Optimization of a DNN program on the CPU+MIC

University of Electronic Secience and Technology of China

Abstract

This article is a part of competition proposal of Asia Supercomputer Student Challenge. We analysis the `DNN` program, put forward different optimization methods, test them and point their pros and cons. In the end we talk about our limitations.

1. Introduction

There is a program based on a standalone hybrid CPU+MIC platform called `DNN(deep neural network)` needed to be parallelized for obtain better performance. Here is some detailed information about hardware in Figure 1, software configuration in Figure 2.

After optimization, the final program is tested on one computing server in the CPU+MIC hybrid cluster. Performance analysis in this proposal is based on the results of this test.

Item	Name	Configuration	Hosts
Server	Inspur NF5280M4 x 4	CPU：Intel Xeon E5-2680v3 x 2, 2.5Ghz, 12 cores	**hostname:** mic1, mic2, mic3, mic4
		Memory: 16G x8, DDR4, 2133Mhz	
		Hard disk: 1T SATA x 1	
		Accelerator card: Intel XEON PHI-31S1P (57 cores, 1.1GHz, 1003GFlops, 8GB GDDR5 Memory)	
Network		Infiniband+Ethernet	

Figure 1. Hardware configuration

Classification	Description	Installation path	Version
OS	GNU/Linux		RHEL 7.1
Compiler	Intel Composer XE Suites	/opt/intel/composer_xe_2015.0.090	2015.0.090
MKL	Intel MKL	/opt/intel/mkl/lib/intel64	
MPI	Intel MPI	/opt/intel/impi/5.0.1.035	5.0.1.035
PBS	Torque	/opt/tsce	3.0.5

Figure 2. Software configuration

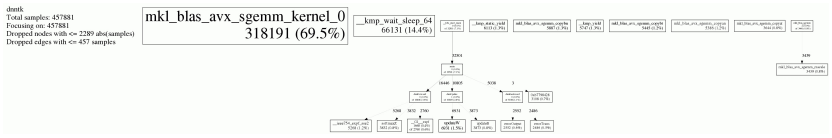

Figure 3. Google Perfools results

2. Analysis of the serial program

2.1. Coarse grain analysis

At first, we generate a call graph(Figure 3) by using `Google perfools`, a open source performance profiler, to have a glance though it. Every square represents a function, and the bigger square is, the more time corresponding function cost.

Obviously, the hot spot is something about `MKL`. After googling and searching Intel document we know that MKL provides `BLAS routinues`, which includes a serial function named `cblas_?gemm` to compute a matrix-matrix product with general matrices.

But giving that MKL function is well-optimized, we search for all position where `cblas_*gemm` is called. Results show the usage of `cblas_*gemm` appear in file `dnn_func.cpp`, more specifically, in three functions:

- `extern "C" int dnnForward(NodeArg &nodeArg)`
- `extern "C" int dnnBackward(NodeArg &nodeArg)`
- `extern "C" int dnnUpdate(NodeArg &nodeArg)`

They call `MKL` function `cblas_sgemm` many times by `for loop` and cost almost 90% of all CPU time. So we guess that those function is what we may optimize, aka, hotspots. The report(see Figure 4.) showed by`Intel VTune`, another profiler, proves our guess.

According to a skim through the source code, we could establish a clear structure about this program. To simplify code, original program could be rewritten in pseudocode:

```
1. GetInitFileConfig(cpuArg)
2. While FetchOneChunk(cpuArg, onChunk) do:
3.       While FetchOneBunch(oneChunk, nodeArg) do:
```

