CS492: Systems for Machine Learning

Project 3 Report

Team 11

Jehoon Park Junoh Moon

Korea Advanced Institute of Science and Technology

June 2, 2020

1 Overview

In this third project, we were required to improve the performance of inferencing Neural Networks (NNs) by offloading compute intensive codes to C and CUDA. Through previous assignments, we saw how fast the well-designed framework (tensorflow) is and how much time is required for computing NN in python. Although codes in the second assignment were accelerated by a "multiprocessing" library in python, it was still too slow to infer images. To accelerate the inference time, we offloaded time-consuming codes to C and parallelized it with 4 different ways. For each device (CPU, GPU), a well designed library (openBLAS, cuBLAS) or a low-level acceleration method (AVX2 + pthread, CUDA) was used for the better performance.

2 Implementation

2.1 openBLAS

openBLAS is a one of libraries that provides Basic Linear Algebra Subprograms (BLAS) operations. There are 3-level BLAS functions: 1. scalar, vector, and vector-vector 2. matrix-vector 3. matrix-matrix operations. Those functions automatically vectorize operations, and hence parallelization could be achieved easily.

The key function we used is `cblas_dgemm(cblas_order, cblas_transpose, cblas_transpose, M, N, K, alpha, A, LDA, B, LDB, beta, C, LDC)`. It performs the Double precision General Matrix-Matrix Multiplication of the following operation.

$$C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$$

The order specifies in which order the data are stored in the matrices (row-major or column-major). If the "transpose" argument is set, it transposes the matrix before multiplication. The arguments LDA, LDB, and LDC are meaning the first dimension of each matrix. They are different from the dimensions of matrix multiplication performed. Therefore, by manipulating the dimension arguments and LDX, we could perform matrix multiplication of some portions of big matrices.

2.1.1 Parallelization Strategy

For the parallelization of CNN, we used openBLAS functions for conv2d, bias_add, and batch_normalization. Since the openBLAS only provides operations about vectors and matrices, leakyReLU and maxpool2d could not be accelerated by it because they consisted of comparing operations. Moreover, although

bias_add and batch_normalization used openBLAS, they required almost the same time as just offloading python code to C due to similar reasons.

For the conv2d, we used the Toeplitz matrix for implementing the convolution. We transformed input tensor to Toeplitz matrix. The tiled_in matrix's size was (output channel size $\times$ # of input channels $\cdot$ kernel size), and the weight matrix's size was (# of input channels $\cdot$ kernel size $\times$ # of output channels). By multiplying tiled_in and weight, we could get the output of (output channel size $\times$ # of output channels). In conclusion, we could calculate the convolution by only one function call. The parallelization was automatically done by the openBLAS library.

For the bias_add and batch_normalization, we used `cblas_daxpy` and `cblas_dscal` : vector-vector addition and vector scalar operation. Since there were no efficient way to duplicate parameters vector (*e.g.* biases, mean, gamma), we executed those functions for # of input tensor size or # of input channels times. It was also possible just manually duplicating the vectors to the size of input tensors then executing the addition by one function call. However, since the overhead of duplicating was not negligible, we implemented like we wrote. Actually, those two layers required almost the same time as just porting the python code to C.

2.2 cuBLAS

The cuBLAS is a GPU implementation of the BLAS library. The API is almost the same as the openBLAS library. The one important difference is that cuBLAS matrix functions are based on the column-major order. This is due to the GPU's memory structure, so we should pay attention to the data's stored order.

The key function we used is `cublasSgemm(cublas_handle, cublas_transpose, cublas_transpose, M, N, K, alpha, A, LDA, B, LDB, beta, C, LDC)`. It performs the Single precision General Matrix-Matrix Multiplication of the following operation.

$$C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$$

The most arguments are the same as the `cblas_dgemm`. However, there is no matrix order argument because the matrix order was fixed to the column-major. Also, a new argument of the handle to the cuBLAS library context was added. Matrices A, B, C should be in the GPU memory. We used `cublasSetMatrix` function to copy the matrix in the host memory to GPU memory.

