



Intel® Math Kernel Library for Windows* OS

User's Guide

Intel® MKL - Windows* OS

Document Number: 315930-021US

Legal Information

Contents

Legal Information.....	7
Introducing the Intel® Math Kernel Library.....	9
Getting Help and Support.....	11
Notational Conventions.....	13
Chapter 1: Overview	
Document Overview.....	15
What's New.....	15
Related Information.....	15
Chapter 2: Getting Started	
Checking Your Installation.....	17
Setting Environment Variables	17
Compiler Support.....	19
Using Code Examples.....	19
What You Need to Know Before You Begin Using the Intel® Math Kernel Library.....	19
Chapter 3: Structure of the Intel® Math Kernel Library	
Architecture Support.....	23
High-level Directory Structure.....	23
Layered Model Concept.....	25
Contents of the Documentation Directories.....	26
Chapter 4: Linking Your Application with the Intel® Math Kernel Library	
Linking Quick Start.....	27
Using the /Qmkl Compiler Option.....	27
Automatically Linking a Project in the Visual Studio* Integrated Development Environment with Intel® MKL.....	28
Automatically Linking Your Microsoft Visual C/C++* Project with Intel® MKL.....	28
Automatically Linking Your Intel® Visual Fortran Project with Intel® MKL.....	28
Using the Single Dynamic Library.....	28
Selecting Libraries to Link with.....	29
Using the Link-line Advisor.....	29
Using the Command-line Link Tool.....	30
Linking Examples.....	30
Linking on IA-32 Architecture Systems.....	30
Linking on Intel(R) 64 Architecture Systems.....	31
Linking in Detail.....	31
Dynamically Selecting the Interface and Threading Layer.....	32
Linking with Interface Libraries.....	33
Using the cdecl and stdcall Interfaces.....	33
Using the ILP64 Interface vs. LP64 Interface.....	34
Linking with Fortran 95 Interface Libraries.....	36

- Linking with Threading Libraries.....36
 - Sequential Mode of the Library.....36
 - Selecting the Threading Layer.....36
- Linking with Computational Libraries.....37
- Linking with Compiler Run-time Libraries.....38
- Linking with System Libraries.....39
- Building Custom Dynamic-link Libraries.....39
 - Using the Custom Dynamic-link Library Builder in the Command-line Mode.....39
 - Composing a List of Functions41
 - Specifying Function Names.....41
 - Building a Custom Dynamic-link Library in the Visual Studio* Development System.....42
 - Distributing Your Custom Dynamic-link Library.....43

Chapter 5: Managing Performance and Memory

- Using Parallelism of the Intel® Math Kernel Library.....45
 - Threaded Functions and Problems.....45
 - Avoiding Conflicts in the Execution Environment.....47
 - Techniques to Set the Number of Threads.....48
 - Setting the Number of Threads Using an OpenMP* Environment Variable.....48
 - Changing the Number of Threads at Run Time.....48
 - Using Additional Threading Control.....50
 - Intel MKL-specific Environment Variables for Threading Control....50
 - MKL_DYNAMIC.....51
 - MKL_DOMAIN_NUM_THREADS.....52
 - Setting the Environment Variables for Threading Control.....53
- Tips and Techniques to Improve Performance.....53
 - Coding Techniques.....54
 - Hardware Configuration Tips.....54
 - Managing Multi-core Performance.....55
 - Operating on Denormals.....56
 - FFT Optimized Radices.....56
- Using Memory Management56
 - Intel MKL Memory Management Software.....56
 - Redefining Memory Functions.....56

Chapter 6: Language-specific Usage Options

- Using Language-Specific Interfaces with Intel® Math Kernel Library.....59
 - Interface Libraries and Modules.....59
 - Fortran 95 Interfaces to LAPACK and BLAS.....61
 - Compiler-dependent Functions and Fortran 90 Modules.....61
 - Using the stdcall Calling Convention in C/C+.....62
 - Compiling an Application that Calls the Intel® Math Kernel Library and Uses the CVF Calling Conventions.....62
- Mixed-language Programming with the Intel Math Kernel Library.....63
 - Calling LAPACK, BLAS, and CBLAS Routines from C/C++ Language Environments.....63
 - Using Complex Types in C/C+.....64

Calling BLAS Functions that Return the Complex Values in C/C++ Code.....	65
Support for Boost uBLAS Matrix-matrix Multiplication.....	66
Invoking Intel MKL Functions from Java* Applications.....	68
Intel MKL Java* Examples.....	68
Running the Java* Examples.....	69
Known Limitations of the Java* Examples.....	70
Chapter 7: Coding Tips	
Aligning Data for Consistent Results.....	71
Using Predefined Preprocessor Symbols for Intel® MKL Version-Dependent Compilation.....	72
Chapter 8: Working with the Intel® Math Kernel Library Cluster Software	
MPI Support.....	75
Linking with ScaLAPACK and Cluster FFTs.....	75
Determining the Number of Threads.....	77
Using DLLs.....	77
Setting Environment Variables on a Cluster.....	78
Building ScaLAPACK Tests.....	78
Examples for Linking with ScaLAPACK and Cluster FFT.....	78
Examples for Linking a C Application.....	79
Examples for Linking a Fortran Application.....	79
Chapter 9: Programming with Intel® Math Kernel Library in Integrated Development Environments (IDE)	
Configuring Your Integrated Development Environment to Link with Intel Math Kernel Library	81
Configuring the Microsoft Visual C/C++* Development System to Link with Intel® MKL.....	81
Configuring Intel® Visual Fortran to Link with Intel MKL.....	81
Running an Intel MKL Example in the Visual Studio* 2008 IDE.....	82
Creating, Configuring, and Running the Intel® C/C++ and/or Visual C++* 2008 Project.....	82
Creating, Configuring, and Running the Intel Visual Fortran Project.....	84
Support Files for Intel® Math Kernel Library Examples.....	85
Known Limitations of the Project Creation Procedure.....	86
Getting Assistance for Programming in the Microsoft Visual Studio* IDE	86
Viewing Intel MKL Documentation in Visual Studio* IDE.....	86
Using Context-Sensitive Help.....	87
Using the IntelliSense* Capability.....	88
Chapter 10: LINPACK and MP LINPACK Benchmarks	
Intel® Optimized LINPACK Benchmark for Windows* OS.....	91
Contents of the Intel® Optimized LINPACK Benchmark.....	91
Running the Software.....	92
Known Limitations of the Intel® Optimized LINPACK Benchmark.....	93
Intel® Optimized MP LINPACK Benchmark for Clusters.....	93
Overview of the Intel® Optimized MP LINPACK Benchmark for Clusters....	93

