

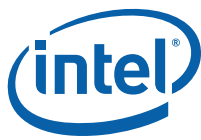
Intel® Math Kernel Library for Mac OS* X

User's Guide

March 2009

Document Number: 315932-008US

World Wide Web: <http://www.intel.com/software/products/>



Version	Version Information	Date
-001	Original issue. Documents Intel® Math Kernel Library (Intel® MKL) 9.1 beta release.	January 2007
-002	Documents Intel® MKL 9.1 gold release. Document restructured. Section "Selecting Between Static and Dynamic Linking" added to chapter 5; sections "Changing the Number of Processors for Threading at Run Time" and "Using Intel MKL Memory Management" added to chapter 6.	June 2007
-003	Documents Intel® MKL 10.0 Beta release. Layered design model has been described in chapter 3 and the content of the entire book adjusted to the model. ILP64 interface has been described in chapter 3. New Intel MKL threading controls have been described in chapter 6.	September 2007
-004	Documents Intel® MKL 10.0 Gold release.	October 2007
-005	Documents Intel® MKL 10.1 beta release. Section "Accessing Man Pages" has been added to chapter 3. Screenshots have been added to instructions for configuring the Apple Xcode* development software. Intel® Compatibility OpenMP* run-time compiler library (<code>libiomp</code>) has been described. Section "Mixed-language programming with Intel® MKL" in chapter 7 has been considerably extended with description of techniques for calling LAPACK and BLAS functions from C. Section "Support for Boost uBLAS Matrix-Matrix Multiplication" has been added to chapter 7.	May 2008
-006	Documents Intel® MKL 10.1 gold release. Linking examples for IA-32 architecture and section "Linking with Computational Libraries" have been added to chapter 5. Integration of DSS/PARDISO into the layered structure has been documented. Two Fortran code examples have been added.	August 2008
-007	Documents Intel® MKL 10.2 beta release. Prebuilt Fortran 95 interface libraries and modules for BLAS and LAPACK have been described. Chapter 5 has been restructured.	January 2009
-008	Documents Intel® MKL 10.2 gold release. The document has been considerably restructured. The "Getting Started" chapter has been enhanced, as well as the description of the layered model concept. Description of the SP2DP interface has been added to Chapter 3. The Web-based linking advisor has been described and referenced in chapters 2 and 5.	March 2009



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.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

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 by visiting [Intel's Web Site](#).

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

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

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2007 - 2009, Intel Corporation. All rights reserved.

Contents

Chapter 1 Overview

Technical Support	1-1
About This Document	1-1
Document Organization	1-2
Term and Notational Conventions	1-3

Chapter 2 Getting Started

Before You Begin	2-1
Compiler Support	2-3
Steps to Get Started	2-3
Check Your Installation	2-3
Set Environment Variables	2-4
Use a Web-based Linking Advisor	2-4
Use Intel MKL Code Examples	2-4

Chapter 3 Intel® Math Kernel Library Structure

High-level Directory Structure	3-1
Layered Model Concept	3-2
Sequential Mode of the Library	3-4
Support for ILP64 Programming	3-5
Architecture Support	3-6
Directory Structure in Detail	3-7
Accessing the Intel® MKL Documentation	3-14
Contents of the Documentation Directory	3-14
Accessing Man Pages	3-14

Chapter 4 Configuring Your Development Environment

Configuring the Apple Xcode* Developer Software to Link with Intel® MKL.....	4-1
Configuring the Out-of-Core (OOC) DSS/PARDISO* Solver.....	4-3

Chapter 5 Linking Your Application with the Intel® Math Kernel Library

Listing Libraries on a Link Line	5-2
Selecting Libraries to Link.....	5-2
Linking with Fortran 95 Interface Libraries	5-3
Linking with Threading Libraries	5-3
Linking with Computational Libraries	5-4
Linking with Compiler Support RTLs	5-6
Linking with System Libraries.....	5-6
Linking Examples	5-6
Building Custom Dynamically Linked Shared Libraries.....	5-9
Intel MKL Custom Dynamically Linked Shared Library Builder	5-10
Specifying Makefile Parameters	5-10
Specifying a List of Functions	5-11

Chapter 6 Managing Performance and Memory

Using the Intel® MKL Parallelism.....	6-1
Techniques to Set the Number of Threads	6-3
Avoiding Conflicts in the Execution Environment	6-3
Setting the Number of Threads Using OpenMP* Environment Variable	6-4
Changing the Number of Threads at Run Time.....	6-5
Using Additional Threading Control	6-8
Tips and Techniques to Improve Performance.....	6-13
Coding Techniques.....	6-13
Hardware Configuration Tips	6-14
Operating on Denormals	6-14
FFT Optimized Radices	6-15
Using the Intel® MKL Memory Management.....	6-15
Redefining Memory Functions.....	6-16

Chapter 7	Language-specific Usage Options	
	Using Language-Specific Interfaces with Intel® MKL	7-1
	Mixed-language Programming with Intel® MKL	7-4
	Calling LAPACK, BLAS, and CBLAS Routines from C Language Environments	7-5
	Using Complex Types in C/C++	7-7
	Calling BLAS Functions that Return the Complex Values in C/C++ Code	7-8
	Support for Boost uBLAS Matrix-matrix Multiplication	7-10
	Invoking Intel® MKL Functions from Java* Applications	7-12
Chapter 8	Coding Tips	
	Aligning Data for Numerical Stability	8-1
Chapter 9	Intel® Optimized LINPACK Benchmark	
	Contents	9-1
	Running the Software	9-2
	Known Limitations	9-3
Appendix A Intel® Math Kernel Library Language Interfaces Support		
Appendix B Support for Third-Party Interfaces		
	GMP* Functions	B-1
	FFTW Interface Support	B-1
Index		
List of Tables		
	Table 1-1 Notational Conventions	1-3
	Table 2-1 What You Need to Know Before You Begin.....	2-1
	Table 3-1 High-level Directory Structure	3-1
	Table 3-2 Compiling for the ILP64 and LP64 Interfaces	3-5
	Table 3-3 Integer Types.....	3-6
	Table 3-4 Architecture-specific Implementations	3-7
	Table 3-5 Detailed Structure of the IA-32 Architecture-specific	

Directory lib/32	3-8
Table 3-6 Detailed Structure of the Intel® 64 Architecture-specific Directory lib/em64t	3-10
Table 3-7 Detailed Structure of the Universal Directory for the IA-32 and Intel® 64 Architectures	3-12
Table 3-8 Contents of the doc Directory	3-14
Table 5-1 Typical Link Libraries	5-1
Table 5-2 Selecting Threading Libraries	5-4
Table 5-3 Computational Libraries to Link, by Function Domain	5-5
Table 6-1 How to Avoid Conflicts in the Execution Environment for Your Threading Model	6-4
Table 6-2 Environment Variables for Threading Controls	6-9
Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS Values ..	6-11
Table 7-1 Interface Libraries and Modules	7-1
Table 9-1 Contents of the LINPACK Benchmark	9-2

List of Examples

Example 6-1 Changing the Number of Threads	6-5
Example 6-2 Setting the Number of Threads to One	6-9
Example 6-3 Redefining Memory Functions	6-16
Example 7-1 Calling a Complex BLAS Level 1 Function from C++	7-9
Example 7-2 Using CBLAS Interface Instead of Calling BLAS Directly from C	7-10
Example 8-1 Aligning Addresses at 16-byte Boundaries	8-2

List of Figures

Figure 4-1 Opening the active project target in the Apple Xcode* IDE	4-2
Figure 4-2 Specifying build settings in the Apple Xcode* IDE	4-3
Figure 7-1 Column-major Order versus Row-major Order	7-6

Overview

1

The Intel® Math Kernel Library (Intel® MKL) offers highly optimized, thread-safe math routines for science, engineering, and financial applications that require maximum performance.

Technical Support

Intel provides a support web site, which 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/>.

About This Document

The *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:

- Focuses on the usage information needed to call Intel MKL routines from user's applications running on Mac OS* X.
- Describes OS-specific usage of Intel MKL, along with OS-independent features.
- Contains usage information for all Intel MKL function domains, listed in [Table A-1](#) (in Appendix A).
- Assumes you have completed the installation of Intel MKL on your system. If you have not completed the installation, see the *Intel® Math Kernel Library Installation Guide* (file `Install.txt`).

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

- *The Intel MKL 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*.

This User's Guide is intended to assist you in mastering the usage of Intel MKL on Mac OS X. In particular, it:

- 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 to the library and provides simple usage scenarios
- Describes how to code, compile, and run your application with Intel MKL for Mac OS X.

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

Document Organization

The document contains the following chapters and appendices:

- | | |
|-----------|--|
| Chapter 1 | Overview . Introduces the Intel MKL usage information and describes this document's notational conventions. |
| Chapter 2 | Getting Started . Describes post-installation steps and gives information needed to start using Intel MKL after its installation. |
| Chapter 3 | Intel® Math Kernel Library Structure . Discusses the structure of the Intel MKL directory after installation. |
| Chapter 4 | Configuring Your Development Environment . Explains how to configure Intel MKL with your development environment. |
| Chapter 5 | Linking Your Application with the Intel® Math Kernel Library . Explains which libraries should be linked with your application for your particular platform; discusses how to build custom dynamic libraries. |
| Chapter 6 | Managing Performance and Memory . Discusses Intel MKL threading; shows coding techniques and gives hardware configuration tips for improving performance of the library; explains features of the Intel MKL memory management and, in particular, shows how to replace memory functions that the library uses by default with your own ones. |
| Chapter 7 | Language-specific Usage Options . Discusses mixed-language programming and the use of language-specific interfaces. |

Chapter 8	Coding Tips . Presents coding tips that may be helpful to your specific needs.
Chapter 9	Intel® Optimized LINPACK Benchmark . Describes the Intel® Optimized LINPACK Benchmark for Mac OS* X.
Appendix A	Intel® Math Kernel Library Language Interfaces Support . Summarizes information on language interfaces that Intel MKL provides for each function domain, including the respective header files.
Appendix B	Support for Third-Party Interfaces . Describes some interfaces that Intel MKL supports.

The document also includes an [Index](#).

Term and Notational Conventions

The following term is used to refer to the operating system:

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

The following notation is used in reference to Intel MKL directories:

<code><mkl_directory></code>	The main directory where Intel MKL is installed. Replace this placeholder with the specific pathname in the configuring, linking, and building instructions. For more information, see Getting Started .
<code><Intel Compiler Pro directory></code>	The installation directory for the Intel® C++ Compiler Professional Edition or Intel® Fortran Compiler Professional Edition. For more information, see Getting Started .

[Table 1-1](#) lists the other notational conventions:

Table 1-1 Notational Conventions

<i>Italic</i>	Italic is used for emphasis and also indicates document names in body text, for example: see <i>Intel MKL Reference Manual</i>
---------------	---

Table 1-1 Notational Conventions (continued)

<p>Monospace lowercase mixed with uppercase</p>	<p>Indicates commands and command-line options, for example: <code>icc myprog.c -L\$MKLPATH -I\$MKLINCLUDE -lmkl -lguid -lpthread ;</code> filenames, directory names and pathnames, for example: <code>/Library/Frameworks/Intel_MKL.framework/Versions/ 10.2.0.004</code> C/C++ code fragments, for example: <code>a = new double [SIZE*SIZE];</code></p>
<p>UPPERCASE MONOSPACE</p>	<p>Indicates system variables, for example, \$MKLPATH</p>
<p><i>Monospace italic</i></p>	<p>Indicates a parameter in discussions: routine parameters, for example, <i>lda</i>; makefile parameters, for example, <i>functions_list</i>; etc. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, <i><mkl directory></i>. Substitute one of these items for the placeholder.</p>
<p>[items]</p>	<p>Square brackets indicate that the items enclosed in brackets are optional.</p>
<p>{ item item }</p>	<p>Braces indicate that only one of the items listed between braces should be selected. A vertical bar () separates the items</p>

Getting Started

2

This chapter provides some basic usage information and describes post-installation steps to help you start using the Intel® Math Kernel Library (Intel® MKL) on Mac OS* X.

