Development of Harp-DAAL Interface

Langshi Chen and Judy Qiu
School of Informatics and Computing
Indiana University

Intel® Data Analytics Acceleration Library (DAAL) is a library from Intel that aims to provide the users of some highly optimized building blocks for data analytics and machine learning applications. The latest version is the 2017 beta version that is already made open source on the Github homepage[^https://github.com/01org/daal]. For each of its kernel, DAAL has three modes:

- A *Batch Processing* mode is the default mode that works on an entire dataset that fits into the memory space of a single node.
- A *Online Processing* mode works on the blocked dataset that is streamed into the memory space of a single node.
- A *Distributed Processing* mode works on datasets that are stored in distributed systems like multiple nodes of a cluster.

Nowadays, many data analytics and machine learning problems contain millions or billions of training data and parameter data, it is obvious that the *Distributed Processing* mode is the only choice for many applications. Within DAAL's framework, the communication layer of the *Distributed Processing* mode is left to the users, which could be Hadoop, Spark, MPI, or any of the user-defined middleware. The goal of our project is thus to fit Harp, a plug-in into Hadoop ecosystem, into the *Distributed Processing* mode of DAAL. Compared to contemporary communication libraries, Harp has the advantages as follows:

- Harp has MPI-like collective communication operations that are highly optimized for big data problems.
- Harp has efficient and innovative computation models for different machine learning problems.

The original Harp project has all of its codes written in Java, which is a common choice within the Hadoop ecosystem. The downside of the pure Java implementation is the slow speed of the computation kernels that are limited by Java's data management. Since manycore architectures devices are becoming a mainstream choice for both server and personal computer market, the computation kernels should also fully take advantage of the architecture's new features, which are also beyond the capability of the Java language. Thus, a reasonable solution for Harp is to accomplish the computation tasks by invoking C++ based kernels from libraries such as DAAL. The implementation and the challenges of such an interface between Harp and DAAL will be discussed in the following sections.

Data Structure of Harp and DAAL

The major roadblock of the interface between Harp and DAAL is their data structures. Harp is written in Java while DAAL has most of its codes written in C++. Fortunately, DAAL has already provide the users of a Java API, which invokes the C++ codes at low level. Nevertheless, Harp and DAAL have different data structures shown in Figure 1.

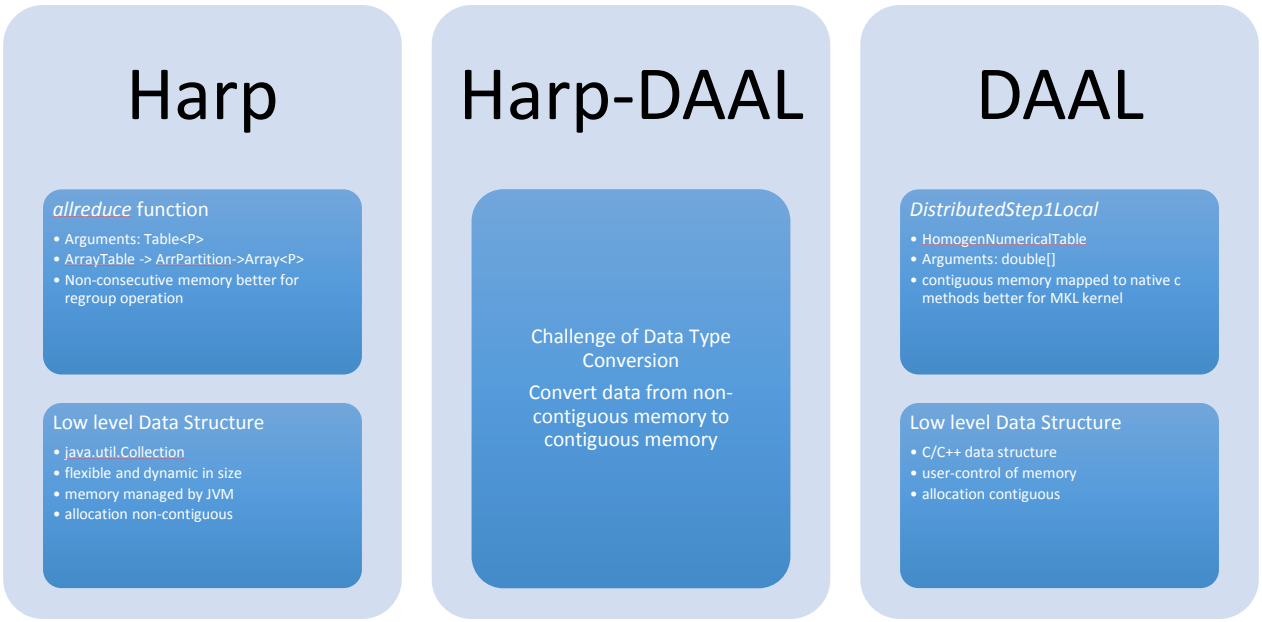


Figure 1. Data structure of Harp and DAAL

Harp defines its own three-level data structure. At the top level is its *ArrTbale*, which extends the Table class of Java. At the middle level is the *ArrPartition*, which owns an *identifier* and an *Array*. At the low level is the *Array* that derives from the primitive Array class of Java. Therefore, an *ArrTable* of Harp could contains a substantial number of *ArrPartition*, and the data resides on non-consecutive memory space.

DAAL also has a variety of data structures. In general, it has *data source*, *data dictionary*, *NumericTable*, *matrix*, and so forth. E.g., *NumericTable* is a major data structure used in DAAL's algorithms. *NumericTable* has many sub-classes such as *HomogenNumericTable*, *AOSNumericTable*, *SOANumericTable*. In *HomogenNumericTable*, the data is stored in consecutive chunks of memory space, while the *AOSNumericTable* has its data stored in non-contiguous memory space. If the users use the Java APIs of these data structures, they could also decide whether the data's memory space is allocated on the Java side or on the C++ side. There are pros and cons for both of the two sides, and we will investigate them in the following sections.

Because of the different data structures, the interface needs to address the data conversion problem, and consequently, will cause additional time overhead. Unless we modify the data structures of Harp and DAAL, we can only use some multithreading copy to reduce the conversion time overhead. The JNI interface used in DAAL's Java APIs will also give us some challenges that will be discussed in the following sections.

Two Case Studies and Preliminary Results

We select two algorithms, K-means and Stochastic Gradient Descent (SGD), to test our interface implementation. K-means represents the category of computation-intensive problems while SGD represents the category of memory-intensive applications.
For each of them, we will compare the performance of the Harp-DAAL interfaced hybrid codes with that of the original pure Java codes.

Harp-DAAL-Kmeans

The interface between Harp's K-means and DAAL's K-means is quite straightforward.

```
//----- Harp codes to get pointPartitions ----------------//

//------ start convert data from Harp to DAAL -----------

HarpDaalParallelConvert converter = new HarpDaalParallelConvert(pointPartitions, pointData, pointPartition
s.size(), pointsPerFile, vectorSize);
converter.HarpToDaal();

HarpDaalParallelConvert converter_cen_to_daal = new HarpDaalParallelConvert(cenTable, daalCenData, numCenP
artitions, 0, vectorSize);
converter_cen_to_daal.HarpCenToDaal();

//------- End of converting data from Harp to DAAL ----------//

//create the DAAL's data structure
HomogenNumericTable ntData = new HomogenNumericTable(daalContext, pointData, vectorSize, totalVectors);
HomogenNumericTable ntCen = new HomogenNumericTable(daalContext, daalCenData, vectorSize, numCentroids);

// compute K-means by DAAL
DistributedStep1Local kmeansLocal = new DistributedStep1Local(daalContext, Double.class, Method.defaultDen
se, numCentroids);

kmeansLocal.input.set(InputId.data, ntData);
kmeansLocal.input.set(InputId.inputCentroids, ntCen);
PartialResult pres = kmeansLocal.compute();
```

HarpDaalParallelConvert is a defined class that uses Java multithreading to copy data from Harp's *pointPartition* to DAAL's *pointData*

In the test, we compiled the codes with the DAAL-2017 released version on Github The test of K-means was done by two mappers on a single node of Juliet. The test of MF-SGD was done by four mappers on two nodes of Juliet.

