Intel® Math Kernel Library for Mac OS* X

User's Guide

Intel® MKL - Mac OS* X

Document Number: 315932-023US

Legal Information

Contents

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order. Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number/

This document contains information on products in the design phase of development.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Moblin, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Java is a registered trademark of Oracle and/or its affiliates.

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

Introducing the Intel® Math Kernel Library

Intel® Math Kernel Library (Intel® MKL) is a computing math library of highly optimized, extensively threaded routines for applications that require maximum performance. Intel MKL provides comprehensive functionality support in these major areas of computation:

- BLAS (level 1, 2, and 3) and LAPACK linear algebra routines, offering vector, vector-matrix, and matrix-matrix operations.
- The PARDISO* direct sparse solver, an iterative sparse solver, and supporting sparse BLAS (level 1, 2, and 3) routines for solving sparse systems of equations.
- ScaLAPACK distributed processing linear algebra routines for Linux* and Windows* operating systems, as well as the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS).
- Fast Fourier transform (FFT) functions in one, two, or three dimensions with support for mixed radices (not limited to sizes that are powers of 2), as well as distributed versions of these functions provided for use on clusters of the Linux* and Windows* operating systems.
- Vector Math Library (VML) routines for optimized mathematical operations on vectors.
- Vector Statistical Library (VSL) routines, which offer high-performance vectorized random number generators (RNG) for several probability distributions, convolution and correlation routines, and summary statistics functions.
- Data Fitting Library, which provides capabilities for spline-based approximation of functions, derivatives and integrals of functions, and search.

For details see the *Intel® MKL Reference Manual*.

Intel MKL is optimized for the latest Intel processors, including processors with multiple cores (see the *Intel® MKL Release Notes* for the full list of supported processors). Intel MKL also performs well on non-Intel processors.

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

Getting Help and Support

Intel provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel MKL support website at http://www.intel.com/software/products/support/.

Notational Conventions

The following term is used in reference to the operating system.

Mac OS * X	This term refers to information that is valid on all Intel®-based systems running the Mac OS* X operating system.

The following notations are used to refer to Intel MKL directories.

<Composer XE directory>	The installation directory for the Intel® C++ Composer XE or Intel® Fortran Composer XE .
<mkl directory>	The main directory where Intel MKL is installed: *<mkl directory>*=*<Composer XE directory>*/mkl. Replace this placeholder with the specific pathname in the configuring, linking, and building instructions.

The following font conventions are used in this document.

Italic	Italic is used for emphasis and also indicates document names in body text, for example: see *Intel MKL Reference Manual*.

Monospace lowercase mixed with uppercase	Indicates:
	• Commands and command-line options, for example,
	`icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -liomp5 -lpthread`
	• Filenames, directory names, and pathnames, for example,
	`/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Home`
	• C/C++ code fragments, for example,
	`a = new double [SIZE*SIZE];`

UPPERCASE MONOSPACE	Indicates system variables, for example, `$MKLPATH`.
Monospace italic	Indicates a parameter in discussions, for example, *lda*. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, *<mkl directory>*. Substitute one of these items for the placeholder.
`[ items ]`	Square brackets indicate that the items enclosed in brackets are optional.
`{ item | item }`	Braces indicate that only one of the items listed between braces should be selected. A vertical bar (|) separates the items.

Overview

1

Document Overview

The Intel® Math Kernel Library (Intel® MKL) User's Guide provides *usage information* for the library. The usage information covers the organization, configuration, performance, and accuracy of Intel MKL, specifics of routine calls in mixed-language programming, linking, and more.

This guide describes OS-specific usage of Intel MKL, along with OS-independent features. The document contains usage information for all Intel MKL function domains.

This User's Guide provides the following information:

- Describes post-installation steps to help you start using the library
- Shows you how to configure the library with your development environment
- Acquaints you with the library structure
- Explains how to link your application with the library and provides simple usage scenarios
- Describes how to code, compile, and run your application with Intel MKL

This guide is intended for Mac OS X programmers with beginner to advanced experience in software development.

See Also

Language Interfaces Support, by Function Domain

What's New

This User's Guide documents Intel® Math Kernel Library (Intel® MKL) 11.0 beta Update 2.

The following changes to the product have been documented:

- New behavior of the conditional bitwise reproducibility if unsupported or invalid values of the environment variable were set. See Values to Specify the CBWR Branch.
- Addition of the `mkl_set_num_threads_local` function, which sets the number of threads on the current execution thread. See Using Additional Threading Control.

Additionally, minor updates have been made to correct errors in the document.

Related Information

To reference how to use the library in your application, use this guide in conjunction with the following documents:

- The *Intel® Math Kernel Library Reference Manual*, which provides *reference* information on routine functionalities, parameter descriptions, interfaces, calling syntaxes, and return values.
- The *Intel® Math Kernel Library for Mac OS * X Release Notes*.

Getting Started

2

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

Checking Your Installation

After installing the Intel® Math Kernel Library (Intel® MKL), verify that the library is properly installed and configured:

1. Intel MKL installs in *<Composer XE directory>*.

   Check that the subdirectory of *<Composer XE directory>* referred to as *<mkl directory>* was created.
2. If you want to keep multiple versions of Intel MKL installed on your system, update your build scripts to point to the correct Intel MKL version.
3. Check that the following files appear in the *<mkl directory>*`/bin` directory and its subdirectories:

   `mklvars.sh`

   `mklvars.csh`

   `ia32/mklvars_ia32.sh`

   `ia32/mklvars_ia32.csh`

   `intel64/mklvars_intel64.sh`

   `intel64/mklvars_intel64.csh`

   Use these files to assign Intel MKL-specific values to several environment variables, as explained in Setting Environment Variables
4. To understand how the Intel MKL directories are structured, see Intel® Math Kernel Library Structure.
5. To make sure that Intel MKL runs on your system, launch an Intel MKL example, as explained in Using Code Examples.

See Also

Notational Conventions

Setting Environment Variables

When the installation of Intel MKL for Mac OS* X is complete, set the `INCLUDE`, `MKLROOT`, `DYLD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `NLSPATH` environment variables in the command shell using one of the script files in the `bin` subdirectory of the Intel MKL installation directory.

Choose the script corresponding to your system architecture and command shell as explained in the following table:

Architecture	Shell	Script File
IA-32	C	`ia32/mklvars_ia32.csh`
IA-32	Bash	`ia32/mklvars_ia32.sh`
Intel® 64	C	`intel64/mklvars_intel64.csh`
Intel® 64	Bash	`intel64/mklvars_intel64.sh`
IA-32 and Intel® 64	C	`mklvars.csh`
IA-32 and Intel® 64	Bash	`mklvars.sh`

Running the Scripts

The parameters of the scripts specify the following:

- Architecture.
- Use of Intel MKL Fortran modules precompiled with the Intel® Fortran compiler. Supply this parameter only if you are using this compiler.
- Programming interface (LP64 or ILP64).

Usage and values of these parameters depend on the name of the script (regardless of the extension). The following table lists values of the script parameters.

Script	Architecture (required, when applicable)	Use of Fortran Modules (optional)	Interface (optional)
`mklvars_ia32`	n/a[†]	`mod`	n/a
`mklvars_intel64`	n/a	`mod`	`lp64`, default `ilp64`
`mklvars`	`ia32` `intel64`	`mod`	`lp64`, default `ilp64`

[†] Not applicable.

For example:

- The command
  `mklvars.sh ia32`
  sets the environment for Intel MKL to use the IA-32 architecture.
- The command
  `mklvars.sh intel64 mod ilp64`
  sets the environment for Intel MKL to use the Intel® 64 architecture, ILP64 programming interface, and Fortran modules.
- The command
  `mklvars.sh intel64 mod`
  sets the environment for Intel MKL to use the Intel® 64 architecture, LP64 interface, and Fortran modules.

NOTE Supply the parameter specifying the architecture first, if it is needed. Values of the other two parameters can be listed in any order.

See Also

High-level Directory Structure
Interface Libraries and Modules

Fortran 95 Interfaces to LAPACK and BLAS
Setting the Number of Threads Using an OpenMP* Environment Variable

Compiler Support

Intel MKL supports compilers identified in the *Release Notes*. However, the library has been successfully used with other compilers as well.

Intel MKL provides a set of include files to simplify program development by specifying enumerated values and prototypes for the respective functions. Calling Intel MKL functions from your application without an appropriate include file may lead to incorrect behavior of the functions.

See Also

Include Files

Using Code Examples

The Intel MKL package includes code examples, located in the `examples` subdirectory of the installation directory. Use the examples to determine:

- Whether Intel MKL is working on your system
- How you should call the library
- How to link the library

The examples are grouped in subdirectories mainly by Intel MKL function domains and programming languages. For example, the `examples/spblas` subdirectory contains a makefile to build the Sparse BLAS examples and the `examples/vmlc` subdirectory contains the makefile to build the C VML examples. Source code for the examples is in the next-level `sources` subdirectory.

See Also

High-level Directory Structure

What You Need to Know Before You Begin Using the Intel® Math Kernel Library

Target platform	Identify the architecture of your target machine: • IA-32 or compatible • Intel® 64 or compatible **Reason:** Linking Examples To configure your development environment for the use with Intel MKL, set your environment variables using the script corresponding to your architecture (see Setting Environment Variables for details).
Mathematical problem	Identify all Intel MKL function domains that you require: • BLAS • Sparse BLAS • LAPACK • Sparse Solver routines • Vector Mathematical Library functions (VML) • Vector Statistical Library functions • Fourier Transform functions (FFT) • Trigonometric Transform routines • Poisson, Laplace, and Helmholtz Solver routines • Optimization (Trust-Region) Solver routines

	• Data Fitting Functions **Reason:** The function domain you intend to use narrows the search in the *Reference Manual* for specific routines you need. Coding tips may also depend on the function domain (see Tips and Techniques to Improve Performance).
Programming language	Intel MKL provides support for both Fortran and C/C++ programming. Identify the language interfaces that your function domains support (see Intel® Math Kernel Library Language Interfaces Support). **Reason:** Intel MKL provides language-specific include files for each function domain to simplify program development (see Language Interfaces Support, by Function Domain). For a list of language-specific interface libraries and modules and an example how to generate them, see also Using Language-Specific Interfaces with Intel® Math Kernel Library.
Range of integer data	If your system is based on the Intel 64 architecture, identify whether your application performs calculations with large data arrays (of more than 2^{31}-1 elements). **Reason:** To operate on large data arrays, you need to select the ILP64 interface, where integers are 64-bit; otherwise, use the default, LP64, interface, where integers are 32-bit (see Using the ILP64 Interface vs. LP64 Interface).
Threading model	Identify whether and how your application is threaded: • Threaded with the Intel compiler • Threaded with a third-party compiler • Not threaded **Reason:** The compiler you use to thread your application determines which threading library you should link with your application. For applications threaded with a third-party compiler you may need to use Intel MKL in the sequential mode (for more information, see Sequential Mode of the Library and Linking with Threading Libraries).
Number of threads	Determine the number of threads you want Intel MKL to use. **Reason:** Intel MKL is based on the OpenMP* threading. By default, the OpenMP* software sets the number of threads that Intel MKL uses. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, see Using Parallelism of the Intel® Math Kernel Library.
Linking model	Decide which linking model is appropriate for linking your application with Intel MKL libraries: • Static • Dynamic **Reason:** The link line syntax and libraries for static and dynamic linking are different. For the list of link libraries for static and dynamic models, linking examples, and other relevant topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with the Intel® Math Kernel Library.

Structure of the Intel® Math Kernel Library

3

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

Architecture Support

Intel® Math Kernel Library (Intel® MKL) for Mac OS* X supports the IA-32, Intel® 64, and compatible architectures in its universal libraries, located in the *<mkl directory>*/lib directory.

NOTE Universal libraries contain both 32-bit and 64-bit code. If these libraries are used for linking, the linker dispatches appropriate code as follows:

- A 32-bit linker dispatches 32-bit code and creates 32-bit executable files.
- A 64-bit linker dispatches 64-bit code and creates 64-bit executable files.

See Also

High-level Directory Structure
Directory Structure in Detail

High-level Directory Structure

Directory	Contents
<mkl directory>	Installation directory of the Intel® Math Kernel Library (Intel® MKL)
Subdirectories of *<mkl directory>*	
bin/	Scripts to set environmental variables in the user shell
bin/ia32	Shell scripts for the IA-32 architecture
bin/intel64	Shell scripts for the Intel® 64 architecture
benchmarks/linpack	Shared-Memory (SMP) version of LINPACK benchmark
examples	Examples directory. Each subdirectory has source and data files
include	INCLUDE files for the library routines, as well as for tests and examples
include/ia32	Fortran 95 .mod files for the IA-32 architecture and Intel® Fortran compiler

Directory	Contents
include/intel64/lp64	Fortran 95 .mod files for the Intel® 64 architecture, Intel Fortran compiler, and LP64 interface
include/intel64/ilp64	Fortran 95 .mod files for the Intel® 64 architecture, Intel® Fortran compiler, and ILP64 interface
include/fftw	Header files for the FFTW2 and FFTW3 interfaces
interfaces/blas95	Fortran 95 interfaces to BLAS and a makefile to build the library
interfaces/fftw2xc	FFTW 2.x interfaces to Intel MKL FFTs (C interface)
interfaces/fftw2xf	FFTW 2.x interfaces to Intel MKL FFTs (Fortran interface)
interfaces/fftw3xc	FFTW 3.x interfaces to Intel MKL FFTs (C interface)
interfaces/fftw3xf	FFTW 3.x interfaces to Intel MKL FFTs (Fortran interface)
interfaces/lapack95	Fortran 95 interfaces to LAPACK and a makefile to build the library
lib	Universal static libraries and shared objects for the IA-32 and Intel® 64 architectures
tests	Source and data files for tests
tools	Tools and plug-ins
tools/builder	Tools for creating custom dynamically linkable libraries
Subdirectories of *<Composer XE directory>*	
Documentation/en_US/mkl	Intel MKL documentation

See Also
Notational Conventions

Layered Model Concept

Intel MKL is structured to support multiple compilers and interfaces, different OpenMP* implementations, both serial and multiple threads, and a wide range of processors. Conceptually Intel MKL can be divided into distinct parts to support different interfaces, threading models, and core computations:

1. Interface Layer
2. Threading Layer
3. Computational Layer

You can combine Intel MKL libraries to meet your needs by linking with one library in each part layer-by-layer. Once the interface library is selected, the threading library you select picks up the chosen interface, and the computational library uses interfaces and OpenMP implementation (or non-threaded mode) chosen in the first two layers.