Before You Begin

Before you begin using Intel MKL, learning a few important concepts will help you get off to a good start, as shown in [Table 2-1](#).

Table 2-1 **What You Need to Know Before You Begin**

Target platform	<p>Action: Identify the architecture of your target machine:</p> <ul style="list-style-type: none">• IA-32 or compatible• Intel® 64 or compatible <p>Reason: Because Intel MKL libraries are located in directories corresponding to your particular architecture (see Architecture Support), you should provide proper paths on your link lines (see 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 Set Environment Variables for details).</p>
-----------------	---

Table 2-1 What You Need to Know Before You Begin (continued)

<p>Mathematical problem</p>	<p>Action: Identify all Intel MKL function domains that you require:</p> <ul style="list-style-type: none"> • BLAS • Sparse BLAS • LAPACK • Sparse Solver routines • Vector Mathematical Library functions • Vector Statistical Library functions • Fourier Transform functions (FFT) • Trigonometric Transform routines • Poisson, Laplace, and Helmholtz Solver routines • Optimization (Trust-Region) Solver routines • GMP* arithmetic functions <p>Reason: The function domain you intend to use narrows the search in the <i>Reference Manual</i> for specific routines you need. Coding tips may also depend on the function domain (see Tips and Techniques to Improve Performance).</p>
<p>Programming language</p>	<p>Action: Though Intel MKL provides support for both Fortran and C/C++ programming, not all the function domains support a particular language environment, for example, C/C++ or Fortran 90/95. Identify the language interfaces that your function domains support (see Intel® Math Kernel Library Language Interfaces Support).</p> <p>Reason: In case your function domain does not directly support the needed environment, you can use mixed-language programming (see Mixed-language Programming with Intel® MKL).</p> <p>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® MKL.</p>
<p>Range of integer data</p>	<p>Action: If your system is based on the Intel 64 or IA-64 architecture, identify whether your application performs calculations with huge data arrays (of more than $2^{31}-1$ elements).</p> <p>Reason: To operate on huge 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 Support for ILP64 Programming).</p>
<p>Threading model</p>	<p>Action: Identify whether and how your application is threaded:</p> <ul style="list-style-type: none"> • Threaded with the Intel® compiler • Threaded with a third-party compiler • Not threaded <p>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).</p> <p>Action: Determine the number of threads you want Intel MKL to use.</p> <p>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 the Intel® MKL Parallelism.</p>

Table 2-1 What You Need to Know Before You Begin (continued)

Linking model	<p>Action: Decide which linking model is appropriate for linking your application with Intel MKL libraries:</p> <ul style="list-style-type: none"> • Static • Dynamic <p>Reason: For information on the benefits of each linking model, link command syntax and examples, link libraries, and other linking topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with the Intel® Math Kernel Library.</p>
---------------	---

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 (for the list of include files, see [Table A-2](#)). Calling Intel MKL functions from your application without an appropriate include file may lead to incorrect behavior of the functions.

Steps to Get Started

This section helps you to get started with Intel MKL on Mac OS* X.

Check Your Installation

After installing Intel MKL, verify that the library has been properly installed and configured.

1. Check that the directory you chose for the installation has been created. The Intel MKL *default* installation directory may be one of the following:
 - `/Library/Frameworks/Intel_MKL.framework/Versions/RR.r.y.xxx`, where RR.r is the version number, y is the release-update number, and xxx is the package number, for example, `/Library/Frameworks/Intel_MKL.framework/Versions/10.2.0.004`
 - `<Intel Compiler Pro directory>/Frameworks/MKL`, for example, `/Developer/Intel/Compiler/11.1/015/Frameworks/MKL`, `/Xcode2.5/Intel/Compiler/11.1/015/Frameworks/MKL`, or `/opt/Intel/Compiler/11.1/015/Frameworks/MKL`.
2. If you choose to keep multiple versions of Intel MKL installed on your system, update your build scripts so that they point to the *desired* version.
3. Check that the following four files are placed in the `tools/environment` directory:

```
mklvars32.sh
mklvars32.csh
mklvarsem64t.sh
mklvarsem64t.csh
```

You can use these files to set environment variables in the current user shell.

4. To check the high-level and detailed structure of the Intel MKL installation directory, see Chapter 3.

Set Environment Variables

When the installation of Intel MKL for Mac OS* X is complete, you can use two scripts `mklvars32` and `mklvarsem64t` with two flavors each (`.sh` and `.csh`) in the `tools/environment` directory to set the environment variables `INCLUDE`, `MKLROOT`, `DYLD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `FPATH` in the user shell. For further configuring the library, see Chapter 4.

Use a Web-based Linking Advisor

Intel MKL provides a web-based linking advisor to help you choose the libraries and options to specify on a link line for your application.

The tool is available at <http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor>.

The advisor types to a screen the link line depending on your Intel MKL usage scenario, specifically: the operating system, architecture, compiler, static or dynamic linking model, length of integers (32-bit or 64-bit), threaded or sequential mode of Intel MKL operation, and so on.

Copy-paste the output to your link line.

For more information on linking with Intel MKL, see Chapter 5 and specifically [Table 5-1](#) for a list of typical link libraries.

Use Intel MKL Code Examples

Intel MKL package includes code examples, located in the `examples` subdirectory of the Intel MKL installation directory. The examples provide the most direct way for you to find out:

- 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 instance, subdirectory `examples/spblas` contains Sparse BLAS examples, and subdirectory `examples/vmlc` contains VML examples in C. Source code for the examples is in the next level `sources` subdirectory.

To compile, build, and run the examples, use the makefile provided. For information on how to use it, refer to the makefile header.

See also:

[High-level Directory Structure.](#)

Intel® Math Kernel Library Structure

3

The chapter discusses the structure of the Intel® Math Kernel Library (Intel® MKL), including the Intel MKL directory structure, architecture-specific implementations, supported programming interfaces, and more.

Starting with version 10.0, Intel MKL employs a layered model to streamline the library structure, reduce its size, and add usage flexibility.

See also: [Layered Model Concept](#).

High-level Directory Structure

[Table 3-1](#) shows a high-level directory structure of Intel MKL after installation.

Table 3-1 High-level Directory Structure

Directory	Comment
<code><mk1 directory></code>	Intel MKL main directory
<code><mk1 directory>/benchmarks/linpack</code>	Contains a Shared-Memory (SMP) version of LINPACK benchmark
<code><mk1 directory>/examples</code>	A source and data for examples
<code><mk1 directory>/include</code>	Contains INCLUDE files for the library routines, as well as for tests and examples
<code><mk1 directory>/include/32</code>	Contains BLAS95 ¹ and LAPACK95 ² .mod files for the IA-32 architecture and Intel® Fortran compiler
<code><mk1 directory>/include/em64t/ilp64</code>	Contains BLAS95 and LAPACK95 .mod files for the Intel® 64 architecture (formerly, Intel® EM64T), Intel® Fortran compiler, and ILP64 interface
<code><mk1 directory>/include/em64t/lp64</code>	Contains BLAS95 and LAPACK95 .mod files for the Intel® 64 architecture, Intel® Fortran compiler, and LP64 interface

Table 3-1 High-level Directory Structure (continued)

Directory	Comment
<code><mkl directory>/interfaces/blas95</code>	Contains Fortran 95 wrappers for BLAS and a makefile to build the library
<code><mkl directory>/interfaces/fftw2xc</code>	Contains wrappers for FFTW version 2.x (C interface) to call Intel MKL FFTs
<code><mkl directory>/interfaces/fftw2xf</code>	Contains wrappers for FFTW version 2.x (Fortran interface) to call Intel MKL FFTs
<code><mkl directory>/interfaces/fftw3xc</code>	Contains wrappers for FFTW version 3.x (C interface) to call Intel MKL FFTs
<code><mkl directory>/interfaces/fftw3xf</code>	Contains wrappers for FFTW version 3.x (Fortran interface) to call Intel MKL FFTs
<code><mkl directory>/interfaces/lapack95</code>	Contains Fortran 95 wrappers for LAPACK and a makefile to build the library
<code><mkl directory>/lib</code>	Contains static libraries and shared objects.
<code><mkl directory>/lib/32</code>	Contains static libraries and shared objects for the IA-32 architecture
<code><mkl directory>/lib/em64t</code>	Contains static libraries and shared objects for the Intel® 64 architecture
<code><mkl directory>/lib/universal</code>	Contains universal static libraries and shared objects for the IA-32 and Intel® 64 architectures.
<code><mkl directory>/tests</code>	Contains source and data files for tests
<code><mkl directory>/tools/builder</code>	Contains tools for creating custom dynamically linkable libraries
<code><mkl directory>/tools/environment</code>	Contains shell scripts to set environmental variables in the user shell
<code><mkl directory>/tools/plugins/com.intel.mkl.help</code>	Contains an Eclipse plug-in with Intel MKL Reference Manual in WebHelp format. See <code>mkl_documentation.htm</code> for comments.
<code><Intel Compiler Pro directory>/documentation/en_US/mkl</code>	Contains the Intel MKL documentation.
<code><Intel Compiler Pro directory>/man/en_US/man3</code>	Contains man pages for the Intel MKL functions.

1. Fortran 95 interface to BLAS
2. Fortran 95 interface to LAPACK

Layered Model Concept

Starting with release 10.0, Intel MKL employs a layered model.

There are four essential parts of the library:

1. Interface layer
2. Threading layer
3. Computational layer
4. Compiler Support Run-time libraries.

The first part adapts Intel MKL to interface-related issues, for example, whether to use 32-bit or 64-bit integer types, or how different compilers return function values. The second part adapts the library to the OpenMP* implementations used by different threading compilers or to the non-threaded usage mode. The Computational layer is the bulk of the library, which is isolated in a separate part to save space. Rather than generate one library for each of the cases within those parts and thus create a geometric explosion of possibilities, Intel MKL has broken down each part into independent libraries corresponding to those cases. You can combine these libraries to meet your needs. Once the interface library is selected, the threading library you select picks up that interface, and the computational libraries use interfaces and threading chosen in the first two layers.

Interface Layer. This layer provides matching between compiled code of your application and the threading and/or computational parts of the library. This layer provides:

- The LP64 and ILP64 interfaces (see [Support for ILP64 Programming](#) for details)
- 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)



NOTE. 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.
- An ability for the user 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. For any given processor architecture (IA-32, IA-64, or Intel® 64) and OS, this layer has only one computational library to link with, regardless of the Interface and Threading layer. The Computational layer accommodates multiple architectures through identification of architecture features and chooses the appropriate binary code at run time.

Compiler Support Run-time Libraries (RTL, for brevity). This layer provides RTL support. Not all RTLs are delivered with Intel MKL. The only RTLs provided are Intel® compiler RTLs: Intel® Compatibility OpenMP* run-time library (`libiomp`) and Intel® Legacy OpenMP* run-time library (`libguide`). To thread using third-party threading compilers, use libraries in the Threading layer or an appropriate compatibility library (for more information, see [Linking with Threading Libraries](#)).

