CS492 Project 3 Report

Anchit Tandon 20206033
Jinwoo Kim 20160171

June 5, 2020

Contents

1 Parallelization Recipe

This section describes the key library functions used for optimizing `dnn.py`.

1.1 Ctypes

To offload Python computations to C, we used following features of `ctypes`:

- `ctypes.cdll.LoadLibrary`, to dynamically link `.so` object files to `dnn.py`.
- `c_float`, `c_int` and `POINTER` function for specifying ctypes.
- `argtypes` to specify argument types of an imported function.
- `ctypes.data_as` method to cast numpy arrays to ctypes scalar or pointer.

1.2 CPU: OpenBLAS

We only used `cblas_sgemm` to parallelize matrix-matrix multiplications. Given input 2D matrices **A**, **B**, **C**, and scalar factors α and β, `cblas_sgemm` performs a matrix multiply-and-add operation and writes the results to **C**, denoted as $\mathbf{C} \leftarrow \alpha * \mathbf{AB} + \beta * \mathbf{C}$.

```
void cblas_sgemm (
    const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa,
    const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k,
    const float alpha, const float *a, const MKL_INT lda, const float *b,
    const MKL_INT ldb, const float beta, float *c, const MKL_INT ldc)
```

A number of parameters define input data and layout of the matrices:

- **A**, **B** and **C** should be organized contiguously in memory, and their pointers are passed through parameters `a`, `b` and `c`. α and β are passed through `alpha` and `beta`.
- `layout` defines memory layout of matrices (either `CblasRowMajor` or `CblasColMajor`).
- `transa` and `transb` defines whether to transpose **A** or **B** before computation (either `CblasNoTrans` or `CblasTrans`, as we consider real matrices).
- `m`, `n`, and `k` contain dimension information such that $\texttt{transa}(\mathbf{A})^{\texttt{m}\times\texttt{k}}$ and $\texttt{transb}(\mathbf{B})^{\texttt{k}\times\texttt{n}}$.
- `lda`, `ldb`, and `ldc` represents leading dimension of **A**, **B** and **C** defined by the caller; in plain matmul we let them one of `m`, `n`, and `k`, depending on `layout` and `trans`.

1.3 CPU: AVX & pthread

We leveraged (1) AVX primitives for SIMD parallelization of eight float32 values and (2) pthread library to exploit multicore CPU via multithreading.

AVX functions for SIMD operation[1]:

- `__m256 _mm256_setzero_ps(void)` Returns a zero-initialized vector.
- `__m256 _mm256_loadu_ps(float const *a)` Moves values of a vector at unaligned pointer `a` to an aligned vector and returns it.
- `__m256 _mm256_mul_ps(__m256 m1, __m256 m2)` Performs an elementwise multiplication of source vectors `m1` and `m2`, and returns the resulting vector.
- `__m256 _mm256_add_ps(__m256 m1, __m256 m2)` Performs a SIMD elementwise addition of source vectors `m1` and `m2`, and returns the resulting vector.
- `__m256 _mm256_set1_ps(float)` Returns a vector with all elements initialized with given float32 value.
- `__m256 _mm256_max_ps(__m256 m1, __m256 m2)` Performs a SIMD elementwise maximum of source vectors `m1` and `m2`, and returns the resulting vector.
- `__m256 _mm256_permute_ps(__m256 m1, int control)` Permutes source vector `m1` following four control fields in the lower 8 bits of `control`, and returns the resulting vector. For example, when `control` is `0b01110100` (control fields `[01,11,01,00]`), lower and upper half `l`, `h` of `m1` are permuted to `l[1],l[3],l[1],l[0]` and `h[1],h[3],h[1],h[0]` respectively, constituting the return vector.

[1]Note: we write "vector" to represent an aligned `m256` vector containing eight float32 values.

