Using Intel® Math Kernel Library (Intel® MKL) to Program for the Intel® Xeon Phi™ Coprocessor

Difficulty Level: Basic-Medium

Estimated Time to Complete: 30 minutes

LAB 1: SGEMM with Automatic Offload

Matrix multiplication is a standard problem in HPC. This computation is exemplified in the Basic Linear Algebra Subroutine (BLAS) function SGEMM (single-precision general matrix multiplication). In this lab we use the SGEMM function from Intel MKL to compute

C = alpha * A * B + beta * C,

Where A, B, and C are matrices of dimension N x N, alpha and beta are scalars.

This lab exercises Intel MKL's Automatic Offload usage model of offloading computation to Intel Xeon Phi coprocessor.

1. Understand the source code.

Open in an editor the source code "~/lab_workspace/MKL/lab_SGEMM_000/mkl_sgemm_ao.cpp".

The code has 3 sections: "executing on HOST", "AO executing", and "executing on TARGET". The last section is originally commented out. Pay attention to the following things while reading the code:

- How is SGEMM called? Refer to Intel MKL Reference Manual for descriptions for SGEMM.
- How is Automatic Offload enabled? What are the two ways of enabling Automatic Offload?

2. Build and run the program.

Follow these steps to build and run the program:

- Set up the build environment.

```
[linux_prompt]$ source /opt/intel/composerxe/bin/compilervars.sh intel64
```

- Build the program.

```
[linux_prompt]$ icc -mkl mkl_sgemm_ao.cpp
```

- Run the program with various matrix sizes (e.g. 1000 and 3000), and note down the performance numbers.

```
[linux_prompt]$ ./a.out <N>
```

- Set the following environment variables and repeat the tests.

```
MIC_ENV_PREFIX=MIC
MIC_USE_2MB_BUFFERS=16K
MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[1-236:1]
KMP_AFFINITY=granularity=fine,compact,1,0
```

Can you explain the performance numbers observed for host executions and AO executions? Can you explain the purposes of the environment variables?

3. Extend the code to include an SGEMM call that is entirely executed on target.
   - Find the section labeled with "TODO" in the source code. Uncomment the section.
   - Insert a call to the Intel MKL support function "mkl_mic_set_workdivision( )" at the beginning of the uncommented section.
   - Save the file. Build and run the program. Repeat the tests in step 2.

Among host execution, AO execution, and target execution, which one gives the best performance? Why?

LAB 2: 1D FFT with pragma offload and Native Execution

In this lab, we learn how to offload Intel MKL function calls using the offload pragma. We will use the offload pragma to control memory allocation/de-allocation on target, data transferring, data persistence, and function offload.

In addition to an offload execution, we will also see how to build and run a program natively.

The example in this lab computes an in-place forward transformation, and an in-place backward transformation, for a 1D input array of one million single precision values.

1. Understand the source code.

Open in an editor the source code
"~/lab_workspace/MKL/lab_FFT_001/mkl_dft_1d.cpp".

The code is based on the DFTI interface for FFT provided by Intel MKL. Please refer to Intel MKL Reference Manual for details about the DFTI interface. The flow of computation has the following stages:

- Create a descriptor by calling function DftiCreateDescriptor( ). The descriptor is an opaque object that captures the configuration of the transforms, such as the rank, size, number of transforms, input and output data, etc.
- Commit the descriptor with a call to DftiCommitDescriptor( ).
- Compute the forward transform by calling DftiComputeForward( ) with the descriptor.
- Compute the backward transform by calling DftiComputeBackward( ) with the same descriptor.
- Deallocate the descriptor with a call to DftiFreeDescriptor( ).

2. Build and run the program on host.

Without modifying the original code provided in this lab, build and run it on host first just to be sure it works and produces the correct results.

```
[linux_prompt]$ icc -no-offload -mkl mkl_dft_1d.c
[linux_prompt]$ ./a.out
```

Note the "-no-offload" option on the compile command line. What is it for?

3. Modify the code to offload DFTI function calls to target.

Make a copy of the original code. All code changes in this step should be done in the copy. We will keep the original code for use in the next exercise.

- Find in the code all sections labeled with "TODO". Each "TODO" is a comment with hints on what code changes are needed.
- For each "TODO" section, make code changes by following the hints. Most of the time, it requires you to insert a properly formed pragma to

perform a memory allocation/de-allocation on the target, or to offload a function call. Pay close attention to the hints about memory reuse and data persistence.

