Intel® Machine Learning Scaling Library

Developer Guide

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 is a library providing an efficient implementation of communication patterns used in deep learning. Some of the library features include:

- Built on top of MPI, allows for use of other communication libraries
- Optimized to drive scalability of communication patterns
- Works on various interconnects: Intel® Omni-Path Architecture, InfiniBand*, and Ethernet
- 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 its configuration reference. For installation instructions, system requirements, and other information, refer to the README file supplied with the library.

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
```

This is sets up the environment both for the C/C++ and Python* bindings.

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

2.2. Generic Workflow

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

1. Initialize the library:

   ```
   Environment::GetEnv().Init(&argc, &argv);
   ```

2. Create a `Session` and a `Distribution` objects:

   ```
   Session *s = Environment::GetEnv().CreateSession();
   Distribution *d = Environment::GetEnv().CreateDistribution(<args>);
   ```

3. Set the global mini-batch size (equal to the sum of local batch sizes):

   ```
   s->SetGlobalMinibatchSize(<args>);
   ```

4. For each layer, create an `Operation` object, as follows:
   a. Create an `OperationRegInfo` object:

      ```
      OperationRegInfo *ori = s->CreateOperationRegInfo(<args>);
      ```

   b. Depending on the type of parallelism, add input/output activation shapes or shapes of parameters (weights or biases) to the `OperationRegInfo` object:

      ```
      ori->AddInput(<args>); // to add input activation shape
      ori->AddOutput(<args>); // to add output activation shape
      ori->AddParameterSet(<args>); // to add weight shape
      ori->AddParameterSet(<args>); // to add bias shape
      ```

   c. Create an `Operation` object using the `OperationRegInfo` and `Distribuiton` objects:

      ```
      size_t opIdx = s->AddOperation(ori, d);
      Operation *op = s->GetOperation(opIdx);
      ```

5. Invoke the `Commit()` method of the `Session` object:

```
s->Commit();
```

6. In each forward propagation iteration for each layer, use `Activation` objects of the `Operation` object to exchange data for model parallelism:

```
op->GetInput(<activation_index>)->WaitComm();
op->GetOutput(<activation_index>)->StartComm();
```

7. In each backward propagation iteration for each layer:
    a. Use `Activation` objects of the `Operation` object to exchange gradients with respect to data for model parallelism:

    ```
    op->GetOutput(<activation_index>)->WaitComm();
    op->GetInput(<activation_index>)->StartComm();
    ```

    b. Use `ParameterSet` objects of the `Operation` object to exchange gradients with respect to parameters (weights or biases) for data parallelism:

    ```
    op->GetParameterSet(<parameter_index>)->StartGradientComm();
    op->GetParameterSet(<parameter_index>)->WaitGradientComm();
    ```

    c. In the case of distributed parameter update, use `ParameterSet` objects of the `Operation` object to exchange parameter increments for data parallelism:

    ```
    op->GetParameterSet(<parameter_index>)->StartIncrementComm();
    op->GetParameterSet(<parameter_index>)->WaitIncrementComm();
    ```

8. Delete the `Session` and `Distribution` objects:

```
Environment::GetEnv().DeleteSession(s);
Environment::GetEnv().DeleteDistribution(d);
```

9. Finalize the library:

```
Environment::GetEnv().Finalize();
```

The workflow described above is implemented in a sample application `mlsl_test.cpp` distributed with Intel® MLSL. There is also a similar sample Python script `mlsl_test.py`, implementing the same workflow.

You can use this sample as a reference when applying Intel MLSL for your framework (for C++ or Python, respectively). See the section below for description and instructions on using the sample application.

For detailed API description, refer to the *Intel MLSL API Reference* at <install_dir>/doc/API_Reference.htm.

2.3. Launching Sample Application

Intel® MLSL supplies a sample application, `mlsl_test.cpp` or `mlsl_test.py`, which demonstrates the usage of Intel MLSL API.

2.3.1. Launching the Sample

1. For the C++ sample, build `mlsl_test.cpp`:

```
$ cd <install_dir>/test
$ icpc -O2 -I<install_dir>/intel64/include/ -L<install_dir>/intel64/lib
-lmlsl -lmpi -ldl -lrt -lpthread -o mlsl_test mlsl_test.cpp
```

2. Launch the `mlsl_test` binary or the `mlsl_test.py` script with `mpirun` on the desired number of nodes (`N`).
   The application takes one argument `num_groups`, which will define the type of parallelism, based on the following logic:
   - `num_groups = 1` – data parallelism, for example:
     ```
     $ mpirun -n 8 -ppn 1 ./mlsl_test[.py] 1
     ```
   - `num_groups = N` – model parallelism, for example:
     ```
     $ mpirun -n 8 -ppn 1 ./mlsl_test[.py] 8
     ```
   - `num_groups > 1` and `num_groups < N` – hybrid parallelism, for example:
     ```
     $ mpirun -n 8 -ppn 1 ./mlsl_test[.py] 2
     ```

2.3.2. Sample Description

The application is set up to run a test for two layers. It sets output on the 1st layer and checks input for the 2nd layer in an `fprop()` call. Similarly, for the `bprop()` call, it sets a gradient with respect to input for the 2nd layer and checks the gradient with respect to output for the 1st layer. For weights, it sets gradients with respect to weights, checks the gradients accumulation, modifies weights in a `wtinc()` call, and then verifies the expected values in an `fprop()` call for both layers.
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.

2.4. Statistics Collection

Intel® MLSL statistics allow you to monitor the time spent by operations in Intel MLSL during the computation and communication phases.

You can start and stop monitoring operations using the `MLSL::Statistics` API.

You can also fetch, print and reset Intel MLSL statistics data to understand whether your neural network is computation or communication bound.

Intel MLSL provides the following statistics:

- The total time spent by all operations in the computation phase
- The total time spent by all operations in the communication phase
- The total time expected to spend in the communication phase in an isolated environment (that is, time spent in blocking communication calls with no computation involved between communication calls)
- The total communication size of all operations
- The time spent per operation in the computation phase
- The time spent per operation in the communication phase

- The expected time spent per operation in the communication phase in an isolated environment
- The communication size for each operation

By analyzing communication time collected in an isolated environment you can understand the impact of the computation phase on the communication phase.

To enable or disable statistics collection, use the `MLSL_STATS` environment variable.

3. Environment Variables

MLSL_ROOT

Syntax

`MLSL_ROOT=<path>`

Arguments

<path>	Installation directory of the Intel® MLSL.

Description

Set this environment variable to specify the installation directory of the Intel® MLSL.

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

MLSL_STATS

Syntax

`MLSL_STATS=`*`<arg>`*

Arguments

<arg>	Binary indicator
`1`	Enable statistics collection.
`0`	Disable statistics collection. This is the default value.

Description

Set this environment variable to enable statistics collection. See Statistics Collection for details.

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_SERVER_CREATION_TYPE

Syntax

`MLSL_SERVER_CREATION_TYPE=<arg>`

Arguments

<arg>	Launch method index
0	Use MPI spawn API
1	Use `mpirun`

Description

Set this environment variable to specify a method for launching endpoint servers.

MLSL_HOSTNAME_TYPE

Syntax

`MLSL_HOSTNAME_TYPE=<arg>`

Arguments

<arg>	Index of host name retrieval method
0	Use `MPI_Get_processor_name`
1	Use `gethostname` and `getaddrinfo`
2	Use the interface IP address

Description

Set this environment variable to specify a method of retrieving host names of endpoint servers.

If using the value `2`, make sure to also specify the interface using one of the variables below:

- `MLSL_IFACE_NAME` – select an interface by prefix name
- `MLSL_IFACE_IDX` – select an interface by index

MLSL_HOSTNAME

Syntax

`MLSL_HOSTNAME=<name>`

Arguments

<name>	The hostname.

Description

Set this environment variable to specify the hostname to be used for launching endpoint servers. The variable must be set for each Intel MLSL process.

For example, to set the variable through Intel® MPI Library `-gtool` option:

```
$ mpirun ... -gtool "env MLSL_HOSTNAME=name1:0; env MLSL_HOSTNAME=name2:1"
```

MLSL_ MSG_PRIORITY

Syntax

`MLSL_MSG_PRIORITY=<arg>`

Arguments

<arg>	Binary indicator
`1`	Enable reversed order of communication calls execution.
`0`	Disable reversed order of communication calls execution. This is the default value.

Description

Set this environment variable to enable executing communication calls in the order opposite to the order they are started.

When performing backward propagation, the gradient reduction communication generally starts from the last layers of the network topology finishing at the first ones. With this experimental feature enabled, the communication for first layers has a higher priority, which allows layer one to minimize the time spent in SGD and start a forward pass for the next iteration as soon as possible.

NOTE: This feature is experimental and has the following limitations: it can be enabled only for a specific implementation of the `Allreduce` algorithm and only when the number of communicating processes is a power of two.

MLSL_ MSG_PRIORITY_THRESHOLD

Syntax

`MLSL_MSG_PRIORITY_THRESHOLD=<arg>`

Arguments

<arg>	Message size in bytes
`> 0`	The default value is 10000

Description

Set this environment variable to define the lower threshold of message sizes where the experimental priority feature is enabled. The variable only takes effect when `MLSL_MSG_PRIORITY` is set.