To support threading with different compilers, one more layer is needed, which contains libraries not included in Intel MKL:

- Compiler run-time libraries (RTL).

The following table provides more details of each layer.

Layer	Description

Interface Layer	This layer matches compiled code of your application with the threading and/or computational parts of the library. This layer provides:
	• LP64 and ILP64 interfaces.
	• Compatibility with compilers that return function values differently.
	• A mapping between single-precision names and double-precision names for applications using Cray*-style naming (SP2DP interface).
	SP2DP interface supports Cray-style naming in applications targeted for the Intel 64 architecture and using the ILP64 interface. SP2DP interface provides a mapping between single-precision names (for both real and complex types) in the application and double-precision names in Intel MKL BLAS and LAPACK. Function names are mapped as shown in the following example for BLAS functions `?GEMM`:
	`SGEMM -> DGEMM`
	`DGEMM -> DGEMM`
	`CGEMM -> ZGEMM`
	`ZGEMM -> ZGEMM`
	Mind that no changes are made to double-precision names.

Threading Layer	This layer: • Provides a way to link threaded Intel MKL with different threading compilers. • Enables you to link with a threaded or sequential mode of the library. This layer is compiled for different environments (threaded or sequential) and compilers (from Intel, GNU*).
Computational Layer	This layer is the heart of Intel MKL. It has only one library for each combination of architecture and supported OS. The Computational layer accommodates multiple architectures through identification of architecture features and chooses the appropriate binary code at run time.
Compiler Run-time Libraries (RTL)	To support threading with Intel compilers, Intel MKL uses RTLs of the Intel® C++ Composer XE or Intel® Fortran Composer XE. To thread using third-party threading compilers, use libraries in the Threading layer or an appropriate compatibility library.

See Also

Using the ILP64 Interface vs. LP64 Interface
Linking Your Application with the Intel® Math Kernel Library
Linking with Threading Libraries

Contents of the Documentation Directories

Most of Intel MKL documentation is installed at *<Composer XE directory>*`/Documentation/`*<locale>*`/mkl`. For example, the documentation in English is installed at *<Composer XE directory>*`/Documentation/en_US/mkl`. However, some Intel MKL-related documents are installed one or two levels up. The following table lists MKL-related documentation.

File name	Comment
Files in *<Composer XE directory>*`/Documentation`	
<locale>`/clicense` or *<locale>*`/flicense`	Common end user license for the Intel® C++ Composer XE or Intel® Fortran Composer XE, respectively
`mklsupport.txt`	Information on package number for customer support reference
Contents of *<Composer XE directory>*`/Documentation/`*<locale>*`/mkl`	
`mkl_documentation.htm`	Overview and links for the Intel MKL documentation

File name	Comment
Release_Notes.htm	Intel MKL Release Notes
mkl_userguide/index.htm	Intel MKL User's Guide in an uncompressed HTML format, this document
mkl_link_line_advisor.htm	Intel MKL Link-line Advisor

For more documents, visit http://software.intel.com/en-us/articles/intel-mkl-11dot0/.

Linking Your Application with the Intel® Math Kernel Library

4

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

Linking Quick Start

Intel® Math Kernel Library (Intel® MKL) provides several options for quick linking of your application, which depend on the way you link:

Using the Intel® Composer XE compiler	see Using the `-mkl` Compiler Option.
Explicit dynamic linking	see Using the Single Dynamic Library for how to simplify your link line.
Explicitly listing libraries on your link line	see Selecting Libraries to Link with for a summary of the libraries.
Using an interactive interface	see Using the Link-line Advisor to determine libraries and options to specify on your link or compilation line.
Using an internally provided tool	see Using the Command-line Link Tool to determine libraries, options, and environment variables or even compile and build your application.

Using the -mkl Compiler Option

The Intel® Composer XE compiler supports the following variants of the `-mkl` compiler option:

`-mkl` or `-mkl=parallel`	to link with standard threaded Intel MKL.
`-mkl=sequential`	to link with sequential version of Intel MKL.
`-mkl=cluster`	to link with Intel MKL cluster components (sequential) that use Intel MPI.

For more information on the `-mkl` compiler option, see the Intel Compiler User and Reference Guides.

On Intel® 64 architecture systems, for each variant of the `-mkl` option, the compiler links your application using the LP64 interface.

If you specify any variant of the `-mkl` compiler option, the compiler automatically includes the Intel MKL libraries. In cases not covered by the option, use the Link-line Advisor or see Linking in Detail.

See Also

Listing Libraries on a Link Line
Using the ILP64 Interface vs. LP64 Interface
Using the Link-line Advisor
Intel® Software Documentation Library

Using the Single Dynamic Library

You can simplify your link line through the use of the Intel MKL Single Dynamic Library (SDL).

To use SDL, place `libmkl_rt.dylib` on your link line. For example:

```
icc application.c -lmkl_rt
```

SDL enables you to select the interface and threading library for Intel MKL at run time. By default, linking with SDL provides:

- LP64 interface on systems based on the Intel® 64 architecture
- Intel threading

To use other interfaces or change threading preferences, including use of the sequential version of Intel MKL, you need to specify your choices using functions or environment variables as explained in section Dynamically Selecting the Interface and Threading Layer.

Selecting Libraries to Link with

To link with Intel MKL:

- Choose one library from the Interface layer and one library from the Threading layer
- Add the only library from the Computational layer and run-time libraries (RTL)

The following table lists Intel MKL libraries to link with your application.

	Interface layer	Threading layer	Computational layer	RTL
IA-32 architecture, static linking	`libmkl_intel.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.dylib`
IA-32 architecture, dynamic linking	`libmkl_intel.dylib`	`libmkl_intel_thread.dylib`	`libmkl_core.dylib`	`libiomp5.dylib`
Intel® 64 architecture, static linking	`libmkl_intel_lp64.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.dylib`
Intel® 64 architecture, dynamic linking	`libmkl_intel_lp64.dylib`	`libmkl_intel_thread.dylib`	`libmkl_core.dylib`	`libiomp5.dylib`

The Single Dynamic Library (SDL) automatically links interface, threading, and computational libraries and thus simplifies linking. The following table lists Intel MKL libraries for dynamic linking using SDL. See Dynamically Selecting the Interface and Threading Layer for how to set the interface and threading layers at run time through function calls or environment settings.

	SDL	RTL
IA-32 and Intel® 64 architectures	`libmkl_rt.dylib`	`libiomp5.dylib`[†]

[†] Use the Link-line Advisor to check whether you need to explicitly link the `libiomp5.dylib` RTL.

For exceptions and alternatives to the libraries listed above, see Linking in Detail.

See Also
Layered Model Concept
Using the Link-line Advisor
Using the -mkl Compiler Option

Using the Link-line Advisor

Use the Intel MKL Link-line Advisor to determine the libraries and options to specify on your link or compilation line.

The latest version of the tool is available at http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor. The tool is also available in the product.

The Advisor requests information about your system and on how you intend to use Intel MKL (link dynamically or statically, use threaded or sequential mode, etc.). The tool automatically generates the appropriate link line for your application.

See Also
Contents of the Documentation Directories

Using the Command-line Link Tool

Use the command-line Link tool provided by Intel MKL to simplify building your application with Intel MKL.

The tool not only provides the options, libraries, and environment variables to use, but also performs compilation and building of your application.

The tool `mkl_link_tool` is installed in the *<mkl directory>*`/tools` directory.

See the knowledge base article at http://software.intel.com/en-us/articles/mkl-command-line-link-tool for more information.

Linking Examples

See Also
Using the Link-line Advisor

Linking on IA-32 Architecture Systems

The following examples illustrate linking that uses Intel(R) compilers.

The examples use the `.f` Fortran source file. C/C++ users should instead specify a `.cpp` (C++) or `.c` (C) file and replace `ifort` with `icc`.

In these examples,
`MKLPATH=$MKLROOT/lib`,
`MKLINCLUDE=$MKLROOT/include`.

NOTE If you successfully completed the Setting Environment Variables step of the Getting Started process, you can omit `-I$MKLINCLUDE` in all the examples and omit `-L$MKLPATH` in the examples for dynamic linking.

- Static linking of `myprog.f` and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
$MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
$MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
```

```
-liomp5 -lpthread -lm
```

- Dynamic linking of `myprog.f` and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

- Static linking of `myprog.f` and sequential version of Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
$MKLPATH/libmkl_intel.a $MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a
$MKLPATH/libmkl_intel.a $MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a
-lpthread -lm
```

- Dynamic linking of `myprog.f` and sequential version of Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_sequential -lmkl_core -lpthread -lm
```

- Dynamic linking of user code `myprog.f` and parallel or sequential Intel MKL (Call the `mkl_set_threading_layer` function or set value of the `MKL_THREADING_LAYER` environment variable to choose threaded or sequential mode):

```
ifort myprog.f -lmkl_rt
```

- Static linking of `myprog.f`, Fortran 95 LAPACK interface, and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_lapack95
$MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -liomp5 -lpthread -lm
```

- Static linking of `myprog.f`, Fortran 95 BLAS interface, and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_blas95 $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/
libmkl_core.a $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/
libmkl_core.a -liomp5 -lpthread -lm
```

See Also

Fortran 95 Interfaces to LAPACK and BLAS
Linking with System Libraries

Linking on Intel(R) 64 Architecture Systems

The following examples illustrate linking that uses Intel(R) compilers.

The examples use the `.f` Fortran source file. C/C++ users should instead specify a `.cpp` (C++) or `.c` (C) file and replace `ifort` with `icc`.

In these examples,
`MKLPATH=$MKLROOT/lib`,
`MKLINCLUDE=$MKLROOT/include`.

NOTE If you successfully completed the Setting Environment Variables step of the Getting Started process, you can omit `-I$MKLINCLUDE` in all the examples and omit `-L$MKLPATH` in the examples for dynamic linking.

- Static linking of `myprog.f` and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
$MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -liomp5 -lpthread -lm
```

- Dynamic linking of `myprog.f` and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
```

```
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core
-liomp5 -lpthread -lm
```

- Static linking of `myprog.f` and sequential version of Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
$MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -lpthread -lm
```

- Dynamic linking of `myprog.f` and sequential version of Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm
```

- Static linking of `myprog.f` and parallel Intel MKL supporting the ILP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
$MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -liomp5 -lpthread -lm
```

- Dynamic linking of `myprog.f` and parallel Intel MKL supporting the ILP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

- Dynamic linking of user code `myprog.f` and parallel or sequential Intel MKL (Call appropriate functions or set environment variables to choose threaded or sequential mode and to set the interface):

```
ifort myprog.f -lmkl_rt
```

- Static linking of `myprog.f`, Fortran 95 LAPACK interface, and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_lapack95_lp64 $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -liomp5 -lpthread -lm
```

- Static linking of `myprog.f`, Fortran 95 BLAS interface, and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_blas95_lp64 $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a $MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -liomp5 -lpthread -lm
```

See Also

Fortran 95 Interfaces to LAPACK and BLAS
Linking with System Libraries

Linking in Detail

This section recommends which libraries to link with depending on your Intel MKL usage scenario and provides details of the linking.

Listing Libraries on a Link Line

To link with Intel MKL, specify paths and libraries on the link line as shown below.

NOTE The syntax below is for dynamic linking. For static linking, replace each library name preceded with "`-l`" with the path to the library file. For example, replace `-lmkl_core` with `$MKLPATH/libmkl_core.a`, where `$MKLPATH` is the appropriate user-defined environment variable.

<files to link>

```
-L<MKL path> -I<MKL include>
[-I<MKL include>/{ia32|intel64|{ilp64|lp64}}]
[-lmkl_blas{95|95_ilp64|95_lp64}]
[-lmkl_lapack{95|95_ilp64|95_lp64}]


-lmkl_{intel|intel_ilp64|intel_lp64}
-lmkl_{intel_thread|sequential}
-lmkl_core
-liomp5 [-lpthread] [-lm] [-ldl]
```

In case of static linking, for all components except BLAS and FFT, repeat interface, threading, and computational libraries two times (for example, `libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a`). For the LAPACK component, repeat the threading and computational libraries three times.

The order of listing libraries on the link line is essential.

See Also

Using the Link-line Advisor
Linking Examples

Dynamically Selecting the Interface and Threading Layer

The Single Dynamic Library (SDL) enables you to dynamically select the interface and threading layer for Intel MKL.

Setting the Interface Layer

Available interfaces depend on the architecture of your system.

On systems based on the Intel® 64 architecture, LP64 and ILP64 interfaces are available. To set one of these interfaces at run time, use the `mkl_set_interface_layer` function or the `MKL_INTERFACE_LAYER` environment variable. The following table provides values to be used to set each interface.