Sequential Mode of the Library

You can use Intel MKL in a sequential (non-threaded) mode. It requires no Compatibility OpenMP* or Legacy OpenMP* run-time library, and does not respond to the environment variable `OMP_NUM_THREADS` or its Intel MKL equivalents. In this mode, Intel MKL runs unthreaded code. However, it is thread-safe¹, which means that you can use it in a parallel region from your own OpenMP* code. 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 may, for various reasons, need a non-threaded version of the library (for instance, in some MPI cases). To obtain Intel MKL running in the sequential mode, in the Threading layer, choose the `*sequential.*` library to link.

Note that the `*sequential.*` library depends on the POSIX threads library (`pthread`), which is used to make the Intel MKL software thread-safe and should be listed on the link line.

See also:

[Directory Structure in Detail](#)

[Using the Intel® MKL Parallelism](#)

[Avoiding Conflicts in the Execution Environment](#)

[Linking Examples](#).

1. Except for LAPACK deprecated routines `?lacon`, `?lasq3`, and `?lasq4`.

Support for ILP64 Programming

The Intel MKL ILP64 libraries use the 64-bit integer type (necessary for indexing huge 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 supported in the Interface layer (see [Layered Model Concept](#) and [Directory Structure in Detail](#) for more information).

The ILP64 interface is provided for the following two reasons:

- To support huge data arrays (with more than $2^{31}-1$ elements)
- To enable compiling your Fortran code with the `-i8` compiler option

It is up to you to choose which interface to use. Choose the LP64 interface for 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 huge 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

[Table 3-2](#) shows how to compile for the ILP64 and LP64 interfaces:

Table 3-2 Compiling for the ILP64 and LP64 Interfaces

Fortran	
Compiling for ILP64	<code>ifort -i8 -I<mk1 drectory>/include ...</code>
Compiling for LP64	<code>ifort -I<mk1 drectory>/include ...</code>
C or C++	
Compiling for ILP64	<code>icc -DMKL_ILP64 -I<mk1 directory>/include ...</code>
Compiling for LP64	<code>icc -I<mk1 directory>/include ...</code>



NOTE. Linking of the 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 (see [Table 3-3](#)):

Table 3-3 Integer Types

	Fortran	C or C++
32-bit integers	INTEGER*4 or INTEGER(KIND=4)	int
Universal integers for ILP64/LP64: <ul style="list-style-type: none"> • 64-bit for ILP64 • 32-bit otherwise 	INTEGER without specifying KIND	MKL_INT
Universal integers for ILP64/LP64: <ul style="list-style-type: none"> • 64-bit integers 	INTEGER*8 or INTEGER(KIND=8)	MKL_INT64
FFT interface integers for ILP64/LP64	INTEGER without specifying KIND	MKL_LONG

Browsing the Intel MKL Include Files

Given a function with integer parameters, the *Reference Manual* does not explain which parameters become 64-bit and which remain 32-bit for ILP64, and you are encouraged to browse the include files, examples, and tests for the ILP64 interface details. For their location, see [Table 3-1](#).

You are encouraged to start with browsing the include files, listed in [Table A-2](#).

Some function domains that support only a Fortran interface (see [Table A-1](#)), provide header files for C/C++ in the include directory. Such *.h files enable using a Fortran binary interface from C/C++ code. These files can also be used to understand the ILP64 usage.

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`.
- GMP* arithmetic functions do not support ILP64.

Architecture Support

Intel MKL for Mac OS* X provides two architecture-specific implementations and universal libraries for both architectures. [Table 3-4](#) lists the supported architectures and directories where each architecture-specific implementation is located.

Table 3-4 Architecture-specific Implementations

Architecture	Location
IA-32 or compatible	<mk1 directory>/lib/32
Intel® 64 or compatible	<mk1 directory>/lib/em64t
IA-32, Intel® 64, or compatible	<mk1 directory>/lib/universal

See a detailed structure of these directories in [Table 3-5](#), [Table 3-6](#), and [Table 3-7](#).



NOTE. The 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.

Directory Structure in Detail

The information in the tables below shows a detailed structure of the Intel MKL architecture-specific directories and the `universal` directory. For the list of additional interface libraries that can be generated in these directories using makefiles in the `interfaces` directory, see [Using Language-Specific Interfaces with Intel® MKL](#). For the contents of the `doc` directory, see [Contents of the Documentation Directory](#). For the contents of the `benchmarks/linpack` directory, see [Intel® Optimized LINPACK Benchmark](#).

Table 3-5 Detailed Structure of the IA-32 Architecture-specific Directory lib/32

File	Contents
Static Libraries	
<i>Interface layer</i>	
libmkl_blas95.a	Fortran 95 interface library for BLAS for the Intel® Fortran compiler
libmkl_intel.a	Interface library for the Intel® compilers
libmkl_lapack95.a	Fortran 95 interface library for LAPACK for the Intel® Fortran compiler.
<i>Threading layer</i>	
libmkl_intel_thread.a	Threading library for the Intel compilers
libmkl_pgi_thread.a	Threading library for the PGI* compilers
libmkl_sequential.a	Sequential library
<i>Computational layer</i>	
libmkl_core.a	Kernel library for IA-32 architecture
libmkl_solver.a	Iterative Sparse Solver, Trust Region Solver, and GMP routines
libmkl_solver_sequential.a	Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routines.
RTL	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking

Table 3-5 Detailed Structure of the IA-32 Architecture-specific Directory lib/32
(continued)

File	Contents
Dynamic Libraries	
Interface layer	
libmkl_intel.dylib	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	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_lapack.dylib	LAPACK and DSS/PARDISO routines and drivers
libmkl_p4m.dylib	Kernel for the Intel® Core™ microarchitecture.
libmkl_p4m3.dylib	Kernel library for the Intel® Core™ i7 processors
libmkl_p4p.dylib	Kernel for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
libmkl_vml_p4m.dylib	VML/VSL for the Intel® Core™ microarchitecture.
libmkl_vml_p4m2.dylib	VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
libmkl_vml_p4m3.dylib	VML/VSL for the Intel® Core™ i7 processors
libmkl_vml_p4p.dylib	VML/VSL for Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
RTL	
libguide.dylib	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.dylib	Intel® Compatibility OpenMP* run-time library for dynamic linking
locale/en_US/mkl_msg.cat	Catalog of Intel MKL messages in English

Table 3-6 Detailed Structure of the Intel® 64 Architecture-specific Directory lib/em64t

File	Contents
Static Libraries	
<i>Interface layer</i>	
libmkl_blas95_ilp64.a	Fortran 95 interface library for BLAS for the Intel® Fortran compiler. Supports the ILP64 interface
libmkl_blas95_lp64.a	Fortran 95 interface library for BLAS for the Intel® Fortran compiler. Supports the LP64 interface
libmkl_intel_ilp64.a	ILP64 interface library for the Intel compilers
libmkl_intel_lp64.a	LP64 interface library for the Intel compilers
libmkl_lapack95_ilp64.a	Fortran 95 interface library for LAPACK for the Intel® Fortran compiler. Supports the ILP64 interface
libmkl_lapack95_lp64.a	Fortran 95 interface library for LAPACK for the Intel® Fortran compiler. Supports the LP64 interface
<i>Threading layer</i>	
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
<i>Computational layer</i>	
libmkl_core.a	Kernel library for the Intel® 64 architecture
libmkl_solver_ilp64.a	Iterative Sparse Solver and Trust Region Solver routine library supporting the ILP64 interface
libmkl_solver_ilp64_sequential.a	Sequential version of Iterative Sparse Solver and Trust Region Solver routine library supporting the ILP64 interface
libmkl_solver_lp64.a	Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting the LP64 interface
libmkl_solver_lp64_sequential.a	Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting the LP64 interface
RTL	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking

Table 3-6 Detailed Structure of the Intel® 64 Architecture-specific Directory lib/em64t (continued)

File	Contents
Dynamic Libraries	
<i>Interface layer</i>	
libmkl_intel_ilp64.dylib	ILP64 interface library for the Intel compilers
libmkl_intel_lp64.dylib	LP64 interface library for the Intel compilers
<i>Threading layer</i>	
libmkl_intel_thread.dylib	Threading library for the Intel compilers
libmkl_sequential.dylib	Sequential library
<i>Computational layer</i>	
libmkl_core.dylib	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_lapack.dylib	LAPACK and DSS/PARDISO routines and drivers
libmkl_mc.dylib	Kernel for processors based on the Intel® Core™ microarchitecture
libmkl_mc3.dylib	Kernel for the Intel® Core™ i7 processors
libmkl_vml_mc.dylib	VML/VSL for processors based on the Intel® Core™ microarchitecture
libmkl_vml_mc2.dylib	VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
libmkl_vml_mc3.dylib	VML/VSL for the Intel® Core™ i7 processors
RTL	
libguide.dylib	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.dylib	Intel® Compatibility OpenMP* run-time library for dynamic linking
locale/en_US/mkl_msg.cat	Catalog of Intel MKL messages in English

Table 3-7 Detailed Structure of the Universal Directory for the IA-32 and Intel® 64 Architectures

File	Contents
Static Libraries	
<i>Interface layer</i>	
libmkl_intel_ilp64.a	ILP64 interface library for the Intel compilers
libmkl_intel_lp64.a	LP64 interface library for the Intel compilers
<i>Threading layer</i>	
libmkl_intel_thread.a	Universal threading library for the Intel compilers
libmkl_pgi_thread.a	Universal threading library for the PGI* compiler
libmkl_sequential.a	Universal sequential library
<i>Computational layer</i>	
libmkl_core.a	Universal kernel library
libmkl_solver_ilp64.a	Iterative Sparse Solver and Trust Region Solver routine library supporting the ILP64 interface
libmkl_solver_ilp64_sequential.a	Sequential version of Iterative Sparse Solver and Trust Region Solver routine library supporting the ILP64 interface
libmkl_solver_lp64.a	Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting the LP64 interface
libmkl_solver_lp64_sequential.a	Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting the LP64 interface
RTL	
libguide.a	Intel® Legacy OpenMP* run-time library for static linking
libiomp5.a	Intel® Compatibility OpenMP* run-time library for static linking

Table 3-7 Detailed Structure of the Universal Directory for the IA-32 and Intel® 64 Architectures (continued)

File	Contents
Dynamic Libraries	
<i>Interface layer</i>	
libmkl_intel_ilp64.dylib	ILP64 interface library for the Intel compilers
libmkl_intel_lp64.dylib	LP64 interface library for the Intel compilers
<i>Threading layer</i>	
libmkl_intel_thread.dylib	Universal threading library for the Intel compilers
libmkl_sequential.dylib	Universal sequential library
<i>Computational layer</i>	
libmkl_core.dylib	Universal library. Contains the dispatcher for dynamic load of the processor-specific kernel library
libmkl_lapack.dylib	Universal binary with 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_p4m.dylib	32-bit kernel for the Intel® Core™ microarchitecture
libmkl_p4m3.dylib	32-bit kernel library for the Intel® Core™ i7 processors
libmkl_p4p.dylib	32-bit kernel for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
libmkl_vml_mc.dylib	64-bit VML for processors based on the Intel® Core™ microarchitecture
libmkl_vml_mc2.dylib	64-bit VML/VSL for 45nm Hi-k Intel® Core™2 and the Intel Xeon® processor families
libmkl_vml_mc3.dylib	64-bit VML/VSL for the Intel® Core™ i7 processors
libmkl_vml_p4m.dylib	32-bit VML for processors based on Intel® Core™ microarchitecture
libmkl_vml_p4m2.dylib	32-bit VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
libmkl_vml_p4m3.dylib	32-bit VML/VSL for the Intel® Core™ i7 processors
libmkl_vml_p4p.dylib	32-bit VML for the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)

Table 3-7 Detailed Structure of the Universal Directory for the IA-32 and Intel® 64 Architectures (continued)

File	Contents
<i>RTL</i>	
libguide.dylib	Intel® Legacy OpenMP* run-time library for dynamic linking
libiomp5.dylib	Intel® Compatibility OpenMP* run-time library for dynamic linking

Accessing the Intel® MKL Documentation

This section details the contents of the Intel MKL documentation directory and explains how to access man pages for the library.