- Harp-Kmeans

  Origianl implementation of Bingjing, the local computatoin kernel of K-means is written in Java multithreading

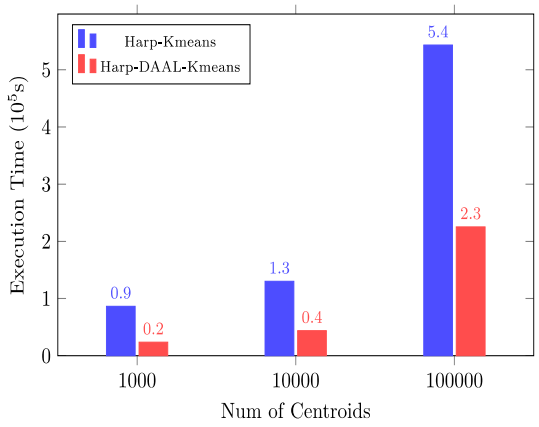
- Harp-DAAL-Kmeans

  Hybrid implementation of Harp with DAAL-2017. The local computation is offloaded to DAAL while the communication layer is handled by Harp. There is a group of Java classes dedicated to the data type conversion between Harp and DAAL

The K-means tests used data points where each point is a vector with a length of 10. Each test had 10 iterations of computing K-means, and each test was repeated by 5 times. Thus, the averaged results could remove the influence from the fluctuation of Hadoop JVM's performance. In order to investigate the benefits of DAAL to Harp-Kmeans, we divided the tests into two groups.

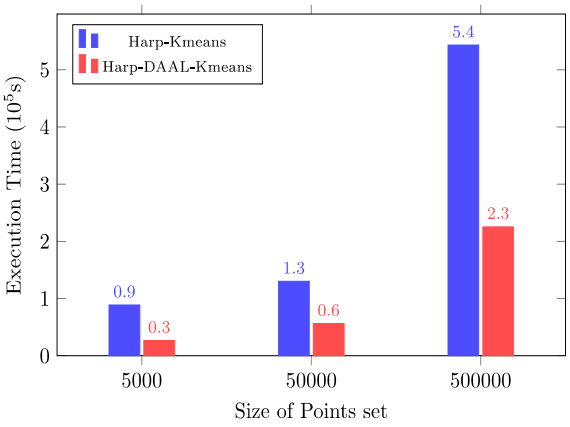
- Group 1: The tests fix the number of data points to 500000 while varying the number of centroids by 1000, 10000, 100000, respectively.
- Group 2: The tests fix the number of centroids to 100000 while varying the number of data points by 5000, 50000, 500000, respectively.

The following two graphs show the experiment results:



Harp-Kmeans vs. Harp-DAAL-Kmeans with 500000 Points

Figure 2. Test Results of Group 1



Harp-Kmeans vs. Harp-DAAL-Kmeans with 100000 Centroids

Figure 3. Test Results of Group 2

In Figure 2., we found that the execution time of both Harp-Kmeans and Harp-DAAL-Kmeans increase rapidly when we augment the number of centroids. The benefits of DAAL is obvious because it saves by 2x to 4x the execution time in average. In Figure 3., Harp-DAAL-Kmeans still saves up to 3 times of the execution time compared to Harp-Kmeans.

Harp-DAAL-SGD

Matrix Factorization based on Stochastic Gradient Descent

Matrix Factorization based on Stochastic Gradient Descent (SGD-MF for short) is an algorithm widely used in recommender systems. MF-SGD is one of the implemented algorithm within Harp, however, DAAL current does not have a MF-SGD kernel. We first implement a MF-SGD kernel inside DAAL's framework and then interface it with that of Harp. MF-SGD aims to factorize a sparse matrix into two dense matrices named mode W and model H as follows.

$V = WH$

Matrix V includes both training data and test data, a machine learning inspired kernel with use the training data to approximate the model matrices W and H. A standard SGD procedure will update the model W and H when it trains each training data, i.e., an entry in matrix V in the following formula.

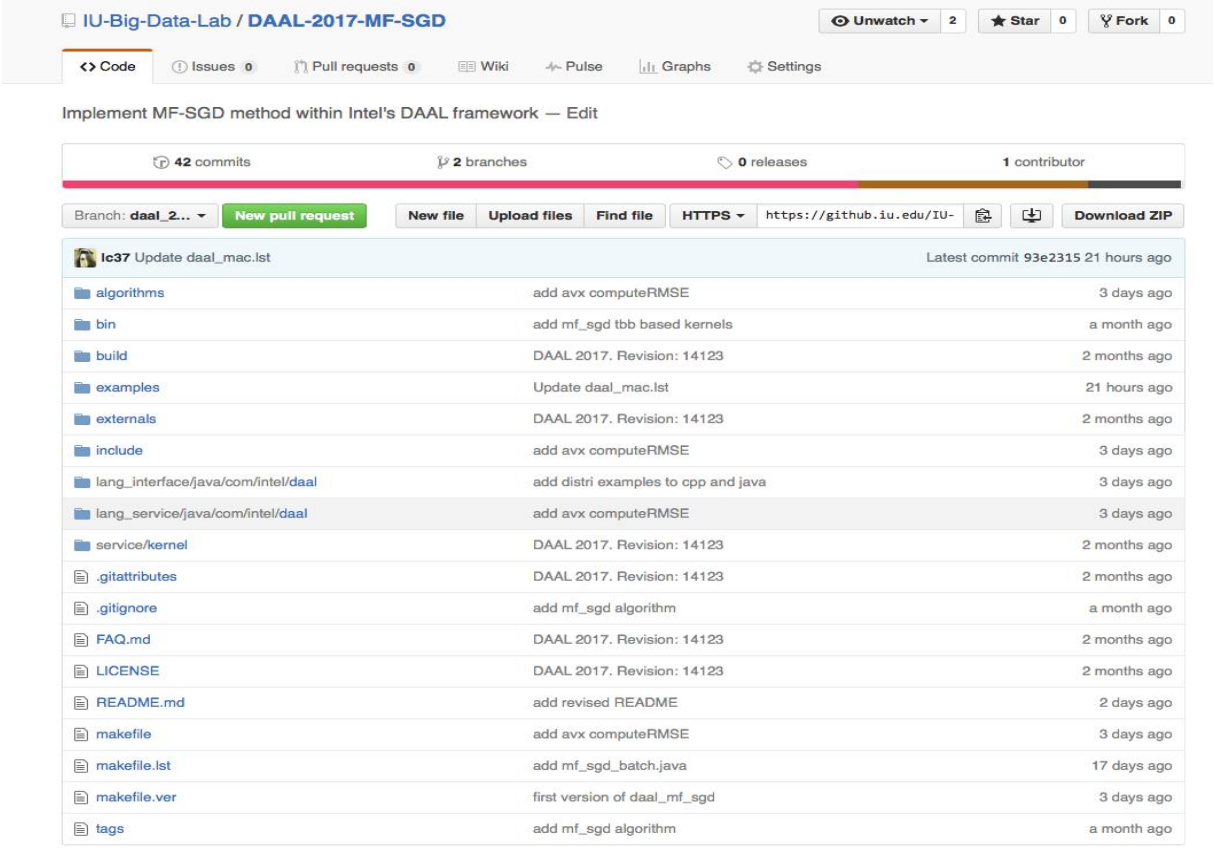
$$E_{ij} = V_{ij} - \sum_{k=0}^{r} W_{ik} H_{kj}$$

$$W_{i*}^{t} = W_{i*}^{t-1} - \eta(E_{ij}^{t-1} \cdot H_{*j}^{t-1} - \lambda \cdot W_{i*}^{t-1})$$

$$H_{*j}^{t} = H_{*j}^{t-1} - \eta(E_{ij}^{t-1} \cdot W_{i*}^{t-1} - \lambda \cdot H_{*j}^{t-1})$$