1.4 GPU: cuBLAS

We used `cublasSgemm` to parallelize `matmul` in an exactly same manner to OpenBLAS, along with following helper functions for host-device communication[2]:

- `cublasCreate(cublasHandle_t *handle)` Initializes cuBLAS library and creates a handle referencing library context, allocating hardware resources on both the host and device.
- `cublasSetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)` Copies the elements from matrix `A` in host memory to `B` in device memory.
- `cublasGetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)` Copies elements from matrix `A` in device memory to `B` in host memory.

-
  ```
  cublasSgemm(cublasHandle_t handle,  cublasOperation_t transa,
          cublasOperation_t transb, int m, int n, int k, const float *alpha,
          const float *A, int lda, const float *B, int ldb, const float *beta,
          float *C, int ldc)
  ```

  Performs parallelized matrix-matrix multiplications. The syntax and operation of this method is quite to similar to `cblas_sgemm`, except that both matrices are stored in column-major format, with the leading dimension of `A` and `B` given in `lda` and `ldb`, respectively.
- `cublasDestroy(cublasHandle_t handle)` Frees hardware resources.

1.5 GPU: CUDA

We leveraged CUDA instruction primitives to manage global and shared GPU memory, and to execute kernel function on in-device data.

CUDA primitives (return type `cudaError_t`):

- `cudaMalloc(void **devPtr, size_t size)` Allocates memory of `size` bytes on the device and sets `*devPtr` a pointer to the allocated memory.
- `cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)` Copies `size` bytes from `src` to `dst`, following `kind` which is either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.
- `cudaFree(void* devPtr)` Frees device memory specified by `devPtr`.
- `__global__ void mykernel()` The `__global__` keyword indicates that the function `mykernel` runs on the device (as a kernel), which we call from host code with `mykernel<<<NUM_BLOCKS, NUM_THREADS, SHARED_MEMORY_BYTES>>>()`. The parameters within brakets denote:
  - `NUM_BLOCKS` Parallel invocation of kernel function, index accessed by `blockIdx.x`
  - `NUM_THREADS` Parallel threads within a block, index accessed by `threadIdx.x`
  - `SHARED_MEMORY_BYTES` Dynamic allocation of on-chip shared memory.
- `void __syncthreads()` Synchronizes all threads within a block.

[2]We explain CUDA primitives in GPU:CUDA section.

2 Parallelization Strategy

2.1 CPU: OpenBLAS & GPU: cuBLAS

OpenBLAS As the goal was outperforming the scalar-operation baseline, we only optimized `Conv2D.run`, the computation bottleneck. We hypothesized that acceleration of operations at each sliding window will be sufficient for beating the baseline. Thus, OpenBLAS loops over output pixels.

Given padded input $\mathbf{I}^{W_i \times H_i \times C_i}$, kernel $\mathbf{K}^{W_k \times H_k \times C_i \times C_o}$ and strides s_w, s_k, we optimized computation of $\mathbf{O}[w, h, :]$, where $\mathbf{O}^{W_o \times H_o \times C_o}$ is the convolved output tensor[3]. If we naively loop over kernel pixels, the computation is done as follows:

$$\mathbf{O}[w, h, :] = \sum_{i=0}^{W_k} \sum_{j=0}^{H_k} \mathbf{I}[ws_w + i, hs_h + j, :]\mathbf{K}[i, j, :, :] \tag{1}$$

Here, we observe that the above formula can be modified to a single vector-matrix multiplication $\mathbf{O}[w, h, :] = \mathbf{I}_{(w,h)}\mathbf{K}_{(2d)}$, where:

- $\mathbf{I}_{(w,h)}^{W_k * H_k * C_i} = \mathbf{I}[ws_w : ws_w + W_k, hs_h : hs_h + H_k, :]$`.flatten()` is input at a sliding window,
- $\mathbf{K}_{(2d)}^{(W_k * H_k * C_i) \times C_o}$ is a 2D matrix formed by reshaping the kernel $\mathbf{K}$.

which we optimize using OpenBLAS `cblas_sgemm` function[4]. To prevent interference with internal multiprocessing of OpenBLAS, we sacrificed speedup with `multiprocessing` library and looped over output pixels to get the final $\mathbf{O}$.

