Optimization of a DNN Program on the CPU+MIC Platform

Competition proposal of ASC

2016-02-25

Liwei Cui
University of Electronic Secience and Technology of China
2014060101003@std.uestc.edu.cn

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.

Contents

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

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 on the CPU+MIC hybrid cluster. Performance analysis in this proposal is based on the results of this test.

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

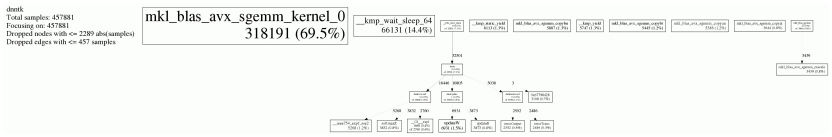

Figure 3. Google Perfools results

Function Stack	CPU Time: Total									
	Effective Time by Utilization		Spin Time			Overhead Time				
	Idle Poor Ok Ideal Over		Imb...	Loc...	Other	Crea...	Sch...	Red...	Ato...	Other
Total	87.5%		12.0%	0.0%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%
_start	87.5%		1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
__libc_start_main	87.5%		1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
main	87.5%		1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnForward	31.6%		0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnUpdate	29.7%		0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dnnBackward	26.1%		0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
FetchOneChunk	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
InitNodeConfig	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
__memcpy_ssse3_rep	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
WriteWts	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Figure 4. Intel VTune top-down tree

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:

```
GetInitFileConfig(cpuArg)
While FetchOneChunk(cpuArg, onChunk) do:
        While FetchOneBunch(oneChunk, nodeArg) do:
             dnnForward(nodeArg)
             dnnBackward(nodeArg)
             dnnUpate(nodeArg)
 WriteWts(nodeArg, cpuArg)
 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 on CPU or ≈60 on 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 loop invoking `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 on 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.

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.

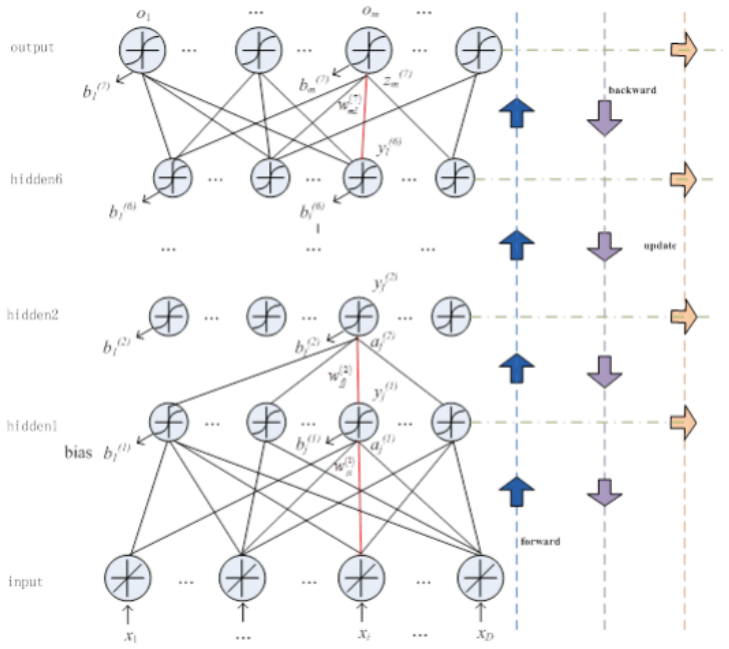

Figure 5. DNN structure

4. Performance Optimization Methods

4.1. Compiler Assisted Offload

4.1.1. Scope of Offloaded Sections

We have talked about the common structure of `DNN` series. In their `for` loop there are serial processing and `MKL` function appear alternately, and serial processing modifies data which is used as input in next `cblas_sgemm`function. Data change between `MIC` and `CPU` is slow because of `PCI-E` speeds so the best known methods[3] is to make the whole section of code an offload unit, which could reduce most of data transfer. In addition, because we nearly make the three `dnn` functions offload units, data transfer between them is less. SO, all we need do is to transfer input data in the start of `for loop` from `CPU` to `MIC` and transfer output after loop to `CPU`. `dnn` Function is like this (issues about array transfer discussed below):

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

  #pragma offload target(mic)\
  in(pd_Y:length(0) REUSE)\
  in(pd_W:length(0) REUSE)\
  ...
  {
    /* 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;
}
```

4.1.2. Dimensionality Reduction

As we know, MIC disallow specifying array of pointer, aka. multi-dimension array, to be use in `in` or `out` clauses. To cap it all element in those arrays used in DNN aren't stored consecutively in memory and have different size. So it's necessary for us to calculate array size to allocate one-dimension array. Code like this:

```
W_len = 0;
/* calculate element size */
for (int i = 1; i < dnnLayerNum; i++) {
    W_pos[i - 1] = W_len;
    W_len += /* i-st element size */;
}
/* allocate memory */
pd_W = (float *)mkl_malloc(W_len * sizeof(float), 64);
/* build two-dimension array */
for (int i = 0; i < dnnLayerNum - 1; i++) {
    d_W[i] = pd_W + W_pos[i];
}
```

4.2. OpenMP Optimization

Though MKL function could decide whether to be threaded after a runtime check[1, 4], other serial functions still run slowly in it. To maximize the utilization it's better to have more multithreading processing[3]. In pseudocode we discuss in 4.1.1, `fun1`, `fun2` and `fun3` are bottleneck. Those function all take one or more matrices as arguments, access and modify them then return nothing. Without data dependency it is easy to parallelize them using OpenMp:

```
extern "C" int errorOutput(float *E, float *Z, ...)
{
   float tmp;
   int idx, j;
  //OpenMp block
  #pragma omp parallel for private(tmp, idx, j)
   for(int i=0; i<row; i++)
   {
        for(j=0; j<col; j++)
        {
             idx = i*col + j;
             tmp = (j == T[i]) ? 1.0f:0.0f;
             E[idx] = Z[idx] - tmp;
        }
   }
   return 0;
```

```
}
```

4.3. Alignment

Alignment is important for MKL function to optimize program further and for compiler to assist vectorization. For example, allocating memory for matrices aligned on 64-byte boundary(or more) allows cblas_sgemm to have better performance. Using mkl_alloc is easy to allocate or deallocate an aligned memory buffer:

```
float *d_X = (float *)mkl_malloc(N * sizeof(float), 2048);
...
mkl_free(d_x);
```

Data transfer rate between CPU and MIC is slow, so align CPU data on a 64B boundary or higher is necessary for improve data transfer rate[3]. Align at 2MB for maximum transfer rate, so we use align modifier to improve it and enable vectorization of code on MIC:

```
#pragma offload target(mic) in(pd_X: length(X_len) align(2048))
...
```

4.4. Environmental Variables Settings

4.4.1. MKL_DYNAMIC=true:

This variable leads to a dynamic reduction of number of OpenMPI threads based on analysis of system workload, so it may reduce possible oversubscription from MKL threading.

4.4.2. MKL_NUM_THREADS=57 OMP_NUM_THREADS=57:

Those variables decide a maximum threads allowed to create. Since parallelization regions of this program run in the MIC which has 57 cores in test platform, we set those variables equal to 57.

4.4.3. MIC_USE_2MB_BUFFERS=64K

2MB pages are needed to improve transfer rate between MIC and CPU.[5]

5. Testing Process and Results on the CPU+MIC Platform

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] Kevin Davis. *Effective Use of the Intel Compiler's Offload Features*. Intel, https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features.

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

[5] 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.