A quick overview of Intel MKL
Usage of MKL on Xeon Phi
- Compiler Assisted Offload
- Automatic Offload
- Native Execution
Hands-on & Performance
Useful links where do we find more information?
What is the Intel MKL?

- Math library for C and Fortran
- Includes
  - BLAS
  - LAPACK
  - ScaLAPACK
  - FFTW
  - …
- Containing optimized routines
  - For Intel CPUs and MIC architecture
- All MKL functions are supported on Xeon Phi

But optimized at different levels
Execution Models on Intel MIC Architectures

Multicore Xeon

MKL AO & CAO

Multicore Hosted
General purpose serial and parallel computing

Offload
Codes with highly-parallel phases

Symmetric
Codes with balanced needs

Many Core Hosted
Highly-parallel codes

Many-core MIC

MKL Native

Intel MIC Programming Workshop @ Ostrava
allalen@lrz.de
MKL Usage In Accelerator Mode

• Compiler Assisted Offload
  • Offloading is explicitly controlled by compiler pragmas or directives.
  • All MKL functions can be inserted inside offload region to run on the Xeon Phi (In comparison, only a subset of MKL is subject to AO).
  • More flexibility in data transfer and remote execution management.

• Automatic Offload Mode
  • MKL functions are automatically offloaded to the accelerator.
  • MKL decides:
    • When to offload
    • Work division between host and targets
  • Data is managed automatically

• Native Execution
MKL Usage In Accelerator Mode

- Compiler Assisted Offload
  - Offloading is explicitly controlled by compiler pragmas or directives.
  - All MKL functions can be inserted inside offload region to run on the Xeon Phi (in comparison, only a subset of MKL is subject to AO).
  - More flexibility in data transfer and remote execution management.

- Automatic Offload Mode
  - MKL functions are automatically offloaded to the accelerator.
  - MKL decides:
    - When to offload
    - Work division between host and targets
  - Data is managed automatically

- Native Execution
How to use CAO

- The same way you would offload any function call to MIC
- An example in C:

```c
#pragma offload target(mic) \
  in(transa, transb, N, alpha, beta) \ 
  in(A:length(matrix_elements)) \ 
  in(B:length(matrix_elements)) \ 
  in(C:length(matrix_elements)) \ 
  out(C:length(matrix_elements) alloc_if(0))
{
  sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,&beta, C, &N);
}
```
How to use CAO

• An example in Fortran:

```fortran
!DEC$ ATTRIBUTES OFFLOAD : TARGET( MIC ) :: SGEMM
!DEC$ OMP OFFLOAD TARGET( MIC ) &
!DEC$ IN( TRANSA, TRANSB, M, N, K, ALPHA, BETA, LDA, LDB, LDC ), &
!DEC$ IN( A: LENGTH( NCOLA * LDA ) ), &
!DEC$ IN( B: LENGTH( NCOLB * LDB ) ), &
!DEC$ INOUT( C: LENGTH( N * LDC ) )

!$OMP PARALLEL SECTIONS
!$OMP SECTION
   CALL SGEMM( TRANSA, TRANSB, M, N, K, ALPHA, &A, LDA, B, LDB BETA, C, LDC )
!$OMP END PARALLEL SECTIONS
```
Tips for Using Compiler Automatic Offload

- Use larger (>2MB) pages for data transferring;

  e.g. `~$export MIC_USE_2MB_BUFFERS=50M`

- This means that for any array allocation larger than 50MB, uses huge pages
MKL Usage In Accelerator Mode

• Compiler Assisted Offload
  • Offloading is explicitly controlled by compiler pragmas or directives.
  • All MKL functions can be inserted inside offload region to run on the Xeon Phi (In comparison, only a subset of MKL is subject to AO).
  • More flexibility in data transfer and remote execution management.

• Automatic Offload Mode
  • MKL functions are automatically offloaded to the accelerator.
  • MKL decides:
    • When to offload
    • Work division between host and targets
  • Data is managed automatically

• Native Execution
How to Use Automatic Offload

- User does not have to change the code at all
- Either by calling the function mkl_mic_enable() or by setting the following environment variable
  
  ```
  ~$export MKL_MIC_ENABLE=1
  ```

- In Intel MKL 11.0.2 the following functions are enabled for automatic offload:

  **Level-3 BLAS functions**
  - *GEMM* (for M,N > 2048, k > 256)
  - *TRSM* (for M,N > 3072)
  - *TRMM* (for M,N > 3072)
  - *SYMM* (for M,N > 2048)

  **LAPACK functions**
  - LU (M,N > 8192)
  - QR
  - Cholesky
How to Use Automatic Offload

- Blas only: work can be divided between host and device using

  \[ \text{mkl\_mic\_set\_workdivision} (\text{TARGET\_TYPE, TARGET\_NUMBER, WORK\_RATIO}) \]

- What if there doesn’t exist a MIC card in the system?
  - Runs on the host as usual \textit{without any penalty} !!