Contents of the Intel® Optimized MP LINPACK Benchmark for Clusters...	94
Building the MP LINPACK.....	95
New Features of Intel® Optimized MP LINPACK Benchmark.....	95
Benchmarking a Cluster.....	96
Options to Reduce Search Time.....	96
Appendix A: Intel® Math Kernel Library Language Interfaces Support	
Language Interfaces Support, by Function Domain.....	99
Include Files.....	100
Appendix B: Support for Third-Party Interfaces	
FFTW Interface Support.....	103
GMP* Functions.....	103
Appendix C: Directory Structure in Detail	
Detailed Structure of the IA-32 Architecture Directories.....	105
Static Libraries in the lib\ia32 Directory.....	105
Dynamic Libraries in the lib\ia32 Directory.....	106
Contents of the redistributable\ia32\mkl Directory.....	106
Detailed Structure of the Intel® 64 Architecture Directories.....	107
Static Libraries in the lib\intel64 Directory.....	108
Dynamic Libraries in the lib\intel64 Directory.....	109
Contents of the redistributable\intel64\mkl Directory.....	109

Legal Information

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

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

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

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

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

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

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

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

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

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

Copyright © 2007 - 2012, Intel Corporation. All rights reserved.

Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Introducing the Intel® Math Kernel Library

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

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

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

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Getting Help and Support

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

The Intel MKL documentation integrates into the Microsoft Visual Studio* integrated development environment (IDE). See [Getting Assistance for Programming in the Microsoft Visual Studio* IDE](#).

Notational Conventions

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

Windows* OS This term refers to information that is valid on all supported Windows* operating systems.

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

`<Composer XE directory>` The installation directory for the Intel® C++ Composer XE or Intel® Visual Fortran Composer XE .

`<mkl directory>` The main directory where Intel MKL is installed:
`<mkl directory>=<Composer XE directory>\mkl.`
Replace this placeholder with the specific pathname in the configuring, linking, and building instructions.

The following font conventions are used in this document.

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

Monospace lowercase mixed with uppercase Indicates:
• Commands and command-line options, for example,

```
ifort myprog.f mkl_blas95.lib mkl_c.lib libiomp5md.lib
```

• Filenames, directory names, and pathnames, for example,

```
C:\Program Files\Java\jdk1.5.0_09
```

• C/C++ code fragments, for example,

```
a = new double [SIZE*SIZE];
```

UPPERCASE Indicates system variables, for example, \$MKLPATH.
MONOSPACE

Monospace italic Indicates a parameter in discussions, for example, *lda*.

When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, `<mkl directory>`.
Substitute one of these items for the placeholder.

[items] Square brackets indicate that the items enclosed in brackets are optional.

{ item | item } Braces indicate that only one of the items listed between braces should be selected. A vertical bar (|) separates the items.

Overview

Document Overview

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

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

This User's Guide provides the following information:

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

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

See Also

[Language Interfaces Support, by Function Domain](#)

What's New

This User's Guide documents Intel® Math Kernel Library (Intel® MKL) 10.3 Update 10.

Minor bugs have been fixed in the document.

Related Information

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

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

Getting Started

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Checking Your Installation

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

- Intel MKL installs in *<Composer XE directory>*.
Check that the subdirectory of *<Composer XE directory>* referred to as *<mkl directory>* was created.
Check that subdirectories for Intel MKL redistributable DLLs `redist\ia32\mkl` and `redist\intel64\mkl` were created in the *<Composer XE directory>* directory (See `redist.txt` in the Intel MKL documentation directory for a list of files that can be redistributed.)
- If you want to keep multiple versions of Intel MKL installed on your system, update your build scripts to point to the correct Intel MKL version.
- Check that the following files appear in the *<mkl directory>\bin* directory and its subdirectories:


```
mklvars.bat
ia32\mklvars_ia32.bat
intel64\mklvars_intel64.bat
```

 Use these files to assign Intel MKL-specific values to several environment variables, as explained in [Setting Environment Variables](#)
- To understand how the Intel MKL directories are structured, see [Intel® Math Kernel Library Structure](#).
- To make sure that Intel MKL runs on your system, do one of the following:
 - Launch an Intel MKL example, as explained in [Using Code Examples](#)
 - In the Visual Studio* IDE, create and run a simple project that uses Intel MKL, as explained in [Running an Intel MKL Example in the Visual Studio IDE](#)

See Also

[Notational Conventions](#)

Setting Environment Variables

When the installation of Intel MKL for Windows* OS is complete, set the `PATH`, `LIB`, and `INCLUDE` environment variables in the command shell using one of the script files in the `bin` subdirectory of the Intel MKL installation directory:

```
ia32\mklvars_ia32.bat
```

for the IA-32 architecture,

intel64\mklvars_intel64.bat
mklvars.bat

for the Intel® 64 architecture,
for the IA-32 and Intel® 64 architectures.

