Competition Proposal

1. The super computing introduction

1.1 The super computing platform of our school

1.2 The supercomputing education and achievement of our school

1.3 Event organization

2. Technical Proposal

2.1 The system platform

2.1.1 The platform configuration

Parts(one)	Power Consumption	Type
CPU	115 W	intel Xeon E5-2670 12cores x 2
Memory Chips	7.5 W	16GB x 8, DDR4, 2133MHz
HCA	9 W	InfinibandMellanox ConnectX®-3 HCA
The Hard Disk	10 W	300G SAS x 1
Accelerator Card	225 w	NVIDIA Tesla K20 x 2

2.2 HPCG Test

2.2.1HPCG Introduction:

The High Performance Conjugate Gradients(HPCG) Benchmark is an effort to create a new metric for ranking HPC systems. HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

HPCG is a complete, stand-alone code that measures the performance of basic operations in a unified code:

* Sparse matrix-vector multiplication.
* Sparse triangular solve.
* Vector updates.
* Global dot products.
* Local symmetric Gauss-Seidel smoother.

* Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids.
* Reference implementation is written in C++ with MPI and OpenMP support.

2.2.2 The HPCG Test Method

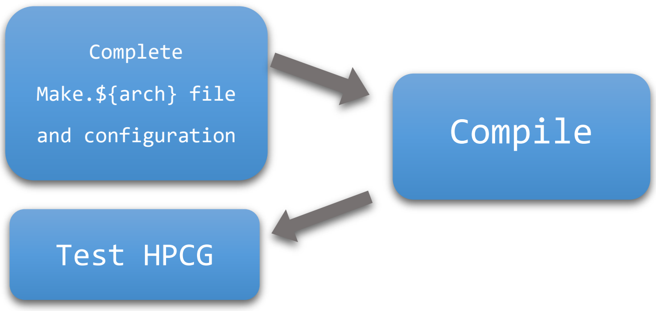


Fig.2.1

2.2.3 The Test Steps:

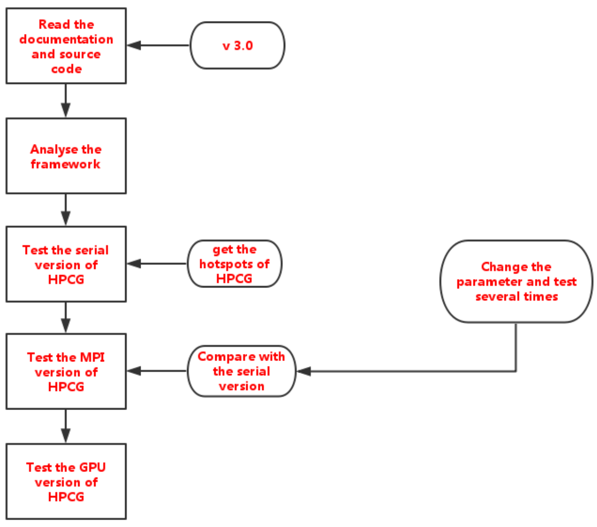
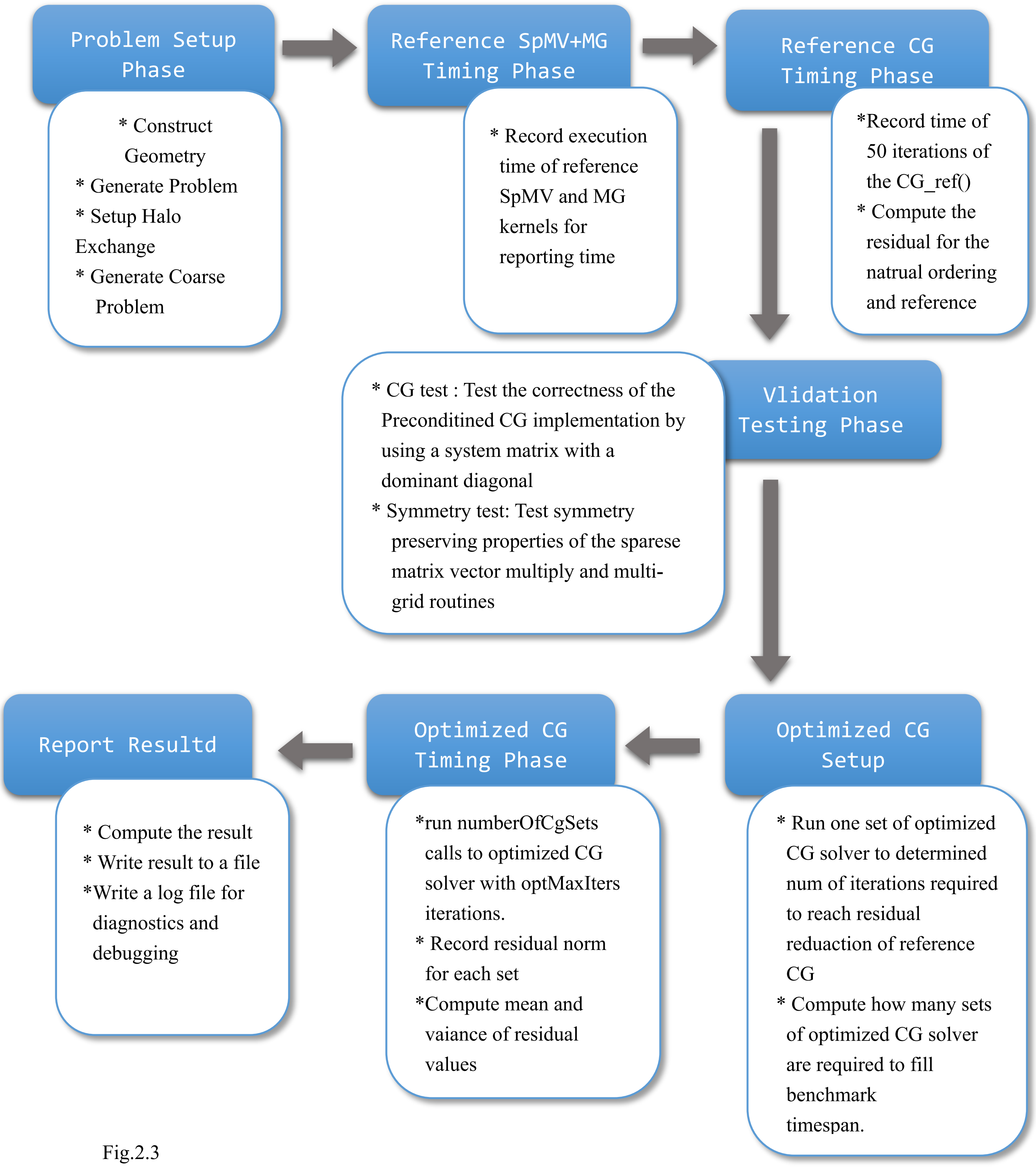

Read the documentation and source code
v 3.0
Analyse the framework
Test the serial version of HPCG
get the hotspots of HPCG
Change the parameter and test several times
Test the MPI version of HPCG
Compare with the serial version
Test the GPU version of HPCG

Fig.2.2

2.2.4 The Code Analysis:

We read the `main.cpp (v3.0)` at first. In main.cpp file, we can find the program process:



Problem Setup Phase
* Construct Geometry
* Generate Problem
* Setup Halo Exchange
* Generate Coarse Problem
Reference SpMV+MG Timing Phase
* Record execution time of reference SpMV and MG kernels for reporting time
Reference CG Timing Phase
*Record time of 50 iterations of the CG_ref()
* Compute the residual for the natrual ordering and reference
Vlidation Testing Phase
* CG test : Test the correctness of the Preconditined CG implementation by using a system matrix with a dominant diagonal
* Symmetry test: Test symmetry preserving properties of the sparese matrix vector multiply and multi-grid routines
Optimized CG Setup
* Run one set of optimized CG solver to determined num of iterations required to reach residual reduaction of reference CG
* Compute how many sets of optimized CG solver are required to fill benchmark timespan.
Optimized CG Timing Phase
*run numberOfCgSets calls to optimized CG solver with optMaxIters iterations.
* Record residual norm for each set
*Compute mean and vaiance of residual values
Report Resultd
* Compute the result
* Write result to a file
*Write a log file for diagnostics and debugging

Fig.2.3

2.2.5 HPCG Principle

The HPCG benchmark generates a synthetic discretized three-dimensional partial differential equation model problem, and computes preconditioned conjugate gradient iterations for the resulting sparse linear system. The model problem can be interpreted as a single degree of freedom heat diffusion model with zero Dirichlet boundary conditions.

The setup phase constructs a logically global, physically distributed sparse linear system using a 27-point stencil at each grid point in the 3D domain such that the equation at point (i, j, k) depends the values at its location and its 26 surrounding neighbors. The matrix is constructed to be weakly diagonally dominant for interior points of the global domain, and strongly diagonally dominant for boundary points, reflecting a synthetic conservation principle for the interior points and the impact of zero Dirichlet boundary values on the boundary equations.