Interface Layer	Value of `MKL_INTERFACE_LAYER`	Value of the Parameter of `mkl_set_interface_layer`
LP64	`LP64`	`MKL_INTERFACE_LP64`
ILP64	`ILP64`	`MKL_INTERFACE_ILP64`

If the `mkl_set_interface_layer` function is called, the environment variable `MKL_INTERFACE_LAYER` is ignored.

By default the LP64 interface is used.

See the *Intel MKL Reference Manual* for details of the `mkl_set_interface_layer` function.

Setting the Threading Layer

To set the threading layer at run time, use the `mkl_set_threading_layer` function or the `MKL_THREADING_LAYER` environment variable. The following table lists available threading layers along with the values to be used to set each layer.

Threading Layer	Value of MKL_THREADING_LAYER	Value of the Parameter of mkl_set_threading_layer
Intel threading	INTEL	MKL_THREADING_INTEL
Sequential mode of Intel MKL	SEQUENTIAL	MKL_THREADING_SEQUENTIAL

If the `mkl_set_threading_layer` function is called, the environment variable `MKL_THREADING_LAYER` is ignored.

By default Intel threading is used.

See the *Intel MKL Reference Manual* for details of the `mkl_set_threading_layer` function.

See Also

Using the Single Dynamic Library
Layered Model Concept
Directory Structure in Detail

Linking with Interface Libraries

Using the ILP64 Interface vs. LP64 Interface

The Intel MKL ILP64 libraries use the 64-bit integer type (necessary for indexing large arrays, with more than 2^{31}-1 elements), whereas the LP64 libraries index arrays with the 32-bit integer type.

The LP64 and ILP64 interfaces are implemented in the Interface layer. Link with the following interface libraries for the LP64 or ILP64 interface, respectively:

- `libmkl_intel_lp64.a` or `libmkl_intel_ilp64.a` for static linking
- `libmkl_intel_lp64.dylib` or `libmkl_intel_ilp64.dylib` for dynamic linking

The ILP64 interface provides for the following:

- Support large data arrays (with more than 2^{31}-1 elements)
- Enable compiling your Fortran code with the `-i8` compiler option

The LP64 interface provides compatibility with the previous Intel MKL versions because "LP64" is just a new name for the only interface that the Intel MKL versions lower than 9.1 provided. Choose the ILP64 interface if your application uses Intel MKL for calculations with large data arrays or the library may be used so in future.

Intel MKL provides the same include directory for the ILP64 and LP64 interfaces.

Compiling for LP64/ILP64

The table below shows how to compile for the ILP64 and LP64 interfaces:

Fortran	
Compiling for ILP64	`ifort -i8 -I<mkl directory>/include ...`
Compiling for LP64	`ifort -I<mkl directory>/include ...`

C or C++	
Compiling for ILP64	`icc -DMKL_ILP64 -I<mkl directory>/include ...`

	C or C++
Compiling for LP64	`icc -I<mkl directory>/include ...`

CAUTION Linking of an application compiled with the `-i8` or `-DMKL_ILP64` option to the LP64 libraries may result in unpredictable consequences and erroneous output.

Coding for ILP64

You do not need to change existing code if you are not using the ILP64 interface.

To migrate to ILP64 or write new code for ILP64, use appropriate types for parameters of the Intel MKL functions and subroutines:

Integer Types	Fortran	C or C++
32-bit integers	`INTEGER*4` or `INTEGER(KIND=4)`	`int`
Universal integers for ILP64/ LP64: • 64-bit for ILP64 • 32-bit otherwise	`INTEGER` without specifying `KIND`	`MKL_INT`
Universal integers for ILP64/ LP64: • 64-bit integers	`INTEGER*8` or `INTEGER(KIND=8)`	`MKL_INT64`
FFT interface integers for ILP64/ LP64	`INTEGER` without specifying `KIND`	`MKL_LONG`

To determine the type of an integer parameter of a function, use appropriate include files. For functions that support only a Fortran interface, use the C/C++ include files `*.h`.

The above table explains which integer parameters of functions become 64-bit and which remain 32-bit for ILP64. The table applies to most Intel MKL functions except some VML and VSL functions, which require integer parameters to be 64-bit or 32-bit regardless of the interface:

- **VML:** The *mode* parameter of VML functions is 64-bit.
- **Random Number Generators (RNG):**

  All discrete RNG except `viRngUniformBits64` are 32-bit.

  The `viRngUniformBits64` generator function and `vslSkipAheadStream` service function are 64-bit.
- **Summary Statistics:** The *estimate* parameter of the `vslsSSCompute/vsldSSCompute` function is 64-bit.

Refer to the *Intel MKL Reference Manual* for more information.

To better understand ILP64 interface details, see also examples and tests.

Limitations

All Intel MKL function domains support ILP64 programming with the following exceptions:

- FFTW interfaces to Intel MKL:
  - FFTW 2.x wrappers do not support ILP64.
  - FFTW 3.2 wrappers support ILP64 by a dedicated set of functions `plan_guru64`.

See Also
High-level Directory Structure
Include Files
Language Interfaces Support, by Function Domain
Layered Model Concept
Directory Structure in Detail

Linking with Fortran 95 Interface Libraries

The `libmkl_blas95*.a` and `libmkl_lapack95*.a` libraries contain Fortran 95 interfaces for BLAS and LAPACK, respectively, which are compiler-dependent. In the Intel MKL package, they are prebuilt for the Intel® Fortran compiler. If you are using a different compiler, build these libraries before using the interface.

See Also
Fortran 95 Interfaces to LAPACK and BLAS
Compiler-dependent Functions and Fortran 90 Modules

Linking with Threading Libraries

Sequential Mode of the Library

You can use Intel MKL in a sequential (non-threaded) mode. In this mode, Intel MKL runs unthreaded code. However, it is thread-safe (except the LAPACK deprecated routine `?lacon)`, which means that you can use it in a parallel region in your OpenMP* code. The sequential mode requires no compatibility OpenMP* run-time library and does not respond to the environment variable `OMP_NUM_THREADS` or its Intel MKL equivalents.

You should use the library in the sequential mode only if you have a particular reason not to use Intel MKL threading. The sequential mode may be helpful when using Intel MKL with programs threaded with some non-Intel compilers or in other situations where you need a non-threaded version of the library (for instance, in some MPI cases). To set the sequential mode, in the Threading layer, choose the `*sequential.*` library.

Add the POSIX threads library `(pthread)` to your link line for the sequential mode because the `*sequential.*` library depends on `pthread` .

See Also
Directory Structure in Detail
Using Parallelism of the Intel® Math Kernel Library
Avoiding Conflicts in the Execution Environment
Linking Examples

Selecting the Threading Layer

Several compilers that Intel MKL supports use the OpenMP* threading technology. Intel MKL supports implementations of the OpenMP* technology that these compilers provide. To make use of this support, you need to link with the appropriate library in the Threading Layer and Compiler Support Run-time Library (RTL).

Threading Layer

Each Intel MKL threading library contains the same code compiled by the respective compiler (Intel, gnu and PGI* compilers on Mac OS X).

RTL

This layer includes `libiomp`, the compatibility OpenMP* run-time library of the Intel compiler. In addition to the Intel compiler, `libiomp` provides support for one more threading compiler on Mac OS X (GNU). That is, a program threaded with a GNU compiler can safely be linked with Intel MKL and `libiomp`.

The table below helps explain what threading library and RTL you should choose under different scenarios when using Intel MKL (static cases only):

Compiler	Application Threaded?	Threading Layer	RTL Recommended	Comment
Intel	Does not matter	libmkl_intel_thread.a	libiomp5.dylib	
PGI	Yes	libmkl_pgi_thread.a or libmkl_sequential.a	PGI* supplied	Use of libmkl_sequential.a removes threading from Intel MKL calls.
PGI	No	libmkl_intel_thread.a	libiomp5.dylib	
PGI	No	libmkl_pgi_thread.a	PGI* supplied	
PGI	No	libmkl_sequential.a	None	
GNU	Yes	libmkl_sequential.a	None	
GNU	No	libmkl_intel_thread.a	libiomp5.dylib	
other	Yes	libmkl_sequential.a	None	
other	No	libmkl_intel_thread.a	libiomp5.dylib	

Linking with Compiler Run-time Libraries

Dynamically link libiomp5, the compatibility OpenMP* run-time library, even if you link other libraries statically.

Linking to the libiomp5 statically can be problematic because the more complex your operating environment or application, the more likely redundant copies of the library are included. This may result in performance issues (oversubscription of threads) and even incorrect results.

To link libiomp5 dynamically, be sure the DYLD_LIBRARY_PATH environment variable is defined correctly.

See Also

Setting Environment Variables
Layered Model Concept

Linking with System Libraries

To use the Intel MKL FFT, Trigonometric Transform, or Poisson, Laplace, and Helmholtz Solver routines, link also the math support system library by adding " -lm " to the link line.

On Mac OS X, the libiomp5 library relies on the native pthread library for multi-threading. Any time libiomp5 is required, add -lpthread to your link line afterwards (the order of listing libraries is important).

NOTE To link with Intel MKL statically using a GNU or PGI compiler, link also the system library `libdl` by adding `-ldl` to your link line. The Intel compiler always passes `-ldl` to the linker.

See Also
Linking Examples

Building Custom Dynamically Linked Shared Libraries

Custom dynamically linked shared libraries reduce the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

The Intel MKL custom dynamically linked shared library builder enables you to create a dynamic ally linked shared library containing the selected functions and located in the `tools/builder` directory. The builder contains a makefile and a definition file with the list of functions.

Using the Custom Dynamically Linked Shared Library Builder

To build a custom dynamically linked shared library, use the following command:

`make target [<options>]`

The following table lists possible values of `target` and explains what the command does for each value:

Value	Comment
`libuni`	The builder uses static Intel MKL interface, threading, and core libraries to build a universal dynamically linked shared library for the IA-32 or Intel® 64 architecture.
`dylibuni`	The builder uses the single dynamic library `libmkl_rt.dylib` to build a universal dynamically linked shared library for the IA-32 or Intel® 64 architecture.
`help`	The command prints Help on the custom dynamically linked shared library builder

The *<options>* placeholder stands for the list of parameters that define macros to be used by the makefile. The following table describes these parameters:

Parameter [Values]	Description
`interface = {lp64|ilp64}`	Defines whether to use LP64 or ILP64 programming interfacefor the Intel 64architecture.The default value is `lp64`.
`threading = {parallel| sequential}`	Defines whether to use the Intel MKL in the threaded or sequential mode. The default value is `parallel`.
`export = <file name>`	Specifies the full name of the file that contains the list of entry-point functions to be included in the shared object. The default name is `user_example_list` (no extension).
`name = <so name>`	Specifies the name of the library to be created. By default, the names of the created library is `mkl_custom.dylib`.
`xerbla = <error handler>`	Specifies the name of the object file *<user_xerbla>*`.o` that contains the user's error handler. The makefile adds this error handler to the library for use instead of the default Intel MKL error handler *xerbla*. If you omit this parameter, the native Intel MKL *xerbla* is used. See the description of the *xerbla* function in the Intel MKL Reference Manual on how to develop your own error handler.

Parameter [Values]	Description
MKLROOT = *<mkl directory>*	Specifies the location of Intel MKL libraries used to build the custom dynamically linked shared library. By default, the builder uses the Intel MKL installation directory.

All the above parameters are optional.

In the simplest case, the command line is `make ia32`, and the missing options have default values. This command creates the `mkl_custom.dylib` library for processors using the IA-32 architecture. The command takes the list of functions from the `user_list` file and uses the native Intel MKL error handler *xerbla*.

An example of a more complex case follows:

```
make ia32 export=my_func_list.txt name=mkl_small xerbla=my_xerbla.o
```

In this case, the command creates the `mkl_small.dylib` library for processors using the IA-32 architecture. The command takes the list of functions from `my_func_list.txt` file and uses the user's error handler `my_xerbla.o`.

The process is similar for processors using the Intel® 64 architecture.

See Also
Using the Single Dynamic Library

Composing a List of Functions

To compose a list of functions for a minimal custom dynamically linked shared library needed for your application, you can use the following procedure:

1. Link your application with installed Intel MKL libraries to make sure the application builds.
2. Remove all Intel MKL libraries from the link line and start linking.

   Unresolved symbols indicate Intel MKL functions that your application uses.
3. Include these functions in the list.

Important Each time your application starts using more Intel MKL functions, update the list to include the new functions.

See Also
Specifying Function Names

Specifying Function Names

In the file with the list of functions for your custom dynamically linked shared library, adjust function names to the required interface. For example, for Fortran functions append an underscore character "_" to the names as a suffix:

```
dgemm_
ddot_
dgetrf_
```

For more examples, see domain-specific lists of functions in the *<mkl directory>*`/tools/builder` folder.

NOTE The lists of functions are provided in the *<mkl directory>*`/tools/builder` folder merely as examples. See Composing a List of Functions for how to compose lists of functions for your custom dynamically linked shared library.

TIP Names of Fortran-style routines (BLAS, LAPACK, etc.) can be both upper-case or lower-case, with or without the trailing underscore. For example, these names are equivalent:
BLAS: `dgemm`, `DGEMM`, `dgemm_`, `DGEMM_`
LAPACK: `dgetrf`, `DGETRF`, `dgetrf_`, `DGETRF_`.

Properly capitalize names of C support functions in the function list. To do this, follow the guidelines below:

1. In the `mkl_service.h` include file, look up a `#define` directive for your function (`mkl_service.h` is included in the `mkl.h` header file).
2. Take the function name from the replacement part of that directive.

For example, the `#define` directive for the `mkl_disable_fast_mm` function is `#define mkl_disable_fast_mm MKL_Disable_Fast_MM`.