As a minor optimization, we avoided re-calculating $\mathbf{K}_{(2d)}$ by reusing it over iterations.

cuBLAS For cuBLAS, we held the optimization strategy exactly as same as OpenBLAS, differing only in usage of `cublasSgemm` function and `cuda` and cuBLAS primitives to support host-device communication. This leads to an interesting phenomenon of cuBLAS running slower than OpenBLAS, which we discuss in the Results section.

2.2 CPU: AVX & pthread

We optimized `run` method of `Conv2D`, `MaxPool2D`, `BiasAdd`, `BatchNorm`, and `LeakyRelu` in two different levels of parallelism:

(1) Thread-level (TLP): Exploiting multicore CPU by multithreading with `pthread`.

(2) Instruction-level (ILP): SIMD vector computations using AVX library.

To keep the report compact, we provide a detailed description of optimization strategy for `Conv2D.run`, and provide simple explanations for other layers.

2.2.1 `Conv2D`

We observed that looped computation (similar to OpenBLAS) results in a prohibitive runtime. So, in AVX, we preprocess the padded input $\mathbf{I}^{W_i * H_i * C_i}$ to a Toeplitz matrix $\mathbf{T}^{(W_o * H_o) \times (W_k * H_k * C_i)}$:

$$\mathbf{T}[w * H_o + h, :] = \mathbf{I}[ws_w : ws_w + W_k, hs_h : hs_h + H_k, :]\texttt{.flatten()} \tag{2}$$

$$(0 \le w \le W_o, 0 \le h \le H_o) \tag{3}$$

Then, output $\mathbf{O}^{W_o \times H_o \times C_o}$ can be computed by multiplying $\mathbf{T}$ and reshaped kernel $\mathbf{K}_{(2d)}^{(W_k * H_k * C_i) \times C_o}$, followed by reshaping to $W_o \times H_o \times C_o$. Thus, our problem reduces to optimizing a single `matmul`.

Thread-level: We divided `matmul` into 16 loads across rows of $\mathbf{T}$ and columns of $\mathbf{K}$, and distributed them to 16 threads. Each thread computes disjoint elements in output matrix in parallel to others.

[3]Note: We omit batch size to simplify notation. Baseline code assumes batch size of 1.

[4]Note: We treat vector-matrix multiplication as an instance of matmul with one of the dimensions being one.

For load-balancing optimization, our code dynamically determines how to split **T** and **K**; For example, if number of rows of **T** is too small (e.g. 2 rows), we split more in the column dimension of **K** (e.g. split columns to 8 sets) so that each of 16 threads always holds non-zero load.

Instruction-level: Given a thread with $\mathbf{T}^i$ and $\mathbf{K}^j$, the thread loops over each rows of $\mathbf{T}^i$ and columns of $\mathbf{K}^j$ and computes dot product between column of $\mathbf{T}^i$ and row of $\mathbf{K}^j$, which is saved at corresponding element of output **O**.

We further divided the computation as following to exploit SIMD of eight floating point values:

$$\mathbf{T}^i\mathbf{K}^j = \sum_{n=1}^{N} \mathbf{T}^i[:, \text{chunk}_n]\mathbf{K}^j[\text{chunk}_n, :] \tag{4}$$

where the length of chunk_n is 8. For each chunk, we used `_mm256_mul_ps` for SIMD multiplication.

2.2.2 `MaxPool2D`

Inspired by performance gain in `Conv2D`, we first construct a Toeplitz-like matrix $\mathbf{T}^{(W_o*H_o*C_o)\times(W_k*H_k)}$ from padded input, which is organized in a way that when `max` operation is done for each rows, the flattened output $\mathbf{O}^{W_o*H_o*C_o}$ is obtained (we omit the details here).

Thread-level: We split **T** over row dimension into 16 threads, where each thread computes `max` of designated rows. Unlike `Conv2D`, $W_o * H_o * C_o$ is guaranteed to be larger than 16, so dynamic load balancing is not needed.