The PCG algorithm is shown as Fig.2.5

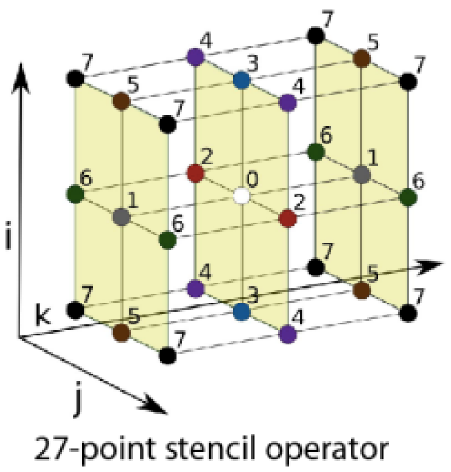
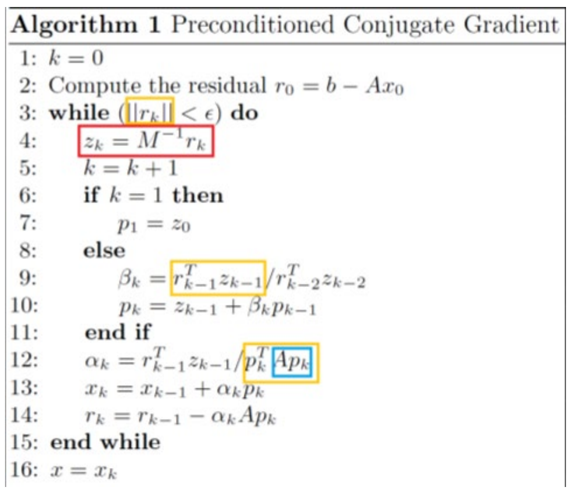


Fig.2.4



```
Algorithm 1 Preconditioned Conjugate Gradient
1: $k = 0$
2: Compute the residual $r_0 = b - Ax_0$
3: while ($\boxed{\|r_k\|} < \epsilon$) do
4:     $\boxed{z_k = M^{-1}r_k}$
5:     $k = k + 1$
6:     if $k = 1$ then
7:         $p_1 = z_0$
8:     else
9:         $\beta_k = \boxed{r_{k-1}^T z_{k-1}} / r_{k-2}^T z_{k-2}$
10:        $p_k = z_{k-1} + \beta_k p_{k-1}$
11:    end if
12:    $\alpha_k = r_{k-1}^T z_{k-1} / \boxed{p_k^T \boxed{Ap_k}}$
13:    $x_k = x_{k-1} + \alpha_k p_k$
14:    $r_k = r_{k-1} - \alpha_k Ap_k$
15: end while
16: $x = x_k$
```

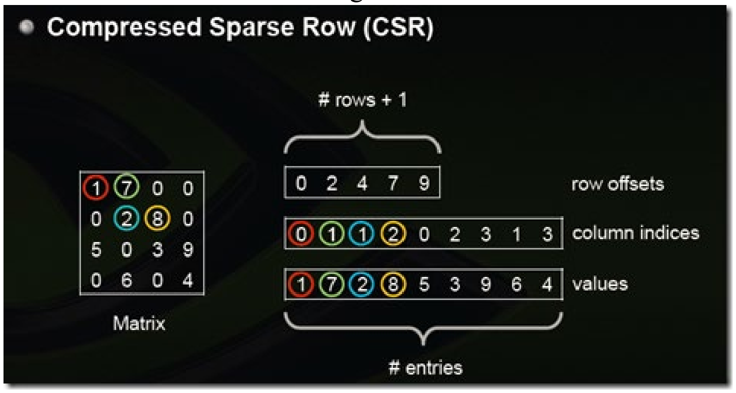
Fig.2.5

Shown in the Fig.2.5, the red color means **Symmetric-Gauss-Seidel(SYMGS)**, the yellow color represents the **Dot Product**, which needs communications among the neighbors--*MPI_Allreduce()*, the blue color shows the **Sparse Matrix Vector Multiply(SPMV)** calculations.

2.2.6 Optimization Methods of HPCG

1) Storage Format of Sparse Matrix

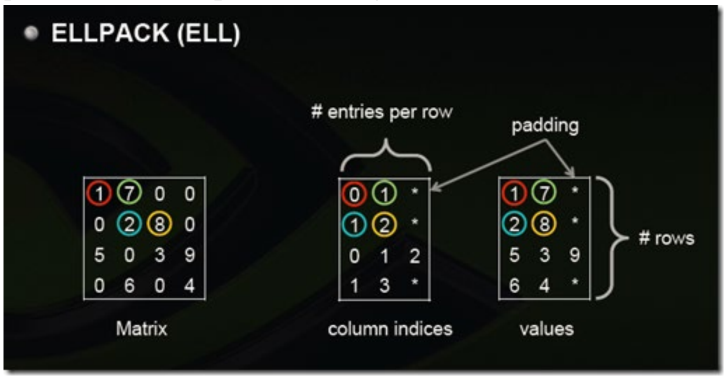
The storage format of sparse matrix plays a key role in SpMV and SymGS performance. In the HPCG reference implementation, a standard **Compressed Sparse Row** (CSR) format is used. The CSR format is shown as Fig.x.



The CSR is a common format to store sparse matrix, which consists of three elements--**values, column indices** and **row offsets.** The row offsets demonstrate the offset of one row's first element.

Fig.2.6

Generally speaking, the **ELLPACK** format performs much better when doing **sparse matrix-vector products** calculations, so it can largely promote the performance of a **sparse linear system**.



Stored in ELLPACK format, a sparse matrix is represented by two matrices with the same number of rows. The first matrix stores the **column indices** and the second one stores **values**. Due to the following two matrices have the same rows as the first one, There is no need to

Fig.2.7

store the row numbers additionally. In the ELLPACK method, if there is no element in one position. There will be a star * occupied.

2) The optimization of Gauss-Siedel algorithm

In numerical linear algebra, the Gauss-Seidel method is an iterative method used to solve a **linear system of equations**.

For a square system of n linear equations $A\boldsymbol{x} = \boldsymbol{b}$, we can solve the expression $\mathbf{x}$ iteratively with the formula $\boldsymbol{D}\boldsymbol{x}^{k+1} = \boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{k+1} - \boldsymbol{U}\boldsymbol{x}^{k}$, where $\mathbf{x}^{\mathbf{k}}$ is the kth approximation of $\mathbf{x}$, $\boldsymbol{x}^{k+1}$ is the next of $\mathbf{x}$, and the matrix A can be decomposed into a diagonal component $\boldsymbol{D}$, a lower triangular component L, and a strictly upper triangular component $\boldsymbol{U}$: $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$.

The Gauss-Seidel method now solves the left hand side of this expression for x, using previous value for x on the right hand side, which can be written as:

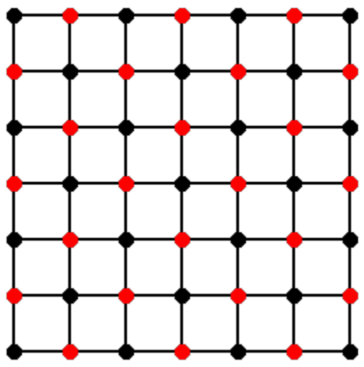
$\boldsymbol{X}^{(k+1)} = (\boldsymbol{D} + \boldsymbol{L})^{-1}(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)})$.

The procedure can be stopped when the residual is small enough.

We have many choices for the order of iterations. In the original code, it is implemented in *natural ordering*—it is the typical ordering that would result in a *for* loop. We can loop successively from the bottom line to the top line.

There is another optimized order—**Red-Black ordering.** Specifically, the gridpoint (i,j) is colored

Red-Black Ordering of Grid Points



Black points have only Red neighbors

Red points have only Black neighbors

red if $i+j$ is even, and black if $i+j$ is odd. During the Gauss-Seidel update, all red points are updated before the black points. The update of a red grid point only requires the information from the black grid points, and vice versa.

The Red-Black ordering is much easier to be implemented in parallelism.

Fig.2.7

2.2.7 The Test Results

2.2.7.1 HPCG results submission

HPCG	
HPCG version	3.0
System Spec	
Compute Nodes	1
OS	CentOS 6.7
MPI	impi 4.1.1
Compiler	icpc 2013
Compiler Flags	-O3 –xHost –Wall
Environment Variables:	
Turbo (ON/ OFF)	ON

	Results
Number of nodes	1
Number of cores	16
HPCG (base or optimized)	7.01
HPCG result	PASSED

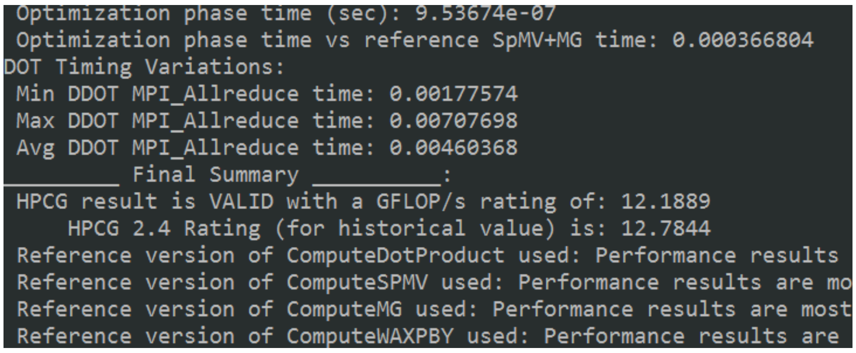
nx,ny,nz	CPU(ICPC+ AVX)	GPU Version	nx,ny,nz	CPU(ICPC+ AVX)	GPU Version
24	1.705	1.713	112	1.069	17.780
32	1.786	6.324	120	1.093	19.947
40	1.502	6.232	128	1.039	22.651
48	1.314	10.107	136	1.054	19.072
56	1.211	10.514	144	1.038	19.111
64	1.133	15.437	152	1.041	20.544
72	1.155	16.124	160	1.046	17.468
80	1.134	14.409	168	1.037	20.141
88	1.100	17.700	176	1.031	19.742
96	1.131	16.338	184	1.023	20.021
104	1.086	18.677	192	1.016	18.076

2.2.7.2 Tests with different *nx,ny,nz*

Table 2.1 Comparison between ICPC+AVX and CUDA version (in one node)

Distributed Processes	GFLPOS/s	Distributed Processes	GFLPOS/s
1	0.823	16	6.109
2	1.728	20	5.695
4	3.117	24	5.849
8	4.866	28	5.856
12	4.594	32	5.769

Table 2.2 Test results of ICPC_MPI version (nx = 64 ny=64 nz=64, CPU: 8 cores x 2, one node)



```
 Optimization phase time (sec): 9.53674e-07
 Optimization phase time vs reference SpMV+MG time: 0.000366804
DOT Timing Variations:
 Min DDOT MPI_Allreduce time: 0.00177574
 Max DDOT MPI_Allreduce time: 0.00707698
 Avg DDOT MPI_Allreduce time: 0.00460368
_________ Final Summary __________:
 HPCG result is VALID with a GFLOP/s rating of: 12.1889
     HPCG 2.4 Rating (for historical value) is: 12.7844
 Reference version of ComputeDotProduct used: Performance results
 Reference version of ComputeSPMV used: Performance results are mo
 Reference version of ComputeMG used: Performance results are most
 Reference version of ComputeWAXPBY used: Performance results are
```

The best result we got on our own platform was 12.1889GFlops(ICPC_MPI, n=16,nx=ny=nz=24),

reaching 4.76% of the peak value.