- Compare your modified code with the provided solution code. The solution code shows just one way to insert offload pragmas. You are welcome to compare and comment on your solution and the provided solution.

4. Build and run the program for offload.

Build either your modified code or the provided solution code for offload and run it. Observe the performance numbers.

```
[linux_prompt]$ icc  -mkl mkl_dft_1d.c
[linux_prompt]$ ./a.out
```

Note the "-no-offload" option is not passed on the command line, because the code now contains portions to be offloaded to the target.

What performance numbers do you see? Set the following environment variables and run again:

```
MIC_ENV_PREFIX=MIC
MIC_USE_2MB_BUFFERS=16K
MIC_KMP_AFFINITY=scatter,granularity=fine
```

What performance numbers do you see now? Can you explain the purposes of the environment variables?

5. Build the original code for native execution.

The original code (without offload pragmas) can be easily built for running exclusively on the target:

```
[linux_prompt]$ icc  -mmic -mkl mkl_dft_1d.c
```

Note the "-mmic" option is used. We now manually upload the binary executable and dependent libraries to the target, then ssh into the target and run from there.

```
[linux_prompt]$ scp $MKLROOT/../compiler/lib/mic/libiomp5.so mic0:/tmp
[linux_prompt]$ scp $MKLROOT/lib/mic/libmkl_core.so mic0:/tmp
[linux_prompt]$ scp $MKLROOT/lib/mic/libmkl_intel_thread.so mic0:/tmp
[linux_prompt]$ scp $MKLROOT/lib/mic/libmkl_intel_lp64.so mic0:/tmp
[linux_prompt]$ scp ./a.out mic0:/tmp
[linux_prompt]$ ssh mic0 LD_LIBRARY_PATH=/tmp /tmp/a.out
```

Note the performance numbers in the output. Then, run again by setting and passing across the environment variable KMP_AFFINITY:

```
[linux_prompt]$ ssh mic0 KMP_AFFINITY=scatter,granularity=fine
LD_LIBRARY_PATH=/tmp /tmp/a.out
```

Do you see better performance this time? Can you explain why?

LAB 3: Intel MKL LINPACK for native execution

Intel MKL provides a LINPACK benchmark, which is a generalization of the LINPACK 1000 benchmark. It solves a dense double precision system of linear equations (Ax=b), measures the amount of time it takes to factor and solve the system, and then reports the performance. This lab exercises native execution on Intel Xeon Phi coprocessor using a pre-built LINPACK executable.

1. Upload LINPACK executable, input data, and dependent library to target.

Change directory to "~/lab_workspace/MKL/lab_LINPACK_002". Then, upload the following files to the /tmp directory on the target (You may need root privilege for uploading to target):

```
[linux_prompt]$ scp ./xlinpack_mic mic0:/tmp
[linux_prompt]$ scp ./lininput_mic mic0:/tmp
[linux_prompt]$ scp ./libiomp5.so mic0:/tmp
[linux_prompt]$ scp ./runme_mic mic0:/tmp
```

- "xlinpack_mic" is the executable pre-built for native execution on the coprocessor.
- "lininput_mic" is the input file.
- "libiomp5.so" is the OpenMP runtime library needed for native execution.
- "runme_mic" is a script for launching the execution.

2. Log into target as root and launch LINPACK benchmark execution.

```
[linux_prompt]$ ssh mic0 LD_LIBRARY_PATH=/tmp /tmp/runme_mic
```

It takes a while for the benchmarking to complete. When it finishes the output is stored in "lin_mic.txt".

3. Fine tune the execution by setting thread affinity on target.

Set OpenMP environment variables for number of threads and thread affinity, and then repeat the test (Assuming a 61-core coprocessor):

```
KMP_AFFINITY=granularity=fine,compact
OMP_NUM_THREADS=244
```

Do you see a difference in performance?

4. BONUS: Enable huge paging for memory allocation on target.

By default, memory allocation on coprocessor uses small page size (4KB). Many Intel MKL functions benefit from large pages. We can use the libhugetlbfs library to enabling huge paging on the coprocessor. The libhugetlbfs library is an independent open source project (http://libhugetlbfs.sourceforge.net/). This lab includes a specially built libhugetlbfs.

- On host, change directory to "~/lab_workspace/MKL/tlb". Read carefully the document "readme.txt" to understand how to use libhugetlbfs.
- Upload (using micput) libhugetlbfs.so to the target.
- Log into the target. Follow steps 1 – 4 in "readme.txt" to enable huge paging on target.
- Rerun the LINPACK benchmark.

Do you see performance improvements?

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