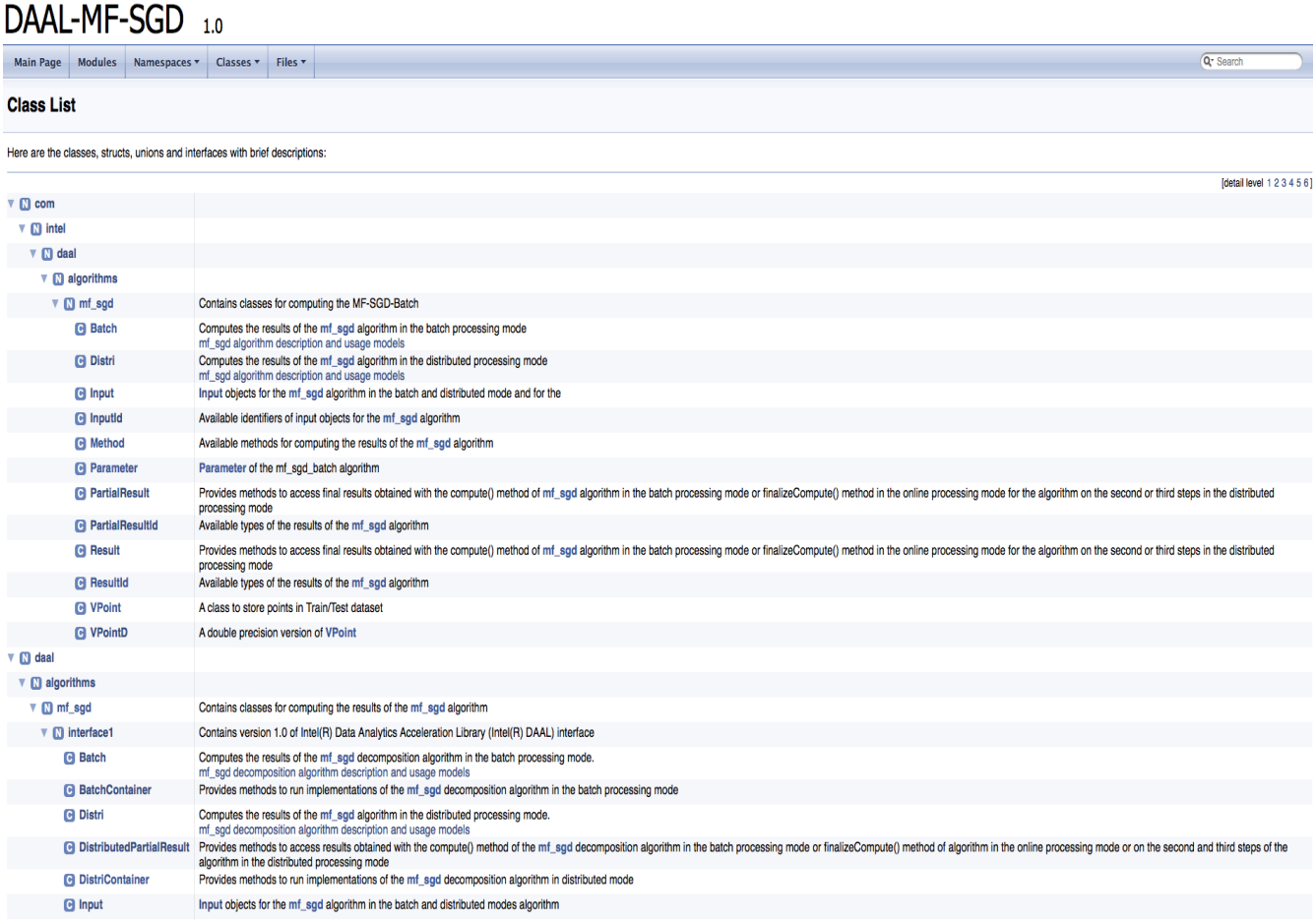
Tests on Harp-SGD and Harp-DAAL-SGD

We implement a MF-SGD module and add it into DAAL-2017's library. Our work has already been released on the following Github page: https://github.iu.edu/IU-Big-Data-Lab/DAAL-2017-MF-SGD



Archive of DAAL-MF-SGD on IU's Github

We also generate an online documentation at https://github.iu.edu/pages/IU-Big-Data-Lab/DAAL-2017-MF-SGD/



DAAL-MF-SGD 1.0

Main Page | Modules | Namespaces | Classes | Files | Search

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

[detail level 1 2 3 4 5 6]

com	
intel	
daal	
algorithms	
mf_sgd	Contains classes for computing the MF-SGD-Batch
Batch	Computes the results of the mf_sgd algorithm in the batch processing mode mf_sgd algorithm description and usage models
Distri	Computes the results of the mf_sgd algorithm in the distributed processing mode mf_sgd algorithm description and usage models
Input	Input objects for the mf_sgd algorithm in the batch and distributed mode and for the
InputId	Available identifiers of input objects for the mf_sgd algorithm
Method	Available methods for computing the results of the mf_sgd algorithm
Parameter	Parameter of the mf_sgd_batch algorithm
PartialResult	Provides methods to access final results obtained with the compute() method of mf_sgd algorithm in the batch processing mode or finalizeCompute() method in the online processing mode for the algorithm on the second or third steps in the distributed processing mode
PartialResultId	Available types of the results of the mf_sgd algorithm
Result	Provides methods to access final results obtained with the compute() method of mf_sgd algorithm in the batch processing mode or finalizeCompute() method in the online processing mode for the algorithm on the second or third steps in the distributed processing mode
ResultId	Available types of the results of the mf_sgd algorithm
VPoint	A class to store points in Train/Test dataset
VPointD	A double precision version of VPoint
daal	
algorithms	
mf_sgd	Contains classes for computing the results of the mf_sgd algorithm
interface1	Contains version 1.0 of Intel(R) Data Analytics Acceleration Library (Intel(R) DAAL) interface
Batch	Computes the results of the mf_sgd decomposition algorithm in the batch processing mode. mf_sgd decomposition algorithm description and usage models
BatchContainer	Provides methods to run implementations of the mf_sgd decomposition algorithm in the batch processing mode
Distri	Computes the results of the mf_sgd algorithm in the distributed processing mode. mf_sgd decomposition algorithm description and usage models
DistributedPartialResult	Provides methods to access results obtained with the compute() method of the mf_sgd decomposition algorithm in the batch processing mode or finalizeCompute() method of algorithm in the online processing mode or on the second and third steps of the algorithm in the distributed processing mode
DistriContainer	Provides methods to run implementations of the mf_sgd decomposition algorithm in distributed mode
Input	Input objects for the mf_sgd algorithm in the batch and distributed modes algorithm

Online Documentation of DAAL-MF-SGD

In the test, we refer the original pure-Java implementation as *Harp-SGD* while the hybrid one with DAAL's kernel as *Harp-DAAL-SGD*. We use two nodes from Juliet cluster located at Indiana University Bloomington in our tests.

Each node has the following CPU information:

Architecture: x86_64 CPU
op-mode(s): 32-bit,
64-bit Byte Order: Little Endian
CPU(s): 48
On-line CPU(s) list: 0-47
Thread(s) per core: 2
Core(s) per socket: 12

The memory configuration of Harp on each node is shown as follows:

Yarn allocates: 128 GBs Each Mapper allocates: 100 GBs

In the test, we use two datasets.