Running the Scripts

The scripts accept parameters to specify the following:

- Architecture.
- Addition of a path to Fortran 95 modules precompiled with the Intel® Fortran compiler to the INCLUDE environment variable. Supply this parameter only if you are using the Intel® Fortran compiler.
- Interface of the Fortran 95 modules. This parameter is needed only if you requested addition of a path to the modules.

Usage and values of these parameters depend on the script. The following table lists values of the script parameters.

Script	Architecture (required, when applicable)	Addition of a Path to Fortran 95 Modules (optional)	Interface (optional)
mklvars_ia32	n/a [†]	mod	n/a
mklvars_intel64	n/a	mod	lp64, default ilp64
mklvars	ia32 intel64	mod	lp64, default ilp64

[†] Not applicable.

For example:

- The command
`mklvars_ia32`
sets environment variables for the IA-32 architecture and adds no path to the Fortran 95 modules.
- The command
`mklvars_intel64 mod ilp64`
sets environment variables for the Intel® 64 architecture and adds the path to the Fortran 95 modules for the ILP64 interface to the INCLUDE environment variable.
- The command
`mklvars intel64 mod`
sets environment variables for the Intel® 64 architecture and adds the path to the Fortran 95 modules for the LP64 interface to the INCLUDE environment variable.



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

See Also

[High-level Directory Structure](#)

[Interface Libraries and Modules](#)

[Fortran 95 Interfaces to LAPACK and BLAS](#)

[Setting the Number of Threads Using an OpenMP* Environment Variable](#)

Compiler Support

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

Although Compaq no longer supports the Compaq Visual Fortran* (CVF) compiler, Intel MKL still preserves the CVF interface in the IA-32 architecture implementation. You can use this interface with the Intel® Fortran Compiler. Intel MKL provides both `stdcall` (default CVF interface) and `cdecl` (default interface of the Microsoft Visual C* application) interfaces for the IA-32 architecture.

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

See Also

[Compiling an Application that Calls the Intel® Math Kernel Library and Uses the CVF Calling Conventions](#)

[Using the `cdecl` and `stdcall` Interfaces](#)

[Include Files](#)

Using Code Examples

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

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

The examples are grouped in subdirectories mainly by Intel MKL function domains and programming languages. For example, the `examples\spblas` subdirectory contains a makefile to build the Sparse BLAS examples and the `examples\vm1c` subdirectory contains the makefile to build the C VML examples. Source code for the examples is in the next-level `sources` subdirectory.

See Also

[High-level Directory Structure](#)

[Running an Intel MKL Example in the Visual Studio* 2008 IDE](#)

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

Target platform	<p>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 Setting Environment Variables for details).</p>
Mathematical problem	<p>Identify all Intel MKL function domains that you require:</p> <ul style="list-style-type: none"> • BLAS • Sparse BLAS

- LAPACK
- PBLAS
- ScaLAPACK
- Sparse Solver routines
- Vector Mathematical Library functions (VML)
- Vector Statistical Library functions
- Fourier Transform functions (FFT)
- Cluster FFT
- Trigonometric Transform routines
- Poisson, Laplace, and Helmholtz Solver routines
- Optimization (Trust-Region) Solver routines
- Data Fitting Functions
- GMP* arithmetic functions.
Deprecated and will be removed in a future release

Reason: The function domain you intend to use narrows the search in the *Reference Manual* for specific routines you need. Additionally, if you are using the Intel MKL cluster software, your link line is function-domain specific (see [Working with the Cluster Software](#)). Coding tips may also depend on the function domain (see [Tips and Techniques to Improve Performance](#)).

Programming language

Intel MKL provides support for both Fortran and C/C++ programming. Identify the language interfaces that your function domains support (see [Intel® Math Kernel Library Language Interfaces Support](#)).

Reason: Intel MKL provides language-specific include files for each function domain to simplify program development (see [Language Interfaces Support, by Function Domain](#)).

For a list of language-specific interface libraries and modules and an example how to generate them, see also [Using Language-Specific Interfaces with Intel® Math Kernel Library](#).

Range of integer data

If your system is based on the Intel 64 architecture, identify whether your application performs calculations with large data arrays (of more than $2^{31}-1$ elements).

Reason: To operate on large data arrays, you need to select the ILP64 interface, where integers are 64-bit; otherwise, use the default, LP64, interface, where integers are 32-bit (see [Using the ILP64 Interface vs. LP64 Interface](#)).

Threading model

Identify whether and how your application is threaded:

- Threaded with the Intel compiler
- Threaded with a third-party compiler
- Not threaded

Reason: The compiler you use to thread your application determines which threading library you should link with your application. For applications threaded with a third-party compiler you may need to use Intel MKL in the sequential mode (for more information, see [Sequential Mode of the Library](#) and [Linking with Threading Libraries](#)).

Number of threads

Determine the number of threads you want Intel MKL to use.

Reason: Intel MKL is based on the OpenMP* threading. By default, the OpenMP* software sets the number of threads that Intel MKL uses. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, see [Using Parallelism of the Intel® Math Kernel Library](#).

Linking model

Decide which linking model is appropriate for linking your application with Intel MKL libraries:

- Static

- Dynamic

Reason: The link libraries for static and dynamic linking are different. For the list of link libraries for static and dynamic models, linking examples, and other relevant topics, like how to save disk space by creating a custom dynamic library, see [Linking Your Application with the Intel® Math Kernel Library](#).

MPI used

Decide what MPI you will use with the Intel MKL cluster software. You are strongly encouraged to use Intel® MPI 3.2 or later.