2.3 Masnum_wave

2.3.1 Background of the software

The third-generation Wave Model LAGFD-WAM was proposed early in 1990s in LAGFD (Laboratory of Geophysical Fluid Dynamics), FIO (First Institute of Oceanography) of SOA (State Oceanic Administration), China. It has marked originality in
1) The energy and action energy spectrum balance equations,
2) The source functions of dissipation and bottom friction,
3) The wave-current interaction source function and
4) The computational scheme.
The model has been compared with WAM model in typical wind fields and gave concerted results in general sea state and good improvement in high sea state. Also the model has been used in China seas for forecast and hindcast practices and got results consistent with filed measured data.
The structure of MASNUM program is shown in Fig.

```
Main program_____precom_____settopog
             |       |_setwave
             |       |_nlweight_____jafu
             |_readwi______setspec
                     |_propagat_____inter
                     |_implsch______mean2
                     |……glb/setspec
                     |_mean1
                     |_output…
```

Fig.2.8

time_mod	Used to deal with the time
netcdf_mod	Used to input/output data through NetCDF format.
wamvar_mod	Include all the global variables used in this model
wamfio_mod	Subroutines for I/O data or model results
wamcpl_mod	Subroutines for coupling w/current model.
wamcor_mod	The core subroutines of this model.
wamnst_mod	The subroutines for model nesting.

The calculation procedure can be divided into 2 steps:

Initialization

a) Read parameters from the file **ctlparams**
b) Call **wamvar_mod_init** to initialize the global variables and allocate the memory for global arrays
c) Call **init_wamcpl** to initialize the control parameters of the model.

d) Call **setwave** to set the discrete wave number space
e) Call **nlweight** to calculate the non-linear interaction between wave and wave.

Read in the wind data and process

a) Call **get_wind** to read in the data of wind site and do the bilinear interpolation.
b) Call **propagate** to solve the equation of the wave propagation.
c) Call **implsch** to solve the local change caused by the source function.
d) Call **mean1** to solve the characteristic quantity of the wave.
e) Call **output** to output the netcdf file according to the type.

The flow chart is shown in Fig.

Test Enviornment

The enviornment of MASNUM test is listed below:

- Operation System:Red Hat Linux 6.4
- Compiler: mpifort
- MPI Distribution:Intel mpi 12.1.4 openmip 1.4.4
- Essential Package: netcdf-3.6.2

3) Test Method

Take exp1 as example:

Change directory to masnum_wave/source/bin

Change parameters in MAKEFILE

```
- F77 = mpifort
- FFLAGS      = -O3 -no-prec-div
```

Running commands:

```
$ make
```

Compiled successfilly when no error occurs and "masnum_wave.mpi" is generated

Go to masnum_wave/exp/exp1

Change parameters in exp_run.csh

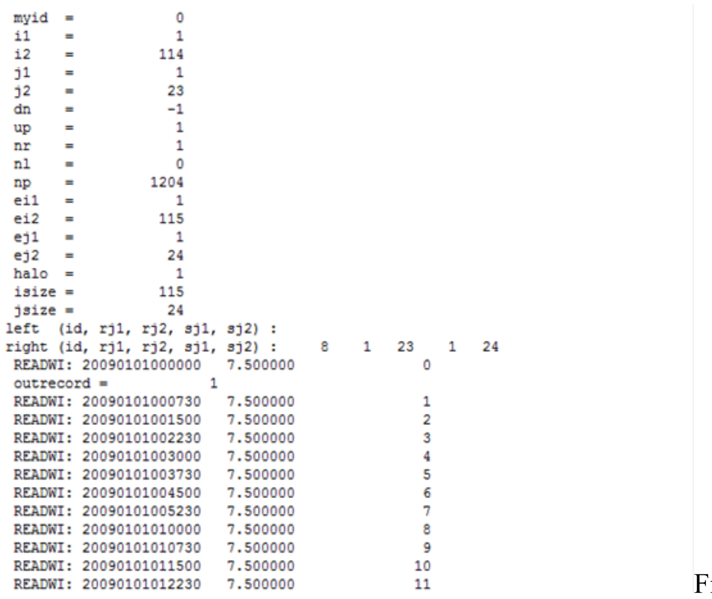
```
- set needmake = "YES"
- set nproc = 24
- set masnum_home = $HOME/***/masnum_wave
```

And change :

```
mpirun -np $nproc ./masnum.wam.mpi > out.qrunout
```

to:

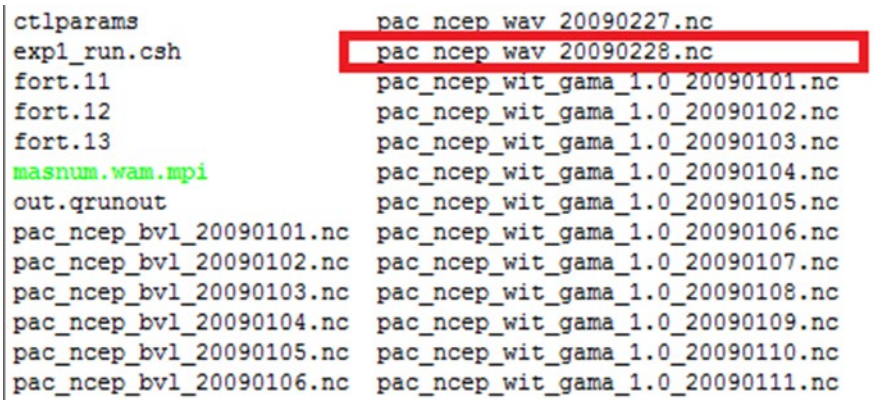
```
yhrun -p klcuda -N 6 -n 48    ./masnum.wam.mpi > out.qrunout &
```



```
 myid  =           0
 i1    =           1
 i2    =         114
 j1    =           1
 j2    =          23
 dn    =          -1
 up    =           1
 nr    =           1
 nl    =           0
 np    =        1204
 ei1   =           1
 ei2   =         115
 ej1   =           1
 ej2   =          24
 halo  =           1
 isize =         115
 jsize =          24
left  (id, rj1, rj2, sj1, sj2) :
right (id, rj1, rj2, sj1, sj2) :     8    1   23    1   24
 READWI: 20090101000000   7.500000            0
 outrecord =            1
 READWI: 20090101000730   7.500000            1
 READWI: 20090101001500   7.500000            2
 READWI: 20090101002230   7.500000            3
 READWI: 20090101003000   7.500000            4
 READWI: 20090101003730   7.500000            5
 READWI: 20090101004500   7.500000            6
 READWI: 20090101005230   7.500000            7
 READWI: 20090101010000   7.500000            8
 READWI: 20090101010730   7.500000            9
 READWI: 20090101011500   7.500000           10
 READWI: 20090101012230   7.500000           11
```

Fig.2.9 The out.qrunout file

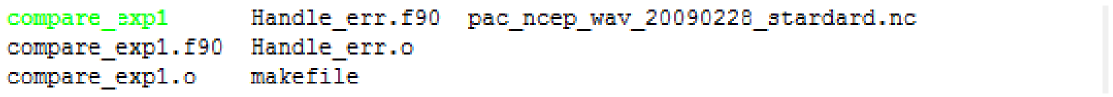
1) Wait until the **pac_ncep_wav_20090228.nc** is generated. Change the directory to **masnum_wave/balidation/exp1**.



```
ctlparams                  pac_ncep_wav_20090227.nc
exp1_run.csh               pac_ncep_wav_20090228.nc
fort.11                    pac_ncep_wit_gama_1.0_20090101.nc
fort.12                    pac_ncep_wit_gama_1.0_20090102.nc
fort.13                    pac_ncep_wit_gama_1.0_20090103.nc
masnum.wam.mpi             pac_ncep_wit_gama_1.0_20090104.nc
out.qrunout                pac_ncep_wit_gama_1.0_20090105.nc
pac_ncep_bvl_20090101.nc   pac_ncep_wit_gama_1.0_20090106.nc
pac_ncep_bvl_20090102.nc   pac_ncep_wit_gama_1.0_20090107.nc
pac_ncep_bvl_20090103.nc   pac_ncep_wit_gama_1.0_20090108.nc
pac_ncep_bvl_20090104.nc   pac_ncep_wit_gama_1.0_20090109.nc
pac_ncep_bvl_20090105.nc   pac_ncep_wit_gama_1.0_20090110.nc
pac_ncep_bvl_20090106.nc   pac_ncep_wit_gama_1.0_20090111.nc
```