Instruction-level: Given a thread with $\mathbf{T}^i$, the thread loops over rows of $\mathbf{T}^i$ and computes maximum over elements. We further split a row to chunks of eight elements and leverage SIMD instructions to find maximum element in the row. We provide a oversimplified pseudocode:

```
Algorithm 1: Finding maximum element in v
Result: max(v)
buf← _mm256_set1_ps(1e-20);
chunk← _mm256_loadu_ps(&v);
while chunk is within v do
    buf ← _mm256_max_ps(chunk, buf);
    chunk ← next(chunk);
end
return find_max_in_mm256(buf);
```

Here, find_max_in_mm256() is a routine that finds maximum element within an `_mm256` vector, by repeatedly rotating the vector using `_mm256_permute_ps` and updating it with `_mm256_max_ps`. For details, please refer to `void * mp_func(void * aux)` in `dnn_avx.c`.

2.2.3 `BiasAdd`

Thread-level: `BiasAdd` is an element-wise addition of input matrix $\mathbf{I}^{W\times H\times C}$ and bias vector $\mathbf{B}^C$, where values of **B** are broadcasted over pixels. We split output channels of **I** and **B** into 16 loads, where addition of each splits ($\mathbf{I}^i$ and $\mathbf{B}^i$) are processed in one of 16 threads.

Instruction-level: Within a thread with $\mathbf{I}^i$ and $\mathbf{B}^i$, for an output channel c, we divided the pixels of $\mathbf{I}^i[:, :, c]$ into chunks of 8 and used `_mm256_add_ps` to perform SIMD addition with $\mathbf{B}^i[c]$ broadcasted using `_mm256_set1_ps`.

2.2.4 `BatchNorm`

`BatchNorm` is an element-wise normalization of input matrix $\mathbf{I}^{W\times H\times C}$ with channelwise parameters $\mathbf{mu}^C$, $\mathbf{gamma}^C$, $\mathbf{var}^C$ and **eps**. As this is a simple extension of `BiasAdd`, we skip the details.

2.2.5 `LeakyRelu`

`LeakyRelu` is an element-wise rectification of input matrix $\mathbf{I}^{W\times H\times C}$ without any channelwise parameters. Even simpler than `BatchNorm`, we simply flatten the input, split input over 16 threads, and rectify 8 inputs in a row using SIMD operations `_mm256_mul_ps` and `_mm256_max_ps`.

2.3 GPU: CUDA

When using CUDA, we observed that refactoring the `Conv2D` into a single matrix multiplication didn't help in performance, presumably due to low utilization of on-chip memory and host-device communication overhead.

Thus, we kept the sliding window scheme for computing `Conv2D` and `MaxPool2D`, and sought for computation routines that utilize on-chip memory and maximize data reuse.

For `BiasAdd`, `BatchNorm` and `LeakyRelu`, we observed that simple routines can compete with NumPy.

2.3.1 `Conv2D`

To compute convolved output $\mathbf{O}^{W_o \times H_o \times C_o}$ from $\mathbf{I}^{W_i \times H_i \times C_i}$ and $\mathbf{K}^{W_k \times H_k \times C_i \times C_o}$, we employ a channel-wise sliding window scheme. For each output pixel (w_o, h_o, c_o), corresponding partition of input and kernel are elementwise-multiplied and accumulated:

$$\mathbf{O}[w_o, h_o, c_o] = (\mathbf{I}[w_o s_w : w_o s_w + W_k, h_o s_h : h_o s_h + H_k, :] \otimes \mathbf{K}[:, :, :, c_o])\texttt{.sum()} \tag{5}$$

The above computation for all the output pixels can be (simply) distributed across multiple blocks and threads. However, our goal is to maximize on-chip data reuse between threads in a block.

For this purpose, we consider the shared memory size dynamically allocated as $(W_k * H_k * C_i)$, same as the number of elements in a sliding window. This gives us two choices:

- Input-Stationary (IS) Dataflow

  Load $\mathbf{I}[w_o s_w : w_o s_w + W_k, h_o s_h : h_o s_h + H_k, :]$ onto shared memory of a block. Within each thread, load $\mathbf{K}[:, :, :, c_o]$ with different c_o from global memory, and compute $\mathbf{O}[w_o, h_o, c_o]$.