- Users can use AO for some MKL calls and use CAO for others in the same program
  - Only supported by Intel compilers
  - Work division must be set explicitly for AO, otherwise, all MKL AO calls are executed on the host
Automatic Offload Mode Example

#include “mkl.h”
err = mkl_mic_enable();

//Offload all work on the Xeon Phi
err = mkl_mic_set_workdivision (MKL_TARGET_HOST, MIC_HOST_DEVICE, 0, 0);

//Let MKL decide of the amount of work to offload on coprocessor 0
err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, MIC_AUTO_WORKDIVISION);

//Offload 50% of work on coprocessor 0
err = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, 0.5);

//Get amount of work on coprocessor 0
err = mkl_mic_get_workdivision(MKL_TARGET_MIC, 0, &wd);
Tips for Using Automatic Offload

- AO works only when matrix sizes are right
  - SGEMM: Offloading only when M, N > 2048
  - Square matrices give much better performance

- These settings may produce better results for SGEMM calculation for 60-core coprocessor:
  
  ```
  export MIC_USE_2MB_BUFFERS=16K
  export MIC_OMP_NUM_THREADS=240
  export MIC_ENV_PREFIX=MIC
  export MIC_KMP_AFFINITY=compact,granularity=fine
  export MIC_PLACE_THREADS=60C,4t
  ```

- Work division settings are just hints to MKL runtime
  
  Threading control tips:
  
  Prevent thread migration on the host using:
  ```
  export KMP_AFFINITY=granularity=fine, compact, 1,0
  ```
MKL Usage In Accelerator Mode

- Compiler Assisted Offload
  - Offloading is explicitly controlled by compiler pragmas or directives.
  - All MKL functions can be inserted inside offload region to run on the Xeon Phi (In comparison, only a subset of MKL is subject to AO).
  - More flexibility in data transfer and remote execution management.

- Automatic Offload Mode
  - MKL functions are automatically offloaded to the accelerator.
  - MKL decides:
    - When to offload
    - Work division between host and targets
  - Data is managed automatically

- Native Execution
Native Execution

- In order to use Intel MKL in a native application, an additional argument `-mkl` is required with the compiler option `-mmic`.

- Native applications with Intel MKL functions operate just like native applications with user-defined functions.

```bash
$ icc -O3 -mmic -mkl sgemm.c -o sgemm.exe
```
Compile to use the Intel MKL

- Compile using \texttt{--mkl} flag
  - \texttt{-mkl=parallel} (default) for parallel execution
  - \texttt{-mkl=sequential} for sequential execution

- AO: The same way of building code on Xeon:
  - \texttt{user@host $ icc -O3 -mkl sgemm.c -o sgemm.exe}

- Native using \texttt{-mmic}
  - \texttt{user@host $ ifort -mmic -mkl myProgram.c -o myExec.mic}

- MKL can also be used in native mode if compiled with \texttt{-mmic}
More code examples:

- module show imkl
- $MKLROOT/examples/examples/examples_mic.tgz
  - sgemm       SGEMM example
  - sgemm_f     SGEMM example (Fortran 90)
  - fft         complex-to-complex 1D FFT
  - solverc     Pardiso examples
  - sgaussian   single precision Gaussian RNG
  - dgaussian   double precision Gaussian RNG
  - ...
Which Model to Choose:

- **native execution for**
  - Highly parallel code.
  - Using coprocessors as independency compute nodes
- **AO if**
  - Sufficient Byte/FLOP ratio makes offload beneficial.
  - Using Level-3 BLAS functions: GEMM, TRMM, TRSM
- **CAO if**
  - There is enough computations to offset data transfer overhead
  - Transferred data can be reused by multiple operations

Lab: MKL
Memory Allocation: Data Alignment

- Compiler-assisted offload
  - Memory alignment is inherited from host!

- General memory alignment (SIMD vectorisation)
  - Align buffers (leading dimension) to a multiple of vector width (64 Byte)
    - mkl_malloc, _mm_malloc(_aligned_malloc),
      tbb::scalable_aligned_malloc, ...
Memory Allocation: Data Alignment

```c
void  * darray;
int    workspace;
int    alignment = 64;
...
darray = mkl_malloc(sizeof(double) * workspace, alignment);
...
    mkl_free(darray);
```
Performance of many Intel MKL routines improves when input and output data reside in memory allocated with 2 MB pages

—>Address more memory with less pages, reduce overhead of translating between host- and MIC address spaces

# Allocate all pointer-based variables with run-time
# length > 64 KB in 2 MB pages:
$ export MIC_USE_2MB_BUFFERS=64K
Memory Allocation: Page Size

Native:
KMP_AFFINITY=balanced
OMP_NUM_THREADS=244

Compiler-Assisted Offload:
MIC_ENV_PREFIX=MIC
MIC_KMP_AFFINITY=balanced
MIC_OMP_NUM_THREADS=240
MIC_USE_2MB_BUFFERS=64K

Automatic Offload:
MKL_MIC_ENABLE=1
OFFLOAD_DEVICES=<list>
MKL_MIC_MAX_MEMORY=2GB
MIC_ENV_PREFIX=MIC
MIC_OMP_NUM_THREADS=240
MIC_KMP_AFFINITY=balanced
+ Compiler-Assisted Offload:
OFFLOAD_ENABLE_ORSL=1

Memory Allocation: Page Size

KMP_AFFINITY=
• Host: e.g., compact,1
• Coprocessor: balanced

MIC_ENV_PREFIX=MIC; MIC_KMP_AFFINITY=
• Coprocessor (CAO): balanced

KMP_PLACE_THREADS
• Note: does not replace KMP_AFFINITY
• Helps to set/achieve pinning on e.g., 60 cores with 3 threads each

kmp_* (or mkl_*) functions take precedence over corresponding env. variables
More MKL documentation

• Intel Many Integrated Core Community website:

• Performance charts online:

• Intel MKL forum

• https://software.intel.com/en-us/node/528430
• https://www.nersc.gov/assets/MKL_for_MIC.pdf
Thank you.