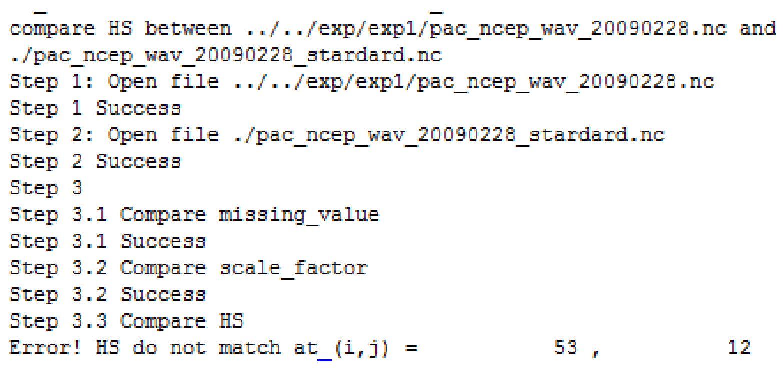
Fig.2.10

2) Change the parameters in makefile
3) **Make**, if "**compare_exp1**" is generated, the compiling procedure is successful.



```
compare_exp1       Handle_err.f90   pac_ncep_wav_20090228_stardard.nc
compare_exp1.f90   Handle_err.o
compare_exp1.o     makefile
```

4) Run the command "**./compare_exp1**". If there is "Compare Success" on the screen, the validation test is passed.



```
compare HS between ../../exp/exp1/pac_ncep_wav_20090228.nc and
./pac_ncep_wav_20090228_stardard.nc
Step 1: Open file ../../exp/exp1/pac_ncep_wav_20090228.nc
Step 1 Success
Step 2: Open file ./pac_ncep_wav_20090228_stardard.nc
Step 2 Success
Step 3
Step 3.1 Compare missing_value
Step 3.1 Success
Step 3.2 Compare scale_factor
Step 3.2 Success
Step 3.3 Compare HS
Error! HS do not match at (i,j) =           53 ,          12
```

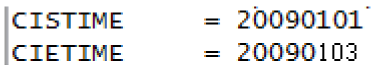
Fig.2.11

Optimization methods

1) Multi-node parallelization

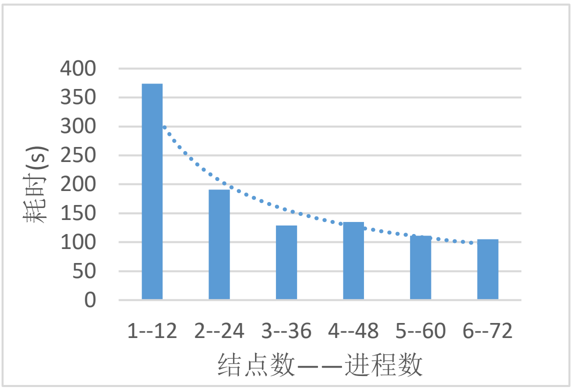
In our code, after calculating the wind speed arrays, **MPI_Bcast()** is called to broadcast the wind speed arrays to all the processes. We divide the **e** array into blocks according to the process number and then call **propagat** function. Then each process broadcast the **e** array to make it updated. The procedure is similar when calling **implsch** function.

To reach better performance, we added more computing nodes and have a test. For the reason that it takes too much time to complete the whole task, we changed the parameters in **ctlparams** file.



```
CISTIME       = 20090101
CIETIME       = 20090103
```

The result is shown in Fig.



From the result we can get the info that **The more nodes used, the less time is consumed. However, it is not a linear relationship.**

Fig.2.12

2.4 Optimization of The DNN Program on The CPU+MIC Platform

2.4.1 Introduction to the platform

2.4.1.1 Introduction to the software and the hardware

Operating system：Red Hat 4.8.3-7

Compiler：Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.0 (gcc version 4.8.3 compatibility)

Configuration of the environment in detail:

Entry	Name	Configuration

Server	Inspur NF5280M4	CPU: Intel Xeon E5-2680v3 × 2, 2.5Ghz, 12 cores Memory: 16G × 8, DDR4, 2133Mhz Hard disk: 1T SATA × 1 Power consumption estimation: E5-2680v3 TDP 120W, memory 7.5W, hard disk 10W
Accelerator	XEON PHI-31S1P	Intel XEON PHI-31S1P (57 cores, 1.1GHz, 1003GFloops, 8GB GDDR5)

Table 2.3

2.4.2 Brief introduction to the DNN

Deep neural network (DNN) is a derivation of artificial neural network model. It owns more hidden layers compared with common artificial neural networks, which leads to its good performance on fitting arbitrary functions. DNN works similarly to human brains with a hierarchy of hidden levels. More hidden layers mean high level of abstraction, which plays an important role in extracting ideal features. Due to these reasons, DNN is widely applied in many research areas, such as image-recognition and speech-recognition.
In this project, the DNN model is shown as the Fig2.13.

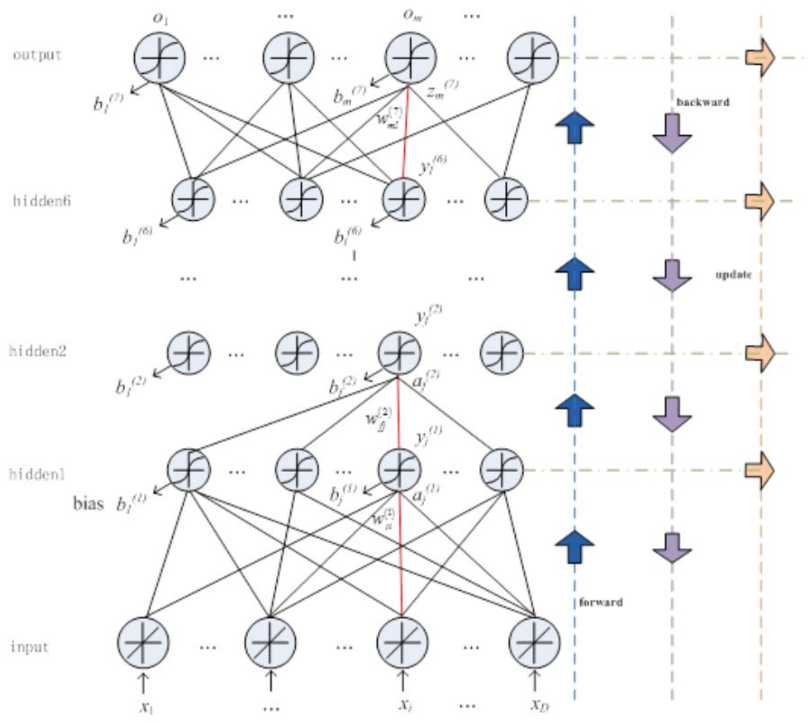


Fig.2.13
It has one input layer, six hidden layers and one output layer. For the **dimension** of the input feature is 39, and the **context** of it is 11, there are **429**(39*11) nodes in the input layer. Each hidden layer has 2048 nodes and the output layer has 8091 ones.

2.4.2.1 Neural Network Model

A neural network is a machine learning algorithm. The common pattern of a machine learning model is shown as Fig.

Train Model
Validate Model

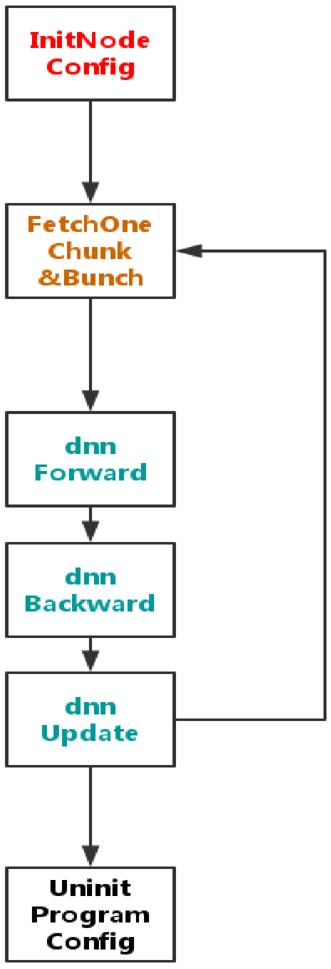
Fig.2.14

In the project, the cpp files in ***dnntk_src*** folder implements the "model training" component and the files in ***dnn_cvtk_src*** folder implements the "model validation" function.

The main parameters of DNN consist of ***weights*** and ***bias***. Before training, function *InitNodeConfig* initializes the weights and bias. During the training process, function *FetchOneChunk* reads the training set data from the disk to the memory. In each iteration, ***bunchsize*** of the samples are used to train the model.

dnnForward, dnnBackward and *dnnUpdate* form the main framework of the training process. They three separately implement the ***forward propagation***, ***backward propagation*** and ***parameter update*** stage of the algorithm.