Capitalize the name of this function in the list like this: `MKL_Disable_Fast_MM`.

For the names of the Fortran support functions, see the tip.

NOTE If selected functions have several processor-specific versions, the builder automatically includes them all in the custom library and the dispatcher manages them.

Distributing Your Custom Dynamically Linked Shared Library

To enable use of your custom dynamically linked shared library in a threaded mode, distribute `libiomp5.dylib` along with the custom dynamically linked shared library.

Managing Performance and Memory

5

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

Using Parallelism of the Intel® Math Kernel Library

Intel MKL is extensively parallelized. See Threaded Functions and Problems for lists of threaded functions and problems that can be threaded.

Intel MKL is *thread-safe*, which means that all Intel MKL functions (except the LAPACK deprecated routine `?lacon`) work correctly during simultaneous execution by multiple threads. In particular, any chunk of threaded Intel MKL code provides access for multiple threads to the same shared data, while permitting only one thread at any given time to access a shared piece of data. Therefore, you can call Intel MKL from multiple threads and not worry about the function instances interfering with each other.

The library uses OpenMP* threading software, so you can use the environment variable `OMP_NUM_THREADS` to specify the number of threads or the equivalent OpenMP run-time function calls. Intel MKL also offers variables that are independent of OpenMP, such as `MKL_NUM_THREADS`, and equivalent Intel MKL functions for thread management. The Intel MKL variables are always inspected first, then the OpenMP variables are examined, and if neither is used, the OpenMP software chooses the default number of threads.

By default, Intel MKL uses the number of threads equal to the number of physical cores on the system.

To achieve higher performance, set the number of threads to the number of real processors or physical cores, as summarized in Techniques to Set the Number of Threads.

Threaded Functions and Problems

The following Intel MKL function domains are threaded:

- Direct sparse solver.
- LAPACK.

  For the list of threaded routines, see Threaded LAPACK Routines.
- Level1 and Level2 BLAS.

  For the list of threaded routines, see Threaded BLAS Level1 and Level2 Routines.
- All Level 3 BLAS and all Sparse BLAS routines except Level 2 Sparse Triangular solvers.
- All mathematical VML functions.
- FFT.

  For the list of FFT transforms that can be threaded, see Threaded FFT Problems.

Threaded LAPACK Routines

In the following list, `?` stands for a precision prefix of *each* flavor of the respective routine and may have the value of `s`, `d`, `c`, or `z`.

The following LAPACK routines are threaded:

- Linear equations, computational routines:
  - Factorization: `?getrf, ?gbtrf, ?potrf, ?pptrf, ?sytrf, ?hetrf, ?sptrf, ?hptrf`
  - Solving: `?dttrsb, ?gbtrs, ?gttrs, ?pptrs, ?pbtrs, ?pttrs, ?sytrs, ?sptrs, ?hptrs, ?tptrs, ?tbtrs`
- Orthogonal factorization, computational routines:
  `?geqrf, ?ormqr, ?unmqr, ?ormlq, ?unmlq, ?ormql, ?unmql, ?ormrq, ?unmrq`
- Singular Value Decomposition, computational routines:
  `?gebrd, ?bdsqr`
- Symmetric Eigenvalue Problems, computational routines:
  `?sytrd, ?hetrd, ?sptrd, ?hptrd, ?steqr, ?stedc`.
- Generalized Nonsymmetric Eigenvalue Problems, computational routines:
  `chgeqz/zhgeqz`.

A number of other LAPACK routines, which are based on threaded LAPACK or BLAS routines, make effective use of parallelism:
`?gesv, ?posv, ?gels, ?gesvd, ?syev, ?heev, cgegs/zgegs, cgegv/zgegv, cgges/zgges, cggesx/zggesx, cggev/zggev, cggevx/zggevx`, and so on.

Threaded BLAS Level1 and Level2 Routines

In the following list, `?` stands for a precision prefix of *each* flavor of the respective routine and may have the value of `s, d, c`, or `z`.

The following routines are threaded for Intel® Core™2 Duo and Intel® Core™ i7 processors:

- Level1 BLAS:
  `?axpy, ?copy, ?swap, ddot/sdot, cdotc, drot/srot`
- Level2 BLAS:
  `?gemv, ?trmv, dsyr/ssyr, dsyr2/ssyr2, dsymv/ssymv`

Threaded FFT Problems

The following characteristics of a specific problem determine whether your FFT computation may be threaded:

- rank
- domain
- size/length
- precision (single or double)
- placement (in-place or out-of-place)
- strides
- number of transforms
- layout (for example, interleaved or split layout of complex data)

Most FFT problems are threaded. In particular, computation of multiple transforms in one call (number of transforms > 1) is threaded. Details of which transforms are threaded follow.

One-dimensional (1D) transforms

1D transforms are threaded in many cases.

1D complex-to-complex (c2c) transforms of size N using interleaved complex data layout are threaded under the following conditions depending on the architecture:

Architecture	Conditions
Intel® 64	N is a power of 2, $log_2(N) > 9$, the transform is double-precision out-of-place, and input/output strides equal 1.
IA-32	N is a power of 2, $log_2(N) > 13$, and the transform is single-precision.
	N is a power of 2, $log_2(N) > 14$, and the transform is double-precision.
Any	N is composite, $log_2(N) > 16$, and input/output strides equal 1.

1D complex-to-complex transforms using split-complex layout are not threaded.

Multidimensional transforms

All multidimensional transforms on large-volume data are threaded.

Avoiding Conflicts in the Execution Environment

Certain situations can cause conflicts in the execution environment that make the use of threads in Intel MKL problematic. This section briefly discusses why these problems exist and how to avoid them.

If you thread the program using OpenMP directives and compile the program with Intel compilers, Intel MKL and the program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads unless you specifically request Intel MKL to do so via the `MKL_DYNAMIC` functionality. However, Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If your program is threaded by some other means, Intel MKL may operate in multithreaded mode, and the performance may suffer due to overuse of the resources.

The following table considers several cases where the conflicts may arise and provides recommendations depending on your threading model:

Threading model	Discussion
You thread the program using OS threads (`pthreads` on Mac OS* X).	If more than one thread calls Intel MKL, and the function being called is threaded, it may be important that you turn off Intel MKL threading. Set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads).
You thread the program using OpenMP directives and/or pragmas and compile the program using a compiler other than a compiler from Intel.	This is more problematic because setting of the `OMP_NUM_THREADS` environment variable affects both the compiler's threading library and `libiomp5`. In this case, choose the threading library that matches the layered Intel MKL with the OpenMP compiler you employ (see Linking Examples on how to do this). If this is not possible, use Intel MKL in the sequential mode. To do this, you should link with the appropriate threading library: `libmkl_sequential.a` or `libmkl_sequential.dylib` (see High-level Directory Structure).
There are multiple programs running on a multiple-cpu system, for example, a parallelized program that runs using MPI for communication in which each processor is treated as a node.	The threading software will see multiple processors on the system even though each processor has a separate MPI process running on it. In this case, one of the solutions is to set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads).

Using the `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions to control parallelism of Intel MKL from parallel user threads may result in a race condition that impacts the performance of the application because these functions operate on internal control variables that are global, that is, apply to all threads. For example, if parallel user threads call these functions to set different numbers of threads for the

same function domain, the number of threads actually set is unpredictable. To avoid this kind of data races, use the `mkl_set_num_threads_local` function (see the "Support Functions" chapter in the *Intel MKL Reference Manual* for the function description).

See Also

Using Additional Threading Control
Linking with Compiler Run-time Libraries

Techniques to Set the Number of Threads

Use the following techniques to specify the number of threads to use in Intel MKL:

- Set one of the OpenMP or Intel MKL environment variables:
  - `OMP_NUM_THREADS`
  - `MKL_NUM_THREADS`
  - `MKL_DOMAIN_NUM_THREADS`
- Call one of the OpenMP or Intel MKL functions:
  - `omp_set_num_threads()`
  - `mkl_set_num_threads()`
  - `mkl_domain_set_num_threads()`
  - `mkl_set_num_threads_local()`

When choosing the appropriate technique, take into account the following rules:

- The Intel MKL threading controls take precedence over the OpenMP controls because they are inspected first.
- A function call takes precedence over any environment settings. The exception, which is a consequence of the previous rule, is that a call to the OpenMP subroutine `omp_set_num_threads()` does not have precedence over the settings of Intel MKL environment variables such as `MKL_NUM_THREADS`. See Using Additional Threading Control for more details.
- You cannot change run-time behavior in the course of the run using the environment variables because they are read only once at the first call to Intel MKL.

Setting the Number of Threads Using an OpenMP* Environment Variable

You can set the number of threads using the environment variable `OMP_NUM_THREADS`. To change the number of threads, in the command shell in which the program is going to run, enter:

`export OMP_NUM_THREADS=<number of threads to use>`.

See Also

Using Additional Threading Control

Changing the Number of Threads at Run Time

You cannot change the number of threads during run time using environment variables. However, you can call OpenMP API functions from your program to change the number of threads during run time. The following sample code shows how to change the number of threads during run time using the `omp_set_num_threads()` routine. See also Techniques to Set the Number of Threads.

The following example shows both C and Fortran code examples. To run this example in the C language, use the `omp.h` header file from the Intel(R) compiler package. If you do not have the Intel compiler but wish to explore the functionality in the example, use Fortran API for `omp_set_num_threads()` rather than the C version. For example, `omp_set_num_threads_( &i_one );`

```
// ******* C language *******
#include "omp.h"
```

```
#include "mkl.h"
#include <stdio.h>
#define SIZE 1000
int main(int args, char *argv[]){
double *a, *b, *c;
a = (double*)malloc(sizeof(double)*SIZE*SIZE);
b = (double*)malloc(sizeof(double)*SIZE*SIZE);
c = (double*)malloc(sizeof(double)*SIZE*SIZE);
double alpha=1, beta=1;
int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
char transa='n', transb='n';
for( i=0; i<SIZE; i++){
for( j=0; j<SIZE; j++){
a[i*SIZE+j]= (double)(i+j);
b[i*SIZE+j]= (double)(i*j);
c[i*SIZE+j]= (double)0;
}
}
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
}
omp_set_num_threads(1);
for( i=0; i<SIZE; i++){
for( j=0; j<SIZE; j++){
a[i*SIZE+j]= (double)(i+j);
b[i*SIZE+j]= (double)(i*j);
c[i*SIZE+j]= (double)0;
}
}
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
}
omp_set_num_threads(2);
for( i=0; i<SIZE; i++){
for( j=0; j<SIZE; j++){
a[i*SIZE+j]= (double)(i+j);
b[i*SIZE+j]= (double)(i*j);
c[i*SIZE+j]= (double)0;
}
}
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
}
free (a);
free (b);
free (c);
return 0;
}
```

```
// ******* Fortran language *******
PROGRAM DGEMM_DIFF_THREADS
INTEGER N, I, J
PARAMETER (N=100)
REAL*8 A(N,N),B(N,N),C(N,N)
REAL*8 ALPHA, BETA

ALPHA = 1.1
BETA = -1.2
DO I=1,N
```

```
DO J=1,N
A(I,J) = I+J
B(I,J) = I*j
C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
CALL OMP_SET_NUM_THREADS(1);
DO I=1,N
DO J=1,N
A(I,J) = I+J
B(I,J) = I*j
C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
CALL OMP_SET_NUM_THREADS(2);
DO I=1,N
DO J=1,N
A(I,J) = I+J
B(I,J) = I*j
C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
STOP
END
```

Using Additional Threading Control

Intel MKL-specific Environment Variables for Threading Control

Intel MKL provides optional threading controls, that is, the environment variables and support functions that are independent of OpenMP. They behave similar to their OpenMP equivalents, but take precedence over them in the meaning that the Intel MKL-specific threading controls are inspected first. By using these controls along with OpenMP variables, you can thread the part of the application that does not call Intel MKL and the library independently of each other.

These controls enable you to specify the number of threads for Intel MKL independently of the OpenMP settings. Although Intel MKL may actually use a different number of threads from the number suggested, the controls will also enable you to instruct the library to try using the suggested number when the number used in the calling application is unavailable.

NOTE Sometimes Intel MKL does not have a choice on the number of threads for certain reasons, such as system resources.

Use of the Intel MKL threading controls in your application is optional. If you do not use them, the library will mainly behave the same way as Intel MKL 9.1 in what relates to threading with the possible exception of a different default number of threads.

Section "Number of User Threads" in the "Fourier Transform Functions" chapter of the *Intel MKL Reference Manual* shows how the Intel MKL threading controls help to set the number of threads for the FFT computation.

The table below lists the Intel MKL environment variables for threading control, their equivalent functions, and OMP counterparts:

Environment Variable	Support Function	Comment	Equivalent OpenMP* Environment Variable
MKL_NUM_THREADS	mkl_set_num_threads mkl_set_num_threads_local	Suggests the number of threads to use.	OMP_NUM_THREADS
MKL_DOMAIN_NUM_THREADS	mkl_domain_set_num_threads	Suggests the number of threads for a particular function domain.	
MKL_DYNAMIC	mkl_set_dynamic	Enables Intel MKL to dynamically change the number of threads.	OMP_DYNAMIC

NOTE The functions take precedence over the respective environment variables. Therefore, if you want Intel MKL to use a given number of threads in your application and do not want users of your application to change this number using environment variables, set the number of threads by a call to `mkl_set_num_threads()`, which will have full precedence over any environment variables being set.