MPI used

Reason: To link your application with ScaLAPACK and/or Cluster FFT, the libraries corresponding to your particular MPI should be listed on the link line (see [Working with the Cluster Software](#)).

Structure of the Intel® Math Kernel Library



Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Architecture Support

Intel® Math Kernel Library (Intel® MKL) for Windows* OS provides architecture-specific implementations for supported platforms. The following table lists the supported architectures and directories where each architecture-specific implementation is located.

Architecture	Location
IA-32 or compatible	<code><mkl directory>\lib\ia32</code> <code><Composer XE directory>\redist\ia32\mkl (DLLs)</code>
Intel® 64 or compatible	<code><mkl directory>\lib\intel64</code> <code><Composer XE directory>\redist\intel64\mkl (DLLs)</code>

See Also

[High-level Directory Structure](#)

[Detailed Structure of the IA-32 Architecture Directories](#)

[Detailed Structure of the Intel® 64 Architecture Directories](#)

High-level Directory Structure

Directory	Contents
<code><mkl directory></code>	Installation directory of the Intel® Math Kernel Library (Intel® MKL)
Subdirectories of <code><mkl directory></code>	
<code>bin</code>	Batch files to set environmental variables in the user shell
<code>bin\ia32</code>	Batch files for the IA-32 architecture
<code>bin\intel64</code>	Batch files for the Intel® 64 architecture
<code>benchmarks\linpack</code>	Shared-Memory (SMP) version of the LINPACK benchmark
<code>benchmarks\mp_linpack</code>	Message-passing interface (MPI) version of the LINPACK benchmark

Directory	Contents
lib\ia32	Static libraries and static interfaces to DLLs for the IA-32 architecture
lib\intel64	Static libraries and static interfaces to DLLs for the Intel® 64 architecture
examples	Examples directory. Each subdirectory has source and data files
include	INCLUDE files for the library routines, as well as for tests and examples
include\ia32	Fortran 95 .mod files for the IA-32 architecture and Intel Fortran compiler
include\intel64\lp64	Fortran 95 .mod files for the Intel® 64 architecture, Intel® Fortran compiler, and LP64 interface
include\intel64\ilp64	Fortran 95 .mod files for the Intel® 64 architecture, Intel Fortran compiler, and ILP64 interface
include\fftw	Header files for the FFTW2 and FFTW3 interfaces
interfaces\blas95	Fortran 95 interfaces to BLAS and a makefile to build the library
interfaces\fftw2x_cdft	MPI FFTW 2.x interfaces to Intel MKL Cluster FFTs
interfaces\fftw3x_cdft	MPI FFTW 3.x interfaces to Intel MKL Cluster FFTs
interfaces\fftw2xc	FFTW 2.x interfaces to the Intel MKL FFTs (C interface)
interfaces\fftw2xf	FFTW 2.x interfaces to the Intel MKL FFTs (Fortran interface)
interfaces\fftw3xc	FFTW 3.x interfaces to the Intel MKL FFTs (C interface)
interfaces\fftw3xf	FFTW 3.x interfaces to the Intel MKL FFTs (Fortran interface)
interfaces\lapack95	Fortran 95 interfaces to LAPACK and a makefile to build the library
tests	Source and data files for tests
tools	Command-line link tool and tools for creating custom dynamically linkable libraries
tools\builder	Tools for creating custom dynamically linkable libraries
Subdirectories of <Composer XE directory>	
redist\ia32\mkl	DLLs for applications running on processors with the IA-32 architecture
redist\intel64\mkl	DLLs for applications running on processors with Intel® 64 architecture
Documentation\en_US\MKL	Intel MKL documentation
Documentation\vshelp \1033\ intel.mkldocs	Help2-format files for integration of the Intel MKL documentation with the Microsoft Visual Studio* 2005/2008 IDE
Documentation\msvhelp \1033\mkl	Microsoft Help Viewer*-format files for integration of the Intel MKL documentation with the Microsoft Visual Studio* 2010 IDE

See Also
[Notational Conventions](#)

Layered Model Concept

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

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

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

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

- Compiler run-time libraries (RTL).

The following table provides more details of each layer.

Layer	Description
Interface Layer	<p>This layer matches compiled code of your application with the threading and/or computational parts of the library. This layer provides:</p> <ul style="list-style-type: none"> • cdecl and CVF default interfaces. • LP64 and ILP64 interfaces. • Compatibility with compilers that return function values differently. • A mapping between single-precision names and double-precision names for applications using Cray*-style naming (SP2DP interface). SP2DP interface supports Cray-style naming in applications targeted for the Intel 64 architecture and using the ILP64 interface. SP2DP interface provides a mapping between single-precision names (for both real and complex types) in the application and double-precision names in Intel MKL BLAS and LAPACK. Function names are mapped as shown in the following example for BLAS functions ?GEMM: <pre> SGEMM -> DGEMM DGEMM -> DGEMM CGEMM -> ZGEMM ZGEMM -> ZGEMM </pre> <p>Mind that no changes are made to double-precision names.</p>
Threading Layer	<p>This layer:</p> <ul style="list-style-type: none"> • Provides a way to link threaded Intel MKL with different threading compilers. • Enables you to link with a threaded or sequential mode of the library. <p>This layer is compiled for different environments (threaded or sequential) and compilers (from Intel, Microsoft, and so on).</p>
Computational Layer	<p>This layer is the heart of Intel MKL. It has only one library for each combination of architecture and supported OS. The Computational layer accommodates multiple architectures through identification of architecture features and chooses the appropriate binary code at run time.</p>
Compiler Run-time Libraries (RTL)	<p>To support threading with Intel compilers, Intel MKL uses RTLs of the Intel® C++ Composer XE or Intel® Visual Fortran Composer XE. To thread using third-party threading compilers, use libraries in the Threading layer or an appropriate compatibility library.</p>