The structure of a single neuron is shown as Fig2.15. ,which can be displayed by a formula: $t = f(\sum_{i=1}^{n} w_i x_i + b) = f(\mathbf{w}^{\mathrm{T}}\mathbf{x})$ (**w**—weights,**b**—bias,***f***—the activation function)



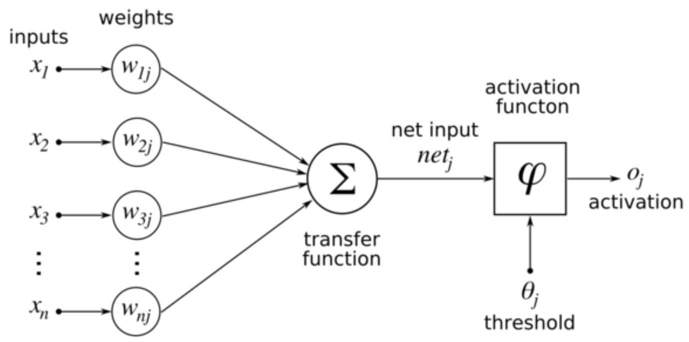


Fig.2.15

Fig.2.14

2.4.2.1.1 Forward Propagation

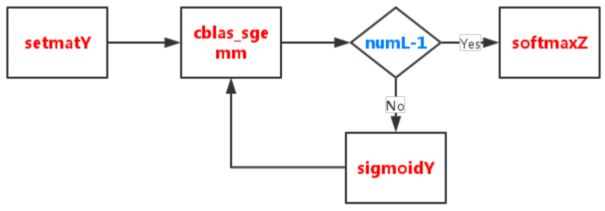


Fig.2.16

The input data is stored in variable ***nodeArg.d_X***, which is transformed into a **429*2048** matrix after processed by function ***setmatY***. Each row of the matrix is the ***bias vector*** of the next layer.
Due to the formula $t = f(\sum_{i=1}^{n} w_i x_i + b) = f(\mathbf{w}^{\mathrm{T}}\mathbf{x})$, the input data will do Cartesian product with the weights between the input layer and the first hidden layer, then add the bias to be the input value of the next layer. To implement this process, function ***cblas_sgemm*** is used. The prototype of *cblas_sgemm* is shown as below:

```
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);
```

The description of this function is: The ?gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

$$C := alpha * op(A) * op(B) + beta * C$$

In the project, ***alpha*** and ***beta*** are both 1.0, so what the function does is to calculate $\mathrm{d_y}$ according to the formula

$$d_Y := d_Y * d_W + d_Y$$

The result calculated by the cblas_sgemm will be processed by the activation function before being input to the next layer. In our project, the activation function is *sigmoidY*. The forward propagation procedures from the second hidden layer to the sixth one are similar. When **d_Y ($\mathbf{d_y}$)** is transferred from the last hidden layer to the output layer, function *sigmoidY* is replaced with *softmaxZ*.

The prototype of *softmaxZ* is shown as below:

$$extern\,"C"\;\; int\; softmaxZ(float * in_{vec}, float * out_{vec}, int\; row, int\; col)$$

Softmax function takes the form $f_j(z) = \frac{e^{x_j}}{\sum_k e^{x_k}}$, it takes a real-number vector as the input, and normalizes it in the exponential domain(to ensure that the total sum of the output values equals 1). For an input vector x , the output of the softmax function can give us the k probabilities of $p(y = j|x)$ for each value of $j = 1,2,3 \ldots \ldots k$

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)};\theta) \\ p(y^{(i)} = 2|x^{(i)};\theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)};\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

Fig.2.17

We can know from the fact that there are 8991 nodes in the output layer of the neural network, that the dnn model is a k-classification model where k=8991.

2.4.2.1.2 Backforward Propagation

The backward propagation is the central part of the DNN model. The most important purpose of backward propagation is to calculate the $\frac{\partial C}{\partial w}$, which is the partial derivative of the cost function(C) with respect to the weight(w). According to $\frac{\partial C}{\partial w}$, we can know the direction and magnitude of the Δw

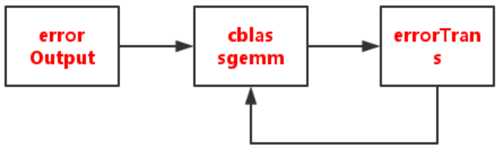


Fig.2.18

First, the d_E will be processed by function *errorOutput.*
The prototype of errorOutput is
*int errorOutput(float *E, float* Z, int *T, int row, int col)*

Float ***E** stores the error of the classification, float ***Z** is the classification result of the output layer, int ***T** stores the indexes of the target values of the samples used for training. The output of *softmaxZ* stores the probabilities of being classified into the **k*th* class** for a sample, which is between zero and one. After the process of *errorOutput*, $E_j = \begin{cases} Z_j - 1, \ j = T[i] \\ \quad Z_j, \ j \neq T[i] \end{cases}$.
After that, the error **E** starts to be transferred to the last hidden layer. *Cblas_sgemm* solves the equation $\mathrm{d_E}[n-2] = d_E[n-1] * d_w^T[n-1]$, which is equivalent to $d_E[n-2] * d_w[n-1] = d_E[n-1]$.

Let's take a neural network with only one hidden layer as an example to demonstrate the algorithm of the backward propagation. The network is shown in the Fig.

We have:

$$\mathrm{net}_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\mathrm{out}_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

The backward propagation aims to know the effect of a change in weights that affects the total error,

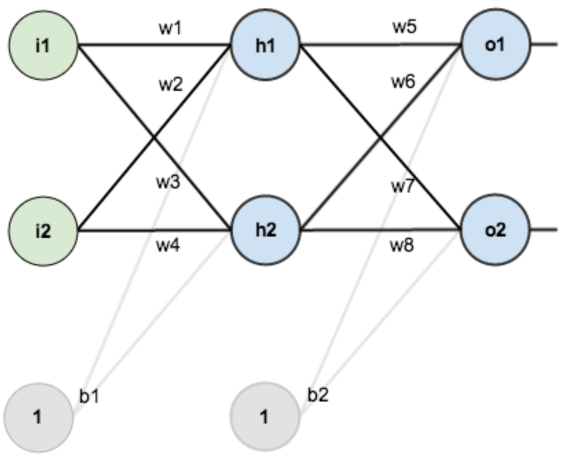


Fig.2.19

According to the chain rule, we can deduce a new equation $\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial out} * \frac{\partial out}{\partial net} * \frac{\partial net}{\partial w_i}$, in the program, $\frac{\partial E}{\partial out}$ is represented by d_E. The function sigmoid takes the form $S(t) = \frac{1}{1+e^{-t}}$, so $\frac{\partial S(t)}{\partial(t)} = s(t) * (1 - s(t))$, function errorTrans compute $\frac{\partial E}{\partial out} * \frac{\partial out}{\partial net}$ and d_E is updated to $\frac{\partial E}{\partial out} * \frac{\partial out}{\partial net}$.

2.4.2.1.3 Parameters updates

In this process, the weights and bias are updated.

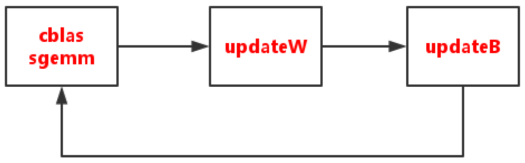


Fig.2.20

We can use a formula to describe the parameter-updates routine $d_{Wdta}[i] = alpha * d_Y^T[i-1] * d_E[i]$, it is also worth noting that the value of $\mathrm{d_E}$ has already been $\frac{\partial E}{\partial out} * \frac{\partial out}{\partial net}$, and from the Fig. we can know that d_Y is the result of $\frac{\partial net}{\partial w_i}$ now.

Finally the weights can be updated by the formula $\mathrm{w}^+ = \mathrm{w} + \mathrm{d_w}$ (because alpha is negative, so $\mathrm{d_w}$ is also negative)

The bias-updating procedure is a little different from that of weights. From the Fig. we know that each layer's bias value is fixed. So every time the bias will be updated by column, which is implemented in function update.

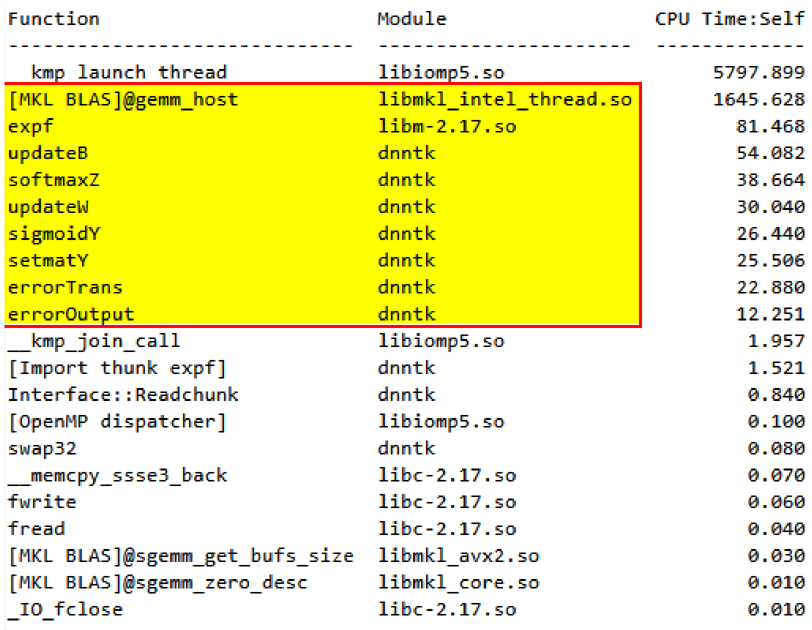
2.4.3 The optimization methods

Hotspots collection

First, we use Intel Vtune tools to collect and analyse the hotspots of the project.

```
amplxe-cl –collect hotspots –result-dir r0001hs
–target-pid XXX
```

```
amplxe-cl –report hotspots –result-dir r00001hs
-group-by function
```

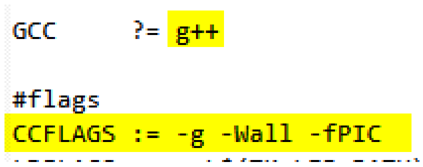


```
Function                        Module                 CPU Time:Self
------------------------------  ---------------------  -------------
__kmp_launch_thread             libiomp5.so                 5797.899
[MKL BLAS]@gemm_host            libmkl_intel_thread.so      1645.628
expf                            libm-2.17.so                  81.468
updateB                         dnntk                         54.082
softmaxZ                        dnntk                         38.664
updateW                         dnntk                         30.040
sigmoidY                        dnntk                         26.440
setmatY                         dnntk                         25.506
errorTrans                      dnntk                         22.880
errorOutput                     dnntk                         12.251
__kmp_join_call                 libiomp5.so                    1.957
[Import thunk expf]             dnntk                          1.521
Interface::Readchunk            dnntk                          0.840
[OpenMP dispatcher]             libiomp5.so                    0.100
swap32                          dnntk                          0.080
__memcpy_ssse3_back             libc-2.17.so                   0.070
fwrite                          libc-2.17.so                   0.060
fread                           libc-2.17.so                   0.040
[MKL BLAS]@sgemm_get_bufs_size  libmkl_avx2.so                 0.030
[MKL BLAS]@sgemm_zero_desc      libmkl_core.so                 0.010
_IO_fclose                      libc-2.17.so                   0.010
```

Fig.2.21

We can find from the results that most of the time is consumed by the functions marked by the yellow color. All of them are in ***dnn_kernel.cpp*** and ***dnn_func.cpp.*** So we regard them as main targets to reach the goal of optimization.

