

Optimization of a DNN program on the CPU+MIC

University of Electronic Science 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 limits.

1. Introduction

There is a program based on a standalone hybrid CPU+MIC platform called DNN(**d**eep **n**eural **n**etwork) 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:
		Memory: 16G x8, DDR4, 2133Mhz	mic1,
		Hard disk: 1T SATA x 1	mic2,
		Accelerator card: Intel XEON PHI-31S1P (57 cores, 1.1GHz, 1003GFlops, 8GB GDDR5 Memory)	mic3, mic4
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

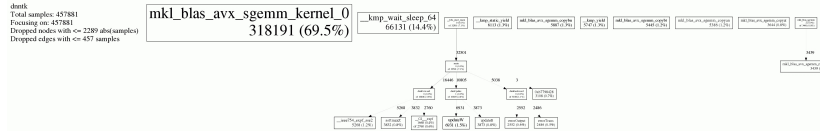


Figure 3. Google Perfools results

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 routines**, which includes a serial function named **cblas_*sgemm** to compute a matrix-matrix product with general matrices.

But giving that MKL function is well-optimized, we search for all position where **cblas_*sgemm** is called. Results show the usage of **cblas_*sgemm** 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:`

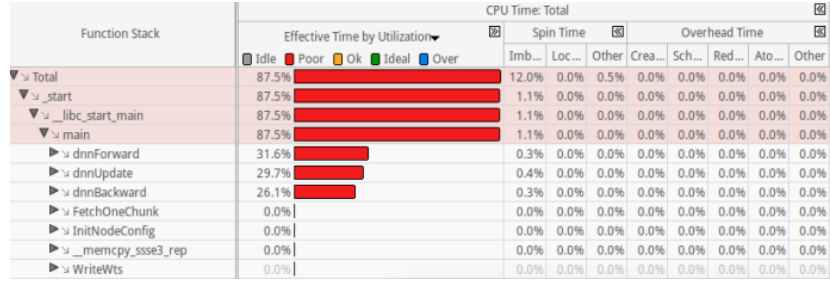


Figure 4. Intel VTune top-down tree

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.[1]

2.2.2. Cycles index

In the `dnn*` series every loop call `cblas_sgemv 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_sgemv`, which nearly cost all CPU time consumed by this function.

3.2. Coarse grain parallelism

To implement 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.

4. Performance optimization methods

4.1. Serial MKL function with OpenMP

MKL function could decide whether to be threaded after a runtime check[2]. To maximize the degree of parallelisation we use as mentioned above, there isn't large enough cycles index to improve performance.

4.2. MPI method

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

References

- [1] Noah Clemons. *Intel MKL Resource*. Intel, <https://software.intel.com/en-us/articles/recommendations-to-choose-the-right-mkl-usage-model-for-xeon-phi>. Mar. 2013.
- [2] Intel. *Parallelism in MKL*. <https://software.intel.com/en-us/articles/parallelism-in-the-intel-math-kernel-library>.