See Also

[Using the ILP64 Interface vs. LP64 Interface](#)

Linking Your Application with the Intel® Math Kernel Library Linking with Threading Libraries

Contents of the Documentation Directories

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

File name	Comment
Files in <code><Composer XE directory>\Documentation</code>	
<code><locale>\clicense.rtf</code> or <code><locale>\flicense.rtf</code>	Common end user license for the Intel® C++ Composer XE or Intel® Visual Fortran Composer XE, respectively
<code>mklsupport.txt</code>	Information on package number for customer support reference
Contents of <code><Composer XE directory>\Documentation\<locale>\mkl</code>	
<code>redist.txt</code>	List of redistributable files
<code>mkl_documentation.htm</code>	Overview and links for the Intel MKL documentation
<code>mkl_manual\index.htm</code>	Intel MKL Reference Manual in an uncompressed HTML format
<code>mklman90.pdf</code> [†]	Intel MKL 9.0 Reference Manual in Japanese
<code>Release_Notes.htm</code>	Intel MKL Release Notes
<code>mkl_userguide\index.htm</code>	Intel MKL User's Guide in an uncompressed HTML format, this document
<code>mkl_link_line_advisor.htm</code>	Intel MKL Link-line Advisor

[†] Included only in the Japanese versions of the Intel® C++ Composer XE and Intel® Visual Fortran Composer XE.

Linking Your Application with the Intel® Math Kernel Library

4

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Linking Quick Start

Intel® Math Kernel Library (Intel® MKL) provides several options for quick linking of your application. The simplest options depend on your development environment:

Intel® Composer XE compiler

see [Using the /Qmkl Compiler Option](#).

Microsoft Visual Studio* Integrated Development Environment (IDE)

see [Automatically Linking a Project in the Visual Studio* IDE with Intel MKL](#).

Other options are independent of your development environment, but depend on the way you link:

Explicit dynamic linking

see [Using the Single Dynamic Library](#) for how to simplify your link line.

Explicitly listing libraries on your link line

see [Selecting Libraries to Link with](#) for a summary of the libraries.

Using an interactive interface

see [Using the Link-line Advisor](#) to determine libraries and options to specify on your link or compilation line.

Using an internally provided tool

see [Using the Command-line Link Tool](#) to determine libraries, options, and environment variables or even compile and build your application.

Using the /Qmkl Compiler Option

The Intel® Composer XE compiler supports the following variants of the /Qmkl compiler option:

/Qmkl or

to link with standard threaded Intel MKL.

/Qmkl:parallel

/Qmkl:sequential

to link with sequential version of Intel MKL.

/Qmkl:cluster

to link with Intel MKL cluster components (sequential) that use Intel MPI.

For more information on the /Qmkl compiler option, see the Intel Compiler User and Reference Guides.

For each variant of the /Qmkl option, the compiler links your application using the following conventions:

- cdecl for the IA-32 architecture
- LP64 for the Intel® 64 architecture

If you specify any variant of the /Qmkl compiler option, the compiler automatically includes the Intel MKL libraries. In cases not covered by the option, use the Link-line Advisor or see [Linking in Detail](#).

See Also

[Using the ILP64 Interface vs. LP64 Interface](#)

[Using the Link-line Advisor](#)

[Intel® Software Documentation Library](#)

Automatically Linking a Project in the Visual Studio* Integrated Development Environment with Intel® MKL

After a default installation of the Intel® Math Kernel Library (Intel® MKL), Intel® C++ Composer XE, or Intel® Visual Fortran Composer XE, you can easily configure your project to automatically link with Intel MKL.

Automatically Linking Your Microsoft Visual C/C++* Project with Intel® MKL

Configure your Microsoft Visual C/C++* project for automatic linking with Intel MKL as follows:

- For the Visual Studio* 2010 development system:
 1. Go to **Project>Properties>Configuration Properties>Intel Performance Libraries**.
 2. Change the **Use MKL** property setting by selecting **Parallel**, **Sequential**, or **Cluster** as appropriate.
- For the Visual Studio 2005/2008 development system:
 1. Go to **Project>Intel C++ Composer XE 2011>Select Build Components**.
 2. From the **Use MKL** drop-down menu, select **Parallel**, **Sequential**, or **Cluster** as appropriate.

Specific Intel MKL libraries that link with your application may depend on more project settings. For details, see the Intel® Composer XE documentation.

See Also

[Intel® Software Documentation Library](#)

Automatically Linking Your Intel® Visual Fortran Project with Intel® MKL

Configure your Intel® Visual Fortran project for automatic linking with Intel MKL as follows:

Go to **Project > Properties > Libraries > Use Intel Math Kernel Library** and select **Parallel**, **Sequential**, or **Cluster** as appropriate.

Specific Intel MKL libraries that link with your application may depend on more project settings. For details see the Intel® Visual Fortran Compiler XE User and Reference Guides.

See Also

[Intel® Software Documentation Library](#)

Using the Single Dynamic Library

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

To use SDL, place `mkl_rt.lib` on your link line. For example:

```
icl.exe application.c mkl_rt.lib
```

`mkl_rt.lib` is the import library for `mkl_rt.dll`.