1.MovieLens 2.Yahoomusic

Other configurations and parameters are listed below:

Total Iteration times: 5
LearningRate: 0.005
Lambda: 0.003
Threads: 40
Total Mapper 2, Each node has 1 mapper

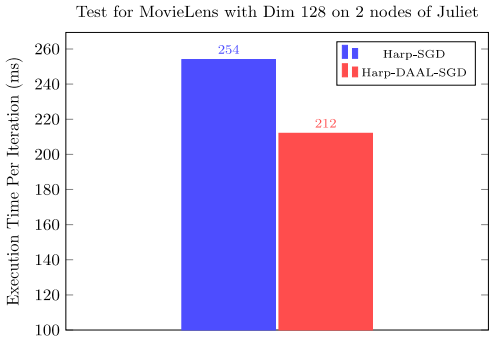


Figure 4. Test on MovieLens with 128 dimensional Feature Vector

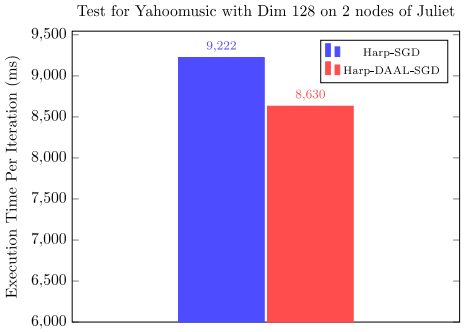


Figure 5. Test on Yahoomusic with 128 dimensional Feature Vector

In Figure 4 and 5, we find that, by integrating DAAL's kernel into Harp, the Harp-DAAL-SGD does have less execution time than the original Harp-SGD. However, the performance speedup is only around 6% for Yahoomusic and 16.5% for MovieLens. As we know, Yahoomusic has a larger data size than MovieLens, which indicates that it should have more computation workload. However, the speedup brought by DAAL's computation kernel is less effective on large dataset, which requires a further investigation.

By decomposing the execution time into different stages within Harp-DAAL-SGD, we have the following pie chart:

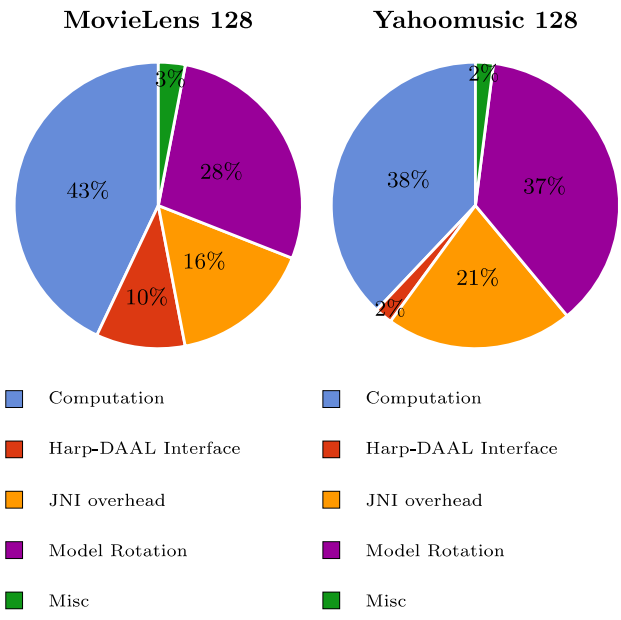


Figure 6. Time Ratio of Harp-DAAL-SGD on MovieLens and Yahoomusic

In Figure 6, we have seen two additional overhead beside computation and data communication between distributed nodes (refers to model rotation).

1.Interfacing Overhead

2.JNI Overhead

They all together take up to 25% of the whole execution time. These overheads do not exist at the pure-Java implementation of Harp-SGD, they are, actually the side-effect of interfacing two languages and two libraries.

Interface Overhead Between DAAL and Harp

DAAL and Harp have their own different data abstracts. DAAL is likely to use contiguous allocated memory either at native side, or at the heap of Java side. In contrast, Harp uses a *Table* structure, where each of its element is allocated on Java Heap individually. The collective communication functions provided by Harp only supports its own *Table* structure, therefore, when we need to transfer data, like the updated model W and H to other nodes, we have to decompose the contiguous table from DAAL into Harp's *Table* structure. In our Harp-DAAL-SGD implementation, we use Java multi-threading to copy data between the two data containers, and the overhead could be controlled in large data cases.

JNI overhead in data transfer

Another interface overhead is the data transfer between two languages. The model data W and H are initially created at the heap space of Java, and DAAL's native kernel could not access them directly. In DAAL's Java interface, we have two solutions.

1.Explicitly allocating data at the native code side in Java programming codes. The data is actually allocated at the DirectBuffer of Java, and the memory address of this buffer could be passed to the native kernel of DAAL by JNI interface.

2.Keeping the data at the Java heap side. Each time the native kernel needs to access the data, it will callback to the Java side and temporarily create a DirectBuffer for the required data at the native code side.

Since the MF-SGD kernel will usually update the same model data by a multiple times, we find the first choice has less time overhead than the second one. Therefore, we pass the model data W and H from Heap side to the DirectBuffer before the computation kernels start. After the computation finishes, we retrieve the updated model data to the Java heap side for rotation. This two-side data transfer contributes a roughly 20% of the total execution time. Java does not have a *pointer* type as that of C/C++, thus, we could not execute the data transfer between DirectBuffer and Heap segmentation by segmentation in parallel. With the data size increases, the JNI overhead increases accordingly.

Future Work on Harp-DAAL interface

The bottleneck of interfacing Harp and DAAL lies on the data transfer between different memory space. Although DAAL's kernel runs faster than the Java written computation kernel, we could only use small model data because of the transfer overhead. To remove this barrier, we need to do the following things:

- Re-implement the communication function of Harp, making it compatible to DirectBuffer memory beside the heap memory
- Currently DAAL only has homogeneous table that supports buffer implementation. If the SOANumericTable can also have such buffer implementation support, we may eliminate the data copy from contiguous memory to separated memory space.

Performance Evaluation of DAAL-SGD-MF and Libmf

Figures 7 and 8 show the comparison of DAAL-SGD-MF and LIBMF. The execution time per iteration shows that LIBMF works faster than DAAL-SGD-MF, however, LIBMF has a worse convergence than DAAL.

We also observe that LIBMF quickly drops out of performance when it runs more than 64 threads, which is the number of physic cores (threads) of our KNL. These differences and phenomena come from the fact that, although both of LIBMF and DAAL follow the same standard SGD algorithm for matrix factorization, they adopt different computation model (synchronization pattern) and multi-threading programming paradigm.

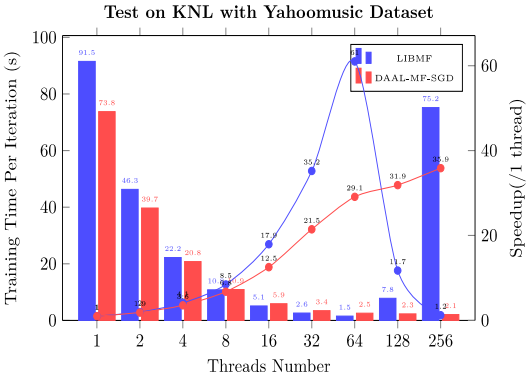


Figure 7.Time per iteration and Threads scalability

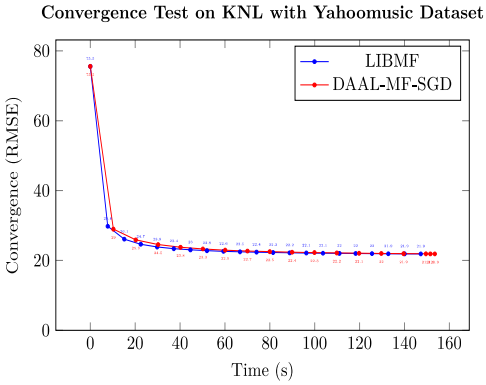


Figure 8. Convergence by Timeline

Model Rotation vs. Asynchronous communication

LIBMF uses the model B in Zhang and Peng's classification, which rotates a portion of the model data among parallel computing elements (e.g., threads in shared memory, or processes in distributed memory). This strategy aims to reduce the confliction of updating the same data by different parallel computing elements. The downside is the lack of load balance for some matrices where the nonzero entries per column/row is significantly unbalanced. In contrast, DAAL-SGD-MF chooses the model D in Zhang and Peng's classification, where each parallel computing elements freely updates the global model and uses stale local model. If the matrix is very sparse, and the workload for each thread is very light, such conflict of updating the same model data by different parallel computing elements rarely happens. The upside of model D is that it could achieve a good load balance if the parallel computing elements are well scheduled.