2.2.1 Parallelization Strategy

We used cuBLAS for the conv2d only. Other layers are the same as openBLAS implementation. We tried to use cuBLAS for the bias_add and batch_normalization. However, we had to allocate GPU memory and copy the matrix from the host memory, which incurred a lot of extra overhead.

For the conv2d layer, we allocated GPU memory, copied the data from host memory, called the `cublasSgemm`, and copied the result back to the host memory. Since cuBLAS performed in column-major order, we used the transpose property that if we transposed the row-major order matrix, it became column-major order. The tiled_input, weight, and output matrices were stored in row-major order, so they could be interpreted as transposed matrices in column-major view. By transposing the equation `cublasSgemm`, we could get $C^T = B^T \times A^T$. Therefore, by switching the order of matrix multiplication to weight $\times$ tiled_input, we could achieve the same output as the openBLAS.

2.3 AVX + pthread

In this approach, AVX2 intrinsics and *pthread* were used to exploit computer resources. AVX2 intrinsics are intrinsics functions for instruction-level-parallelism (ILP) to force AVX2 instructions. Since AVX2 can process 256-bit values at once, eight floating points can be calculated simultaneously, and hence the process is able to fully exploit system resources. In this code, `_mm256_loadu_ps` and `_mm256_set1_ps` were used for fetching values into AVX2 registers; `_mm256_add_ps`, `_mm256_mul_ps`, `_mm256_sub_ps`, `_mm256_div_ps`, `_mm256_max_ps`,and `_mm256_sqrt_ps` were used for calculation; `_mm256_storeu_ps` was used for writing data into the memory. Among them, `_mm256_mul_ps` and `_mm256_add_ps` intrinsics were the most used because the conv2d layer was a bottleneck and it relied heavily on MAC operations. In contrast, pthread is a library for thread-level parallelism (TLP). This library can swawn threads by invoking the `pthread_create` function. Although `pthread_create` tells us that a given function is running, it does not tell us termination of the function. Thus, we used `pthread_join` to ensure all spawned threads were terminated.

2.3.1 Parallelization Strategy

Since the originally given codes were run on a CPython interpreter, all layers took too much time. Hence, we should have ported all layers into C. Some layers (leakyReLU, bias_add, batch_normalization) were relatively easy to port since all layers are calculated in element-wise. Thus, we could interpret the layer as a 1-dimensional array and vectorize easily.

Other layers, however, could not be interpreted as 1-dimensional arrays since the maxpool2d and conv2d layers view data as matrices. Thus, we added a preprocess step that vectorizes each receptive field while striding the input tensor. In detail, while a kernel-sized window strides a padded input, this window is flattened into a $(1, kl)$ 1-dimensional array, where *kl* is a size of kernels (weights) for each filter; to flatten into a 1-D array, each row is concatenated into an array.

By doing this, the maxpool2d layer could be calculated as follows: because the shape of receptive field is $(1, kl)$ and the shape of a weight is $(kl, \#filters)$, an output of "receptive field $\times$ weight" is shaped as $(1, \#filters)$. This means each output of the receptive field generates only a pixel of the output and does not affect each other, ensuring no race condition among them. We made a simplified thread pool for parallelizing matrix multiplication; each worker in a pool is responsible for a receptive field and the corresponding output pixel. As written in Listing 1, moreover, matrix multiplication codes were carefully designed for exploiting cache locality.

```
clear(C)      //Clear all values to zero
//C = A x B
for(int j = 0; j < J; ++j) {
    for(int i = 0; i < I; ++i) {
        for(int k = 0; k < K; ++k) {
            C[i, k] += A[i, j] * B[j, k]
        }
    }
}
```

Listing 1: A cache-friendly matrix multiplication pseudo-code. Unlike normally used codes (I, J, K ordered loop), this code minimized horizontal traversal to fully exploit cache prefetchers and cache locality. Note that the shapes of A, B, and C are (I x J), (J x K), and (I x K).

