Intel® Machine Learning Scaling Library

Developer Guide and Reference

Legal Information

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Contents

1. Introduction

Intel® Machine Learning Scaling Library (Intel® MLSL) is a library providing an efficient implementation of communication patterns used in deep learning. The library is intended for developers of deep learning frameworks, who would like to benefit from scalability in their projects.

Some of the Intel MLSL features include:

- Built on top of MPI, allows for use of other communication libraries
- Communication patters support synchronous Stochastic Gradient Descent (SGD) and its variations (AdaGrad, Momentum, etc.)
- Common API to support deep learning frameworks (Caffe*, Theano*, Torch*, etc.)

The Intel® MLSL package comprises the Intel MLSL Software Development Kit (SDK) and the Intel® MPI Library Runtime components.

This document provides usage instructions for Intel MLSL, and also its API and configuration reference. For installation instructions, system requirements, and other information, refer to the README file.

2. Using Intel® Machine Learning Scaling Library

2.1. Prerequisites

Before you start using the Intel® MLSL, make sure to set up the library environment. Use the command:

```
$ source <install_dir>/intel64/bin/mlslvars.sh
```

Here and below, `<install_dir>` is the Intel MLSL installation directory, which is `/opt/intel/mlsl_<version>.<update>.<package>` by default.

2.2. Generic Workflow

Below is a generic flow of using the Intel® MLSL:

1. Initialize the library:
   ```
   Init(&argc, &argv);
   ```
2. Set the global mini-batch size (equal to the sum of local batch sizes):
   ```
   SetMinibatchSize(<args>);
   ```
3. Depending on the type of parallelism, create an appropriate `Distribution` object:
   ```
   Distribution *d = new Distribution(<args>);
   ```
4. For each layer, create a `ComputeOp` object, as follows:
   a. Create a `ComputeOpRegInfo` object:
      ```
      ComputeOpRegInfo *cori = new ComputeOpRegInfo(<args>);
      ```
   b. Depending on the type of parallelism, add input/output feature map shapes and weights shapes to the `ComputeOpRegInfo` object:
      ```
      cori->AddInputFeatureMap(<args>); // to add input feature map
      shape
      cori->AddOutputFeatureMap(<args>); // to add output feature map
      shape
      cori->AddWeights(<args>); // to add weight shape
      ```
   c. Create a `ComputeOp` object using the `ComputeOpRegInfo` and `Distribuiton` objects:
      ```
      ComputeOp *co = new ComputeOp(d, cori);
      ```
   d. Invoke the `SetPrev()` or `SetNext()` method to link the `ComputeOp` object with the other operations in the graph:
      ```
      co->SetPrev(<args>); // or
      co->SetNext(<args>);
      ```
   e. Invoke the `Finalize()` and `AllocCommsBufs()` methods to fully initialize the `ComputeOp` object:
      ```
      co->Finalize();
      co->AllocCommsBufs();
      ```

5. In each forward propagation iteration for each layer, use `FeatureMap` objects of the `ComputeOp` object to exchange data for model parallelism:
   ```
   co->InputFeatureMap(<feature_map_index>)->CommsWait();
   co->OutputFeatureMap(<feature_map_index>)->CommsStart();
   ```
6. In each backward propagation iteration for each layer:
   a. Use `FeatureMap` objects of the `ComputeOp` object to exchange gradients with respect to data for model parallelism:
      ```
      co->OutputFeatureMap(<feature_map_index>)->CommsWait();
      co->InputFeatureMap(<feature_map_index>)->CommsStart();
      ```
   b. Use `Weights` objects of the `ComputeOp` object to exchange gradients with respect to weights for data parallelism:
      ```
      co->GetWeights(<weight_index>)->CommStartGradient();
      co->GetWeights(<weight_index>)->CommWaitGradient();
      ```
   c. In the case of distributed weight update, use `Weights` objects of the `ComputeOp` object to exchange weight increments for data parallelism:
      ```
      co->GetWeights(<weight_index>)->CommsStartWtInc();
      co->GetWeights(<weight_index>)->CommsWaitWtInc();
      ```
7. Finalize the library:
   ```
   Finalize();
   ```

The workflow described above is implemented in a sample application `mlsl_test.cpp` distributed with Intel® MLSL. You can use this file as a reference when applying Intel MLSL for your framework. See the section below for description and instructions on using the sample application.