- Weight-Stationary (WS) Dataflow

  Load $\mathbf{K}[:, :, :, c_o]$ onto shared memory of a block. Within each thread, load $\mathbf{I}[w_o s_w : w_o s_w + W_k, h_o s_h : h_o s_h + H_k, :]$ with different w_o, h_o from global memory, and compute $\mathbf{O}[w_o, h_o, c_o]$.

A critical observation here is that the extent of data reuse can be different between IS and WS, depending on the values of w_o, h_o and c_o. For example, if $c_o \ll 512$, IS would fail to achieve the maximum data reuse because the shared input can only be reused up to c_o times. On the other hand, if $(w_o * h_o) \gg 512$, WS can reuse the shared kernel data exactly 512 times.

To maximize the data reuse, we leverage this property and dynamically determine whether to use input-stationary dataflow or weight-stationary dataflow on runtime.

As a load-balancing optimization, we distribute the amount of data-loading into shared memory across 512 threads.

2.3.2 `MaxPool2D`

Although there is room for data reuse across multiple sliding windows during the `MaxPool2D` computation, we observed that (1) allocating shared data considering 2D layout and shared memory capacity is error-prone and (2) `MaxPool2D` runs faster than vectorized NumPy when the entire input is loaded to global memory. Hence, we don't allocate shared data and compute an output pixel for each thread independently.

2.3.3 `BiasAdd`

When adding $\mathbf{I}^{W \times H \times C}$ to channelwise $\mathbf{B}^{C}$, we observe that allocating a single output channel c_o to a block and sharing $\mathbf{B}[c_o]$ across threads gives a minor improvement, so we do so.

2.3.4 `BatchNorm`

A simple extension of `BiasAdd`, we skip the details.

2.3.5 `LeakyRelu`

We simply flatten the input and process the elements in threads within blocks. There is no room for data reuse.

3 Performance Analysis

We make following comparisons:

- BLAS vs. baseline scalar operations
  - Preformance gain of OpenBLAS and cuBLAS from `SCALAR`
  - Additional discussion: OpenBLAS and cuBLAS
- AVX/CUDA vs. fully vectorized numpy operations
  - Performance analysis of AVX and CUDA from NumPy
  - Additional discussion: effect of dynamic stationary in CUDA

3.1 BLAS vs. Baseline

We observe a considerable end-to-end performance gain of 20×-40× on offloading only the sliding window operations of `Conv2D` to the BLAS-provided matrix multiplication function.

time (sec)	Baseline	OpenBLAS	cuBLAS
End-to-end	3822.6	114.7	221.0
DNN inference	3822.2	114.7	221.0
`Conv2D` summed	3753.9	21.7	124.7

- `Conv2D` takes up majority (98.2%) of total runtime in the baseline, making it a primary optimization target.
- BLAS functions reduce `Conv2D` running times to a large extent: 180× faster in OpenBLAS, 30× in cuBLAS.
- Faster `Conv2D` leads to an end-to-end performance gain of 34× in OpenBLAS and 17× in cuBLAS.
- It is observed that `Conv2D` runs about 6× slower in cuBLAS than OpenBLAS. Considering that we call the offloaded C function for each sliding window, and data loading from host to device occurs in every C function call, it is highly likely that host-device data communication overhead causes this inefficiency. Nevertheless, we compensate for this problem in CUDA code by data sharing and massive parallelization.