The example below illustrates the use of the Intel MKL function `mkl_set_num_threads()` to set one thread.

```
// ******* C language *******
#include <omp.h>
#include <mkl.h>
...
mkl_set_num_threads ( 1 );
```

```
// ******* Fortran language *******
...
call mkl_set_num_threads( 1 )
```

See the *Intel MKL Reference Manual* for the detailed description of the threading control functions, their parameters, calling syntax, and more code examples.

MKL_DYNAMIC

The `MKL_DYNAMIC` environment variable enables Intel MKL to dynamically change the number of threads.

The default value of `MKL_DYNAMIC` is `TRUE`, regardless of `OMP_DYNAMIC`, whose default value may be `FALSE`.

When `MKL_DYNAMIC` is `TRUE`, Intel MKL tries to use what it considers the best number of threads, up to the maximum number you specify.

For example, `MKL_DYNAMIC` set to `TRUE` enables optimal choice of the number of threads in the following cases:

- If the requested number of threads exceeds the number of physical cores (perhaps because of using the Intel® Hyper-Threading Technology), and `MKL_DYNAMIC` is not changed from its default value of `TRUE`, Intel MKL will scale down the number of threads to the number of physical cores.

- If you are able to detect the presence of MPI, but cannot determine if it has been called in a thread-safe mode (it is impossible to detect this with MPICH 1.2.x, for instance), and MKL_DYNAMIC has not been changed from its default value of TRUE, Intel MKL will run one thread.

When MKL_DYNAMIC is FALSE, Intel MKL tries not to deviate from the number of threads the user requested. However, setting MKL_DYNAMIC=FALSE does not ensure that Intel MKL will use the number of threads that you request. The library may have no choice on this number for such reasons as system resources. Additionally, the library may examine the problem and use a different number of threads than the value suggested. For example, if you attempt to do a size one matrix-matrix multiply across eight threads, the library may instead choose to use only one thread because it is impractical to use eight threads in this event.

Note also that if Intel MKL is called in a parallel region, it will use only one thread by default. If you want the library to use nested parallelism, and the thread within a parallel region is compiled with the same OpenMP compiler as Intel MKL is using, you may experiment with setting MKL_DYNAMIC to FALSE and manually increasing the number of threads.

In general, set MKL_DYNAMIC to FALSE only under circumstances that Intel MKL is unable to detect, for example, to use nested parallelism where the library is already called from a parallel section.

MKL_DOMAIN_NUM_THREADS

The MKL_DOMAIN_NUM_THREADS environment variable suggests the number of threads for a particular function domain.

MKL_DOMAIN_NUM_THREADS accepts a string value <MKL-env-string>, which must have the following format:

<MKL-env-string> ::= *<MKL-domain-env-string>* { *<delimiter><MKL-domain-env-string>* }

<delimiter> ::= [*<space-symbol>**] (*<space-symbol>* | *<comma-symbol>* | *<semicolon-symbol>* | *<colon-symbol>*) [*<space-symbol>**]

<MKL-domain-env-string> ::= *<MKL-domain-env-name><uses><number-of-threads>*

<MKL-domain-env-name> ::= MKL_DOMAIN_ALL | MKL_DOMAIN_BLAS | MKL_DOMAIN_FFT | MKL_DOMAIN_VML | MKL_DOMAIN_PARDISO

<uses> ::= [*<space-symbol>**] (*<space-symbol>* | *<equality-sign>* | *<comma-symbol>*) [*<space-symbol>**]

<number-of-threads> ::= *<positive-number>*

<positive-number> ::= *<decimal-positive-number>* | *<octal-number>* | *<hexadecimal-number>*

In the syntax above, values of *<MKL-domain-env-name>* indicate function domains as follows:

MKL_DOMAIN_ALL	All function domains
MKL_DOMAIN_BLAS	BLAS Routines
MKL_DOMAIN_FFT	Fourier Transform Functions
MKL_DOMAIN_VML	Vector Mathematical Functions
MKL_DOMAIN_PARDISO	PARDISO

For example,

```
MKL_DOMAIN_ALL 2 : MKL_DOMAIN_BLAS 1 : MKL_DOMAIN_FFT 4
MKL_DOMAIN_ALL=2 : MKL_DOMAIN_BLAS=1 : MKL_DOMAIN_FFT=4
MKL_DOMAIN_ALL=2,  MKL_DOMAIN_BLAS=1,  MKL_DOMAIN_FFT=4
MKL_DOMAIN_ALL=2;  MKL_DOMAIN_BLAS=1;  MKL_DOMAIN_FFT=4
MKL_DOMAIN_ALL  = 2  MKL_DOMAIN_BLAS 1 ,  MKL_DOMAIN_FFT  4
MKL_DOMAIN_ALL,2: MKL_DOMAIN_BLAS 1, MKL_DOMAIN_FFT,4 .
```

The global variables MKL_DOMAIN_ALL, MKL_DOMAIN_BLAS, MKL_DOMAIN_FFT, MKL_DOMAIN_VML, and MKL_DOMAIN_PARDISO, as well as the interface for the Intel MKL threading control functions, can be found in the mkl.h header file.

The table below illustrates how values of MKL_DOMAIN_NUM_THREADS are interpreted.

Value of MKL_DOMAIN_NUM_THREADS	Interpretation
MKL_DOMAIN_ALL=4	All parts of Intel MKL should try four threads. The actual number of threads may be still different because of the MKL_DYNAMIC setting or system resource issues. The setting is equivalent to MKL_NUM_THREADS = 4.
MKL_DOMAIN_ALL=1, MKL_DOMAIN_BLAS=4	All parts of Intel MKL should try one thread, except for BLAS, which is suggested to try four threads.
MKL_DOMAIN_VML=2	VML should try two threads. The setting affects no other part of Intel MKL.

Be aware that the domain-specific settings take precedence over the overall ones. For example, the "MKL_DOMAIN_BLAS=4" value of MKL_DOMAIN_NUM_THREADS suggests trying four threads for BLAS, regardless of later setting MKL_NUM_THREADS, and a function call "mkl_domain_set_num_threads (4, MKL_DOMAIN_BLAS);" suggests the same, regardless of later calls to mkl_set_num_threads(). However, a function call with input "MKL_DOMAIN_ALL", such as "mkl_domain_set_num_threads (4, MKL_DOMAIN_ALL);" is equivalent to "mkl_set_num_threads(4)", and thus it will be overwritten by later calls to mkl_set_num_threads. Similarly, the environment setting of MKL_DOMAIN_NUM_THREADS with "MKL_DOMAIN_ALL=4" will be overwritten with MKL_NUM_THREADS = 2.

Whereas the MKL_DOMAIN_NUM_THREADS environment variable enables you set several variables at once, for example, "MKL_DOMAIN_BLAS=4,MKL_DOMAIN_FFT=2", the corresponding function does not take string syntax. So, to do the same with the function calls, you may need to make several calls, which in this example are as follows:

```
mkl_domain_set_num_threads ( 4, MKL_DOMAIN_BLAS );
mkl_domain_set_num_threads ( 2, MKL_DOMAIN_FFT );
```

Setting the Environment Variables for Threading Control

To set the environment variables used for threading control, in the command shell in which the program is going to run, enter:

```
export <VARIABLE NAME>=<value>
```

For example:

```
export MKL_NUM_THREADS=4
export MKL_DOMAIN_NUM_THREADS="MKL_DOMAIN_ALL=1, MKL_DOMAIN_BLAS=4"
export MKL_DYNAMIC=FALSE
```

Tips and Techniques to Improve Performance

Coding Techniques

To obtain the best performance with Intel MKL, ensure the following data alignment in your source code:

- Align arrays on 128-byte boundaries. See Example of Data Alignment for how to do it.

- Make sure leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 128, where `element_size` is the size of an array element in bytes.
- For two-dimensional arrays, avoid leading dimension values divisible by 2048 bytes. For example, for a double-precision array, with `element_size` = 8, avoid leading dimensions 256, 512, 768, 1024, … (elements).

LAPACK Packed Routines

The routines with the names that contain the letters `HP, OP, PP, SP, TP, UP` in the matrix type and storage position (the second and third letters respectively) operate on the matrices in the packed format (see LAPACK "Routine Naming Conventions" sections in the Intel MKL Reference Manual). Their functionality is strictly equivalent to the functionality of the unpacked routines with the names containing the letters `HE, OR, PO, SY, TR, UN` in the same positions, but the performance is significantly lower.

If the memory restriction is not too tight, use an unpacked routine for better performance. In this case, you need to allocate $N^2/2$ more memory than the memory required by a respective packed routine, where N is the problem size (the number of equations).

For example, to speed up solving a symmetric eigenproblem with an expert driver, use the unpacked routine:

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork,
ifail, info)
```

where `a` is the dimension *lda*-by-*n*, which is at least N^2 elements,
instead of the packed routine:

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)
```

where `ap` is the dimension $N*(N+1)/2$.

FFT Functions

Additional conditions can improve performance of the FFT functions.

The addresses of the first elements of arrays and the leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays should be divisible by cache line size, which equals64 bytes.

Hardware Configuration Tips

Dual-Core Intel® Xeon® processor 5100 series systems

To get the best performance with Intel MKL on Dual-Core Intel® Xeon® processor 5100 series systems, enable the Hardware DPL (streaming data) Prefetcher functionality of this processor. To configure this functionality, use the appropriate BIOS settings, as described in your BIOS documentation.

Intel® Hyper-Threading Technology

Intel® Hyper-Threading Technology (Intel® HT Technology) is especially effective when each thread performs different types of operations and when there are under-utilized resources on the processor. However, Intel MKL fits neither of these criteria because the threaded portions of the library execute at high efficiencies using most of the available resources and perform identical operations on each thread. You may obtain higher performance by disabling Intel HT Technology.

If you run with Intel HT Technology enabled, performance may be especially impacted if you run on fewer threads than physical cores. Moreover, if, for example, there are two threads to every physical core, the thread scheduler may assign two threads to some cores and ignore the other cores altogether. If you are using the OpenMP* library of the Intel Compiler, read the respective User Guide on how to best set the thread affinity interface to avoid this situation. For Intel MKL, apply the following setting:

```
set KMP_AFFINITY=granularity=fine,compact,1,0
```

See Also

Using Parallelism of the Intel® Math Kernel Library

Operating on Denormals

The IEEE 754-2008 standard, "An IEEE Standard for Binary Floating-Point Arithmetic", defines *denormal* (or *subnormal*) numbers as non-zero numbers smaller than the smallest possible normalized numbers for a specific floating-point format. Floating-point operations on denormals are slower than on normalized operands because denormal operands and results are usually handled through a software assist mechanism rather than directly in hardware. This software processing causes Intel MKL functions that consume denormals to run slower than with normalized floating-point numbers.

You can mitigate this performance issue by setting the appropriate bit fields in the MXCSR floating-point control register to flush denormals to zero (FTZ) or to replace any denormals loaded from memory with zero (DAZ). Check your compiler documentation to determine whether it has options to control FTZ and DAZ. Note that these compiler options may slightly affect accuracy.

FFT Optimized Radices

You can improve the performance of Intel MKL FFT if the length of your data vector permits factorization into powers of optimized radices.

In Intel MKL, the optimized radices are 2, 3, 5, 7, 11, and 13.

Using Memory Management

Intel MKL Memory Management Software

Intel MKL has memory management software that controls memory buffers for the use by the library functions. New buffers that the library allocates when your application calls Intel MKL are not deallocated until the program ends. To get the amount of memory allocated by the memory management software, call the `mkl_mem_stat()` function. If your program needs to free memory, call `mkl_free_buffers()`. If another call is made to a library function that needs a memory buffer, the memory manager again allocates the buffers and they again remain allocated until either the program ends or the program deallocates the memory. This behavior facilitates better performance. However, some tools may report this behavior as a memory leak.

The memory management software is turned on by default. To turn it off, set the `MKL_DISABLE_FAST_MM` environment variable to any value or call the `mkl_disable_fast_mm()` function. Be aware that this change may negatively impact performance of some Intel MKL routines, especially for small problem sizes.

Redefining Memory Functions

In C/C++ programs, you can replace Intel MKL memory functions that the library uses by default with your own functions. To do this, use the *memory renaming* feature.

Memory Renaming

Intel MKL memory management by default uses standard C run-time memory functions to allocate or free memory. These functions can be replaced using memory renaming.

Intel MKL accesses the memory functions by pointers `i_malloc`, `i_free`, `i_calloc`, and `i_realloc`, which are visible at the application level. These pointers initially hold addresses of the standard C run-time memory functions `malloc`, `free`, `calloc`, and `realloc`, respectively. You can programmatically redefine values of these pointers to the addresses of your application's memory management functions.

Redirecting the pointers is the only correct way to use your own set of memory management functions. If you call your own memory functions without redirecting the pointers, the memory will get managed by two independent memory management packages, which may cause unexpected memory issues.

How to Redefine Memory Functions

To redefine memory functions, use the following procedure:

1. Include the `i_malloc.h` header file in your code.
   This header file contains all declarations required for replacing the memory allocation functions. The header file also describes how memory allocation can be replaced in those Intel libraries that support this feature.
2. Redefine values of pointers `i_malloc`, `i_free`, `i_calloc`, and `i_realloc` prior to the first call to MKL functions, as shown in the following example:

```
#include "i_malloc.h"
  . . .
  i_malloc  = my_malloc;
  i_calloc  = my_calloc;
  i_realloc = my_realloc;
  i_free    = my_free;
  . . .
// Now you may call Intel MKL functions
```

Language-specific Usage Options 6

The Intel® Math Kernel Library (Intel® MKL) provides broad support for Fortran and C/C++ programming. However, not all functions support both Fortran and C interfaces. For example, some LAPACK functions have no C interface. You can call such functions from C using mixed-language programming.