2.3. Launching Sample Application

Intel® MLSL supplies a sample application `mlsl_test.cpp`, which demonstrates the usage of Intel MLSL API. Do the following to launch the application:

1. Set up the compiler environment. For example, for the Intel® compiler:
   ```
   $ source <compiler_install_dir>/bin/compilervars.sh intel64
   ```
2. If you have not done so, set up the Intel MLSL environment:
   ```
   $ source <install_dir>/intel64/bin/mlslvars.sh
   ```
3. Build `mlsl_test.cpp`:
   ```
   $ cd <install_dir>/test
   $ make
   ```
4. Launch the `mlsl_test` binary with `mpirun` on the desired number of nodes (`N`).
   `mlsl_test` takes two arguments:
   - `num_groups` – defines the type of parallelism, based on the following logic:
     - `num_groups == 1` – data parallelism
     - `num_groups == N` – model parallelism
     - `num_groups > 1` and `num_groups < N` – hybrid parallelism
   - `dist_update` – enables distributed weight update

Launch command examples:

```
# use data parallelism
$ mpirun –n 8 -ppn 1 ./mlsl_test 1
# use model parallelism
$ mpirun –n 8 -ppn 1 ./mlsl_test 8
# use hybrid parallelism, enable ditributed weight update
$ mpirun –n 8 -ppn 1 ./mlsl_test 2 1
```

The application is set up to run a test for two layers. In the `Forward()` call, it sets the output feature map on the 1st layer, performs the feature map exchange, and checks the input feature map for the 2nd layer. In the `Backward1()` call, it sets a gradient with respect to the input feature map for the 2nd layer, performs the feature map exchange, and checks the gradient with respect to the output feature map for the 1st layer. In the `Backward2()` call, it sets gradients with respect to weights and starts the exchange of gradients with respect to weights. In the `Update()` call, it finishes the exchange of gradients with respect to weights, checks the gradients accumulation, and modifies the weights, or starts the exchange of weight increments (in the case of distributed weight update). At the next iteration in the `Forward()` call, in the case of distributed weight update, it finishes the exchange of weight increments, modifies and checks the weights.

The application prints parameters for input and output feature maps and weights, whether the communication is required, and what type of communication is required. The test status is printed as PASSED or FAILED. You can `grep` for FAILED to see if a test failed.

3. API Reference

NOTE: The API provided in this document refers to the Intel® MLSL 2017 Beta release and will change in the next stable release.

3.1. Global Functions

int GetVersion()

Returns the Intel MLSL API version. Can be called before the library initialization.

int Init(int *argc, char **argv[])

Initializes the library.

int Finalize()

Finalizes the library. Cleans up and frees all the internally allocated memory.

int SetMinibatchSize(int globalMinibatchSize)

Sets the global mini-batch size. Should be called before any `ComputeOp` creation.

Parameters:

- `globalMinibatchSize` – global mini-batch size – the sum of local mini-batch sizes from all nodes

int GetNodeId()

Returns the global node ID (number of the corresponding MPI rank).

int GetNumNodes()

Returns the number of nodes. Equal to the number of MPI ranks.

void Barrier()

Synchronizes processes on all the nodes.

int GetError()

Returns the last error number.

void *Alloc(size_t size, int align)

Intel MLSL specific `malloc()` to allocate communication buffers.

Parameters:

- `size` – bytes to allocate
- `align` – memory alignment

void Free(void *ptr)

Intel MLSL specific `free()` to free communication buffers.

