/pthreads respectively.

References:
[1]https://medium.com/@_init_/an-illustrated-explanation-of-performing-2d-convolutions-using-matrix-multiplications-1e8de8cd2544
[2] CS231n Stanford