Contents of the Documentation Directory

[Table 3-8](#) shows the contents of the `doc` subdirectory in the Intel MKL installation directory:

Table 3-8 Contents of the doc Directory

File name	Comment
Install.txt	Intel MKL Installation Guide
mkl_documentation.htm	Overview and links for the Intel MKL documentation
mklEULA.txt	Intel MKL end user license
mklman.pdf	Intel MKL Reference Manual
mklman90_j.pdf	Intel MKL Reference Manual in Japanese
mklsupport.txt	Information on package number for customer support reference
redist.txt	List of redistributable files
Release_Notes.pdf	Intel MKL Release Notes
userguide.pdf	Intel MKL User's Guide, this document.

Accessing Man Pages

During installation, the man pages for the Intel MKL functions are copied to subdirectory `man/en_US/man3` of the Intel MKL installation directory.

To make the man pages accessible through the `man` command in your command shell, add the directory with the man pages to the `MANPATH` environment variable (see [Set Environment Variables](#)).

Once the environment variable is set, to view the man page for an Intel MKL function, enter the following command in your command shell:

```
man <function base name>
```

In this release, *<function base name>* is the function name with omitted prefixes denoting data type, precision, or function domain.

Examples:

- For the BLAS function `ddot`, enter `man dot`
- For the FFT function `DftiCommitDescriptor`, enter `man CommitDescriptor`



NOTE. Function names in the `man` command are case-sensitive.

Configuring Your Development Environment

4

This chapter explains how to configure your development environment for the use with the Intel® Math Kernel Library (Intel® MKL).

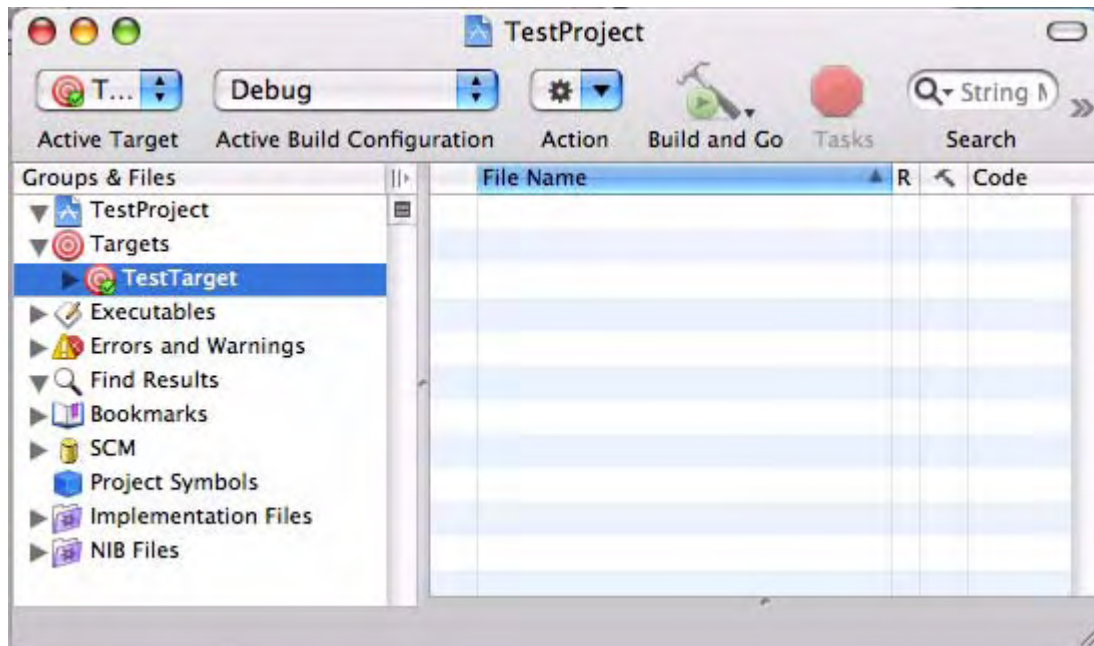
Chapter 2 explains how to set environment variables `INCLUDE`, `MKLROOT`, `DYLD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `FPATH`.

Configuring the Apple Xcode* Developer Software to Link with Intel® MKL

This section provides information on linking of Intel MKL with the Apple Xcode* developer software. Please note that the screenshots are from Apple Xcode* 2.4 and may differ from other versions. The fundamental steps to configuring Xcode* for use with Intel MKL are more widely applicable:

1. Open your project that uses Intel MKL.
2. Under **Targets**, double-click the active target (see [Figure 4-1](#)).

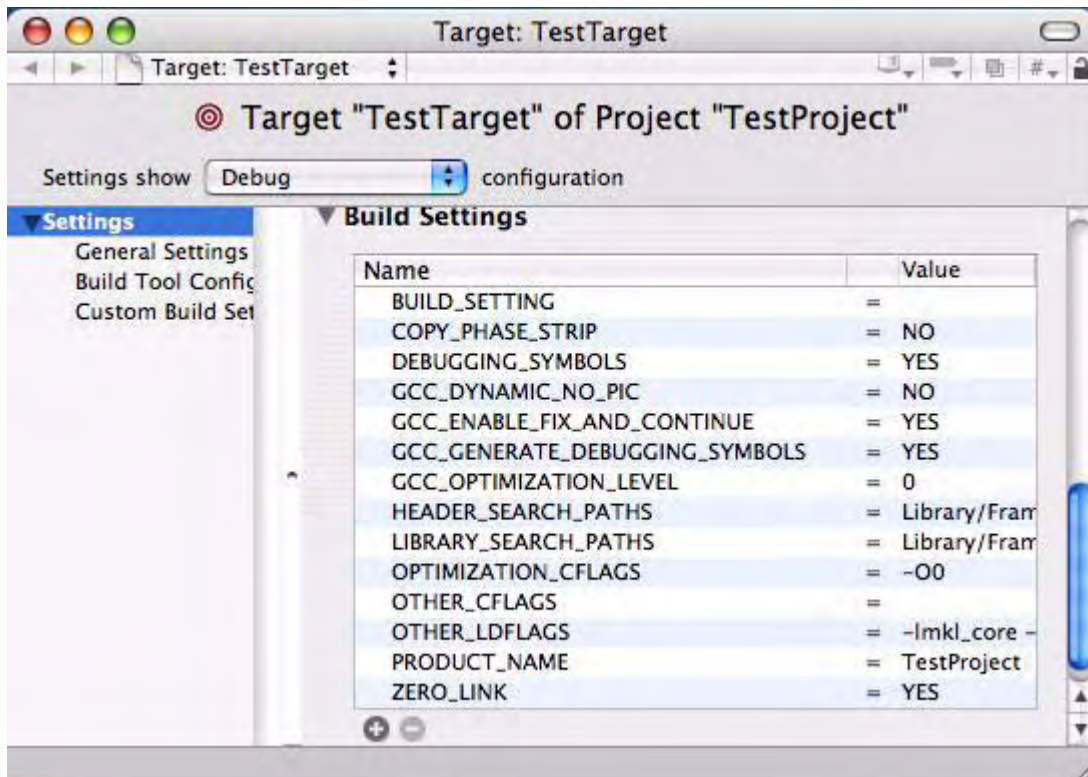
Figure 4-1 Opening the active project target in the Apple Xcode* IDE



In the **Target** dialog box, assign values to the build settings as follows (see [Figure 4-2](#)).

3. Click the plus icon under the **Build Settings** table, located at the bottom of the dialog box, to add a row. In the new row, type **HEADER_SEARCH_PATHS** under **Name** and the path to the Intel® MKL include files, that is, `<mk1 directory>/include`, under **Value**.

Figure 4-2 Specifying build settings in the Apple Xcode* IDE



4. Click the plus icon under the **Build Settings** table to add another row, in which type **LIBRARY_SEARCH_PATHS** under **Name** and the path to the Intel MKL libraries under **Value** (for example, `<mkl directory>/lib/32`).
5. Double-click the setting **OTHER_LDFLAGS** and under **Value**, type linker options for additional libraries (for example, `-lmkl_core -lguid -lpthread`). To learn how to choose the libraries, see [Selecting Libraries to Link](#).

Configuring the Out-of-Core (OOC) DSS/PARDISO* Solver

When using the configuration file for the OOC DSS/PARDISO* Solver, be aware that the maximum length of the path lines in the file is 1000 characters.

For more information, see the "Sparse Solver Routines" chapter in the *Intel MKL Reference Manual*.

Linking Your Application with the Intel® Math Kernel Library

5

This chapter discusses linking your applications with the Intel® Math Kernel Library (Intel® MKL) for Mac OS* X. The chapter provides information on the libraries that should be linked with your application, presents linking examples, and explains building of custom dynamically linked shared libraries.

To link with Intel MKL, which employs the layered linking model, choose one library from the Interface layer, one library from the Threading layer, the library from the Computational layer, and, if necessary, add run-time libraries. [Table 5-1](#) lists typical sets of libraries that suffice to link with Intel MKL.

Table 5-1 Typical Link Libraries

	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_lp64.a	libmkl_core.a	libiomp5.dylib
Intel® 64 architecture, dynamic linking	libmkl_intel_lp64.dylib	libmkl_intel_thread_lp64.dylib	libmkl_core.dylib	libiomp5.dylib

For exceptions and alternatives to the libraries listed above, see [Selecting Libraries to Link](#).

See also: [Listing Libraries on a Link Line](#).

Listing Libraries on a Link Line

To link with Intel MKL libraries, 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 `$MKL_PATH/libmkl_core.a`, where `$MKL_PATH` is the appropriate user-defined environment variable. See specific examples in the [Linking Examples](#) section.

```
<files to link>
-L<MKL path> -I<MKL include>
[-I<MKL include>/{32|em64t|{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_solver_{lp64|ilp64}]
[-lmkl_lapack] -lmkl_core
{-liomp5|-lguide} [-lpthread] [-lm]
```

See [Selecting Libraries to Link](#) for details of this syntax usage and specific recommendations on which libraries to link depending on your Intel MKL usage scenario.

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. See specific examples in the [Linking Examples](#) section.

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

Selecting Libraries to Link

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

[Linking with Fortran 95 Interface Libraries](#)

[Linking with Threading Libraries](#)

[Linking with Computational Libraries](#)

[Linking with Compiler Support RTLs](#)

[Linking with System Libraries](#)

[Linking Examples](#)

See also:

[Note on linking with Universal libraries.](#)

Linking with Fortran 95 Interface Libraries

The `libmkl_blas95.a`, `libmkl_blas95_lp64.a`, `libmkl_blas95_ilp64.a`, `libmkl_lapack95.a`, `libmkl_lapack95_lp64.a`, and `libmkl_lapack95_ilp64.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 [Fortran 95 Interfaces to LAPACK and BLAS](#) and [Compiler-dependent Functions and Fortran 90 Modules](#) for details on building the libraries and on why *source code* is distributed in this case.)