Parameters:

- `ptr` – pointer to memory

3.2. Enumerations

DataType

The data type used for representing input/output feature maps and weights.

Values:

- `DT_FLOAT = 1`
- `DT_DOUBLE = 2`

OpType

Compute operation types (buckets).

Each operation type defines specific relationship between the input and output feature maps and associated weights.

Values:

(abbreviations: IFM – input feature map, OFM – output feature map)

- `COMP_OP_TYPE_CC = 0`
  Cross-correlation – IFM and OFM independent and has weights
- `COMP_OP_TYPE_BIAS = 1`
  BIAS – the same IFM and OFM (dependent) but has weights
- `COMP_OP_TYPE_ACT = 2`
  Activation operation – the same IFM and OFM and no weights
- `COMP_OP_TYPE_POOL = 3`
  Activation operation – the same IFM and OFM and no weights
- `COMP_OP_TYPE_SPLIT = 4`
  OFM depends on IFM (= OFM_1 + OFM_2 ...) and no weights
- `COMP_OP_TYPE_CONCAT = 5`
  OFM depends on IFM_1+IFM_2+... and no weights
- `COMP_OP_TYPE_BCAST = 6`
  OFM_1 = IFM, OFM_2 = IFM, ... and no weights
- `COMP_OP_TYPE_REDUCE = 7`
  OFM = IFM_1 + IFM_2 + ... and no weights
- `COMP_OP_TYPE_DATA = 8`
  Only OFM, no IFM
- `COMP_OP_TYPE_EVAL = 9`
  Only IFM, no OFM

3.3. Classes

BlockInfo

A class to hold block information for packing/unpacking.

Methods:

int MBStart()

Returns the start of the mini-batch portion.

int MBLen()

Returns the length of the mini-batch portion.

int FMStart()

Returns the start of the feature map portion.

int FMLen()

Returns the length of the feature map portion.

int FMSize()

Returns the size of a feature map.

DataType FMType()

Returns the type of the feature map elements.

int BufOffset()

Returns the offset within the communication buffer where to pack to/unpack from.

CommsBuf

A class to hold a communication buffer used for feature map/weight exchange.

Methods:

CommsBuf(size_t size)

The constructor.

Parameters:

- `size` – bytes to allocate

size_t Size()

Returns the buffer size in bytes.

int Alloc()

Allocates a buffer.

void Free()

Frees the buffer.

int SetPtr(void *ptr)

Sets the internal buffer pointer to a new value.

Parameters:

- `ptr` – new pointer value

void *GetPtr()

Returns the internal buffer pointer.

FeatureMap

A wrapper class for layer input and output feature maps. A `FeatureMap` object may contain one or more feature maps.

Methods:

int GlobalLen()

Returns the total number of feature maps.

int GlobalOffset()

Returns the offset of the local portion of feature maps in the total number of feature maps.

int LocalLen()

Returns the local number of feature maps.

CommsBuf *CBuf()

Returns a communication buffer used for feature map data/gradient exchange.

int NumPackBlocks()

Returns the number of `BlockInfo` objects for packing, corresponding to this `FeatureMap` instance.

int NumUnpackBlocks()

Returns the number of `BlockInfo` objects for unpacking, corresponding to this `FeatureMap` instance.

BlockInfo *GetPackBlock(int idx)

Returns the `BlockInfo` object for packing with the index `idx`.

BlockInfo *GetUnpackBlock(int idx)

Returns the `BlockInfo` object for unpacking with the index `idx`.

DataType GetDType()

Returns the type of a feature map.

int FMSize()

Returns the size of a feature map in bytes.

int DoComms(void *buf)

Starts a blocking feature map exchange.

Parameters:

- `buf` – buffer containing the packed feature maps

int CommStart(void *buf)

Starts a non-blocking feature map exchange.

Parameters:

- `buf` – buffer containing the packed feature maps

void *CommWait()

Waits for completion of the feature map exchange.

Returns a pointer to the buffer containing the feature maps to be unpacked.