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

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

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

Selecting Libraries to Link with

To link with Intel MKL:

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

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

	Interface layer	Threading layer	Computational layer	RTL
IA-32 architecture, static linking	mkl_intel_c.lib	mkl_intel_thread.lib	mkl_core.lib	libiomp5md.lib
IA-32 architecture, dynamic linking	mkl_intel_c_dll.lib	mkl_intel_thread_dll.lib	mkl_core_dll.lib	libiomp5md.lib
Intel® 64 architecture, static linking	mkl_intel_lp64.lib	mkl_intel_thread.lib	mkl_core.lib	libiomp5md.lib
Intel® 64 architecture, dynamic linking	mkl_intel_lp64_dll.lib	mkl_intel_thread_dll.lib	mkl_core_dll.lib	libiomp5md.lib

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

	SDL	RTL
IA-32 and Intel® 64 architectures	mkl_rt.lib	libiomp5md.lib [†]

[†] Linking with libiomp5md.lib is not required.

For exceptions and alternatives to the libraries listed above, see [Linking in Detail](#).

See Also

[Layered Model Concept](#)

[Using the Link-line Advisor](#)

[Using the /Qmkl Compiler Option](#)

[Working with the Intel® Math Kernel Library Cluster Software](#)

Using the Link-line Advisor

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

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

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

See Also

[Contents of the Documentation Directories](#)

Using the Command-line Link Tool

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

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

The tool `mkl_link_tool.exe` is installed in the `<mkl_directory>\tools` directory.

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

Linking Examples

See Also

[Using the Link-line Advisor](#)

[Examples for Linking with ScaLAPACK and Cluster FFT](#)

Linking on IA-32 Architecture Systems

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

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

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

```
ifort myprog.f mkl_intel_c.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_c_dll.lib mkl_intel_thread_dll.lib mkl_core_dll.lib  
libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_c.lib mkl_sequential.lib mkl_core.lib
```

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

```
ifort myprog.f mkl_intel_c_dll.lib mkl_sequential_dll.lib mkl_core_dll.lib
```

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

```
ifort myprog.f mkl_intel_s.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_s_dll.lib mkl_intel_thread_dll.lib mkl_core_dll.lib  
libiomp5md.lib
```

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

```
ifort myprog.f mkl_rt.lib
```

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

```
ifort myprog.f mkl_lapack95.lib mkl_intel_c.lib mkl_intel_thread.lib mkl_core.lib  
libiomp5md.lib
```

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

```
ifort myprog.f mkl_blas95.lib mkl_intel_c.lib mkl_intel_thread.lib mkl_core.lib  
libiomp5md.lib
```

See Also

[Fortran 95 Interfaces to LAPACK and BLAS](#)
[Examples for Linking a C Application](#)
[Examples for Linking a Fortran Application](#)
[Using the Single Dynamic Library](#)

Linking on Intel(R) 64 Architecture Systems

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

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

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

```
ifort myprog.f mkl_intel_lp64.lib mkl_intel_thread.lib mkl_core.lib
libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_lp64_dll.lib mkl_intel_thread_dll.lib mkl_core_dll.lib
libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_lp64.lib mkl_sequential.lib mkl_core.lib
```

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

```
ifort myprog.f mkl_intel_lp64_dll.lib mkl_sequential_dll.lib mkl_core_dll.lib
```

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

```
ifort myprog.f mkl_intel_ilp64.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib
```

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

```
ifort myprog.f mkl_intel_ilp64_dll.lib mkl_intel_thread_dll.lib mkl_core_dll.lib
libiomp5md.lib
```

- Dynamic linking of user code `myprog.f` and parallel or sequential Intel MKL supporting the LP64 or ILP64 interface (Call appropriate functions or set environment variables to choose threaded or sequential mode and to set the interface):

```
ifort myprog.f mkl_rt.lib
```

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

```
ifort myprog.f mkl_lapack95_lp64.lib mkl_intel_lp64.lib mkl_intel_thread.lib
mkl_core.lib libiomp5md.lib
```

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

```
ifort myprog.f mkl_blas95_lp64.lib mkl_intel_lp64.lib mkl_intel_thread.lib
mkl_core.lib libiomp5md.lib
```

See Also

[Fortran 95 Interfaces to LAPACK and BLAS](#)
[Examples for Linking a C Application](#)
[Examples for Linking a Fortran Application](#)
[Using the Single Dynamic Library](#)

Linking in Detail

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

Dynamically Selecting the Interface and Threading Layer

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

Setting the Interface Layer

Available interfaces depend on the architecture of your system.

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

Interface Layer	Value of <code>MKL_INTERFACE_LAYER</code>	Value of the Parameter of <code>mkl_set_interface_layer</code>
LP64	LP64	<code>MKL_INTERFACE_LP64</code>
ILP64	ILP64	<code>MKL_INTERFACE_ILP64</code>

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

By default the LP64 interface is used.

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

On systems based on the IA-32 architecture, the `cdecl` and `stdcall` interfaces are available. These interfaces have different function naming conventions, and SDL selects between `cdecl` and `stdcall` at link time according to the function names.

Setting the Threading Layer

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

Threading Layer	Value of <code>MKL_THREADING_LAYER</code>	Value of the Parameter of <code>mkl_set_threading_layer</code>
Intel threading	INTEL	<code>MKL_THREADING_INTEL</code>
Sequential mode of Intel MKL	SEQUENTIAL	<code>MKL_THREADING_SEQUENTIAL</code>
PGI threading	PGI	<code>MKL_THREADING_PGI</code>

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

By default Intel threading is used.

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

Replacing Error Handling and Progress Information Routines

You can replace the Intel MKL error handling routine `xerbla` or progress information routine `mkl_progress` with your own function. If you are using SDL, to replace `xerbla` or `mkl_progress`, call the `mkl_set_xerbla` and `mkl_set_progress` function, respectively. See the *Intel MKL Reference Manual* for details.



NOTE If you are using SDL, you cannot perform the replacement by linking the object file with your implementation of `xerbla` or `mkl_progress`.