Thread based Scheduling vs. Task based Scheduling

LIBMF uses a raw pthread based multi-threading programming, the developers write their own threads scheduler that follows the model rotation strategy and a dynamic threads scheduling. DAAL-SGD-MF chooses the Task based scheduling, which is accomplished by Intel's TBB library. Though both of them has a dynamic thread scheduling policy, there are some fundamental differences between thread based scheduling and tasks based scheduling. According to an explanation provided by Intel.

The threads you create with a threading package are logical threads, which map onto the physical threads of the hardware. For computations that do not wait on external devices, highest efficiency usually occurs when there is exactly one running logical thread per physical thread. Otherwise, there can be inefficiencies from the mismatch.

This explains the drop down of LIBMF when it uses more than the number of physic threads. DAAL-SGD-MF, thanks to the task-based scheduling by TBB, could take advantage of hyper threading and achieve a higher CPU/Threads utilization than that of LIBMF. (See Figure 9)

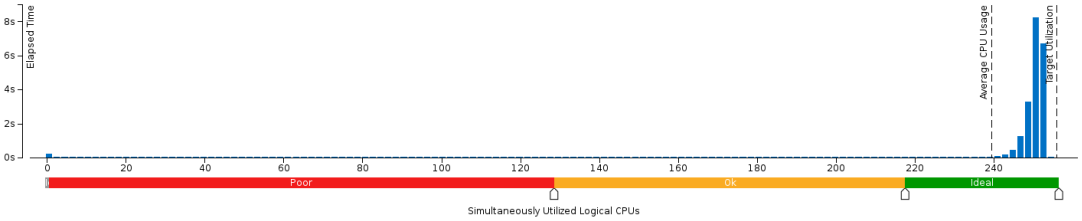


Figure 9. CPU utilization of DAAL-SGD-MF

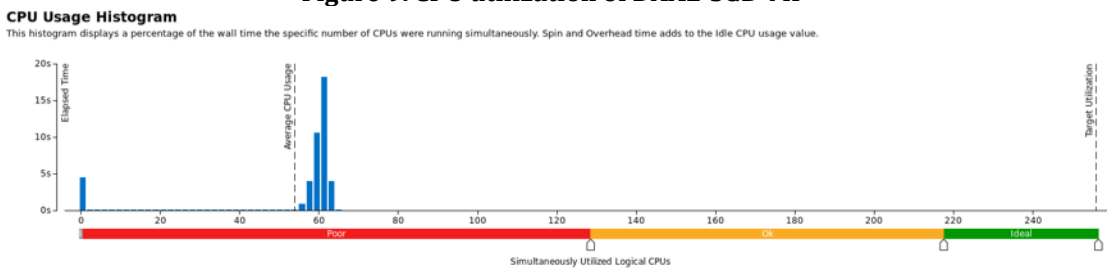


Figure 10. CPU utilization of LIBMF

L2 cache hit Rate and Memory Access Latency

From a detailed profiling by VTune, we find that both of LIBMF and DAAL-SGD-MF own a relatively low L2 cache hit rate, which are 0.541 and 0.404, respectively. This low cache hit rate suggests that the standard SGD's random memory access pattern makes the data hard to be cached. If a L2 cache miss happens, the tiles in KNL will go through a long path to query the L2 cache of other tiles and finally fetch the data from memory. This substantial data access latency is brought either by the access latency of memory, or by the latency time spent on the on-die communication among tiles. According to the bandwidth utilization histogram by VTune (See Figures 11 and 12), it is clear that both of LIBMF and DAAL-SGD-MF do not fully utilize the provided bandwidth (around 400 Gbs/s) of MCDRAM. Thus, the latency of accessing memory should be low, and we suggest the major barrier of the latency performance comes from the poor data locality of tiles on KNL. This conclusion inspires us of using the SNC mode of KNL, which is declared to own the best data locality of threads on sub-NUMA nodes among all the clustering modes of KNL.

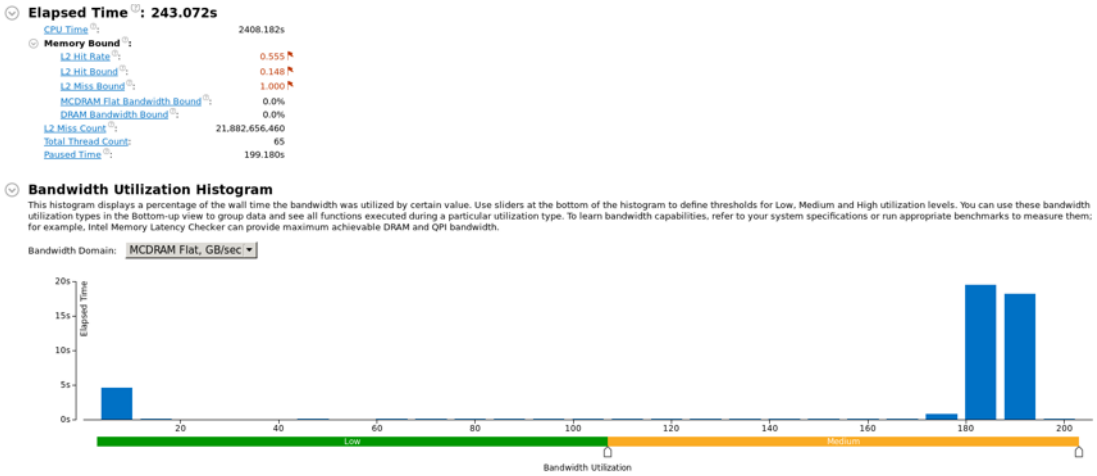


Figure 11. Bandwidth Utilization of LIBMF

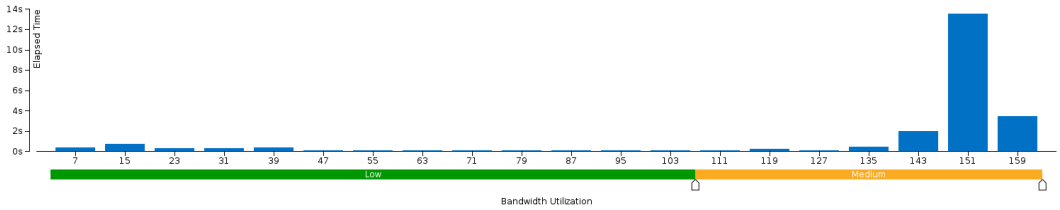


Figure 12. Bandwidth Utilization of DAAL-SGD-MF

SGD-MF Optimization on KNL

Matrix Factorization is one of the best techniques to solve a collaborative filtering problem in order to build a recommender system. It is a typical big data analysis algorithm, which deal with both large volume of input data (user-item ratings) and large volume of model(user-latent and item-latent matrix).