Linking with Threading Libraries

Several compilers that Intel MKL supports use the OpenMP* threading technology. Starting with version 10.0, Intel MKL supports the implementations of OpenMP* that those compilers provide. If an application using OpenMP* and compiled with such a compiler calls Intel MKL 10.0 or lower, which is threaded with Intel® compilers, performance issues and even failures may arise because threading libraries from different compilers are not compatible. Starting with Intel MKL 10.0, solutions for these issues are provided from the Threading Layer and the Compiler Support Run-time Libraries (RTL).

Threading Layer. Because of the internal structure of the library, threading represents a small amount of code. This code is compiled by different compilers (Intel, gnu and PGI* compilers on Mac OS* X), and you should link in the appropriate threading library.

RTL. This layer includes run-time libraries of the Intel compiler: the Intel® Compatibility OpenMP* run-time library `libiomp` and Intel® Legacy OpenMP* run-time library `libguide`. The Compatibility library `libiomp` is an extension of `libguide` that provides support for one additional 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` and execute efficiently and effectively. So, you are encouraged to use `libiomp` rather than `libguide`.

[Table 5-2](#) helps explain what threading library you should choose under different scenarios when using Intel MKL (static cases only):

Table 5-2 Selecting Threading Libraries

Compiler	Application Threaded?	Threading Layer	RTL Recommended	
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 Computational Libraries

Typically, with the layered linking model, you must link your application with only one computational library. However, certain Intel MKL function domains require several computational link libraries.

For each Intel MKL function domain, [Table 5-3](#) lists computational libraries that you must include in the link line.

Table 5-3 Computational Libraries to Link, by Function Domain

Function domain	IA-32 Architecture		Intel® 64 Architecture	
	Static	Dynamic	Static	Dynamic
BLAS, CBLAS, Sparse BLAS, LAPACK, VML, VSL, FFT, Trigonometric Transform Functions, Poisson Library	libmkl_core.a	libmkl_core.dylib	libmkl_core.a	libmkl_core.dylib
Iterative Sparse Solvers, Trust Region Solver, and GMP routines	libmkl_solver.a or libmkl_solver_sequential.a	n/a	See below	n/a
Iterative Sparse Solvers, Trust Region Solver, and GMP routines, LP64 interface	n/a	n/a	libmkl_solver_lp64.a or libmkl_solver_lp64_sequential.a libmkl_core.a	n/a
Iterative Sparse Solvers, Trust Region Solver, and GMP routines, ILP64 interface	n/a	n/a	libmkl_solver_ilp64.a or libmkl_solver_ilp64_sequential.a libmkl_core.a	n/a
Direct Sparse Solver/ PARDISO* Solver	libmkl_core.a	libmkl_lapack.dylib libmkl_core.dylib	libmkl_core.a	libmkl_lapack.dylib libmkl_core.dylib

See also: [Linking with Compiler Support RTLs](#).

Linking with Compiler Support RTLs

You are strongly encouraged to dynamically link in the Intel Compatibility OpenMP* run-time library `libiomp` or Intel Legacy OpenMP* run-time library `libguide`. Linking to static OpenMP* run-time library is not recommended because it is very easy with complex software to link in more than one copy of the library. This causes performance problems (too many threads) and may cause correctness problems if more than one copy is initialized.

You are advised to link with `libiomp` and `libguide` dynamically even if other libraries are linked statically.

However, if you link with `libiomp/libguide` statically, the version of `libiomp/libguide` you link with depends on which compiler you use:

- If you use the Intel compiler, link in the `libiomp/libguide` version that comes with the compiler, that is, use the `-openmp` option.
- If you do not use the Intel compiler, link in the `libiomp/libguide` version that comes with Intel MKL.

If you link with dynamic versions of `libiomp/libguide` (recommended), that is, use `libiomp5.dylib` or `libguide.dylib`, make sure `DYLD_LIBRARY_PATH` is defined correctly. See [Set Environment Variables](#) for details.

Linking with System Libraries

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

On Mac OS X*, `libiomp/libguide` both rely on the native `pthread` library for multi-threading. Any time `libiomp/libguide` is required, add `-lpthread` to your link line afterwards (the order of listing libraries is important).

Linking Examples

The section provides specific linking examples that use Intel® compilers on systems based on the IA-32, Intel® 64, and IA-64 architectures. In these examples, `<MKL path>` and `<MKL include>` placeholders are replaced with user-defined environment variables `$MKLPATH` and `$MKLINCLUDE`, respectively.

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

For assistance in finding the right link line, use the Web-based linking advisor available from <http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor>.

Linking on Systems Based on the IA-32 Architecture

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


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE
$MKLSPATH/libmkl_intel.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel.a
$MKLSPATH/libmkl_intel_thread.a $MKLSPATH/libmkl_core.a -liomp5
-lpthread
```
2. Dynamic linking of `myprog.f` and parallel Intel MKL:


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```
3. Static linking of `myprog.f` and sequential version of Intel MKL:


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE
$MKLSPATH/libmkl_intel.a $MKLSPATH/libmkl_sequential.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel.a
$MKLSPATH/libmkl_sequential.a $MKLSPATH/libmkl_core.a -lpthread
```
4. Dynamic linking of `myprog.f` and sequential version of Intel MKL:


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_sequential -lmkl_core -lpthread
```
5. Static linking of `myprog.f`, Fortran 95 LAPACK interface¹, and parallel Intel MKL:


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -I$MKLINCLUDE/32
-lmkl_lapack95
$MKLSPATH/libmkl_intel.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a -liomp5 -lpthread
```
6. Static linking of `myprog.f`, Fortran 95 BLAS interface¹, and parallel Intel MKL:


```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -I$MKLINCLUDE/32 -lmkl_blas95
$MKLSPATH/libmkl_intel.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel.a
$MKLSPATH/libmkl_intel_thread.a $MKLSPATH/libmkl_core.a -liomp5
-lpthread
```
7. Static linking of `myprog.f`, parallel version of an iterative sparse solver, and parallel Intel MKL:

1. See [Fortran 95 Interfaces to LAPACK and BLAS](#) for information on how to build Fortran 95 LAPACK and BLAS interface libraries.

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

8. Static linking of `myprog.f`, sequential version of an iterative sparse solver, and sequential Intel MKL:

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

Linking on Systems Based on the Intel® 64 Architecture

1. Static linking of `myprog.f` and parallel Intel MKL supporting 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
```

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

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

3. Static linking of `myprog.f` and sequential version of Intel MKL supporting 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
```

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

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

5. Static linking of `myprog.f` and parallel Intel MKL supporting 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
```

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

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

7. Static linking of myprog.f, Fortran 95 LAPACK interface¹, and parallel Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -I$MKLINCLUDE/em64t/lp64
-lmkl_lapack95_lp64
$MKLSPATH/libmkl_intel_lp64.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a -liomp5 -lpthread
```

8. Static linking of myprog.f, Fortran 95 BLAS interface¹, and parallel Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -I$MKLINCLUDE/em64t/lp64
-lmkl_blas95_lp64
$MKLSPATH/libmkl_intel_lp64.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_lp64.a
$MKLSPATH/libmkl_intel_thread.a $MKLSPATH/libmkl_core.a -liomp5
-lpthread
```

9. Static linking of myprog.f, parallel version of an iterative sparse solver, and parallel Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -lmkl_solver_lp64
$MKLSPATH/libmkl_intel_lp64.a $MKLSPATH/libmkl_intel_thread.a
$MKLSPATH/libmkl_core.a $MKLSPATH/libmkl_intel_lp64.a
$MKLSPATH/libmkl_intel_thread.a $MKLSPATH/libmkl_core.a -liomp5
-lpthread
```

10. Static linking of myprog.f, sequential version of an iterative sparse solver, and sequential Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLSPATH -I$MKLINCLUDE -lmkl_solver_lp64_sequential
$MKLSPATH/libmkl_intel_lp64
$MKLSPATH/libmkl_sequential.a $MKLSPATH/libmkl_core.a
$MKLSPATH/libmkl_intel_lp64.a $MKLSPATH/libmkl_sequential.a
$MKLSPATH/libmkl_core.a -lpthread
```

Building Custom Dynamically Linked Shared Libraries

Custom dynamically linked shared libraries enable reducing 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.

1. See [Fortran 95 Interfaces to LAPACK and BLAS](#) for information on how to build Fortran 95 LAPACK and BLAS interface libraries.

Intel MKL Custom Dynamically Linked Shared Library Builder

Custom dynamically linked shared library builder enables creation of a dynamically 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. The makefile has two targets: `ia32`, and `em64t`. Use the `ia32` target for processors the use the IA-32 architecture and use `em64t` for processors that use the Intel® 64 architecture.

Specifying Makefile Parameters

There are several macros (parameters) for the makefile:

`export=user_list`

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_list` (no extension).

`name=mkl_custom`

Specifies the name of the library to be created. By default, the name of the created library is `mkl_custom.dylib`.

`xerbla=user_xerbla.o`

Specifies the name of the object file that contains the user's error handler. This error handler will be added to the library and used 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.

`MKLROOT=<MKL_directory>`

Specifies the location of Intel MKL libraries used to build the custom dynamically linked shared library. The installation directory for the current Intel MKL release is used by default.

All parameters are optional.

In the simplest case, the command line is `make ia32`, and the missing parameters 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`.

Specifying a List of Functions

Adjust entry points in the *user_list* file to the required interface. For example, for Fortran functions append an underscore character "_" to the names as a suffix:

dgemm_

ddot_

dgetrf_

If selected functions have several processor-specific versions, they all will be included in the custom library and managed by the dispatcher.

Managing Performance and Memory

6

This chapter features different ways to obtain the best performance with the Intel® Math Kernel Library (Intel® MKL): primarily, it discusses threading (see [Using the Intel® MKL Parallelism](#)), then shows coding techniques and gives hardware configuration tips for improving performance. The chapter also discusses the Intel MKL memory management and shows how to redefine memory functions used by the library.

Using the Intel® MKL Parallelism

Intel MKL is threaded in a number of places:

- Direct sparse solver.
- LAPACK
 - Linear equations, computational routines:
 - factorization: *getrf, *gbtrf, *potrf, *pptrf, *sytrf, *hetrf, *sptrf, *hptrf
 - solving: *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, *stegr, *stedc.

Note that 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, etc.

- Level1 and Level2 BLAS functions:
 - Level1 BLAS: *axpy, *copy, *swap, ddot/sdot, drot/srot
 - Level2 BLAS: *gemv, *trmv, dsyr/ssyr, dsyr2/ssyr2, dsymv/ssymvNote that these functions are threaded only for:
 - Intel® 64 architecture
 - Intel® Core™2 Duo and Intel® Core™ i7 processors
- All Level 3 BLAS and all Sparse BLAS routines except Level 2 triangular solvers.
- VML.
- FFT.

Because it is designed for multi-threaded programming, Intel MKL is *thread-safe*, which means that all Intel MKL functions¹ 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. Due to thread-safety, 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 environmental variable `OMP_NUM_THREADS` to specify the number of threads to use. There are different means to set the number of threads. In Intel MKL releases earlier than 10.0, you could use the environment variable `OMP_NUM_THREADS` (see [Setting the Number of Threads Using OpenMP* Environment Variable](#) for details) or the equivalent OpenMP run-time function calls (detailed in section [Changing the Number of Threads at Run Time](#)). Starting with version 10.0, Intel MKL also offers variables that are independent of OpenMP, such as `MKL_NUM_THREADS`, and equivalent Intel MKL functions for thread management (see [Using Additional Threading Control](#) for details). The Intel MKL variables are always inspected first, then the OpenMP variables are examined, and if neither are used, the OpenMP software chooses the default number of threads. This is a change from the Intel MKL versions 9.x or lower.



NOTE. Starting with Intel MKL 10.0, the OpenMP* software determines the default number of threads. The default number of threads is equal to the number of logical processors in your system for Intel OpenMP* libraries.

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](#).

1. Except LAPACK deprecated routines ?lacon, ?lasq3, and ?lasq4.

Techniques to Set the Number of Threads

You can employ different techniques to change 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()`

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

- The Intel MKL threading controls take precedence over the OpenMP techniques.
- A function call takes precedence over any environment variables. The exception, which is a consequence of the previous rule, is the OpenMP subroutine `omp_set_num_threads()`, which does not have precedence over Intel MKL environment variables, such as `MKL_NUM_THREADS`.
- The environment variables cannot be used to change run-time behavior in the course of the run, because they are read only once at the first call to Intel MKL.

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. First, this section briefly discusses why the problems exist.

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 the user specifically requests Intel MKL to do so via the `MKL_DYNAMIC` functionality (see [Using Additional Threading Control](#) for details). 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 the user's 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.

Here are several cases with recommendations depending on the threading model you employ:

Table 6-1 How to Avoid Conflicts in the Execution Environment for 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 libiomp (libguide). 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).

See also:

[Linking with Compiler Support RTLs](#).

Setting the Number of Threads Using 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 [Using Additional Threading Control](#) on how to set the number of threads using Intel MKL environment variables, for example, MKL_NUM_THREADS.

Changing the Number of Threads at Run Time

You cannot change the number of threads during run time using the 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® 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.

Example 6-1 Changing the Number of Threads