See Also

[Using the Single Dynamic Library](#)

[Layered Model Concept](#)

[Using the `cdecl` and `stdcall` Interfaces](#)

[Directory Structure in Detail](#)

Linking with Interface Libraries

Using the `cdecl` and `stdcall` Interfaces

Intel MKL provides the following interfaces in its IA-32 architecture implementation:

- `stdcall`
Default Compaq Visual Fortran* (CVF) interface. Use it with the Intel® Fortran Compiler.
- `cdecl`
Default interface of the Microsoft Visual C/C++* application.

To use each of these interfaces, link with the appropriate library, as specified in the following table:

Interface	Library for Static Linking	Library for Dynamic Linking
<code>cdecl</code>	<code>mkl_intel_c.lib</code>	<code>mkl_intel_c_dll.lib</code>
<code>stdcall</code>	<code>mkl_intel_s.lib</code>	<code>mkl_intel_s_dll.lib</code>

To link with the `cdecl` or `stdcall` interface library, use appropriate calling syntax in C applications and appropriate compiler options for Fortran applications.

If you are using a C compiler, to link with the `cdecl` or `stdcall` interface library, call Intel MKL routines in your code as explained in the table below:

Interface Library	Calling Intel MKL Routines
<code>mkl_intel_s[_dll].lib</code>	<p>Call a routine with the following statement:</p> <pre>extern __stdcall name(<prototype variable1>, <prototype variable2>, ..);</pre> <p>where <code>stdcall</code> is actually the CVF compiler default compilation, which differs from the regular <code>stdcall</code> compilation in the way how strings are passed to the routine. Because the default CVF format is not identical with <code>stdcall</code>, you must specially handle strings in the calling sequence. See how to do it in sections on interfaces in the CVF documentation.</p>
<code>mkl_intel_c[_dll].lib</code>	<p>Use the following declaration:</p> <pre><type> name(<prototype variable1>, <prototype variable2>, ..);</pre>

If you are using a Fortran compiler, to link with the `cdecl` or `stdcall` interface library, provide compiler options as explained in the table below:

Interface Library	Compiler Options	Comment
CVF compiler		
<code>mkl_intel_s[_dll].lib</code>	Default	
<code>mkl_intel_c[_dll].lib</code>	<code>/iface=(cref, nomixed_str_len_arg)</code>	

Interface Library	Compiler Options	Comment
Intel® Fortran compiler		
mkl_intel_c[_dll].lib	Default	
mkl_intel_s[_dll].lib	/Gm or /iface:cvf	/Gm and /iface:cvf options enable compatibility of the CVF and Powerstation calling conventions

See Also

[Using the stdcall Calling Convention in C/C++](#)

[Compiling an Application that Calls the Intel® Math Kernel Library and Uses the CVF Calling Conventions](#)

Using the ILP64 Interface vs. LP64 Interface

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

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

- mkl_intel_lp64.lib or mkl_intel_ilp64.lib for static linking
- mkl_intel_lp64_dll.lib or mkl_intel_ilp64_dll.lib for dynamic linking

The ILP64 interface provides for the following:

- Support large data arrays (with more than $2^{31}-1$ elements)
- Enable compiling your Fortran code with the /4I8 compiler option

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

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

Compiling for LP64/ILP64

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

Fortran	
Compiling for ILP64	ifort /4I8 /I<mkl directory>\include ...
Compiling for LP64	ifort /I<mkl directory>\include ...
C or C++	
Compiling for ILP64	icl /DMKL_ILP64 /I<mkl directory>\include ...
Compiling for LP64	icl /I<mkl directory>\include ...



CAUTION Linking of an application compiled with the /4I8 or /DMKL_ILP64 option to the LP64 libraries may result in unpredictable consequences and erroneous output.

Coding for ILP64

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

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

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

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

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

- **VML:** The *mode* parameter of VML functions is 64-bit.
- **Random Number Generators (RNG):**
All discrete RNG except `viRngUniformBits64` are 32-bit.
The `viRngUniformBits64` generator function and `vslSkipAheadStream` service function are 64-bit.
- **Summary Statistics:** The *estimate* parameter of the `vslsSSCompute/vslsSSCompute` function is 64-bit.

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

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

Limitations

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

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



NOTE GMP Arithmetic Functions are deprecated and will be removed in a future release.

See Also

[High-level Directory Structure](#)

[Include Files](#)

[Language Interfaces Support, by Function Domain](#)

[Layered Model Concept](#)

Directory Structure in Detail

Linking with Fortran 95 Interface Libraries

The `mkl_blas95*.lib` and `mkl_lapack95*.lib` libraries contain Fortran 95 interfaces for BLAS and LAPACK, respectively, which are compiler-dependent. In the Intel MKL package, they are prebuilt for the Intel® Fortran compiler. If you are using a different compiler, build these libraries before using the interface.

See Also

[Fortran 95 Interfaces to LAPACK and BLAS](#)

[Compiler-dependent Functions and Fortran 90 Modules](#)

Linking with Threading Libraries

Sequential Mode of the Library

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

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

See Also

[Directory Structure in Detail](#)

[Using Parallelism of the Intel® Math Kernel Library](#)

[Avoiding Conflicts in the Execution Environment](#)

[Linking Examples](#)