3.2 AVX/CUDA vs. Vectorized NumPy

To make the problem more challenging, we implemented a fully vectorized version of baseline code (`dnn_vec.py`) that leverages highly optimized functions of NumPy library. We report that our CUDA code outruns NumPy with 2.5× performance gain, and AVX code performs competitively[5].

time (sec)	Baseline	Numpy	AVX	CUDA
End-to-end	3822.6	1.125	1.383	0.449
DNN inference	3822.2	1.038	1.332	0.399
`Conv2D` summed	3753.9	0.650	1.157	0.258
`BiasAdd` summed	12.7	0.022	0.058	0.036
`BatchNorm` summed	37.7	0.033	0.036	0.034
`LeakyReLU` summed	11.7	0.044	0.038	0.024
`MaxPool2D` summed	6.3	0.274	0.187	0.024

- CUDA outperforms Numpy with 2.5× performance gain, and AVX is comparable to Numpy (1.2× slower). This performance gain comes from 2.5× faster `Conv2D`, and 11× faster `MaxPool2D`.
- AVX fails to overrun `Conv2D` of NumPy (1.8× slower). `Conv2D` takes up majority (>60%) of total runtime in all cases, making it the primary source of performance difference in AVX and CUDA.

[5]All running time was measured for five trials each, then averaged.

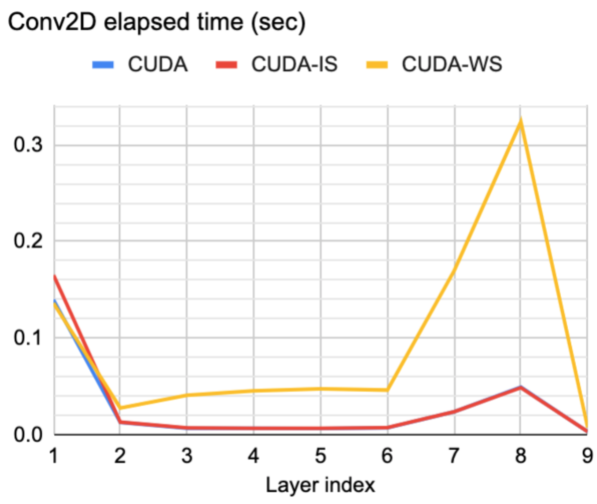
3.3 Analysis: CUDA Dynamic Stationary Dataflow

To verify the effect of dynamic stationary dataflow (deciding input stationary dataflow (IS) or weight stationary dataflow (WS) based on input sizes), we address an ablation study of dynamic decision.

time (sec)	CUDA (dynamic)	CUDA (IS only)	CUDA (WS only)
End-to-end	0.449	0.479	1.08
DNN inference	0.399	0.428	1.03
`Conv2D` summed	0.258	0.283	0.846

- When `Conv2D` relies only on either one of input or weight stationary dataflow, the performance drops.
- We observe the best performance when each layer switches to weight stationary when number of pixels is $\geq$(100$\times$max # of threads within a GPU block); In YOLOv2-tiny, this corresponds to the first `Conv2D`.
- The input to first layer has excessive number of pixels and less number of channels, and therefore serves as a critical bottleneck in algorithms that parallelize over output channels.
- Although weight stationary dataflow performs poorly in most of the cases (as can be seen in WS only), it maximizes reuse of kernel weights over many pixels, effectively reducing inference time when adopted dynamically.

To further identify how adaptive stationary dataflow reduces runtime, we present a runtime plot for all `Conv2D`:



- Weight stationary is inefficient in most cases where number of pixels is small, except in the first `Conv2D`.
- As the bottleneck of input stationary scheme (IS) is the first layer, adaptation of WS here makes a noticeable impact on the summed running time (1.1$\times$ acceleration on `Conv2D` elapsed time).
- To avoid inefficiency, we suggest using IS as the 'default' dataflow and conservatively choosing to use WS. We assure this by using WS only when number of pixels is $\geq$(100$\times$max # of threads within a GPU block).

4 Conclusion

- We achieve improvement over baseline code of DNN inference in four different parallelization schemes (OpenBLAS, cuBLAS, AVX, CUDA) by offloading the python code to C.
- As seen in OpenBLAS / cuBLAS comparison, with an inappropriate use of GPU interface, host-device data communication can form a bottleneck.
- By leveraging both thread-level parallelism and instruction-level parallelism, AVX and pthread-based code achieves performance comparable to highly vectorized NumPy.
- By dynamically determining between weight or input stationary dataflow in utilization of on-chip shared memory, GPU-based CUDA code achieves 2.5$\times$ acceleration compared to NumPy.