If you want to use LAPACK or BLAS functions that support Fortran 77 in the Fortran 95 environment, additional effort may be initially required to build compiler-specific interface libraries and modules from the source code provided with Intel MKL.

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

Using Language-Specific Interfaces with Intel® Math Kernel Library

This section discusses mixed-language programming and the use of language-specific interfaces with Intel MKL.

See also *Appendix G in the Intel MKL Reference Manual* for details of the FFTW interfaces to Intel MKL.

Interface Libraries and Modules

You can create the following interface libraries and modules using the respective makefiles located in the interfaces directory.

File name	Contains
Libraries, in Intel MKL architecture-specific directories	
`libmkl_blas95.a`[1]	Fortran 95 wrappers for BLAS (BLAS95) for IA-32 architecture.
`libmkl_blas95_ilp64.a`[1]	Fortran 95 wrappers for BLAS (BLAS95) supporting LP64 interface.
`libmkl_blas95_lp64.a`[1]	Fortran 95 wrappers for BLAS (BLAS95) supporting ILP64 interface.
`libmkl_lapack95.a`[1]	Fortran 95 wrappers for LAPACK (LAPACK95) for IA-32 architecture.
`libmkl_lapack95_lp64.a`[1]	Fortran 95 wrappers for LAPACK (LAPACK95) supporting LP64 interface.
`libmkl_lapack95_ilp64.a`[1]	Fortran 95 wrappers for LAPACK (LAPACK95) supporting ILP64 interface.

File name	Contains
libfftw2xc_intel.a[1]	Interfaces for FFTW version 2.x (C interface for Intel compilers) to call Intel MKL FFTs.
libfftw2xc_gnu.a	Interfaces for FFTW version 2.x (C interface for GNU compilers) to call Intel MKL FFTs.
libfftw2xf_intel.a	Interfaces for FFTW version 2.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
libfftw2xf_gnu.a	Interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
libfftw3xc_intel.a[2]	Interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs.
libfftw3xc_gnu.a	Interfaces for FFTW version 3.x (C interface for GNU compilers) to call Intel MKL FFTs.
libfftw3xf_intel.a[2]	Interfaces for FFTW version 3.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
libfftw3xf_gnu.a	Interfaces for FFTW version 3.x (Fortran interface for GNU compilers) to call Intel MKL FFTs.
Modules, in architecture- and interface-specific subdirectories of the Intel MKL include directory	
blas95.mod[1]	Fortran 95 interface module for BLAS (BLAS95).
lapack95.mod [1]	Fortran 95 interface module for LAPACK (LAPACK95).
f95_precision.mod[1]	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95.
mkl95_blas.mod[1]	Fortran 95 interface module for BLAS (BLAS95), identical to blas95.mod. To be removed in one of the future releases.
mkl95_lapack.mod[1]	Fortran 95 interface module for LAPACK (LAPACK95), identical to lapack95.mod. To be removed in one of the future releases.
mkl95_precision.mod[1]	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95, identical to f95_precision.mod. To be removed in one of the future releases.
mkl_service.mod[1]	Fortran 95 interface module for Intel MKL support functions.

[1] Prebuilt for the Intel® Fortran compiler

[2] FFTW3 interfaces are integrated with Intel MKL. Look into *<mkl directory>*/interfaces/fftw3x*/makefile for options defining how to build and where to place the standalone library with the wrappers.

See Also
Fortran 95 Interfaces to LAPACK and BLAS

Fortran 95 Interfaces to LAPACK and BLAS

Fortran 95 interfaces are compiler-dependent. Intel MKL provides the interface libraries and modules precompiled with the Intel® Fortran compiler. Additionally, the Fortran 95 interfaces and wrappers are delivered as sources. (For more information, see Compiler-dependent Functions and Fortran 90 Modules). If you are using a different compiler, build the appropriate library and modules with your compiler and link the library as a user's library:

1. Go to the respective directory *<mkl directory>*/interfaces/blas95 or *<mkl directory>*/interfaces/lapack95

2. Type one of the following commands depending on your architecture:
   - For the IA-32 architecture,

     `make libia32 INSTALL_DIR=<user dir>`
   - For the Intel® 64 architecture,

     `make libintel64 [interface=lp64|ilp64] INSTALL_DIR=<user dir>`

Important The parameter `INSTALL_DIR` is required.

As a result, the required library is built and installed in the `<user dir>/lib` directory, and the `.mod` files are built and installed in the `<user dir>/include/<arch>[/{lp64|ilp64}]` directory, where `<arch>` is one of `{ia32, intel64}`.

By default, the ifort compiler is assumed. You may change the compiler with an additional parameter of `make`:
`FC=<compiler>`.

For example, the command

`make libintel64 FC=pgf95 INSTALL_DIR=<userpgf95 dir> interface=lp64`

builds the required library and `.mod` files and installs them in subdirectories of `<userpgf95 dir>`.

To delete the library from the building directory, use one of the following commands:

- For the IA-32 architecture,

  `make cleania32 INSTALL_DIR=<user dir>`
- For the Intel® 64 architecture,

  `make cleanintel64 [interface=lp64|ilp64] INSTALL_DIR=<user dir>`
- For all the architectures,

  `make clean INSTALL_DIR=<user dir>`

CAUTION Even if you have administrative rights, avoid setting `INSTALL_DIR=../..` or `INSTALL_DIR=<mkl directory>` in a build or clean command above because these settings replace or delete the Intel MKL prebuilt Fortran 95 library and modules.

Compiler-dependent Functions and Fortran 90 Modules

Compiler-dependent functions occur whenever the compiler inserts into the object code function calls that are resolved in its run-time library (RTL). Linking of such code without the appropriate RTL will result in undefined symbols. Intel MKL has been designed to minimize RTL dependencies.

In cases where RTL dependencies might arise, the functions are delivered as source code and you need to compile the code with whatever compiler you are using for your application.

In particular, Fortran 90 modules result in the compiler-specific code generation requiring RTL support. Therefore, Intel MKL delivers these modules compiled with the Intel compiler, along with source code, to be used with different compilers.

Mixed-language Programming with the Intel Math Kernel Library

Appendix A: Intel(R) Math Kernel Library Language Interfaces Support lists the programming languages supported for each Intel MKL function domain. However, you can call Intel MKL routines from different language environments.

Calling LAPACK, BLAS, and CBLAS Routines from C/C++ Language Environments

Not all Intel MKL function domains support both C and Fortran environments. To use Intel MKL Fortran-style functions in C/C++ environments, you should observe certain conventions, which are discussed for LAPACK and BLAS in the subsections below.

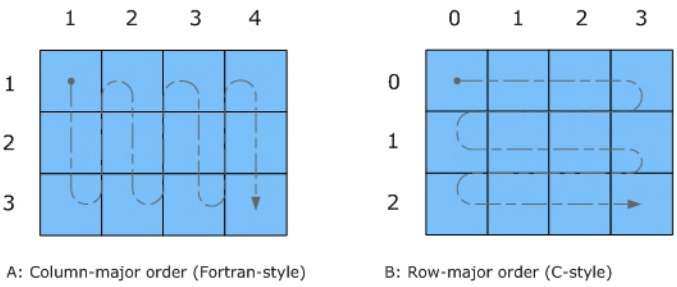
CAUTION Avoid calling BLAS 95/LAPACK 95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

LAPACK and BLAS

Because LAPACK and BLAS routines are Fortran-style, when calling them from C-language programs, follow the Fortran-style calling conventions:

- Pass variables by *address*, not by *value*.
  Function calls in Example "Calling a Complex BLAS Level 1 Function from C++" and Example "Using CBLAS Interface Instead of Calling BLAS Directly from C" illustrate this.
- Store your data in Fortran style, that is, column-major rather than row-major order.

With row-major order, adopted in C, the last array index changes most quickly and the first one changes most slowly when traversing the memory segment where the array is stored. With Fortran-style column-major order, the last index changes most slowly whereas the first index changes most quickly (as illustrated by the figure below for a two-dimensional array).



For example, if a two-dimensional matrix A of size `mxn` is stored densely in a one-dimensional array B, you can access a matrix element like this:

`A[i][j] = B[i*n+j]` in C (`i=0, ... , m-1, j=0, ... , -1`)

`A(i,j) = B((j-1)*m+i)` in Fortran (`i=1, ... , m, j=1, ... , n`).

When calling LAPACK or BLAS routines from C, be aware that because the Fortran language is case-insensitive, the routine names can be both upper-case or lower-case, with or without the trailing underscore. For example, the following names are equivalent:

- LAPACK: `dgetrf, DGETRF, dgetrf_`, and `DGETRF_`
- BLAS: `dgemm, DGEMM, dgemm_`, and `DGEMM_`

See Example "Calling a Complex BLAS Level 1 Function from C++" on how to call BLAS routines from C.

See also the Intel(R) MKL Reference Manual for a description of the C interface to LAPACK functions.

CBLAS

Instead of calling BLAS routines from a C-language program, you can use the CBLAS interface.

CBLAS is a C-style interface to the BLAS routines. You can call CBLAS routines using regular C-style calls. Use the `mkl.h` header file with the CBLAS interface. The header file specifies enumerated values and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. Example "Using CBLAS Interface Instead of Calling BLAS Directly from C" illustrates the use of the CBLAS interface.

C Interface to LAPACK

Instead of calling LAPACK routines from a C-language program, you can use the C interface to LAPACK provided by Intel MKL.

The C interface to LAPACK is a C-style interface to the LAPACK routines. This interface supports matrices in row-major and column-major order, which you can define in the first function argument *`matrix_order`*. Use the `mkl.h` header file with the C interface to LAPACK. `mkl.h` includes the `mkl_lapacke.h` header file, which specifies constants and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. You can find examples of the C interface to LAPACK in the `examples/lapacke` subdirectory in the Intel MKL installation directory.

Using Complex Types in C/C++

As described in the documentation for the Intel® Fortran Compiler XE, C/C++ does not directly implement the Fortran types `COMPLEX(4)` and `COMPLEX(8)`. However, you can write equivalent structures. The type `COMPLEX(4)` consists of two 4-byte floating-point numbers. The first of them is the real-number component, and the second one is the imaginary-number component. The type `COMPLEX(8)` is similar to `COMPLEX(4)` except that it contains two 8-byte floating-point numbers.

Intel MKL provides complex types `MKL_Complex8` and `MKL_Complex16`, which are structures equivalent to the Fortran complex types `COMPLEX(4)` and `COMPLEX(8)`, respectively. The `MKL_Complex8` and `MKL_Complex16` types are defined in the `mkl_types.h` header file. You can use these types to define complex data. You can also redefine the types with your own types before including the `mkl_types.h` header file. The only requirement is that the types must be compatible with the Fortran complex layout, that is, the complex type must be a pair of real numbers for the values of real and imaginary parts.

For example, you can use the following definitions in your C++ code:

```
#define MKL_Complex8 std::complex<float>
```

and

```
#define MKL_Complex16 std::complex<double>
```

See Example "Calling a Complex BLAS Level 1 Function from C++" for details. You can also define these types in the command line:

```
-DMKL_Complex8="std::complex<float>"
-DMKL_Complex16="std::complex<double>"
```

See Also

Intel® Software Documentation Library

Calling BLAS Functions that Return the Complex Values in C/C++ Code

Complex values that functions return are handled differently in C and Fortran. Because BLAS is Fortran-style, you need to be careful when handling a call from C to a BLAS function that returns complex values. However, in addition to normal function calls, Fortran enables calling functions as though they were subroutines, which provides a mechanism for returning the complex value correctly when the function is called from a C program. When a Fortran function is called as a subroutine, the return value is the first parameter in the calling sequence. You can use this feature to call a BLAS function from C.

The following example shows how a call to a Fortran function as a subroutine converts to a call from C and the hidden parameter result gets exposed:

Normal Fortran function call: `result = cdotc( n, x, 1, y, 1 )`

A call to the function as a subroutine: `call cdotc( result, n, x, 1, y, 1)`

A call to the function from C: `cdotc( &result, &n, x, &one, y, &one )`

NOTE Intel MKL has both upper-case and lower-case entry points in the Fortran-style (case-insensitive) BLAS, with or without the trailing underscore. So, all these names are equivalent and acceptable: `cdotc`, `CDOTC`, `cdotc_`, and `CDOTC_`.

The above example shows one of the ways to call several level 1 BLAS functions that return complex values from your C and C++ applications. An easier way is to use the CBLAS interface. For instance, you can call the same function using the CBLAS interface as follows:

```
cblas_cdotu( n, x, 1, y, 1, &result )
```

NOTE The complex value comes last on the argument list in this case.

The following examples show use of the Fortran-style BLAS interface from C and C++, as well as the CBLAS (C language) interface:

- Example "Calling a Complex BLAS Level 1 Function from C"
- Example "Calling a Complex BLAS Level 1 Function from C++"
- Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"

Example "Calling a Complex BLAS Level 1 Function from C"

The example below illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

```
#include "mkl.h"
#define N 5
int main()
{
int n = N, inca = 1, incb = 1, i;
MKL_Complex16 a[N], b[N], c;
for( i = 0; i < n; i++ ){
a[i].real = (double)i; a[i].imag = (double)i * 2.0;
b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.real, c.imag );
return 0;
}
```

Example "Calling a Complex BLAS Level 1 Function from C++"

Below is the C++ implementation:

```
#include <complex>
#include <iostream>
#define MKL_Complex16 std::complex<double>
#include "mkl.h"

#define N 5

int main()
{
    int n, inca = 1, incb = 1, i;
    std::complex<double> a[N], b[N], c;
```

```
    n = N;

    for( i = 0; i < n; i++ ){
        a[i] = std::complex<double>(i,i*2.0);
        b[i] = std::complex<double>(n-i,i*2.0);
    }
    zdotc(&c, &n, a, &inca, b, &incb );
    std::cout << "The complex dot product is: " << c << std::endl;
    return 0;
}
```

Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"

This example uses CBLAS:

```
#include <stdio.h>
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
#define N 5
int main()
{
int n, inca = 1, incb = 1, i;
complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb, &c );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
return 0;
}
```

Support for Boost uBLAS Matrix-matrix Multiplication

If you are used to uBLAS, you can perform BLAS matrix-matrix multiplication in C++ using Intel MKL substitution of Boost uBLAS functions. uBLAS is the Boost C++ open-source library that provides BLAS functionality for dense, packed, and sparse matrices. The library uses an expression template technique for passing expressions as function arguments, which enables evaluating vector and matrix expressions in one pass without temporary matrices. uBLAS provides two modes:

- Debug (safe) mode, default.
  Checks types and conformance.
- Release (fast) mode.
  Does not check types and conformance. To enable this mode, use the `NDEBUG` preprocessor symbol.