Consequently, the most inner loop is accelerated by AVX2 intrinsics as Listing 2.

```
clear(C)      //Clear all values to zero
//C = A x B
for(int j = 0; j < J; ++j) {
    for(int i = 0; i < I; ++i) {
        int k;
        for(k = 0; k + AVX_SIZE < K; k += AVX_SIZE) {
            vec_A = _mm256_loadu_ps(&A[i, j])
            vec_B = _mm256_loadu_ps(&B[j, k])
            vec_C = _mm256_loadu_ps(&C[i, k])
            vec_C = _mm256_add_ps(vec_C, _mm256_mul_ps(vec_A, vec_B))
            _mm256_storeu_ps(&C[i, k], vec_C)
        }
        for(; k < K; ++k) {
            C[i, k] += A[i, j] * B[j, k]
        }
    }
}
```

Listing 2: A revised matrix multiplication pseudo-code with AVX2. In the first loop of k, *AVX_SIZE* elements are calcuated at once if elements are sufficient. In the second loop, the remaining elements are calcuated. Note that *AVX_SIZE* is 8

since 32-bit floating points were used and AVX2 can hold 256-bit values at once.

Similarly, a basic unit per thread of the maxpool2d layer was a receptive field. The difference is, however, the maximum inside a receptive field can be determined while interpreting the receptive field as a 1-D array. Since this receptive field can be viewed as a 1-D array, it was relatively easy to apply AVX2 intrinsics.

2.4 CUDA

Since CUDA C++ extension is a programming language rather than a multiprocessing library, the key work for parallelizing was setting up indices, as apposed to picking up a proper high-level function. When invoking a CUDA function, we have to set the number of blocks and threads. Thus, a matrix is divided into the number of blocks and calculated in parallel. Moreover, a sub-matrix in a block is calculated by multiple threads to hide memory latency.

2.4.1 Parallelization Strategy

Although AVX2 and pthread dramatically reduced the time, conv2d still took 98% of inference time. Thus, we decided to make it faster by GPU. By setting up $\lceil \frac{\#filter}{16} \rceil \times \lceil \frac{n\_strides}{16} \rceil$ blocks per grid and 16×16 threads per block, GPU split a *(n_stride* $\times$ *#filter)* shaped matrix into $\lceil \frac{\#filter}{16} \rceil \times \lceil \frac{n\_strides}{16} \rceil$ blocks and each block spawns 16^2 threads simultaneously. Unlike AVX2 codes, matrix multiplication codes were slightly changed due to two things: a GPU parallelizes matrix multiplication by calculating elements simultaneously, and hence a critical section must be adjusted; the input matrix was not exactly partitioned into blocks, we also added a guard code.

Thus, the revised code is as Listing 3.

```
__global__ void matmul(float *A, float *B, float *C, int I, int J, int K)
    {
    int row = blockIdx.y * blockDim.y + threadIdx.y
    int col = blockIdx.x * blockDim.x + threadIdx.x

    if(row < I && col < K) {
        float acc = 0.f
        for (int k = 0; k < K; ++k) {
            acc += A[row, k] * B[k, col]
        }
        C[row, col] = acc
    }
}

// Calling usage
```

```
int blockSize = 16
dim3 dimBlock(blockSize, blockSize)
dim3 dimGrid(ceil((float)K / dimBlock.x), ceil((float)I / dimBlock.y))

matmul<<<dimGrid, dimBlock>>>(A, B, C, I, J, K)
```

Listing 3: A GPU-acclerated matrix multiplication pseudo-code. For simplicity, some subordinate codes (cudaMalloc, cudaMemcpy, cudaFree) were not represented in this example.

3 Evaluation and Discussion

	Baseline		**openBLAS**		**cuBLAS**		**AVX**		**CUDA**	
Conv2D	5919.118	97.94%	0.434	49.10%	0.520	61.61%	2.363	93.88%	0.375	72.39%
BatchNorm	75.989	1.26%	0.087	9.84%	0.060	7.11%	0.064	2.54%	0.056	10.81%
MaxPool2D	6.291	0.10%	0.210	23.76%	0.165	19.55%	0.037	1.47%	0.034	6.56%
BiasAdd	21.774	0.36%	0.050	5.66%	0.029	3.44%	0.031	1.23%	0.028	5.41%
LeakyReLU	20.149	0.33%	0.101	11.43%	0.069	8.18%	0.021	0.83%	0.021	4.05%
DnnInferenceEngine (inference time)	6043.331	100.00%	0.884	99.77%	0.844	99.88%	2.517	99.96%	0.518	99.23%

Table 1: The table shows the execution time of each layer's `run` method. The baseline represents the given full-python codes. The left column of each item means the total execution time in seconds and the right column means the percentage over the inference time.