Weights

A wrapper class for weights. A `Weights` object may contain one or more weights.

Methods:

int GlobalLen()

Returns the total number of weights.

int GlobalOffset()

Returns the offset of the local portion of weights in the total number of weights.

int LocalLen()

Returns the local number of weights.

int OwnedLen()

Returns the number of weights on which this node performs synchronous Stochastic Gradient Descent (SGD). Applicable only when `distributedUpdate == true`.

int OwnedStart()

Returns the weight offset with respect to `OwnedLen().`

DataType GetDType()

Returns the type of a weight.

Int WTSize()

Returns the size of a weight in bytes.

int DoDelWt(void *buf)

Starts a blocking weight gradient exchange.

Parameters:

- `buf` – buffer containing the gradient

int DoWtInc(void *buf)

Starts a blocking weight increment exchange.

Parameters:

- `buf` – buffer containing the increment

int CommStartDelWt(void *buf)

Starts a non-blocking weight gradient exchange.

Parameters:

- `buf` – buffer containing the gradient

int CommStartWtInc(void *buf)

Starts a non-blocking weight increment exchange.

Parameters:

- `buf` – buffer containing the increment

void *CommsWaitDelWt()

Waits for completion of the weight gradient exchange.

Returns a pointer to the buffer containing the aggregated weight gradient.

void *CommsWaitWtInc()

Waits for completion of the weight increment exchange.

Returns a pointer to the buffer containing the increment obtained with the synchronous SGD.

Distribution

A class to hold the type of parallelism and the parameters for hybrid parallelism.

Methods:

Distribution(int nMBParts, int nFMParts)

The constructor.

Parameters:

- `nMBParts` – number of partitions in the mini-batch
- `nFMParts` – number of partitions in the input feature maps

int GetMBGroupSize()

Returns the number of partitions in the mini-batch.

int GetFMGroupSize()

Returns the number of partitions in the input feature map.

int GetMBGroupId()

Returns the group ID with respect to the mini-batch partitioning.

int GetFMGroupId()

Returns the group ID with respect to the input feature-map partitioning.

ComputeOpRegInfo

A class to hold a compute operation's registration information. All the input and output feature maps and weights (if any) should be added and validated before calling the `ComputeOp` constructor.

Methods:

ComputeOpRegInfo(OpType operationType, std::string name = "")

The constructor.

Parameters:

- `operationType` – operation type describing relationship between the input and output feature maps and associated weights
- `name` – operation name

void SetName(const char *name)

Sets the compute operation name (for debugging purposes).

Parameters:

- `name` – operation name

int AddInputFeatureMap(int numFeatureMaps, int featureMapSize, DataType featureMapType)

Adds an input feature map shape to the compute operation.

Parameters:

- `numFeatureMaps` – number of feature maps
- `featureMapSize` – size of a feature map in `DataType` elements
- `featureMapType` – data type representing the feature map

int AddOutputFeatureMap(int numFeatureMaps, int featureMapSize, DataType featureMapType)

Adds an output feature map shape to the compute operation.

Parameters:

- `numFeatureMaps` – number of feature maps
- `featureMapSize` – size of a feature map in `DataType` elements
- `featureMapType` – data type representing the feature map

int AddWeights(int numWeights, int weightSize, DataType weightType, bool distributedWeightUpdate = false)

Adds a weight to the compute operation.

Parameters:

- `numWeights` – number of weights
- `weightSize` – size of a weight in `DataType` elements
- `weightType` – data type representing the weight
- `distributedWeightUpdate` – use the distributed weight update

int Validate()

Validates the feature map and weight shapes.

ComputeOp

A class to hold the compute operation parameters and related communication information.

Methods:

int SetPrev(ComputeOp *prev, int id = 0, int prevOpId = -1)

Sets the previous compute operation in the graph.

Parameters:

- `id` – index of input if the operation has multiple inputs
- `prevOpId` – index of input within the previous operation

int SetNext(ComputeOp *next, int id = 0, int nextOpId = -1)

Sets the next compute operation in the graph.