Libmf[1] is the state-of-art implementation on multi-threading and share memory platform using SGD solver for the matrix factorization problem. It builds up by two basic ideas. First, it achieves free-of-conflicts model updates by splitting the input matrix into blocks and using a dynamic scheduler to select blocks without conflicts for the running threads. Second, it re-organizes the processing order of the training points into a more cache friendly way, with a compromise of the randomness required by the algorithm. It's always a trade-off between efficiency and effectiveness.

In this report, we investigate the performance issue of a sgd-mf trainer on the new KNL architecture by using libmf. We try to find out the bottleneck of this kind of application and possible solutions. Currently, it has the following sections:

- Performance Evaluation

[1] Libmf, https://www.csie.ntu.edu.tw/~cjlin/libmf/

- Bottleneck Analysis
- Optimization
  - Optimization By Reorganize Memory Access Order: Tiling
  - Optimization By Reorganize Memory Layout: Nopermute
  - Optimization By Repeat Computation on Training Points: Repeat
  - Optimization By Decrease Memory Access Latency: NUMA

Performance Evaluation of Haswell vs. KNL

Experiment setting:
Run libmf on the standard yahoomusic dataset, with different thread number under a fixed eta(learning rate), two platform(hsw72, knl), two vectorization optimization(avx512 on/off knl & avx on/off hsw72):

mf-train -l2 1 -k 128 -r 0.0001 -t 10 -s #threadnum -p yahoomusic-test.mm yahoomusic-train.mm

KNL runs with mcdram in flat mode, all datasets are allocated into mcdram.

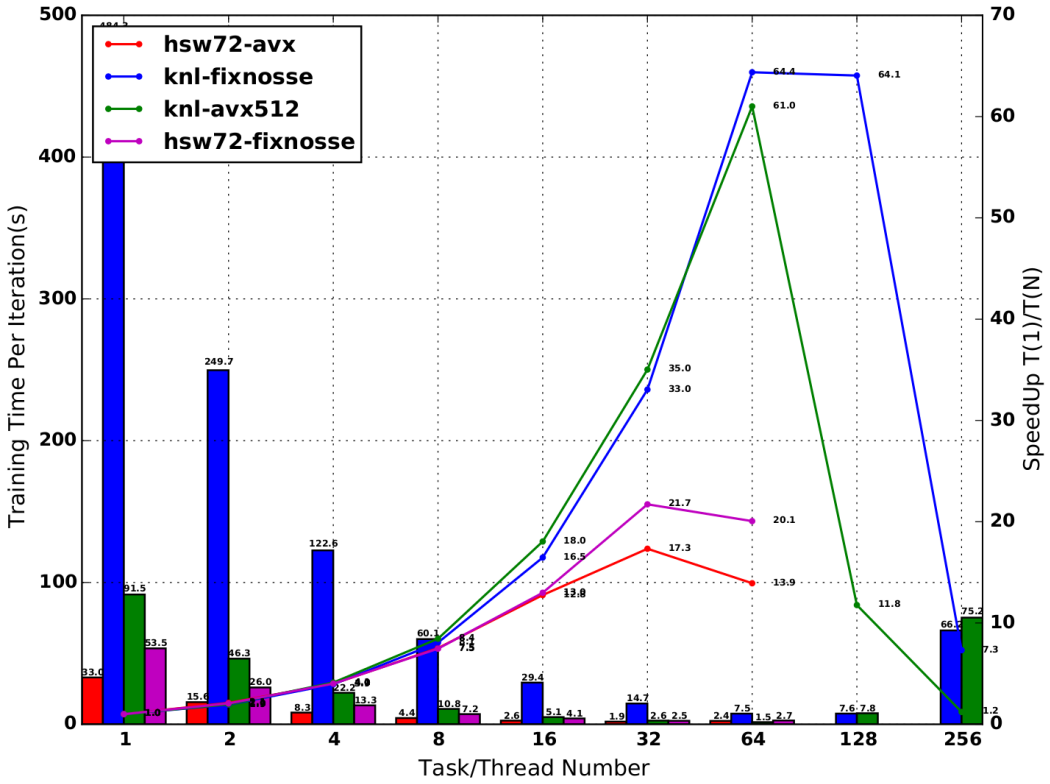


Figure 13 Performance Evaluation of Libmf

According to Figure13, we have the following observations from this experiment:
A). Avx512

a) On knl, avx512 runs about 5x faster than nosse (vectorization off)
b) On hsw72, avx runs about 1.6x faster than nosse(vectorization off)

B). Hyper-Threading

a) Hyper-threading brings no gain on both knl and hsw72

b) It goes even worse when vectorization on.

C). Speedup

a) On hsw72, speedup is below the linear, e.g., 21.7 at S32 (thread=32)
b) On Knl, speedup is near (super) linear, e.g., 61 at S64.

It seems libmf has better scalability on knl. But the very slow performance of one thread on knl can contribute to this.

D). Performance

a) Each core of Hsw72 is more powerful than the counter part of KNL
b) But as in Figure 14, the advantage of HSW decreases when more threads are used. Later, we find out that the memory bandwidth on HSW becomes the bottleneck.

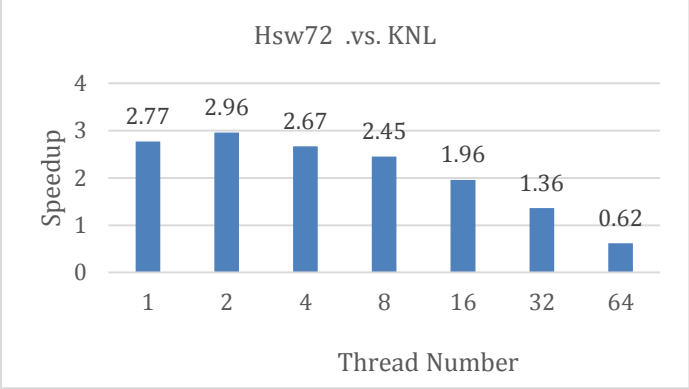


Figure 14 Performance Comparison Between HSW and KNL

c) KNL finally runs faster

KNL get best performance at 1.5(s/iter) on S64, which is about 1.27x faster than Hsw72's best preformance of 1.9 on S32.

Bottleneck Analysis

Performance analysis with VTUNE shows that this application has high l2-cache miss rate and low FPU utilization. (see Appendix)

L2 hit rate	0.541
L2 miss bound	1.00
FPU utilization upper bound	10.7%
Bandwidth Utilization	MCDRAM flat ~180GB/s
SIMD compute-to-l2 access ratio	24.810

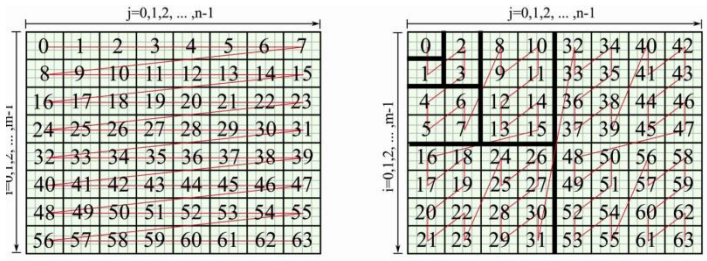
There is a low compute-to-memory load ratio, thus the power of FPU can not be delivered fully.

Then we try different ways to optimize, either to increase the l2 hit rate by reorganize the memory access order or decrease the latency of memory access by utilize the NUMA topology, or increase the computation for each memory load by modifying the algorithm.

Optimization By Reorganize Memory Access Order: Tiling

Tiling, a better organization of memory access, can be used to make the memory access more cache friendly, thus may ameliorate this kind of memory-bounded problem.

There are different tiling schemes: a) row-based, b)col-based, c)z-filling, (cache oblivious filling). Libmf itself split the input training data matrix into blocks, and do partial random training on the training points by select blocks randomly but select points inside one block with a special order. By default, it uses a row-based order.