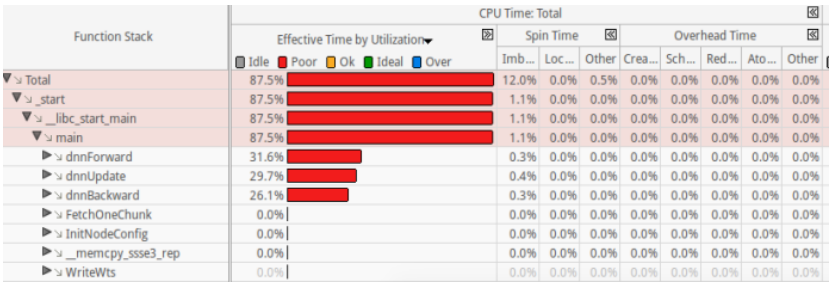

Figure 4. Intel VTune top-down tree

```
4.              dnnForward(nodeArg)
5.              dnnBackward(nodeArg)
6.              dnnUpate(nodeArg)
7. WriteWts(nodeArg, cpuArg)
8. UninitProgramConfig(cpuArg)
```

There are two nested loop before `dnn*()` series, and in each of those processing function many matrix-matrix product are executed. Whether those hotspots could be parallelized or not depends on data scale, dependency and so on. Before we discuss some methods and weighed their pros and cons the implementation of `DNN` should be most carefully checked.

2.2. Fine grain analysis

2.2.1. Matrix size

All `cblas_sgemm` is called like this:

```
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,\
      numN, numA[i], numA[i-1], \
      one, d_Y[i-1], numA[i-1], d_W[i], numA[i], one, d_Y[i], numA[i]);
```

The arguments `numN`, `numA[i]`, `numA[i-1]` indicating the size of the matrices:

- `d_Y[i-1]` is a `numN` row by `numA[i]` column matrix;
- `d_W[i]` is a `numN` row by `numA[i-1]` column matrix;
- `d_Y[i]` is a `numA[i-1]` row by `numA[i]` column matrix.

As we known the bigger matrix size is, the higher degree of `MKL` parallelism is. But in the `DNN` program, the size of matrix is decided by `bunchSize`, a constant integer (≈1024), and element (≈1024) of `dnnLayerArr`, a constant integer array. The two integers are configured by specified file, and we are not allowed to modify it. For this reason there are no sufficiently large matrix to enable `auto offload model` to speed up `DNN`.[2]

2.2.2. Cycles index

In the `dnn*` series every loop call `cblas_sgemm` `numN`(≈7) times, which indicates the length of `dnnLayerArr`. It's regretful that the value cannot be modified by us. Giving the number of core(≈24 in CPU or ≈60 in MIC) and constant `numN`, it's not wise to parallelize those loops.

3. Parallelization design methods

3.1. Fine grain parallelism

In function `dnnForward`, it's easy to observe there is a `for loop` calling `cblas_sgemm`, which nearly cost all CPU time consumed by this function. So it's same with the `dnnBackward` and `dnnUpdate`. A rough thoughts occurred to us that we could parallelize this `for loop` using multi-threads. But data dependency in `for loop` and relatively small cycle index make it inefficient to parallelize. So we should consider generalizing the parallelization region to smaller scale. Taking the feature of `MIC` in to account we hope to execute highly parallel code and/or compute intensive code in the `MIC`. Focusing on `dnn*` series it's easy to find an abstract structure to generalize them:

```
  extern "C" int dnn***(NodeArg &nodeArg)
{
    /* Variables definition */
    float  *d_X = nodeArg.d_X;
      ...
    /* Preprcess function func1 */
    func1(...);

   /* a for loop where a function, a MKL and another function are invoked oderly */
   for (int i = num; ...) {
        func2(...);
        cblas_sgemm(CblasRowMajor, CblasNoTrans, ...);
        func3(...);
        }
    }
    return 0;
}
```

Arguments of `func1`, `func2`, `func3` is matrix or vector, which is easy to parallelized. Taking all factors into consider, we have two choose: a) use serial `MKL` then parallelize the whole `for loop`, b) use multi-thread `MKL` and parallelize `func1, 2, 3`. The two measures all support `MIC`, but we prefer the b) because we could benefit from the parallel optimization of `MKL` and cooperation among `MKL` and `MIC`.