The documentation for the Boost uBLAS is available at www.boost.org.

Intel MKL provides overloaded `prod()` functions for substituting uBLAS dense matrix-matrix multiplication with the Intel MKL `gemm` calls. Though these functions break uBLAS expression templates and introduce temporary matrices, the performance advantage can be considerable for matrix sizes that are not too small (roughly, over 50).

You do not need to change your source code to use the functions. To call them:

- Include the header file `mkl_boost_ublas_matrix_prod.hpp` in your code (from the Intel MKL include directory)
- Add appropriate Intel MKL libraries to the link line.

The list of expressions that are substituted follows:

```
prod( m1, m2 )

prod( trans(m1), m2 )
```

```
prod( trans(conj(m1)), m2 )
prod( conj(trans(m1)), m2 )
prod( m1, trans(m2) )
prod( trans(m1), trans(m2) )
prod( trans(conj(m1)), trans(m2) )
prod( conj(trans(m1)), trans(m2) )
prod( m1, trans(conj(m2)) )
prod( trans(m1), trans(conj(m2)) )
prod( trans(conj(m1)), trans(conj(m2)) )
prod( conj(trans(m1)), trans(conj(m2)) )
prod( m1, conj(trans(m2)) )
prod( trans(m1), conj(trans(m2)) )
prod( trans(conj(m1)), conj(trans(m2)) )
prod( conj(trans(m1)), conj(trans(m2)) )
```

These expressions are substituted in the *release* mode only (with `NDEBUG` preprocessor symbol defined). Supported uBLAS versions are Boost 1.34.1 and higher. To get them, visit www.boost.org.

A code example provided in the *<mkl directory>*`/examples/ublas/source/sylvester.cpp` file illustrates usage of the Intel MKL uBLAS header file for solving a special case of the Sylvester equation.

To run the Intel MKL ublas examples, specify the `BOOST_ROOT` parameter in the `make` command, for instance, when using Boost version 1.37.0:

```
make libia32 BOOST_ROOT = <your_path>/boost_1_37_0
```

See Also

Using Code Examples

Invoking Intel MKL Functions from Java* Applications

Intel MKL Java* Examples

To demonstrate binding with Java, Intel MKL includes a set of Java examples in the following directory:

<mkl directory>`/examples/java`.

The examples are provided for the following MKL functions:

- `?gemm`, `?gemv`, and `?dot` families from CBLAS
- The complete set of FFT functions
- ESSL[1]-like functions for one-dimensional convolution and correlation
- VSL Random Number Generators (RNG), except user-defined ones and file subroutines
- VML functions, except `GetErrorCallBack`, `SetErrorCallBack`, and `ClearErrorCallBack`

You can see the example sources in the following directory:

<mkl directory>`/examples/java/examples`.

The examples are written in Java. They demonstrate usage of the MKL functions with the following variety of data:

- 1- and 2-dimensional data sequences
- Real and complex types of the data
- Single and double precision

However, the wrappers, used in the examples, do not:

- Demonstrate the use of large arrays (>2 billion elements)
- Demonstrate processing of arrays in native memory
- Check correctness of function parameters
- Demonstrate performance optimizations

The examples use the Java Native Interface (JNI* developer framework) to bind with Intel MKL. The JNI documentation is available from http://java.sun.com/javase/6/docs/technotes/guides/jni/.

The Java example set includes JNI wrappers that perform the binding. The wrappers do not depend on the examples and may be used in your Java applications. The wrappers for CBLAS, FFT, VML, VSL RNG, and ESSL-like convolution and correlation functions do not depend on each other.

To build the wrappers, just run the examples. The makefile builds the wrapper binaries. After running the makefile, you can run the examples, which will determine whether the wrappers were built correctly. As a result of running the examples, the following directories will be created in *<mkl directory>*/examples/java:

- docs
- include
- classes
- bin
- _results

The directories docs, include, classes, and bin will contain the wrapper binaries and documentation; the directory _results will contain the testing results.

For a Java programmer, the wrappers are the following Java classes:

- com.intel.mkl.CBLAS
- com.intel.mkl.DFTI
- com.intel.mkl.ESSL
- com.intel.mkl.VML
- com.intel.mkl.VSL

Documentation for the particular wrapper and example classes will be generated from the Java sources while building and running the examples. To browse the documentation, open the index file in the docs directory (created by the build script):

<mkl directory>/examples/java/docs/index.html.

The Java wrappers for CBLAS, VML, VSL RNG, and FFT establish the interface that directly corresponds to the underlying native functions, so you can refer to the Intel MKL Reference Manual for their functionality and parameters. Interfaces for the ESSL-like functions are described in the generated documentation for the com.intel.mkl.ESSL class.

Each wrapper consists of the interface part for Java and JNI stub written in C. You can find the sources in the following directory:

<mkl directory>/examples/java/wrappers.

Both Java and C parts of the wrapper for CBLAS and VML demonstrate the straightforward approach, which you may use to cover additional CBLAS functions.

The wrapper for FFT is more complicated because it needs to support the lifecycle for FFT descriptor objects. To compute a single Fourier transform, an application needs to call the FFT software several times with the same copy of the native FFT descriptor. The wrapper provides the handler class to hold the native descriptor, while the virtual machine runs Java bytecode.

The wrapper for VSL RNG is similar to the one for FFT. The wrapper provides the handler class to hold the native descriptor of the stream state.

The wrapper for the convolution and correlation functions mitigates the same difficulty of the VSL interface, which assumes a similar lifecycle for "task descriptors". The wrapper utilizes the ESSL-like interface for those functions, which is simpler for the case of 1-dimensional data. The JNI stub additionally encapsulates the MKL functions into the ESSL-like wrappers written in C and so "packs" the lifecycle of a task descriptor into a single call to the native method.

The wrappers meet the JNI Specification versions 1.1 and 5.0 and should work with virtually every modern implementation of Java.

The examples and the Java part of the wrappers are written for the Java language described in "The Java Language Specification (First Edition)" and extended with the feature of "inner classes" (this refers to late 1990s). This level of language version is supported by all versions of the Sun Java Development Kit* (JDK*) developer toolkit and compatible implementations starting from version 1.1.5, or by all modern versions of Java.

The level of C language is "Standard C" (that is, C89) with additional assumptions about integer and floating-point data types required by the Intel MKL interfaces and the JNI header files. That is, the native `float` and `double` data types must be the same as JNI `jfloat` and `jdouble` data types, respectively, and the native `int` must be 4 bytes long.

[1] IBM Engineering Scientific Subroutine Library (ESSL*).

See Also

Running the Java* Examples

Running the Java* Examples

The Java examples support all the C and C++ compilers that Intel MKL does. The makefile intended to run the examples also needs the make utility, which is typically provided with the Mac OS* X distribution.

To run Java examples, the JDK* developer toolkit is required for compiling and running Java code. A Java implementation must be installed on the computer or available via the network. You may download the JDK from the vendor website.

The examples should work for all versions of JDK. However, they were tested only with the following Java implementation for all the supported architectures:

- J2SE* SDK 1.4.2 and JDK 5.0 from Apple Computer, Inc. (http://apple.com/).

Note that the Java run-time environment* (JRE*) system, which may be pre-installed on your computer, is not enough. You need the JDK* developer toolkit that supports the following set of tools:

- java
- javac
- javah
- javadoc

To make these tools available for the examples makefile, set the `JAVA_HOME` environment variable and add the JDK binaries directory to the system `PATH`, for example :

```
export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/1.5/Home
export PATH=${JAVA_HOME}/bin:${PATH}
```

You may also need to clear the `JDK_HOME` environment variable, if it is assigned a value:

```
unset JDK_HOME
```

To start the examples, use the makefile found in the Intel MKL Java examples directory:

```
make {dylibia32|libia32} [function=...] [compiler=...]
```

If you type the make command and omit the target (for example, `dylibia32`), the makefile prints the help info, which explains the targets and parameters.

For the examples list, see the `examples.lst` file in the Java examples directory.

Known Limitations of the Java* Examples

This section explains limitations of Java examples.

Functionality

Some Intel MKL functions may fail to work if called from the Java environment by using a wrapper, like those provided with the Intel MKL Java examples. Only those specific CBLAS, FFT, VML, VSL RNG, and the convolution/correlation functions listed in the Intel MKL Java Examples section were tested with the Java environment. So, you may use the Java wrappers for these CBLAS, FFT, VML, VSL RNG, and convolution/correlation functions in your Java applications.

Performance

The Intel MKL functions must work faster than similar functions written in pure Java. However, the main goal of these wrappers is to provide code examples, not maximum performance. So, an Intel MKL function called from a Java application will probably work slower than the same function called from a program written in C/C++ or Fortran.

Known bugs

There are a number of known bugs in Intel MKL (identified in the Release Notes), as well as incompatibilities between different versions of JDK. The examples and wrappers include workarounds for these problems. Look at the source code in the examples and wrappers for comments that describe the workarounds.

Known Limitations of the Java* Examples

7

This section explains limitations of Java examples.

Functionality

Some Intel MKL functions may fail to work if called from the Java environment by using a wrapper, like those provided with the Intel MKL Java examples. Only those specific CBLAS, FFT, VML, VSL RNG, and the convolution/correlation functions listed in the Intel MKL Java Examples section were tested with the Java environment. So, you may use the Java wrappers for these CBLAS, FFT, VML, VSL RNG, and convolution/correlation functions in your Java applications.

Performance

The Intel MKL functions must work faster than similar functions written in pure Java. However, the main goal of these wrappers is to provide code examples, not maximum performance. So, an Intel MKL function called from a Java application will probably work slower than the same function called from a program written in C/C++ or Fortran.

Known bugs

There are a number of known bugs in Intel MKL (identified in the Release Notes), as well as incompatibilities between different versions of JDK. The examples and wrappers include workarounds for these problems. Look at the source code in the examples and wrappers for comments that describe the workarounds.

8 Coding Tips

This section discusses programming with the Intel® Math Kernel Library (Intel® MKL) to provide coding tips that meet certain, specific needs, such as data alignment or conditional compilation.

Example of Data Alignment

Needs for best performance with Intel MKL or for reproducible results from run to run of Intel MKL functions require alignment of data arrays.

To align an array on 128-byte boundaries, use `mkl_malloc()` in place of system provided memory allocators, as shown in the code example below.

Aligning Addresses on 128-byte Boundaries

```
// ******* C language *******
...
#include <stdlib.h>
#include <mkl.h>
...
void *darray;
int workspace;
// Set value of alignment
int alignment=128;
...
// Allocate aligned workspace
darray = mkl_malloc( sizeof(double)*workspace, alignment );
...
// call the program using MKL
mkl_app( darray );
...
// Free workspace
mkl_free( darray );
```

```
! ******* Fortran language *******
...
! Set value of alignment
integer    alignment
parameter (alignment=128)
...
! Declare Intel MKL routines
#ifdef _IA32
integer mkl_malloc
#else
integer*8 mkl_malloc
#endif
external mkl_malloc, mkl_free, mkl_app
...
double precision darray
pointer (p_wrk,darray(1))
integer workspace
...
! Allocate aligned workspace
p_wrk = mkl_malloc( %val(8*workspace), %val(alignment) )
...
! call the program using Intel MKL
call mkl_app( darray )
...
! Free workspace
call mkl_free(p_wrk)
```

Using Predefined Preprocessor Symbols for Intel® MKL Version-Dependent Compilation

Preprocessor symbols (macros) substitute values in a program before it is compiled. The substitution is performed in the preprocessing phase.

The following preprocessor symbols are available:

Predefined Preprocessor Symbol	Description
`__INTEL_MKL__`	Intel MKL major version
`__INTEL_MKL_MINOR__`	Intel MKL minor version
`__INTEL_MKL_UPDATE__`	Intel MKL update number
`INTEL_MKL_VERSION`	Intel MKL full version in the following format: `INTEL_MKL_VERSION = (__INTEL_MKL__*100+__INTEL_MKL_MINOR__)*100+__INTEL_MKL_UPDATE__`

These symbols enable conditional compilation of code that uses new features introduced in a particular version of the library.