First, we test different tiling schemes on libmf with one thread and one block setting, avx512 open.

problem[m=1000990, n=624961,nnz=252800275]

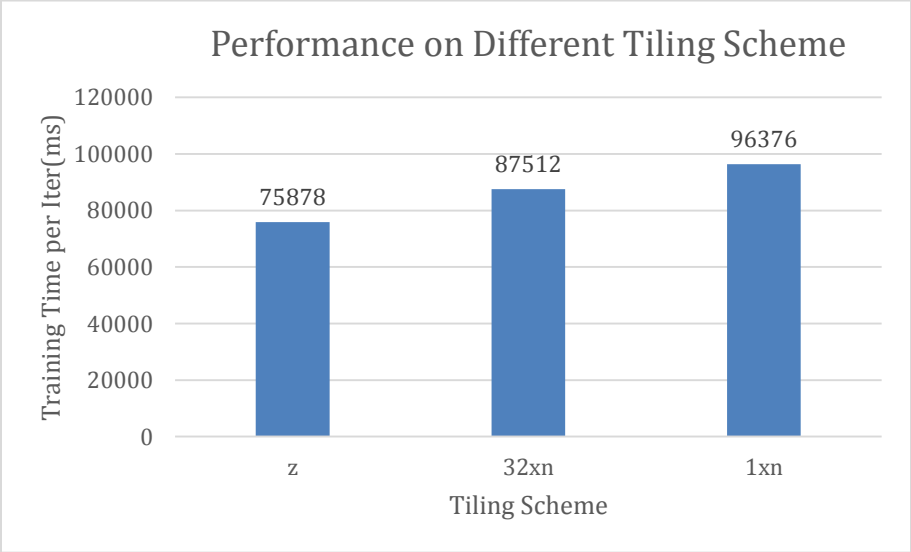


Figure 15 Performance on Different Tiling Scheme

Z filling is about 1.27x faster than non-tiling.

VTune analysis on tiling:

	Non-tiling	Z-filing
	FPU Utilization Upper Bound: 9.0% GFLOPS Upper Bound: 3.742 Scalar GFLOPS Upper Bound: 0.009 Packed GFLOPS Upper Bound: 3.732	FPU Utilization Upper Bound: 12.3% GFLOPS Upper Bound: 5.083 Scalar GFLOPS Upper Bound: 0.006 Packed GFLOPS Upper Bound: 5.077

	Back-End Bound: 45.5% L2 Hit Bound: 0.160 L2 Miss Bound: 1.000 MCDRAM Flat Bandwidth Bound: 0.0% DRAM Bandwidth Bound: 0.0%	Back-End Bound: 50.4% L2 Hit Bound: 0.233 L2 Miss Bound: 1.000 MCDRAM Flat Bandwidth Bound: 0.0% DRAM Bandwidth Bound: 0.0%
	CPU Utilization: 0.4% Average CPU Usage: 0.996 Out of 256 logical CPUs	CPU Utilization: 0.4% Average CPU Usage: 0.996 Out of 256 logical CPUs

There is minor improvement on L2 hit count, subsequently a small improvement on FPU utilization. But it does not change the L2 Miss Bound anyway. Furthermore, the benefits from z-filing will decrease when the thread number increases, for this application deals with a sparse matrix and there are less chances to reuse the data loaded when the blocks size get smaller.

Optimization By Reorganize Memory Layout: Nopermute

Applying permutation on the rows and columns of the training matrix is a standard pre-process step in Libmf, which can make the load more balance among the threads. However, the randomness from permutation make the cache harder to work efficiently.

Here we try skip the permutation in preproces, and on the contrary, we reorder the rows and columns by their frequency of non-zero items in each block. And following the idea of partial randomness in libmf, we rely on the dynamic scheduler randomly select blocks to make the converge rate not deteriorate much comparison with the totally random training point selection.

Thread=64, Blocks=129, Iternum=10

	With Permutation	Nopermute(ms)	Nopermute(ms)+z-filing
TrainTime/Iter(ms)	1490		1280
Test RMSE	26.1194		26.1604

It can achieve about 10% times faster by **nopermute,** but not much.

Optimization By Repeat Computation on Training Points: Repeat

Here we do repeat computation on a chunk of training data to increase the data reuse and the computation-to-memload ratio.

Parameters:

chunksize	How many points will be retrained as a group/chunk
repeatcnt	How many times the chunk data will be retrained

VTune Profiling

S64N0,xtile=3, repeat=3,chunksize=16	S64N0,xtile=3 no repeat
CPU Utilization: 24.8% Average CPU Usage: 63.546 Out of 256 logical CPUs	CPU Utilization: 21.4% Average CPU Usage: 54.850 Out of 256 logical CPUs
Back-End Bound: 42.5%	Back-End Bound: 61.4%

L2 Hit Bound: 0.088 L2 Miss Bound: 0.640 MCDRAM Flat Bandwidth Bound: 0.0% DRAM Bandwidth Bound: 0.0%	L2 Hit Bound: 0.148 L2 Miss Bound: 1.000 MCDRAM Flat Bandwidth Bound: 0.0% DRAM Bandwidth Bound: 0.0%
FPU Utilization Upper Bound: **15.3%** GFLOPS Upper Bound: 405.943 Scalar GFLOPS Upper Bound: 0.450 Packed GFLOPS Upper Bound: 405.493	FPU Utilization Upper Bound: **10.5%** GFLOPS Upper Bound: 239.319 Scalar GFLOPS Upper Bound: 0.206 Packed GFLOPS Upper Bound: 239.113
Top 5 hotspot loops (functions) by FPU usage Function CPU Time FPU Utilization Upper Bound Loop Characterization [Loop@0x41eeb2 in mf::(anonymous namespace)::SolverBase::run] 1512.121s 7.7% [Loop@0x41f1a0 in mf::(anonymous namespace)::SolverBase::run] 1268.951s 25.2% [Loop@0x41efc5 in mf::(anonymous namespace)::SolverBase::run] 1233.001s 2.2% mf::(anonymous namespace)::Block::move_next 611.510s 21.9% [Loop@0x41f1f8 in mf::(anonymous namespace)::SolverBase::run] 607.590s 41.1% [Others] 120.650s N/A*	Top 5 hotspot loops (functions) by FPU usage Function CPU Time FPU Utilization Upper Bound Loop Characterization [Loop@0x41e375 in mf::(anonymous namespace)::SolverBase::run] 1133.191s 1.0% [Loop@0x41e262 in mf::(anonymous namespace)::SolverBase::run] 495.050s 8.7% [Loop@0x41e550 in mf::(anonymous namespace)::SolverBase::run] 424.420s 23.9% [Loop@0x41e5a8 in mf::(anonymous namespace)::SolverBase::run] 196.480s 40.8% mf::(anonymous namespace)::Block::move_next 88.750s 9.6% [Others] 114.550s N/A*

From the VTUNE report, the **Repeat** technique can really increase the FPU utilization and decrease the l2 miss bound in the back-end.

TestRMSE	iter	avg time(ms)	Total(ms)	speedup	chunksizeXrepeatcnt
24.1733	17	1773	30141	1	
	9	1961	17649	1.707802142	8x1
	6	2460	14760	2.042073171	8x2
	5	3088	15440	1.952137306	8x3

This table shows the speedup by “repeat” to converge to the same level in training with different *repeatcnt* settings.

But another group experiments with different chunksize settings show little differences in terms of performance. It means, we can not benefit more from repeat on a larger chunk of points than repeat on a single point. The problem here is that repeat on a single point actually has little differences with doubling the learning rate parameter. Thus, the repeat technique doesn’t work effectively on such kind of sparse problem.

Optimization By Decrease Memory Access Latency: NUMA

Because this application is memory-bound with high l2-miss rate, we can try to decrease the memory access latency by utilize the topology information of the numa architecture.