Parameters:

- `id` – index of output if the operation has multiple outputs
- `nextOpId` – index of output within the next operation

int Finalize()

Finalizes the operation creation. `Finalize()` must be called after all the `SetPrev()` and `SetNext()` calls are made and before any communication operation is performed.

std::string Name()

Returns the compute operation name.

Distribution *GetDistribution()

Returns the distribution used by the current compute operation.

int GlobalMinibatchLen()

Returns the length of the global mini-batch.

int LocalMinibatchLen()

Returns the length of the local mini-batch portion.

int GlobalMinibatchOffset()

Returns the start of the local mini-batch portion within the global mini-batch.

int NumInputFeatureMaps()

Returns the number of input feature maps for the current compute operation.

FeatureMap *InputFeatureMap(int id)

Returns the input feature map by ID.

int NumOutputFeatureMaps()

Returns the number of output feature maps for the current compute operation.

FeatureMap *OutputFeatureMap(int id)

Returns the output feature map by ID.

bool HasWeights()

Returns `true` if this compute operation has `Weights` objects, `false` otherwise.

int NumWeights()

Returns the number of `Weights` objects associated with this compute operation.

Weights *GetWeights(int id)

Returns the `Weights` object by ID.

int AllocCommsBufs()

Internally allocates memory required for the communication.

int FreeCommsBufs()

Frees the memory previously allocated for the communication.

4. Environment Variables

MLSL_LOG_LEVEL

Syntax

`MLSL_LOG_LEVEL=<level>`

Arguments

`<level>`	Logging information level
0	The *error* level. Prints out critical errors that lead to application termination. This is the default value.
1	The *informational* level. Prints out informational messages about the application progress.
2	The *debug* level. Prints out detailed informational messages that are most useful to debug an application.
3	The *trace* level. Prints out more detailed informational messages than in the *debug* level.

Description

Set this environment variable to print logging information about the application. Higher levels include information logged in the lower levels.

MLSL_NUM_SERVERS

Syntax

`MLSL_NUM_SERVERS=<nservers>`

Arguments

`<nservers>`	The number of servers per node.
`>= 0`	The default value is `4`. The maximum value is `16`.

Description

Set this environment variable to define the number of endpoint servers per node.

NOTE: Each server is a separate process, which uses CPU resources. Take that into account when setting the `OMP_NUM_THREADS` variable for OpenMP*. The recommended value for `OMP_NUM_THREADS` is `max_cores - num_servers`, where `max_cores` is the maximum number of cores, and `num_servers` is the number of endpoint servers per node.

MLSL_SERVER_AFFINITY

Syntax

`MLSL_SERVER_AFFINITY=<proclist>`

Arguments

`<proclist>`	A comma-separated list of logical core numbers. The server with the `i`-th index is pinned to the `i`-th core in the list. The number should not exceed the number of cores on the node.
`n-1,n-2,n-3,n-4,…`	This is the default value – servers are pinned to cores in the reversed order. `n` – the number of available cores

Description

Set this environment variable to define the processor core affinity for endpoint servers for best performance.

NOTE: The recommended values can be retrieved by querying `/proc/interrupts`:

- For the Intel® Omni-Path Fabric you are recommended to map servers to the cores handling the send direct memory access (SDMA) interrupts.
- For Ethernet you are recommended to map servers to the cores handling the `Tx/Rx` interrupts.

MLSL_HEAP_SIZE_GB

Syntax

`MLSL_HEAP_SIZE_GB=<ngigabytes>`

Arguments

`<ngigabytes>`	The size of the heap memory maintained by Intel MLSL in gigabytes.
`> 0`	The default value is `32`. The maximum value depends on the maximum memory available on the node .

Description

Set this environment variable to define the size of the heap memory for Intel MLSL.

NOTE: The `MLSL_HEAP_SIZE_GB` value should be large enough to hold all the communication buffers.