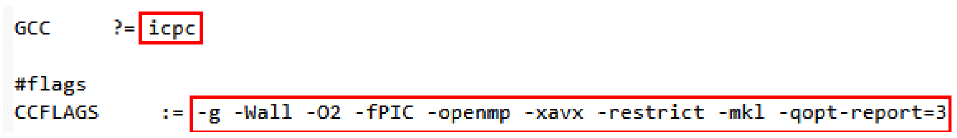
Changing the compiler and modify the compiler options



```
GCC      ?= g++

#flags
CCFLAGS := -g -Wall -fPIC
```

Fig.2.22

In the origin **Makefile**, *g++* compiler is used and few of the optimization option flags are added. To enhance the optimization level, we modified the Makefile like below.



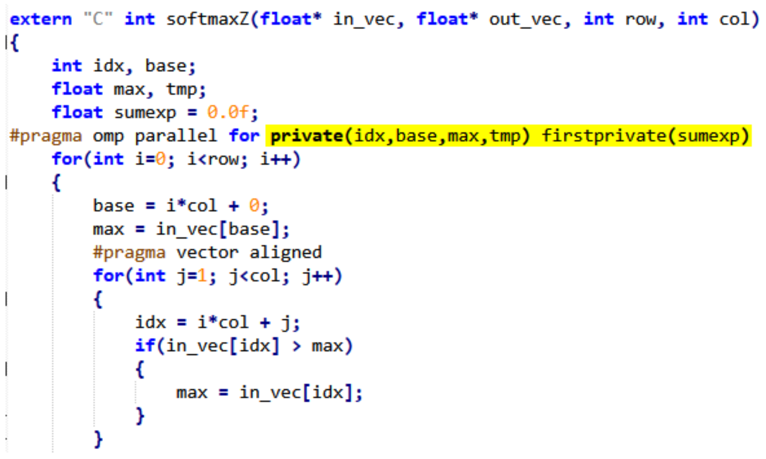
```
GCC       ?= icpc

#flags
CCFLAGS      := -g -Wall -O2 -fPIC -openmp -xavx -restrict -mkl -qopt-report=3
```

Fig.2.23

Option	Effect
-O2	With this option, the compiler performs some basic loop optimizations, inlining of intrinsic, Intra-file interprocedural optimization, and most common compiler optimization technologies.
-openmp	OpenMP support
-xHost	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.
-restrict	By qualifying a pointer with the restrict keyword, you assert that an object accessed by the pointer is only accessed by that pointer in the given scope, which can help the compiler produce more optimizing code.
-qopt-report=3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.

Parallelize the code

In **dnn_kernel.cpp**, all of the functions are ***single-threaded***, we use OpenMP directives to make the code parallelized.

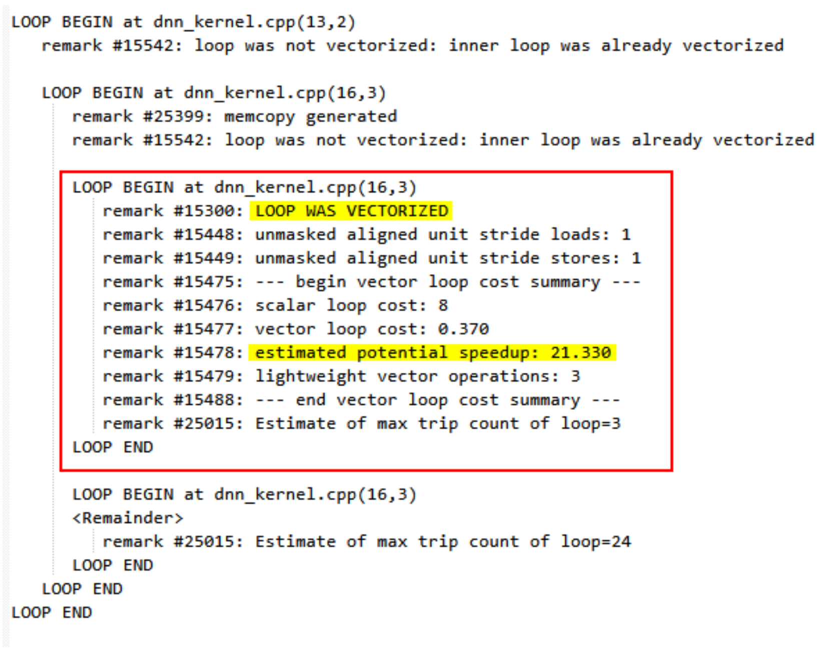


```
extern "C" int softmaxZ(float* in_vec, float* out_vec, int row, int col)
{
    int idx, base;
    float max, tmp;
    float sumexp = 0.0f;
#pragma omp parallel for private(idx,base,max,tmp) firstprivate(sumexp)
    for(int i=0; i<row; i++)
    {
        base = i*col + 0;
        max = in_vec[base];
        #pragma vector aligned
        for(int j=1; j<col; j++)
        {
            idx = i*col + j;
            if(in_vec[idx] > max)
            {
                max = in_vec[idx];
            }
        }
```

Fig.2.24

Vectorization

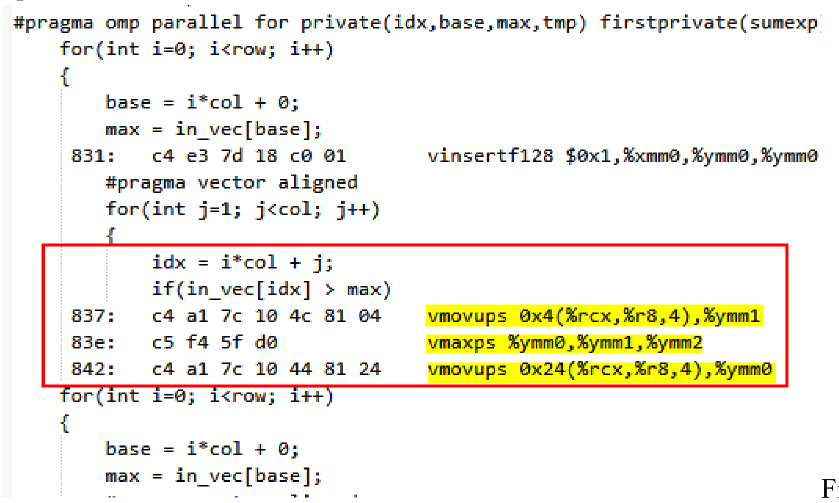
Intel compiler automatically vectorize the code and generate vectorization report with **-qopt-report=3** option. Through studying the report, we can know whether the code snippets are vectorized successfully or not. The intel compiler also provides us with advice so that we can modify our original implementation so that it can achieve much better performance.



```
LOOP BEGIN at dnn_kernel.cpp(13,2)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at dnn_kernel.cpp(16,3)
      remark #25399: memcopy generated
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at dnn_kernel.cpp(16,3)
         remark #15300: LOOP WAS VECTORIZED
         remark #15448: unmasked aligned unit stride loads: 1
         remark #15449: unmasked aligned unit stride stores: 1
         remark #15475: --- begin vector loop cost summary ---
         remark #15476: scalar loop cost: 8
         remark #15477: vector loop cost: 0.370
         remark #15478: estimated potential speedup: 21.330
         remark #15479: lightweight vector operations: 3
         remark #15488: --- end vector loop cost summary ---
         remark #25015: Estimate of max trip count of loop=3
      LOOP END

      LOOP BEGIN at dnn_kernel.cpp(16,3)
      <Remainder>
         remark #25015: Estimate of max trip count of loop=24
      LOOP END
   LOOP END
LOOP END
```

Fig.2.25

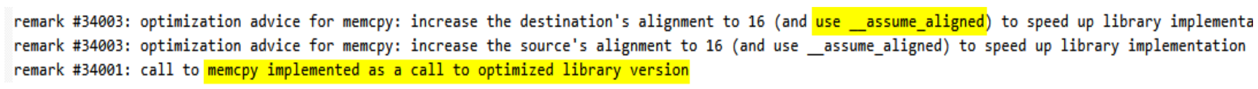
Using ***objdump*** tool, we can generate the **disassemble** code. Comparing the disassemble code with the C language code we can see that the compiler uses **SSE** and **AVX** instruction set to produce the optimized code.



```
#pragma omp parallel for private(idx,base,max,tmp) firstprivate(sumexp
    for(int i=0; i<row; i++)
    {
        base = i*col + 0;
        max = in_vec[base];
     831:   c4 e3 7d 18 c0 01        vinsertf128 $0x1,%xmm0,%ymm0,%ymm0
        #pragma vector aligned
        for(int j=1; j<col; j++)
        {
            idx = i*col + j;
            if(in_vec[idx] > max)
     837:   c4 a1 7c 10 4c 81 04     vmovups 0x4(%rcx,%r8,4),%ymm1
     83e:   c5 f4 5f d0              vmaxps %ymm0,%ymm1,%ymm2
     842:   c4 a1 7c 10 44 81 24     vmovups 0x24(%rcx,%r8,4),%ymm0
    for(int i=0; i<row; i++)
    {
        base = i*col + 0;
        max = in_vec[base];
```