Set KNL into SNC-4 mode, and run 4 processes in this distributed mode by model rotation.

Performance Evaluation of Harp-SGD-MF on KNL

Learning parameters
Yahoomusic Dataset
Number of Training Points 252800275
Number of Rows: 1000990
Number of columns: 624961
Number of Dimensions: 100
Lambda: 1
Epsilon: 0.0001

The results are shown in the chart below. There is (super) linear speedup on 1, 8, 16 and 32 threads. However, the speedup couldn't grow when the number of threads is higher than 32 because the computation time doesn't change much.

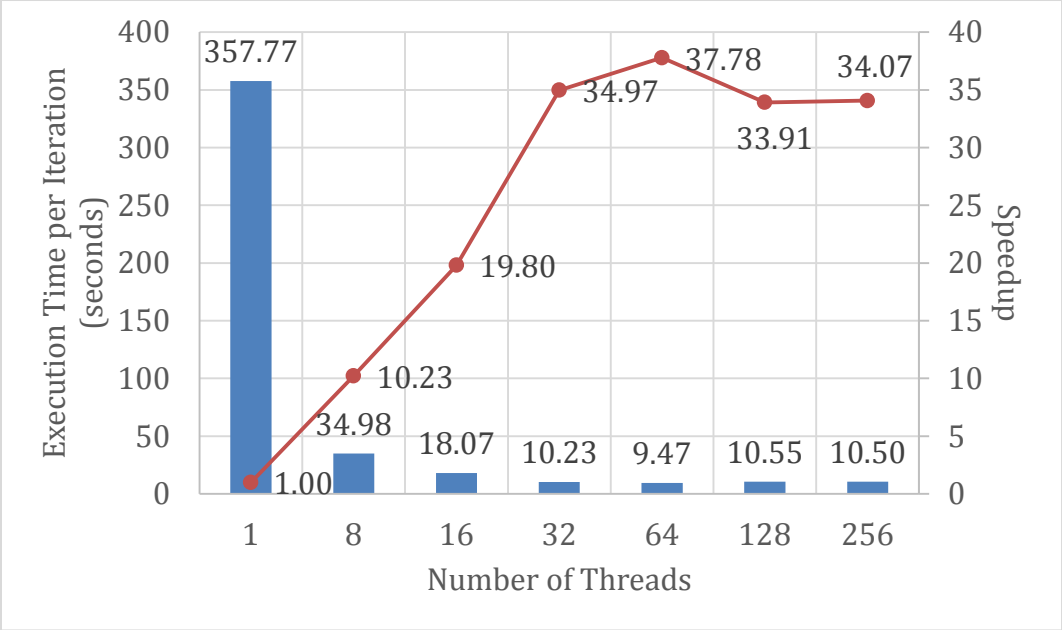


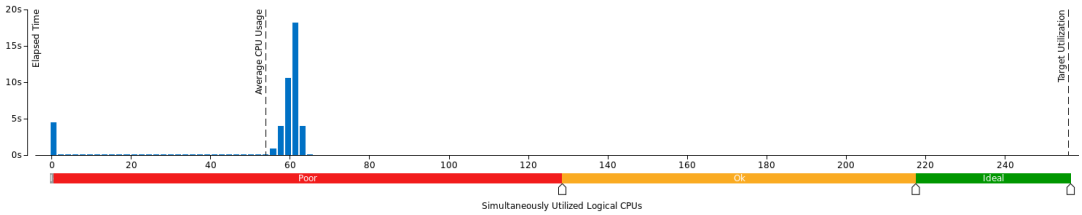
Figure 16 Performance Evaluation of Harp SGD for MF

Appendix
VTUNE Analysis Report
Libmf Parameters
- Thread number: 64
- K: 128
- Lambda: 1
- Eta: 0.0001
- AVX512: on
- MCDRAM: on

VTune--General info

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



Elapsed Time: 240.384s

Clockticks:	3,016,478,400,000
Instructions Retired:	1,421,940,000,000
CPI Rate:	2.121
MUX Reliability:	1.000
Front-End Bound:	**14.7%**
ICache Misses:	0.054
ITLB Overhead:	0.001
BACLEARS:	0.039
MS Entry:	0.020
ICache Line Fetch:	0.025
Bad Speculation:	**6.7%**
Branch Mispredict:	6.7%
SMC Machine Clear:	0.001
MO Machine Clear Overhead:	0.000
Back-End Bound:	**52.5%**
Memory Latency:	
L1 Hit Rate:	0.915
L2 Hit Rate:	0.541
L2 Hit Bound:	0.148
L2 Miss Bound:	1.000
UTLB Overhead:	0.037
SIMD Compute-to-L1 Access Ratio:	1.133
SIMD Compute-to-L2 Access Ratio:	24.810
Contested Accesses (Intra-Tile):	0.000
Page Walk:	0.003
Memory Reissues:	
Split Stores:	0.000
Loads Blocked by Store Forwarding:	0.014
Retiring:	**26.0%**
VPU Utilization:	0.980
Divider:	0.000
MS Assists:	0.107
FP Assists:	0.000
Total Thread Count:	65
Paused Time:	197.395s

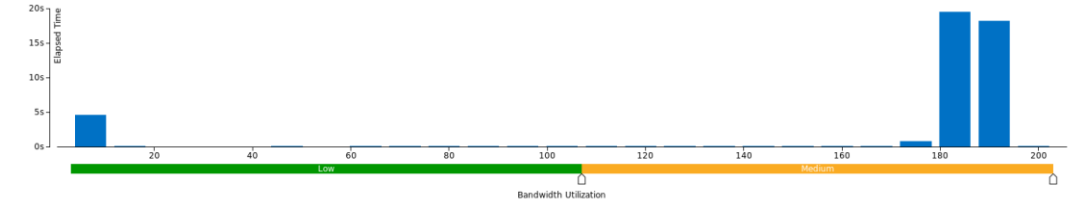
Elapsed Time: 243.072s

- CPU Time: 2408.182s
- **Memory Bound:**
  - L2 Hit Rate: 0.555
  - L2 Hit Bound: 0.148
  - L2 Miss Bound: 1.000
  - MCDRAM Flat Bandwidth Bound: 0.0%
  - DRAM Bandwidth Bound: 0.0%
- L2 Miss Count: 21,882,656,460
- Total Thread Count: 65
- Paused Time: 199.180s

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: MCDRAM Flat, GB/sec



Elapsed Time: 43.370s

GFLOPS Upper Bound: 244.341

CPU Utilization: 21.3%

Average CPU Usage: 54.604 Out of 256 logical CPUs

CPU Usage Histogram

Back-End Bound: 59.9%

- L2 Hit Bound: 0.148
- L2 Miss Bound: 1.000
- MCDRAM Flat Bandwidth Bound: 0.0%
- DRAM Bandwidth Bound: 0.0%

Bandwidth Utilization Histogram

FPU Utilization Upper Bound: 10.7%

- **GFLOPS Upper Bound: 244.341**
  - Scalar GFLOPS Upper Bound: 0.207
  - Packed GFLOPS Upper Bound: 244.134
- **Top 5 hotspot loops (functions) by FPU usage**

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	FPU Utilization Upper Bound	Loop Characterization
[Loop@0x41d0e5 in mf::{anonymous namespace}::SolverBase::run]	1093.271s	1.0%	
[Loop@0x41cfd2 in mf::{anonymous namespace}::SolverBase::run]	474.790s	8.4%	
[Loop@0x41d2c0 in mf::{anonymous namespace}::SolverBase::run]	380.900s	25.6%	
[Loop@0x41d318 in mf::{anonymous namespace}::SolverBase::run]	182.300s	44.5%	
mf::{anonymous namespace}::Block::move_next	85.080s	8.2%	
[Others]	114.020s	N/A*	

*N/A is applied to non-summable metrics.