```
// ***** C language *****

#include "omp.h"
#include "mkl.h"
#include <stdio.h>

#define SIZE 1000

void main(int args, char *argv[]){

    double *a, *b, *c;
    a = new double [SIZE*SIZE];
    b = new double [SIZE*SIZE];
    c = new 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++){
```

Example 6-1 Changing the Number of Threads (continued)

```

        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]);
    }

    delete [] a;
    delete [] b;
    delete [] c;
}

// ***** Fortran language *****

PROGRAM DGEMM_DIFF_THREADS

INTEGER      N, I, J
PARAMETER   (N=1000)

REAL*8      A(N,N),B(N,N),C(N,N)
REAL*8      ALPHA, BETA

INTEGER*8    MKL_MALLOC
integer     ALLOC_SIZE

integer     NTHRS

```

Example 6-1 Changing the Number of Threads (continued)

```
ALLOC_SIZE = 8*N*N
A_PTR = MKL_MALLOC(ALLOC_SIZE,128)
B_PTR = MKL_MALLOC(ALLOC_SIZE,128)
C_PTR = MKL_MALLOC(ALLOC_SIZE,128)

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)
```

Example 6-1 Changing the Number of Threads (continued)

```
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 has new optional threading controls, that is, the new environment variables and service functions. They behave similar to their OpenMP equivalents, but take precedence over them. 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 from 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. Intel MKL does not always 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.

[Table 6-2](#) lists the Intel MKL environment variables for threading control, their equivalent functions, and OMP counterparts:

Table 6-2 Environment Variables for Threading Controls

Environment Variable	Service Function	Comment	Equivalent OpenMP* Environment Variable
MKL_NUM_THREADS	mkl_set_num_threads	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.

In particular, if in your application, you want Intel MKL to use a given number of threads and do not want users of your application to change this via environment variables, set this 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 mimic the Intel MKL 9.x default behavior, that is, running on one thread.

Example 6-2 Setting the Number of Threads to One

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

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

The section further explains the Intel MKL environment variables for threading control. 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 default value of `MKL_DYNAMIC` is `TRUE`, regardless of `OMP_DYNAMIC`, whose default value may be `FALSE`.

`MKL_DYNAMIC` being `TRUE` means that Intel MKL will always try to pick what it considers the best number of threads, up to the maximum specified by the user.

`MKL_DYNAMIC` being `FALSE` means that Intel MKL will normally try 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 mainly because the library may have no choice on this number for such reasons as system resources. Moreover, the library may examine the problem and pick a different number of threads than the value suggested. For example, if you attempt to do a size 1 matrix-matrix multiply across 8 threads, the library may instead choose to use only one thread because it is impractical to use 8 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, you should set `MKL_DYNAMIC` to `FALSE` only under circumstances that Intel MKL is unable to detect, for example, when nested parallelism is desired where the library is called already from a parallel section.

`MKL_DYNAMIC` being `TRUE`, in particular, provides for 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 hyper-threading), 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.

MKL_DOMAIN_NUM_THREADS

`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_ALL | MKL_BLAS | MKL_FFT | MKL_VML
<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, MKL_BLAS indicates the BLAS function domain, MKL_FFT indicates FFTs, and MKL_VML indicates the Vector Mathematics Library.

For example,

```

MKL_ALL 2 : MKL_BLAS 1 : MKL_FFT 4
MKL_ALL=2 : MKL_BLAS=1 : MKL_FFT=4
MKL_ALL=2, MKL_BLAS=1, MKL_FFT=4
MKL_ALL=2; MKL_BLAS=1; MKL_FFT=4
MKL_ALL = 2 MKL_BLAS 1 , MKL_FFT 4
MKL_ALL,2: MKL_BLAS 1, MKL_FFT,4 .
    
```

The global variables MKL_ALL, MKL_BLAS, MKL_FFT, and MKL_VML, as well as the interface for the Intel MKL threading control functions, can be found in the `mk1.h` header file.

[Table 6-3](#) illustrates how values of MKL_DOMAIN_NUM_THREADS are interpreted.

Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS Values

Value of MKL_DOMAIN_NUM_THREADS	Interpretation
MKL_ALL=4	All parts of Intel MKL are suggested to try using 4 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_ALL=1, MKL_BLAS=4	All parts of Intel MKL are suggested to use 1 thread, except for BLAS, which is suggested to try 4 threads.

Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS Values

MKL_VML = 2	VML is suggested to try 2 threads. The setting affects no other part of Intel MKL.
-------------	--



NOTE. The domain-specific settings take precedence over the overall ones. For example, the "MKL_BLAS=4" value of MKL_DOMAIN_NUM_THREADS suggests trying 4 threads for BLAS, regardless of later setting MKL_NUM_THREADS, and a function call "mkl_domain_set_num_threads (4, MKL_BLAS);" suggests the same, regardless of later calls to mkl_set_num_threads(). However, pay attention to that a function call with input "MKL_ALL", such as "mkl_domain_set_num_threads (4, MKL_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_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_BLAS=4,MKL_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_BLAS );
```

```
mkl_domain_set_num_threads ( 2, MKL_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_ALL=1, MKL_BLAS=4"
```

```
export MKL_DYNAMIC=FALSE
```

Tips and Techniques to Improve Performance

To obtain the best performance with Intel MKL, follow the recommendations given in the subsections below.

Coding Techniques

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

- Align arrays at 16-byte boundaries.
- Make sure leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 16.
- For two-dimensional arrays, avoid leading dimension values divisible by 2048.

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 by using an 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

```
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 ($n * \text{element_size}$), of two-dimensional arrays should be divisible by cache line size, which equals 64 bytes.

Hardware Configuration Tips

Dual-Core Intel® Xeon® processor 5100 series systems. To get the best Intel MKL performance 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 where, as described in your BIOS documentation.

The use of Hyper-Threading Technology. Hyper-Threading Technology (HT Technology) is especially effective when each thread is performing 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 *HT Technology*. See [Using the Intel® MKL Parallelism](#) for information on the default number of threads, changing this number, and other relevant details.

If you run with HT 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 ones 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, you are recommended to set `KMP_AFFINITY=granularity=fine,compact,1,0`.

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, and 11.

Using the Intel® MKL Memory Management

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 certain functions (Level 3 BLAS or FFT) 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 will again allocate the buffers and they will 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. In addition to calling the `mkl_free_buffers()` function, you can release (free) memory in your program by setting an environment variable.

The memory management software is turned on by default, which leaves memory allocated by calls to Level 3 BLAS and FFT until the program ends. To disable this behavior of the memory management software, set the `MKL_DISABLE_FAST_MM` environment variable to any value, which will cause memory to be allocated and freed from call to call. Disabling this feature will negatively impact performance of routines such as the level 3 BLAS, especially for small problem sizes.

Using one of these methods to release memory will not necessarily stop programs from reporting memory leaks, and, in fact, may increase the number of such reports in case you make multiple calls to the library, thereby requiring new allocations with each call. Memory not released by one of the methods described previously will be released by the system when the program ends.

Redefining Memory Functions

C/C++ users of Intel MKL can replace memory functions that the library uses by default with their own ones. The *memory renaming* feature enables this replacement.

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 an application developer to replace 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`, `i_realloc` prior to the first call to MKL functions:

Example 6-3 Redefining Memory Functions

```
#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

7

The Intel® Math Kernel Library (Intel® MKL) provides broad support for Fortran and C/C++ programming. However, not all function domains support both Fortran and C interfaces (see [Table A-1](#) in Appendix A). For example, LAPACK has no C interface. You can call functions comprising such domains from C using mixed-language programming.

If you want to use LAPACK or BLAS, which support Fortran, 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.

This chapter mainly focuses on mixed-language programming and the use of language-specific interfaces. It expands upon the use of Intel MKL in C language environments for function domains that provide only Fortran interfaces, as well as explains usage of language-specific interfaces, specifically the Fortran 95 interfaces to LAPACK and BLAS. The chapter also discusses compiler-dependent functions to explain why Fortran 90 modules are supplied as sources. A separate section guides you through the process of running examples to invoke Intel MKL functions from Java*.

Using Language-Specific Interfaces with Intel® MKL

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

Table 7-1 Interface Libraries and Modules

File name	Contains
Libraries, in Intel MKL architecture-specific directories	
<code>libmkl_blas95.a¹</code>	Fortran 95 wrappers for BLAS (BLAS95) for IA-32 architecture.
<code>libmkl_blas95_ilp64.a¹</code>	Fortran 95 wrappers for BLAS (BLAS95) supporting LP64 interface.

Table 7-1 Interface Libraries and Modules (continued)

File name	Contains
<code>libmkl_blas95_lp64.a</code> ¹	Fortran 95 wrappers for BLAS (BLAS95) supporting ILP64 interface.
<code>libmkl_lapack95.a</code> ¹	Fortran 95 wrappers for LAPACK (LAPACK95) for IA-32 architecture.
<code>libmkl_lapack95_lp64.a</code> ¹	Fortran 95 wrappers for LAPACK (LAPACK95) supporting LP64 interface.
<code>libmkl_lapack95_ilp64.a</code> ¹	Fortran 95 wrappers for LAPACK (LAPACK95) supporting ILP64 interface.
<code>libfftw2xc_intel.a</code> ¹	Interfaces for FFTW version 2.x (C interface for Intel® compiler) to call Intel MKL FFTs.
<code>libfftw2xc_gnu.a</code>	Interfaces for FFTW version 2.x (C interface for GNU compiler) to call Intel MKL FFTs.
<code>libfftw2xf_intel.a</code>	Interfaces for FFTW version 2.x (Fortran interface for Intel compiler) to call Intel MKL FFTs.
<code>libfftw2xf_gnu.a</code>	Interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
<code>libfftw3xc_intel.a</code> ²	Interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs.
<code>libfftw3xc_gnu.a</code>	Interfaces for FFTW version 3.x (C interface for GNU compiler) to call Intel MKL FFTs.
<code>libfftw3xf_intel.a</code> ²	Interfaces for FFTW version 3.x (Fortran interface for Intel compiler) to call Intel MKL FFTs.
<code>libfftw3xf_gnu.a</code>	Interfaces for FFTW version 3.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
Modules, in architecture- and interface-specific subdirectories of the Intel MKL include directory	
<code>blas95.mod</code> ¹	Fortran 95 interface module for BLAS (BLAS95).
<code>lapack95.mod</code> ¹	Fortran 95 interface module for LAPACK (LAPACK95).
<code>f95_precision.mod</code> ¹	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95.
<code>mk195_blas.mod</code> ¹	Fortran 95 interface module for BLAS (BLAS95), identical to <code>blas95.mod</code> . To be removed in one of the future releases.
<code>mk195_lapack.mod</code> ¹	Fortran 95 interface module for LAPACK (LAPACK95), identical to <code>lapack95.mod</code> . To be removed in one of the future releases.
<code>mk195_precision.mod</code> ¹	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95, identical to <code>f95_precision.mod</code> . To be removed in one of the future releases.

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 [Fortran 95 Interfaces to LAPACK and BLAS](#), which shows by example how to generate these libraries and modules.

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

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:

```
make lib32 INSTALL_DIR=<user_dir>           for the IA-32 architecture.
make libem64t [interface=lp64|ilp64]       for the Intel® 64 architecture.
INSTALL_DIR=<user_dir>
```



NOTE. Parameter `INSTALL_DIR` is required.

As a result, the required library will be built and installed in the `<user_dir>/lib/<arch>` directory, and the `.mod` files will be built and installed in the `<user_dir>/include/<arch>[/lp64]` directory, where `<arch>` is one of `{32, em64t}`.

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

For example, command

```
make libem64t FC=pgf95 INSTALL_DIR=<user_pgf95_dir> interface=lp64
```

builds the required library and `.mod` files and installs them in subdirectories of `<user_pgf95_dir>`.

There is also a way to use the interfaces without building the libraries.

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

<code>make clean32 INSTALL_DIR=<user_dir></code>	for the IA-32 architecture.
<code>make cleanem64t INSTALL_DIR=<user_dir></code>	for the Intel® 64 architecture.
<code>make clean INSTALL_DIR=<user_dir></code>	for all the architectures.



NOTE. Setting `INSTALL_DIR=../..` or `INSTALL_DIR=<mkl_directory>` in a build or clean command above will replace or delete the Intel MKL prebuilt Fortran 95 library and modules. Though this is possible only if you have administrative rights, you are strongly discouraged from doing this.

Compiler-dependent Functions and Fortran 90 Modules

Compiler-dependent functions occur whenever the compiler places 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.

Where dependencies occur, a supporting RTL is shipped with Intel MKL. The only examples of such RTLs are `libiomp` and `libguide`, which are the libraries for the OpenMP* code compiled with an Intel® compiler. Both `libiomp` and `libguide` support the threaded code in Intel MKL.

In other 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, so, Intel MKL delivers these modules as source code.

Mixed-language Programming with Intel® MKL

Appendix A lists the programming languages supported for each Intel MKL function domain. However, you can call Intel MKL routines from different language environments. This section explains how to do this using mixed-language programming.

Calling LAPACK, BLAS, and CBLAS Routines from 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.



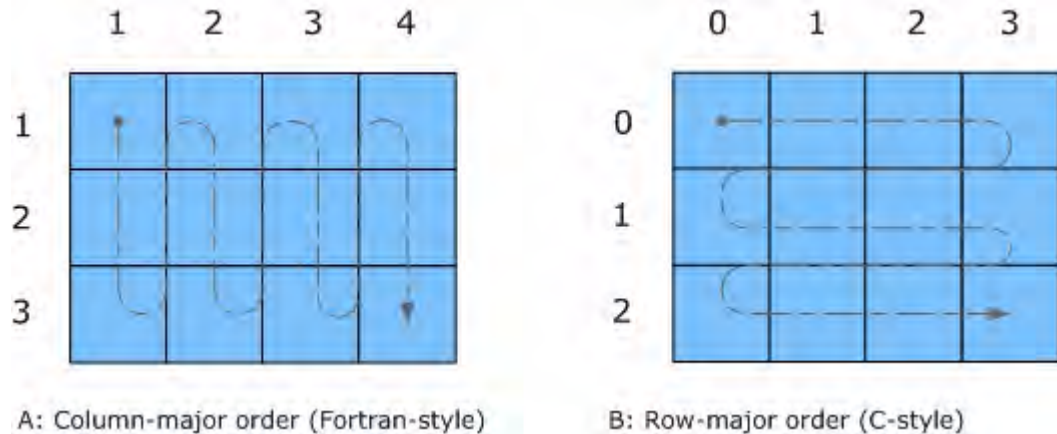
WARNING. Avoid calling BLAS95/LAPACK95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

LAPACK

Because LAPACK routines are Fortran-style, when calling them from C-language programs, make sure that you follow the Fortran-style calling conventions:

- Pass variables by *address* as opposed to pass by *value*.
Function calls in [Example 7-1](#) and [Example 7-2](#) 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 one changes most quickly (as illustrated by [Figure 7-1](#) for a two-dimensional array).

Figure 7-1 Column-major Order versus Row-major Order



For example, if a two-dimensional matrix A of size $m \times n$ is stored densely in a one-dimensional array B , you can access a matrix element like this:

$$A[i][j] = B[i*n+j] \text{ in C} \quad (i=0, \dots, m-1, j=0, \dots, n-1)$$

$$A(i,j) = B(j*m+i) \text{ in Fortran} \quad (i=1, \dots, m, j=1, \dots, n).$$

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

BLAS

BLAS routines are Fortran-style routines. If you call BLAS routines from a C-language program, you must follow the Fortran-style calling conventions:

- Pass variables by *address* as opposed to passing by *value*
- Store data in Fortran style, that is, column-major rather than row-major order

Refer to the [LAPACK](#) section for details of these conventions. See [Example 7-1](#) on how to call BLAS routines from C.

When calling BLAS routines from C, be aware that because the Fortran language is case-insensitive, the BLAS routine names can be both upper-case and lower-case, with or without the trailing underscore. For example, these names are equivalent: `dgemm`, `DGEMM`, `dgemm_`, `DGEMM_`.

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. When using the CBLAS interface, the header file `mkl.h` will simplify the program development because it specifies enumerated values as well as prototypes of all the functions. The header determines if 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 7-2](#) illustrates the use of the CBLAS interface.

Using Complex Types in C/C++

As described in the *Building Applications document for the Intel® Fortran Compiler, 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. These 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 7-1](#) for details. You can also define these types in the command line:

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

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

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

You must be careful when handling a call from C to a BLAS function that returns complex values. The problem arises because BLAS comprises Fortran functions, and complex values they return are handled quite differently for C and Fortran. 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. This feature can be used by the C programmer.

The following example shows how this works.

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

(notice that the hidden

parameter gets exposed): `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_`, `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.

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`.

Below is the C++ implementation:

Example 7-1 Calling a Complex BLAS Level 1 Function from C++

```
#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;
}
```

The example below uses CBLAS:

Example 7-2 Using CBLAS Interface Instead of Calling BLAS Directly from C

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;

extern "C" void cblas_zdotc_sub ( const int , const complex16 *,
    const int , const complex16 *, const int, const complex16*);

#define N 5

void 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 );
}
```

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 pertains to the Boost C++ open-source libraries and 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.
Type and conformance checking is performed.
- Release (fast) mode.
Enabled by 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 `mk1_boost_ublas_matrix_prod.hpp` in your code (from the Intel MKL include directory).
- Add appropriate Intel MKL libraries to the link line (see [Linking Your Application with the Intel® Math Kernel Library](#)).

Only the following expressions are substituted:

```
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, 1.35.0, 1.36.0, and 1.37.0. To get them, visit www.boost.org.

A code example provided in the `<mkldirectory>/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 lib32 BOOST_ROOT=<your_path>/boost_1_37_0
```

Invoking Intel® MKL Functions from Java* Applications

This section describes examples that are provided with the Intel MKL package and illustrate calling the library functions from Java.

Intel MKL Java Examples

Java was positioned by its inventor, the Sun Microsystems Corporation as a "Write Once Run Anywhere" (WORA) language. Intel MKL may help to speed-up Java applications, while partially supporting the WORA philosophy, because Intel MKL editions are intended for a wide variety of operating systems and processors including most kinds of laptops and desktops, as well as many workstations and servers.

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

```
<mkldirectory>/examples/java .
```

The examples are provided for the following MKL functions:

- `?gemm`, `?gemv`, and `?dot` families from CBLAS
- The complete set of non-cluster FFT functions
- ESSL¹-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:

```
<mkldirectory>/examples/java/examples .
```

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

```

<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_ALL | MKL_BLAS | MKL_FFT | MKL_VML
<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, MKL_BLAS indicates the BLAS function domain, MKL_FFT indicates non-cluster FFTs, and MKL_VML indicates the Vector Mathematics Library.

For example,

```

MKL_ALL 2 : MKL_BLAS 1 : MKL_FFT 4
MKL_ALL=2 : MKL_BLAS=1 : MKL_FFT=4
MKL_ALL=2, MKL_BLAS=1, MKL_FFT=4
MKL_ALL=2; MKL_BLAS=1; MKL_FFT=4
MKL_ALL = 2 MKL_BLAS 1 , MKL_FFT 4
MKL_ALL,2: MKL_BLAS 1, MKL_FFT,4 .
    
```

The global variables MKL_ALL, MKL_BLAS, MKL_FFT, and MKL_VML, as well as the interface for the Intel MKL threading control functions, can be found in the `mk1.h` header file.

[Table 6-3](#) illustrates how values of MKL_DOMAIN_NUM_THREADS are interpreted.

Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS Values

Value of MKL_DOMAIN_NUM_THREADS	Interpretation
MKL_ALL=4	All parts of Intel MKL are suggested to try using 4 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_ALL=1, MKL_BLAS=4	All parts of Intel MKL are suggested to use 1 thread, except for BLAS, which is suggested to try 4 threads.

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):

```
<mk1 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:

```
<mk1 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.

Running the Examples

The Java examples support all the C and C++ compilers that the 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 for IA-32 architecture with the following Java implementation:

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

Also 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 {dylib32|lib32} [function=...] [compiler=...]
```

If you type the make command and omit the target (`dylib32`), 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

There are three kinds of limitations:

- Functionality
- Performance

- Known bugs

Functionality. It is possible that some MKL functions will not 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 functions from Intel MKL 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, so most of the examples work anyway. The source code in the examples and wrappers include comments that describe the workarounds.

This chapter discusses programming with the Intel® Math Kernel Library (Intel® MKL) to provide coding tips that meet certain, specific needs, such as numerical stability. Similarly, Chapter 7 focuses on general language-specific programming options, and Chapter 6 provides tips relevant to performance and memory management.