Fig.2.26

The compiler may also give advice to the user intelligently. For example, function *setmatY* assigns the vector B to each row of matrix Y. Its function is logically similar to that of function *memcpy*. So the compiler reminded that user should make the input variables **aligned to 16** and use **"__assume_aligned" macro**, in order to speed up the library implementation.



```
remark #34003: optimization advice for memcpy: increase the destination's alignment to 16 (and use __assume_aligned) to speed up library implementa
remark #34003: optimization advice for memcpy: increase the source's alignment to 16 (and use __assume_aligned) to speed up library implementation
remark #34001: call to memcpy implemented as a call to optimized library version
```

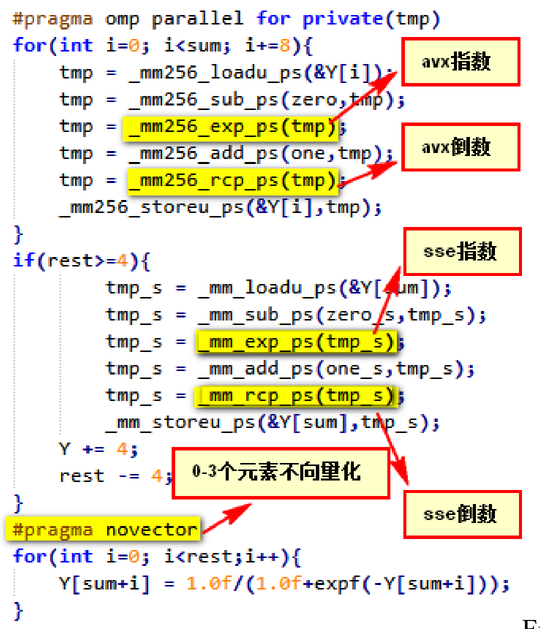
Fig.2.27

the tuning of the functions

SigmoidY and SoftmaxZ

Every time the function *sigmoidY* runs, there will be 1048*2048 calls to the function *expf()* to get the exponential value. Function *expf()* is suboptimal and because it is implemented in the standard library, the compiler will not try to do optimization on it. So we decided to re-implement some code to achieve better performance.

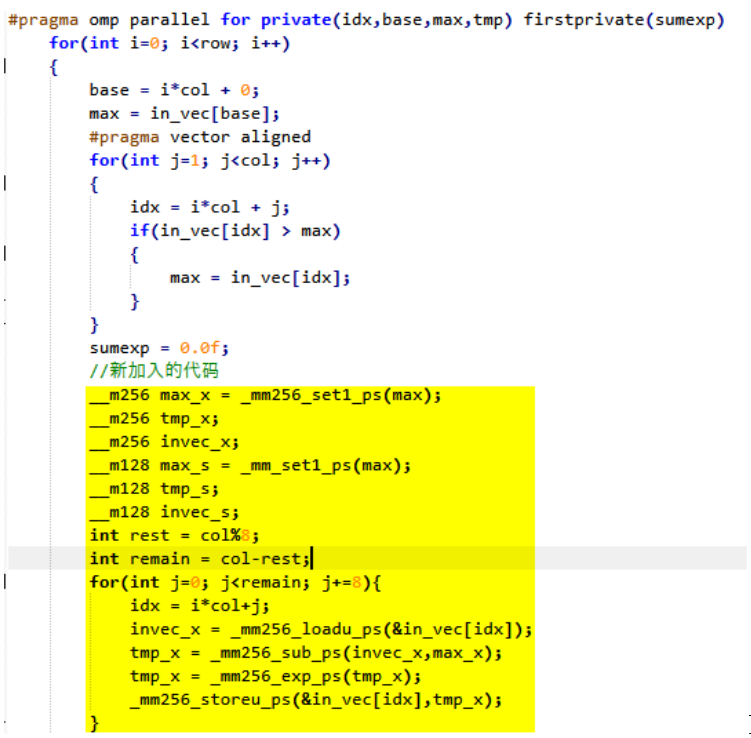
We implement it in C++ Intrinsic syntax format.



```
#pragma omp parallel for private(tmp)
for(int i=0; i<sum; i+=8){
    tmp = _mm256_loadu_ps(&Y[i]);
    tmp = _mm256_sub_ps(zero,tmp);
    tmp = _mm256_exp_ps(tmp);
    tmp = _mm256_add_ps(one,tmp);
    tmp = _mm256_rcp_ps(tmp);
    _mm256_storeu_ps(&Y[i],tmp);
}
if(rest>=4){
        tmp_s = _mm_loadu_ps(&Y[sum]);
        tmp_s = _mm_sub_ps(zero_s,tmp_s);
        tmp_s = _mm_exp_ps(tmp_s);
        tmp_s = _mm_add_ps(one_s,tmp_s);
        tmp_s = _mm_rcp_ps(tmp_s);
        _mm_storeu_ps(&Y[sum],tmp_s);
    Y += 4;
    rest -= 4;
}
#pragma novector
for(int i=0; i<rest;i++){
    Y[sum+i] = 1.0f/(1.0f+expf(-Y[sum+i]));
}
```

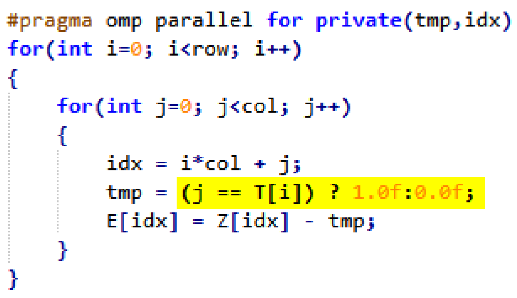
Fig.2.28

So is the function SoftmaxZ, and the updated code is:



```
#pragma omp parallel for private(idx,base,max,tmp) firstprivate(sumexp)
    for(int i=0; i<row; i++)
    {
        base = i*col + 0;
        max = in_vec[base];
        #pragma vector aligned
        for(int j=1; j<col; j++)
        {
            idx = i*col + j;
            if(in_vec[idx] > max)
            {
                max = in_vec[idx];
            }
        }
        sumexp = 0.0f;
        //新加入的代码
        __m256 max_x = _mm256_set1_ps(max);
        __m256 tmp_x;
        __m256 invec_x;
        __m128 max_s = _mm_set1_ps(max);
        __m128 tmp_s;
        __m128 invec_s;
        int rest = col%8;
        int remain = col-rest;
        for(int j=0; j<remain; j+=8){
            idx = i*col+j;
            invec_x = _mm256_loadu_ps(&in_vec[idx]);
            tmp_x = _mm256_sub_ps(invec_x,max_x);
            tmp_x = _mm256_exp_ps(tmp_x);
            _mm256_storeu_ps(&in_vec[idx],tmp_x);
        }
```

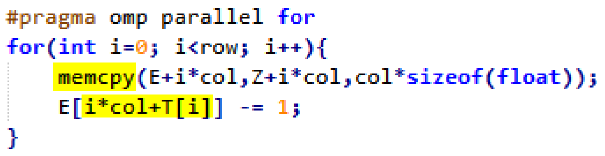
Fig.2.29



```
#pragma omp parallel for private(tmp,idx)
for(int i=0; i<row; i++)
{
    for(int j=0; j<col; j++)
    {
        idx = i*col + j;
        tmp = (j == T[i]) ? 1.0f:0.0f;
        E[idx] = Z[idx] - tmp;
    }
}
```

errorOutput

The original *errorOutput* implementation, there is a conditional statement(tmp = (j == T[i])? 1.0f: 0.0f;) in the ***for*** loop, so that statement will be executed for ***row*col=2048*8091*** times. While for each iteration in the inner loop, there will be ***one*** time that j == T[i] is true, so there will be ***2048*8090*** sub instructions which is vain because it performs "***sub zero***" operation. We modify the original code like Fig2.30.

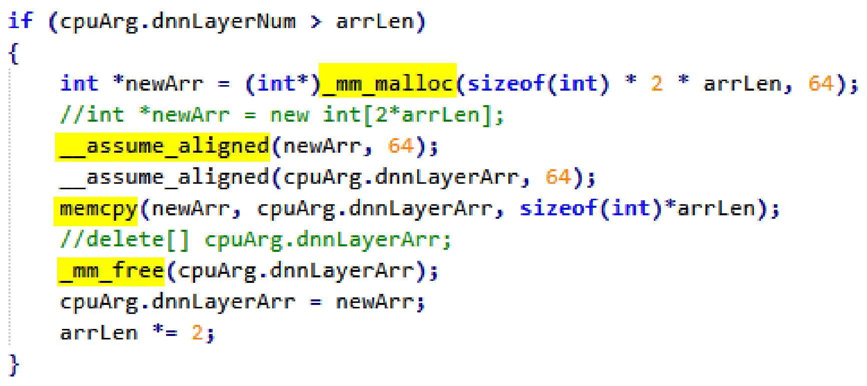


```
#pragma omp parallel for
for(int i=0; i<row; i++){
    memcpy(E+i*col,Z+i*col,col*sizeof(float));
    E[i*col+T[i]] -= 1;
}
```

Fig.2.30

The replacement of *new* and *delete*

In ***dnn_utility.cpp*** file, there are many cases where *new* and *delete* are used. In terms of the performance, *new* and *delete* are slower than *malloc* and *free* when creating and deleting very large arrays. Besides, Intel Compiler also provides a set of memory management API--*_mm_malloc* and *_mm_free* which can allocate and free aligned blocks of memory. So we replaced the *new* and *delete* operations with them.



```
if (cpuArg.dnnLayerNum > arrLen)
{
    int *newArr = (int*)_mm_malloc(sizeof(int) * 2 * arrLen, 64);
    //int *newArr = new int[2*arrLen];
    __assume_aligned(newArr, 64);
    __assume_aligned(cpuArg.dnnLayerArr, 64);
    memcpy(newArr, cpuArg.dnnLayerArr, sizeof(int)*arrLen);
    //delete[] cpuArg.dnnLayerArr;
    _mm_free(cpuArg.dnnLayerArr);
    cpuArg.dnnLayerArr = newArr;
    arrLen *= 2;
}
```