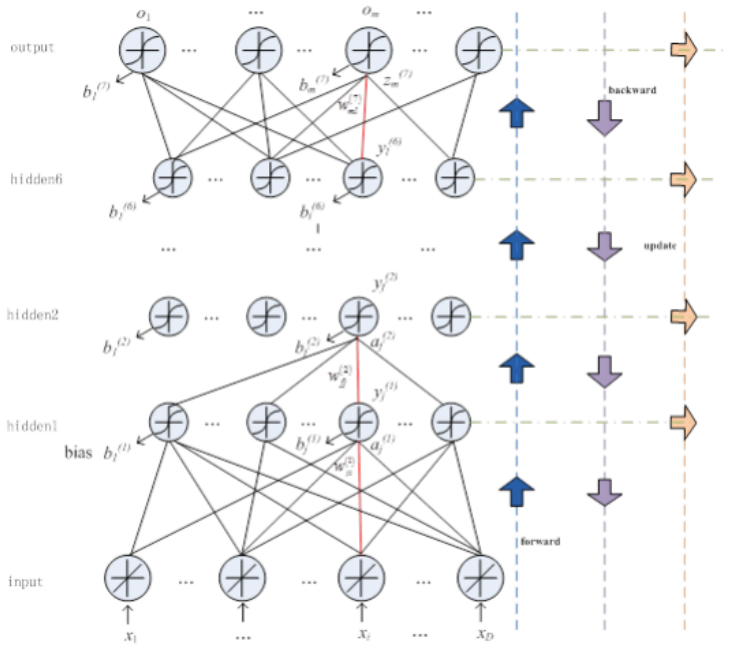

Figure 5. DNN structure

3.2. Coarse grain parallelism

To implenment coarse grain parallelism we hope that each thread/process finish large subcomponents. To achieve this goal `DNN` program should be divided into (mostly) independent and similar proportions, and every proportion should be as large as possible. But considering the structure of `DNN` the default ordering of `dnn*` series couldn't be changed, neither does the processing of file-reading. So in our opinion it's difficult to implement coarse grain parallelism without any change to `DNN` structure and its dataset.

4. Performance optimization methods

4.1. Serial `MKL` function with `OpenMP`

`MKL` function could decide whether to be threaded after a runtime check[3]. To maximize the utilization it's better to execute multithreading application in `MIC`, but as we mentioned above there isn't large enough cycles index to improve performance.

4.2. Compiler assisted offload

To make the `DNN` program scalable we use `offload if` the decide whether to run in the `MIC` or not. After testing a 2048 * 2048 matrix or bigger is suitable to transmit to `MIC` for better performance.

4.3. Environmental variables settings

- `MKL_DYNAMIC=true`:
  This option may reduce possible oversubscription from MKL threading. This option leads to a dynamic reduction of number of OpenMP* threads based on analysis of system workload.

- `MKL_NUM_THREADS=224`:
  `MKL` is designed for high degree parallelization code, so we set the variables to enable `MKL` threads.[1] It's necessary because we choose sequential `for loop` to call `MKL` function.

- `MIC_USE_2MB_BUFFERS=100M`

- `MKL_MIC_ENABLE=FALSE`:
  2MB pages are also needed. the 2MB pages in compiler-assisted offload are used to improve data transfer performance[4], and they could be enabled using the `MIC_USE_2MB_BUFFERS` variable.

5. Testing process and results on the CPU+MIC platform

6. Limitation

References

[1] Konstantin Arturov. *Recommended Settings for Calling Intel MKL Routines from Multi-Threaded Applications*. Intel, https://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-routines-from-multi-threaded-applications.

[2] Noah Clemons. *Recommendations to Choose the Right MKL Usage Model for Xeon Phi*. Intel, https://software.intel.com/en-us/articles/recommendations-to-choose-the-right-mkl-usage-model-for-xeon-phi. Mar. 2013.

[3] Intel. *Parallelism in the Intel® Math Kernel Library*. https://software.intel.com/en-us/articles/parallelism-in-the-intel-math-kernel-library.

[4] Zhang Z. *Performance Tips of Using Intel® MKL on Intel® Xeon Phi™ Coprocessor*. Intel, https://software.intel.com/en-us/articles/performance-tips-of-using-intel-mkl-on-intel-xeon-phi-coprocessor.