Aligning Data for Numerical Stability

If linear algebra routines (LAPACK, BLAS) are applied to input data that are bit-for-bit identical but the arrays are aligned differently or the computations are performed either on different platforms or with different numbers of threads, the output may not be bit-for-bit identical, though they will deviate within the appropriate error bounds. The Intel MKL version may also affect numerical stability of the output, as the routines may be implemented differently in different versions. With a given Intel MKL version, the outputs will be bit-for-bit identical provided all the following conditions are met:

- the outputs are obtained on the same platform
- the inputs are bit-for-bit identical
- the input arrays are aligned identically at 16-byte boundaries
- Intel MKL is run in the sequential mode

Unlike the first two conditions, which are under the users' control, the alignment of arrays, by default, is not. For instance, arrays dynamically allocated using `malloc` are aligned at 8-byte boundaries, not 16-byte boundaries. If you need numerically identical outputs, use `mkl_malloc()` to get the properly aligned workspace, as shown below:

Example 8-1 Aligning Addresses at 16-byte Boundaries

```
// ***** C language *****  
...  
#include <stdlib.h>  
...  
void *darray;  
int workspace;  
...  
// Allocate workspace aligned on 16-bit boundary  
darray = mkl_malloc( sizeof(double)*workspace, 16 );  
...  
// call the program using MKL  
mkl_app( darray );  
...  
// Free workspace  
mkl_free( darray );  
  
! ***** Fortran language *****  
...  
double precision darray  
pointer (p_wrk,darray(1))  
integer workspace  
...  
! Allocate workspace aligned on 16-bit boundary  
p_wrk = mkl_malloc( 8*workspace, 16 )  
...  
! call the program using MKL  
call mkl_app( darray )  
...  
! Free workspace  
call mkl_free(p_wrk)
```

Intel® Optimized LINPACK Benchmark

9

This chapter describes the Intel® Optimized LINPACK Benchmark for Mac OS* X .

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.

This benchmark should not be used to report LINPACK 100 performance, as 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 systems based on genuine Intel® processors 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

The Intel Optimized LINPACK Benchmark for Mac OS* X contains the following files, located in the `./benchmarks/linpack/` subdirectory in the Intel MKL directory (see [Table 3-1](#)):

Table 9-1 Contents of the LINPACK Benchmark

./benchmarks/linpack/	
<code>linpack_cd32.app</code>	The 32-bit program executable for a system using Intel® Core™ Duo processor on Mac OS* X.
<code>linpack_cd64.app</code>	The 64-bit program executable for a system using Intel® Core™ microarchitecture on Mac OS* X.
<code>runme32</code>	A sample shell script for executing a pre-determined problem set for <code>linpack_cd32.app</code> . <code>OMP_NUM_THREADS</code> set to 2 cores.
<code>runme64</code>	A sample shell script for executing a pre-determined problem set for <code>linpack_cd64.app</code> . <code>OMP_NUM_THREADS</code> set to 2 cores.
<code>lininput</code>	Input file for pre-determined problem for the <code>runme32</code> script.
<code>lin_cd32.txt</code>	Result of the <code>runme32</code> script execution.
<code>lin_cd64.txt</code>	Result of the <code>runme64</code> script execution.
<code>help.lpk</code>	Simple help file.
<code>xhelp.lpk</code>	Extended help file.

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, please refer to 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 amount 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, in particular, 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 already match the situation for your machine, edit the script.

Known Limitations

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



[Table A-1](#) shows language interfaces that Intel® Math Kernel Library (Intel® MKL) provides for each function domain, and [Table A-2](#) lists the respective header files. However, Intel MKL routines can be called from other languages using mixed-language programming. For example, see [Mixed-language Programming with Intel® MKL](#) on how to call Fortran routines from C/C++.

Table A-1 Language Interfaces Support

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	†
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	Yes	Yes	†
Auxiliary and utility LAPACK routines	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

Table A-1 Language Interfaces Support (continued)

Function Domain	FORTRAN 77 interface	Fortran 90/95 interface	C/C++ interface
GMP* arithmetic functions			Yes
Service routines (including memory allocation)			Yes

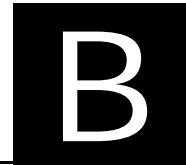
† Supported using a mixed language programming call. See [Table A-2](#) for the respective header file.

[Table A-2](#) lists available header files for all Intel MKL function domains.

Table A-2 Include Files

Function domain	Include files	
	Fortran	C or C++
All function domains	<code>mkl.fi</code>	<code>mkl.h</code>
BLAS Routines	<code>blas.f90</code> <code>mkl_blas.fi</code>	<code>mkl_blas.h</code>
BLAS-like Extension Transposition Routines	<code>mkl_trans.fi</code>	<code>mkl_trans.h</code>
CBLAS Interface to BLAS		<code>mkl_cblas.h</code>
Sparse BLAS Routines	<code>mkl_spblas.fi</code>	<code>mkl_spblas.h</code>
LAPACK Routines	<code>lapack.f90</code> <code>mkl_lapack.fi</code>	<code>mkl_lapack.h</code>
All Sparse Solver Routines	<code>mkl_solver.f90</code>	<code>mkl_solver.h</code>
• PARDISO	<code>mkl_pardiso.f77</code> <code>mkl_pardiso.f90</code>	<code>mkl_pardiso.h</code>
• DSS Interface	<code>mkl_dss.f77</code> <code>mkl_dss.f90</code>	<code>mkl_dss.h</code>
• RCI Iterative Solvers	<code>mkl_rci.fi</code>	<code>mkl_rci.h</code>
• ILU Factorization		
Optimization Solver Routines	<code>mkl_rci.fi</code>	<code>mkl_rci.h</code>
Vector Mathematical Functions	<code>mkl_vml.f77</code> <code>mkl_vml.fi</code>	<code>mkl_vml.h</code>
Vector Statistical Functions	<code>mkl_vml.f77</code> <code>mkl_vsl.fi</code>	<code>mkl_vsl.h</code>
Fourier Transform Functions	<code>mkl_dfti.f90</code>	<code>mkl_dfti.h</code>
Partial Differential Equations Support Routines		
• Trigonometric Transforms	<code>mkl_trig_transforms.f90</code>	<code>mkl_trig_transforms.h</code>
• Poisson Solvers	<code>mkl_poisson.f90</code>	<code>mkl_poisson.h</code>
GMP interface		<code>mkl_gmp.h</code>
Service routines		<code>mkl_service.h</code>
Memory allocation routines		<code>i_malloc.h</code>
MKL examples interface		<code>mkl_example.h</code>

Support for Third-Party Interfaces



This appendix describes in brief certain interfaces that Intel® Math Kernel Library (Intel® MKL) supports.

GMP* Functions

Intel MKL implementation of GMP* arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision* (GMP) Arithmetic Library. For specifications of these functions, please see <http://www.intel.com/software/products/mkl/docs/gnump/WebHelp/>.

If you currently use the GMP* library, you need to modify `INCLUDE` statements in your programs to `mkl_gmp.h`.

FFTW Interface Support

Intel MKL offers two collections of wrappers being the FFTW interface (www.fftw.org) superstructure 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 developers whose programs currently use FFTW to gain performance with the Intel MKL Fourier transforms without changing the program source code. See the *"FFTW to Intel® MKL Wrappers"* appendix in the *Intel MKL Reference Manual* for details on the use of the wrappers.

Index

A

aligning data, 8-2
Apple Xcode*, configuring, 4-1

B

benchmark, 9-1
BLAS
 calling routines from C, 7-6
 Fortran-95 interfaces to, 7-3
BLAS, Fortran-95 interfaces to, 7-3

C

C, calling LAPACK, BLAS, CBLAS from, 7-5
calling
 BLAS functions in C, 7-8
 complex BLAS Level 1 function from C++, 7-9
 Fortran-style routines from C, 7-5
CBLAS, 7-7
CBLAS, code example, 7-10
coding
 data alignment, 8-1
 mixed-language calls, 7-8
 techniques to improve performance, 6-13
Compatibility OpenMP* run-time library, 3-4
compiler support, 2-4
compiler support run-time libraries, 3-4
compiler-dependent function, 7-4
computational layer, 3-4
configuration file, for OOC DSS/PARDISO*, 4-3

configuring Apple Xcode* project, 4-1
configuring development environment, 4-1
custom dynamically linked shared library, 5-9
 building, 5-10
 specifying list of functions, 5-11
 specifying makefile parameters, 5-10

D

data alignment, 8-2
denormal number, performance, 6-14
denormal, performance, 6-14
development environment, configuring, 4-1
directory structure
 documentation, 3-14
 high-level, 3-1
 in-detail, 3-7
documentation, 3-14

E

environment variables, setting, 4-1
examples
 code, 2-4
 linking, general, 5-6

F

FFT functions, data alignment, 6-13
FFT interface
 optimized radices, 6-15
FFTW interface support, B-1

Fortran-95, interfaces to LAPACK and BLAS, 7-3

G

GNU* Multiple Precision Arithmetic Library, B-1

H

HT Technology, *see* Hyper-Threading technology
Hyper-Threading Technology, configuration tip, 6-14

I

ILP64 programming, support for, 3-5
instability, numerical, getting rid of, 8-1
installation, checking, 2-3
interface layer, 3-3

J

Java* examples, 7-12

L

language interfaces support, A-1
 language-specific interfaces, 7-1
LAPACK
 calling routines from C, 7-5
 Fortran-95 interfaces to, 7-3
 packed routines performance, 6-13
layer
 compiler support RTL, 3-4
 computational, 3-4
 interface, 3-3
 threading, 3-3
layered model, 3-2
library
 run-time, Compatibility OpenMP*, 3-4
 run-time, Legacy OpenMP*, 3-4
library structure, 3-1
link libraries
 computational, 5-5
 for Intel(R) 64 architecture, 5-5
 threading, 5-4
linking, 5-1

LINPACK benchmark, 9-1

M

memory functions, redefining, 6-16
memory management, 6-15
memory renaming, 6-16
mixed-language programming, 7-4
module, Fortran-95, 7-4

N

notational conventions, 1-3
number of threads
 changing at run time, 6-5
 changing with OpenMP* environment variable,
 6-4
 Intel(R) MKL choice, particular cases, 6-10
 techniques to set, 6-3
numerical stability, 8-1

O

OpenMP*
 Compatibility run-time library, 3-4
 Legacy run-time library, 3-4
OpenMP*, run-time library, 5-3

P

parallel performance, 6-4
parallelism, 6-1
PARDISO* OOC, configuration file, 4-3
performance, 6-1
 coding techniques to gain, 6-13
 hardware tips to gain, 6-14
 of LAPACK packed routines, 6-13
 with denormals, 6-14
 with subnormals, 6-14

R

RTL, 7-4
run-time library, 7-4
 Compatibility OpenMP*, 3-4

Legacy OpenMP*, 3-4

S

stability, numerical, 8-1

subnormal number, performance, 6-14

support, technical, 1-1

T

technical support, 1-1

thread safety, of Intel(R) MKL, 6-2

threading

- avoiding conflicts, 6-4

- environment variables and functions, 6-8

- Intel(R) MKL behavior, particular cases, 6-10

- Intel(R) MKL controls, 6-8

- see also* number of threads

threading layer, 3-3

U

uBLAS, matrix-matrix multiplication, substitution
with Intel MKL functions, 7-10

unstable output, numerically, getting rid of, 8-1

usage information, 1-1

X

Xcode*, configuring, 4-1