To perform conditional compilation:

1. Include in your code the file where the macros are defined:
   - `mkl.h` for C/C++
   - `mkl.fi` for Fortran
2. [Optionally] Use the following preprocessor directives to check whether the macro is defined:
   - `#ifdef, #endif` for C/C++
   - `!DEC$IF DEFINED, !DEC$ENDIF` for Fortran
3. Use preprocessor directives for conditional inclusion of code:
   - `#if, #endif` for C/C++
   - `!DEC$IF, !DEC$ENDIF` for Fortran

Example

Compile a part of the code if Intel MKL version is MKL 10.3 update 4:

C/C++:

```
#include "mkl.h"
#ifdef INTEL_MKL_VERSION
#if INTEL_MKL_VERSION ==  100304
//     Code to be conditionally compiled
#endif
#endif
```

Fortran:

```
    include "mkl.fi"
!DEC$IF DEFINED INTEL_MKL_VERSION
!DEC$IF INTEL_MKL_VERSION .EQ. 100304
*     Code to be conditionally compiled
!DEC$ENDIF
!DEC$ENDIF
```

Configuring Your Integrated Development Environment to Link with Intel Math Kernel Library

9

Configuring the Apple Xcode* Developer Software to Link with Intel® Math Kernel Library

This section provides information on configuring the Apple Xcode* developer software for linking with Intel MKL.

To configure your Xcode developer software to link with Intel MKL, you need to perform the steps explained below. The specific instructions for performing these steps depend on your version of the Xcode developer software. Please refer to the Xcode Help for more details.

To configure your Xcode developer software, do the following:

1. Open your project that uses Intel MKL and select the target you are going to build.
2. Add the Intel MKL include path, that is, `<mkl directory>/include`, to the header search paths.
3. Add the Intel MKL library path for the target architecture to the library search paths. For example, for the Intel® 64 architecture, add `<mkl directory>/lib/intel64`.
4. Specify the linker options for the Intel MKL and system libraries to link with your application. For example, you may need to specify: `-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lpthread -lm`.
5. (Optional, needed only for dynamic linking) For the active executable, add the environment variable `DYLD_LIBRARY_PATH` with the value of `<mkl directory>/lib`.

See Also

Notational Conventions
Linking in Detail

Intel® Optimized LINPACK Benchmark for Mac OS* X

10

Intel® Optimized LINPACK Benchmark is a generalization of the LINPACK 1000 benchmark. It solves a dense (`real`*8) system of linear equations (*Ax=b*), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate, and tests the results for accuracy. The generalization is in the number of equations (*N*) it can solve, which is not limited to 1000. It uses partial pivoting to assure the accuracy of the results.

Do not use this benchmark to report LINPACK 100 performance because that is a compiled-code only benchmark. This is a shared-memory (SMP) implementation which runs on a single platform. Do not confuse this benchmark with LINPACK, the library, which has been expanded upon by the LAPACK library.

Intel provides optimized versions of the LINPACK benchmarks to help you obtain high LINPACK benchmark results on your genuine Intel processor systems more easily than with the High Performance Linpack (HPL) benchmark. Use this package to benchmark your SMP machine.

Additional information on this software as well as other Intel software performance products is available at http://www.intel.com/software/products/.

Contents of the Intel® Optimized LINPACK Benchmark

The Intel Optimized LINPACK Benchmark for Mac OS* X contains the following files, located in the `./benchmarks/linpack/` subdirectory of the Intel® Math Kernel Library (Intel® MKL) directory:

File in `./benchmarks/linpack/`	**Description**
`linpack_cd32.app`	The 32-bit program executable for a system using Intel® Core™ Duo processor on Mac OS* X.
`linpack_cd64.app`	The 64-bit program executable for a system using Intel® Core™ microarchitecture on Mac OS* X.
`runme32`	A sample shell script for executing a pre-determined problem set for `linpack_cd32.app`.
`runme64`	A sample shell script for executing a pre-determined problem set for `linpack_cd64.app`.
`lininput`	Input file for a pre-determined problem for the `runme32` script.
`lin_cd32.txt`	Result of the `runme32` script execution.
`lin_cd64.txt`	Result of the `runme64` script execution.
`help.lpk`	Simple help file.
`xhelp.lpk`	Extended help file.

See Also

High-level Directory Structure

Running the Software

To obtain results for the pre-determined sample problem sizes on a given system, type one of the following, as appropriate:

```
./runme32
./runme64
```

To run the software for other problem sizes, see the extended help included with the program. Extended help can be viewed by running the program executable with the `-e` option:

```
./linpack_cd32.app -e
./linpack_cd64.app -e
```

The pre-defined data input file `lininput` is provided merely as an example. Different systems have different amounts of memory and thus require new input files. The extended help can be used for insight into proper ways to change the sample input files.

`lininput` requires at least 2 GB of memory.

If the system has less memory than the above sample data input requires, you may need to edit or create your own data input files, as explained in the extended help.

Each sample script uses the `OMP_NUM_THREADS` environment variable to set the number of processors it is targeting. To optimize performance on a different number of physical processors, change that line appropriately. If you run the Intel Optimized LINPACK Benchmark without setting the number of threads, it will default to the number of cores according to the OS. You can find the settings for this environment variable in the `runme*` sample scripts. If the settings do not yet match the situation for your machine, edit the script.

Known Limitations of the Intel® Optimized LINPACK Benchmark

The following limitations are known for the Intel Optimized LINPACK Benchmark for Mac OS* X:

- If an incomplete data input file is given, the binaries may either hang or fault. See the sample data input files and/or the extended help for insight into creating a correct data input file.
- The binary will hang if it is not given an input file or any other arguments.

Intel® Math Kernel Library Language Interfaces Support

A

Language Interfaces Support, by Function Domain

The following table shows language interfaces that Intel® Math Kernel Library (Intel® MKL) provides for each function domain. However, Intel MKL routines can be called from other languages using mixed-language programming. See Mixed-language Programming with Intel® MKL for an example of how to call Fortran routines from C/C++.

Function Domain	FORTRAN 77 interface	Fortran 90/95 interface	C/C++ interface
Basic Linear Algebra Subprograms (BLAS)	Yes	Yes	via CBLAS
BLAS-like extension transposition routines	Yes		Yes
Sparse BLAS Level 1	Yes	Yes	via CBLAS
Sparse BLAS Level 2 and 3	Yes	Yes	Yes
LAPACK routines for solving systems of linear equations	Yes	Yes	Yes
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	Yes	Yes	Yes
Auxiliary and utility LAPACK routines	Yes		Yes
DSS/PARDISO* solvers	Yes	Yes	Yes
Other Direct and Iterative Sparse Solver routines	Yes	Yes	Yes
Vector Mathematical Library (VML) functions	Yes	Yes	Yes
Vector Statistical Library (VSL) functions	Yes	Yes	Yes
Fourier Transform functions (FFT)		Yes	Yes
Trigonometric Transform routines		Yes	Yes
Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines		Yes	Yes
Optimization (Trust-Region) Solver routines	Yes	Yes	Yes
Data Fitting functions	Yes	Yes	Yes
Support functions (including memory allocation)	Yes	Yes	Yes

Include Files

The table below lists Intel MKL include files.

NOTE The `*.f90` include files supersede the `*.f77` include files and can be used for FORTRAN 77 as well as for later versions of Fortran. However, the `*.f77` files are kept for backward compatibility.

Function domain	Fortran Include Files	C/C++ Include Files
All function domains	`mkl.fi`	`mkl.h`
BLAS Routines	`blas.f90` `mkl_blas.fi`†	`mkl_blas.h`‡
BLAS-like Extension Transposition Routines	`mkl_trans.fi`†	`mkl_trans.h`‡
CBLAS Interface to BLAS		`mkl_cblas.h`‡
Sparse BLAS Routines	`mkl_spblas.fi`†	`mkl_spblas.h`‡
LAPACK Routines	`lapack.f90` `mkl_lapack.fi`†	`mkl_lapack.h`‡
C Interface to LAPACK		`mkl_lapacke.h`‡
All Sparse Solver Routines	`mkl_solver.f90` `mkl_solver.fi`†	`mkl_solver.h`‡
PARDISO	`mkl_pardiso.f90` `mkl_pardiso.fi`†	`mkl_pardiso.h`‡
DSS Interface	`mkl_dss.f90` `mkl_dss.fi`†	`mkl_dss.h`‡
RCI Iterative Solvers ILU Factorization	`mkl_rci.fi`†	`mkl_rci.h`‡
Optimization Solver Routines	`mkl_rci.fi`†	`mkl_rci.h`‡

Vector Mathematical Functions	`mkl_vml.90`
	`mkl_vml.fi`†
	`mkl_vml.f77`
		`mkl_vml.h`‡

Vector Statistical Functions	`mkl_vsl.f90`
	`mkl_vsl.fi`†
	`mkl_vsl.f77`
		`mkl_vsl.h`‡

Fourier Transform Functions	`mkl_dfti.f90`	`mkl_dfti.h`‡
Partial Differential Equations Support Routines		
Trigonometric Transforms	`mkl_trig_transforms.f90`	`mkl_trig_transform.h`‡
Poisson Solvers	`mkl_poisson.f90`	`mkl_poisson.h`‡
Data Fitting functions	`mkl_df.f90` `mkl_df.f77`	`mkl_df.h`‡
Support functions	`mkl_service.f90` `mkl_service.fi`†	`mkl_service.h`‡
Declarations for replacing memory allocation functions. See Redefining Memory Functions for details.		`i_malloc.h`

† You can use the `mkl.fi` include file in your code instead.

‡ You can include the `mkl.h` header file in your code instead.

See Also

Language Interfaces Support, by Function Domain

Support for Third-Party Interfaces B

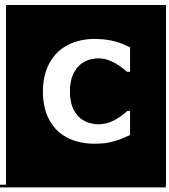
FFTW Interface Support

Intel® Math Kernel Library (Intel® MKL) offers two collections of wrappers for the FFTW interface (www.fftw.org). The wrappers are the superstructure of FFTW to be used for calling the Intel MKL Fourier transform functions. These collections correspond to the FFTW versions 2.x and 3.x and the Intel MKL versions 7.0 and later.

These wrappers enable using Intel MKL Fourier transforms to improve the performance of programs that use FFTW without changing the program source code. See the "*FFTW Interface to Intel® Math Kernel Library*" appendix in the Intel MKL Reference Manual for details on the use of the wrappers.

Important For ease of use, FFTW3 interface is also integrated in Intel MKL.

Directory Structure in Detail



Tables in this section show contents of the *<mkl directory>*/lib directory.

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

Static Libraries in the lib directory

File	Contents
Interface layer	
libmkl_intel.a	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support LP64 interface or on IA-32 architecture systems.
libmkl_intel_lp64.a	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support LP64 interface or on IA-32 architecture systems.
libmkl_intel_ilp64.a	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support ILP64 interface or on IA-32 architecture systems.
libmkl_intel_sp2dp.a	SP2DP interface library for the Intel compilers.
Threading layer	
libmkl_intel_thread.a	Threading library for the Intel compilers
libmkl_pgi_thread.a	Threading library for the PGI* compiler
libmkl_sequential.a	Sequential library
Computational layer	
libmkl_core.a	Kernel library
libmkl_solver_lp64.a	Deprecated. Empty library for backward compatibility
libmkl_solver_lp64_sequential.a	Deprecated. Empty library for backward compatibility
libmkl_solver_ilp64.a	Deprecated. Empty library for backward compatibility
libmkl_solver_ilp64_sequential.a	Deprecated. Empty library for backward compatibility

Dynamic Libraries in the lib directory

File	Contents
libmkl_rt.dylib	Single Dynamic Library
Interface layer	
libmkl_intel.dylib	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support LP64 interface or on IA-32 architecture systems.
libmkl_intel_lp64.dylib	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support LP64 interface or on IA-32 architecture systems.
libmkl_intel_ilp64.dylib	Interface library for the Intel compilers. To be used on Intel® 64 architecture systems to support ILP64 interface or on IA-32 architecture systems.
libmkl_intel_sp2dp.dylib	SP2DP interface library for the Intel compilers.
Threading layer	
libmkl_intel_thread.dylib	Threading library for the Intel compilers
libmkl_sequential.dylib	Sequential library
Computational layer	
libmkl_core.dylib	Contains the dispatcher for dynamic load of the processor-specific kernel library
libmkl_lapack.dylib	LAPACK and DSS/PARDISO routines and drivers
libmkl_mc.dylib	64-bit kernel for processors based on the Intel® Core™ microarchitecture
libmkl_mc3.dylib	64-bit kernel for the Intel® Core™ i7 processors
libmkl_p4p.dylib	32-bit kernel for the Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), including Intel® Core™ Duo and Intel® Core™ Solo processors.
libmkl_p4m.dylib	32-bit kernel for the Intel® Core™ microarchitecture
libmkl_p4m3.dylib	32-bit kernel library for the Intel® Core™ i7 processors
libmkl_vml_mc.dylib	64-bit Vector Math Library (VML)/Vector Statistical Library (VSL)/ Data Fitting (DF) for processors based on the Intel® Core™ microarchitecture
libmkl_vml_mc2.dylib	64-bit VML/VSL/DF for 45nm Hi-k Intel® Core™2 and the Intel Xeon® processor families
libmkl_vml_mc3.dylib	64-bit VML/VSL/DF for the Intel® Core™ i7 processors
libmkl_vml_p4p.dylib	32-bit VML/VSL/DF for the Intel® Pentium® 4 processor with Intel SSE3
libmkl_vml_p4m.dylib	32-bit VML/VSL/DF for processors based on Intel® Core™ microarchitecture

File	Contents
libmkl_vml_p4m2.dylib	32-bit VML/VSL/DF for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
libmkl_vml_p4m3.dylib	32-bit VML/VSL/DF for the Intel® Core™ i7 processors
libmkl_vml_avx.dylib	VML/VSL/DF optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
libmkl_vml_cmpt.dylib	VML/VSL/DF library for conditional bitwise reproducibility
RTL	
locale/en_US/mkl_msg.cat	Catalog of Intel® Math Kernel Library (Intel® MKL) messages in English

Index

N

P

S

T

U

X