To evaluate the increase over plain-python codes, four approaches were evaluated on an Amazon EC2 p2.xlarge instance with an Intel Xeon E5-2686 v4 2C4T processor, 64GB memory, and a Nvidia Tesla K80 GPU. The execution time was evaluated by cProfile, a built-in profiler module in python, and analyzed by pstats, also a built-in python module.

As shown in Table 1, every case offered more than 2400× dramatic speedup over the baseline. In particular, Conv2D accounted for 97.94% of the total inference time in the baseline, so porting Conv2D into C or CUDA played the biggest role in breaking a bottleneck. Comparison over the AVX and the openBLAS codes supported this; although the AVX showed better performance on four layers except Conv2D, the openBLAS code showed 5-fold reduced execution time on Conv2D, and hence the openBLAS offered 2.8× speedup over the AVX. Moreover, GPU-acceleration also helped breaking a bottleneck. Compared to AVX, CUDA reduced the time by 6.3× in Conv2D, so the CUDA reduced it by 4.8× without changing other layers.

However, Conv2D in the cuBLAS implementation consumed more time than openBLAS and CUDA. The reason for the less performance than openBLAS was cuBLAS required additional overhead of GPU memory allocation and copying

data from the host memory. Furthermore, cuBLAS is a library for matrix multiplication, not convolution. From convolutional point of view, cuBLAS was less efficient than manual parallelization in CUDA. In detail, cuBLAS calculated the convolution using single `cublasSgemm` invocation and the library automatically parallelized the computation. In contrast, manual CUDA programming had a room for hand-picked parameter optimizations, in particular when setting up the number of blocks and threads. Thus, this could be one of the explainations why CUDA was faster than than cuBLAS.

	Baseline	**TF CPU**	**TF GPU**	**openBLAS**	**cuBLAS**	**AVX**	**CUDA**
End-to-End Time	3796.482	0.416	0.157	0.935	0.871	2.532	0.535
FPS	0.0002634	2.403	6.369	1.070	1.148	0.395	1.869

Table 2: The table shows the overall performance of the given cases. To evaluate precisely without interruptions, these values were measured without a profiler, unlike Table 1. The end-to-end time stands for the inference time as well as post-processing time in seconds. Two Tensorflow benchmarks came from the first project.

As shown in Table 2, we also measured the end-to-end elapsed time and its corresponding FPS and compared them with the Tensorflow implementation since the baseline was far slower than the others. We measured Tensorflow's FPS in the same environment (Amazon EC2) with the code from the first project. As we already saw in the inference time evaluation, our implementation showed dramatic speedup over the baseline. However, our fastest implementation was still not as fast as the Tensorflow. Tensorflow CPU and GPU offered 2.2$\times$ speedup over the openBLAS, and 3.4$\times$ speedup over the CUDA. These performance gaps came from the fact that Tensorflow is a library optimized for NNs. The way we accelerated NNs was offloading and parallelizing the computations in each layer. Moreover, the graph construction of layers was implemented with `networkx` library, written in python. In contrast, Tensorflow does not only parallelize the computations, but also accelerates graph construction, and other functions since their backends are written in C.

Moreover, Tensorflow is a production-level library and fully optimized by the compiler. With aggressive optimization flag ($e.g.$ -On in GCC), the compiler can aggressively optimize codes: using registers instead of a call stack in the memory, preventing memory aliasing, prefetching caches, hoisting loop-invariant codes, substituting expressions with same meaning but cheaper ones, and sharing common subexpressions. Therefore, the gap would be reduced if our codes are compiled with aggressive optimization.