Fig.2.31

The optimization of matrix multiplication

The matrix multiplication consumes the most part of the total time. Aiming to reach the best performance, we did some experiment with 6 kinds of matrix multiplication:

OpenMP common implementation with OpenMP

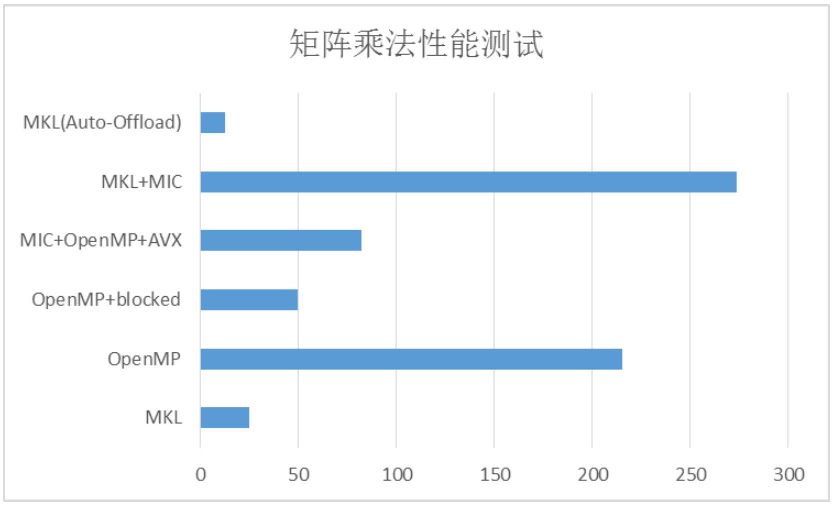
OpenMP+blocked implementation with OpenMP and using blocking

MIC+OpenMP+AVX implementation in offload mode with 512-bit AVX instruction set totally on MIC platform(The time consumed included that spent on transferring data in and out)

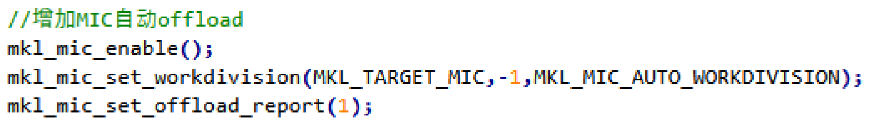
MKL MKL cblas_sgemm on CPU platform.

MKL+MIC MKL cblas_sgemm in offload mode(All the calculation is done on MIC while CPU takes the charge of transferring).

MKL(Auto-Offload) MKL cblas_sgemm in automatic offload mode



Matrices used in the test are A(1024*2048) and B(2048*2048). From the chart we can see that **MKL in automatic offload mode** is the quickest. So we added *mkl_mic_enable()* and *mkl_mic_set_workdivision()* in the **main.cpp**.



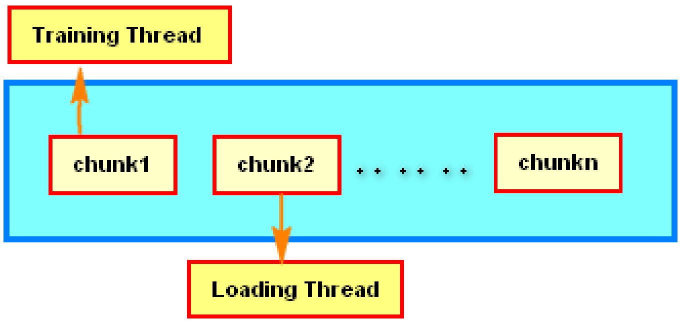
```
//增加MIC自动offload
mkl_mic_enable();
mkl_mic_set_workdivision(MKL_TARGET_MIC,-1,MKL_MIC_AUTO_WORKDIVISION);
mkl_mic_set_offload_report(1);
```

Fig.2.32

2.4.4 Goals in the future

IO optimization

When training a large dnn model, the time consumed on transferring data from the host and the MIC is very prominent. So we can use "**asynchronous transfer mode**".

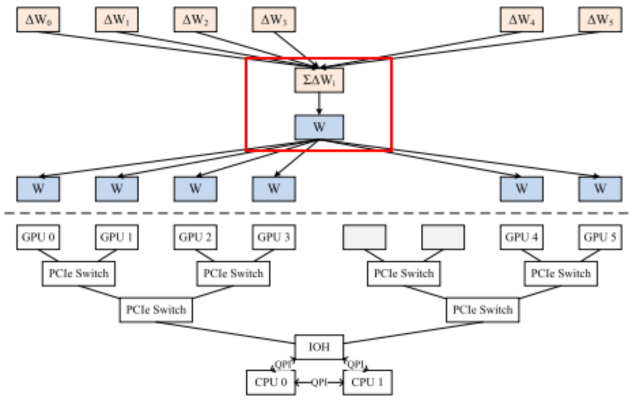


In this mode, threads can be divided into two categories—**training threads** and **loading threads**. The training threads trains the model with $chunk_{i-1}$ data when loading threads are reading $chunk_i$ data. This method is very beneficial to the training process for the reason that it can hide the transmission delay effectively.

We can also divide each *chunk* into many $bunches(bunch_1, bunch_2, \dots \dots, bunch_n)$, the training process starts when $bunch_1$ has been loaded. The time spent on loading is quite less than that spent on training, so it is also a practical way.

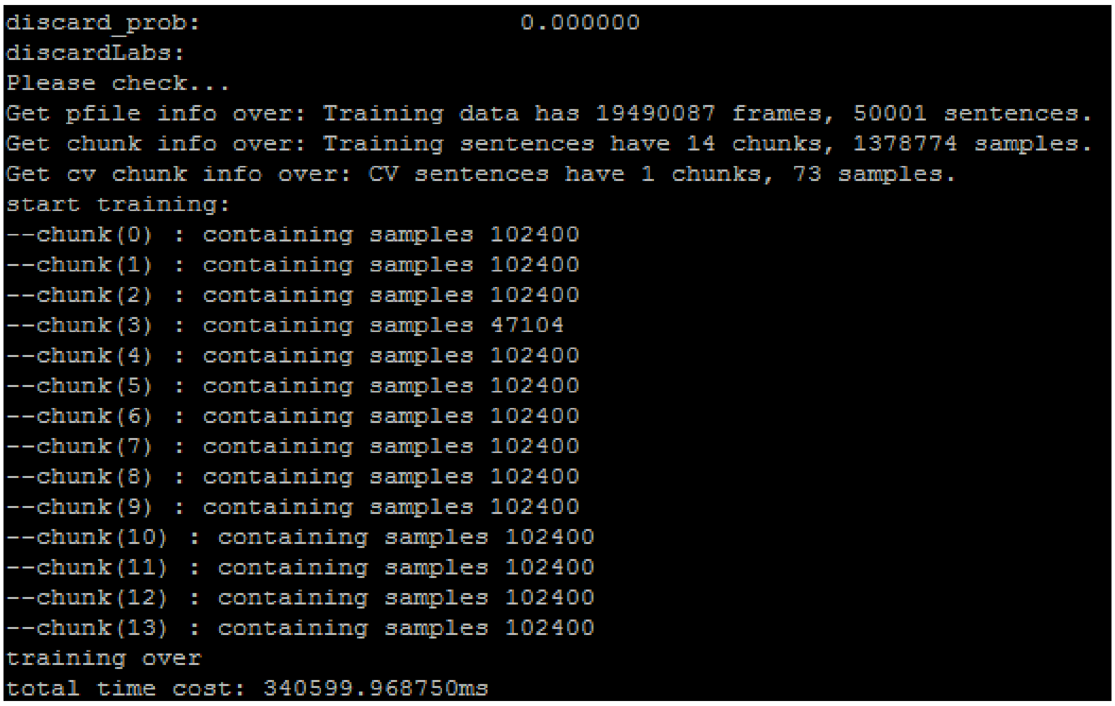
co-training in clusters

The update of weights and bias relies on the ***error*** values generated in training process. So we can establish the same model on each server($model_1, model_2, \dots \dots, model_n$). The data input for each model has the same number. When all the training processes are finished, we can collect the weight updates produced by each model with one process($\Delta W_1, \Delta W_2, \dots \dots, \Delta W_n$). The collecting process reduces all the ΔW and updates the model, then it send the new model parameters to all the working processes. So before the next training procedure, the models on all the servers are the same and updated. This **Data Parallel** method has been used in some famous IT companies.



2.4.5 The Performance

After all methods of optimization, the time cost on **workload2** has been reduced to 33000ms~34000ms, which is big progress compared with the origin code.



```
discard_prob:                  0.000000
discardLabs:
Please check...
Get pfile info over: Training data has 19490087 frames, 50001 sentences.
Get chunk info over: Training sentences have 14 chunks, 1378774 samples.
Get cv chunk info over: CV sentences have 1 chunks, 73 samples.
start training:
--chunk(0) : containing samples 102400
--chunk(1) : containing samples 102400
--chunk(2) : containing samples 102400
--chunk(3) : containing samples 47104
--chunk(4) : containing samples 102400
--chunk(5) : containing samples 102400
--chunk(6) : containing samples 102400
--chunk(7) : containing samples 102400
--chunk(8) : containing samples 102400
--chunk(9) : containing samples 102400
--chunk(10) : containing samples 102400
--chunk(11) : containing samples 102400
--chunk(12) : containing samples 102400
--chunk(13) : containing samples 102400
training over
total time cost: 340599.968750ms
```