Selecting the Threading Layer

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

Threading Layer

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

RTL

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

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

Compiler	Application Threaded?	Threading Layer	RTL Recommended	Comment
Intel	Does not matter	mkl_intel_thread.lib	libiomp5md.lib	
PGI	Yes	mkl_pgi_thread.lib or mkl_sequential.lib	PGI* supplied	Use of mkl_sequential.lib removes threading from Intel MKL calls.
PGI	No	mkl_intel_thread.lib	libiomp5md.lib	
PGI	No	mkl_pgi_thread.lib	PGI* supplied	
PGI	No	mkl_sequential.lib	None	
Microsoft	Yes	mkl_intel_thread.lib	libiomp5md.lib	For the OpenMP* library of the Microsoft Visual Studio* IDE version 2005 or later.
Microsoft	Yes	mkl_sequential.lib	None	For Win32 threading.
Microsoft	No	mkl_intel_thread.lib	libiomp5md.lib	
other	Yes	mkl_sequential.lib	None	
other	No	mkl_intel_thread.lib	libiomp5md.lib	

Linking with Computational Libraries

If you are not using the Intel MKL cluster software, you need to link your application with only one computational library, depending on the linking method:

Static Linking	Dynamic Linking
mkl_core.lib	mkl_core_dll.lib

Computational Libraries for Applications that Use the Intel MKL Cluster Software

ScaLAPACK and Cluster Fourier Transform Functions (Cluster FFT) require more computational libraries, which may depend on your architecture.

The following table lists computational libraries for IA-32 architecture applications that use ScaLAPACK or Cluster FFT.

Computational Libraries for IA-32 Architecture

Function domain	Static Linking	Dynamic Linking
ScaLAPACK [†]	mkl_scalapack_core.lib	mkl_scalapack_core_dll.lib

Function domain	Static Linking	Dynamic Linking
	<code>mkl_core.lib</code>	<code>mkl_core_dll.lib</code>
Cluster Fourier Transform Functions [†]	<code>mkl_cdft_core.lib</code> <code>mkl_core.lib</code>	<code>mkl_cdft_core_dll.lib</code> <code>mkl_core_dll.lib</code>

[†] Also add the library with BLACS routines corresponding to the MPI used.

The following table lists computational libraries for Intel® 64 architecture applications that use ScaLAPACK or Cluster FFT.

Computational Libraries for the Intel® 64 Architecture

Function domain	Static Linking	Dynamic Linking
ScaLAPACK, LP64 interface ¹	<code>mkl_scalapack_lp64.lib</code> <code>mkl_core.lib</code>	<code>mkl_scalapack_lp64_dll.lib</code> <code>mkl_core_dll.lib</code>
ScaLAPACK, ILP64 interface ¹	<code>mkl_scalapack_ilp64.lib</code> <code>mkl_core.lib</code>	<code>mkl_scalapack_ilp64_dll.lib</code> <code>mkl_core_dll.lib</code>
Cluster Fourier Transform Functions ¹	<code>mkl_cdft_core.lib</code> <code>mkl_core.lib</code>	<code>mkl_cdft_core_dll.lib</code> <code>mkl_core_dll.lib</code>

[†] Also add the library with BLACS routines corresponding to the MPI used.

See Also

[Linking with ScaLAPACK and Cluster FFTs](#)

[Using the Link-line Advisor](#)

[Using the ILP64 Interface vs. LP64 Interface](#)

Linking with Compiler Run-time Libraries

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

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

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

Sometimes you may improve performance of your application with threaded Intel MKL by using the `/MT` compiler option. The compiler driver will pass the option to the linker and the latter will load multi-thread (MT) static run-time libraries.

However, to link a Vector Math Library application that uses the `errno` variable for error reporting, compile and link your code using the option that depends on the linking model:

- `/MT` for linking with static Intel MKL libraries
- `/MD` for linking with dynamic Intel MKL libraries

See Also

[Setting Environment Variables](#)

[Layered Model Concept](#)

Linking with System Libraries

If your system is based on the Intel® 64 architecture, be aware that Microsoft SDK builds 1289 or higher provide the `bufferoverflowu.lib` library to resolve the `__security_cookie` external references. Makefiles for examples and tests include this library by using the `buf_lib=bufferoverflowu.lib` macro. If you are using older SDKs, leave this macro empty on your command line as follows: `buf_lib=` .

See Also

[Linking Examples](#)

Building Custom Dynamic-link Libraries

Custom dynamic-link libraries (DLL) reduce the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

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

Using the Custom Dynamic-link Library Builder in the Command-line Mode

To build a custom DLL, use the following command:


```
nmake target [<options>]
```

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

Value	Comment
<code>libia32</code>	The builder uses static Intel MKL interface, threading, and core libraries to build a custom DLL for the IA-32 architecture.
<code>libintel64</code>	The builder uses static Intel MKL interface, threading, and core libraries to build a custom DLL for the Intel® 64 architecture.
<code>dllia32</code>	The builder uses the single dynamic library <code>libmkl_rt.dll</code> to build a custom DLL for the IA-32 architecture.
<code>dllintel64</code>	The builder uses the single dynamic library <code>libmkl_rt.dll</code> to build a custom DLL for the Intel® 64 architecture.
<code>help</code>	The command prints Help on the custom DLL builder

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

Parameter [Values]	Description
<code>interface</code>	Defines which programming interface to use. Possible values: <ul style="list-style-type: none"> For the IA-32 architecture, <code>{cdecl stdcall}</code>. The default value is <code>cdecl</code>. For the Intel 64 architecture, <code>{lp64 ilp64}</code>. The default value is <code>lp64</code>.
<code>threading = {parallel sequential}</code>	Defines whether to use the Intel MKL in the threaded or sequential mode. The default value is <code>parallel</code> .

Parameter [Values]	Description
<code>export = <file name></code>	Specifies the full name of the file that contains the list of entry-point functions to be included in the DLL. The default name is <code>user_example_list</code> (no extension).
<code>name = <dll name></code>	Specifies the name of the dll and interface library to be created. By default, the names of the created libraries are <code>mkl_custom.dll</code> and <code>mkl_custom.lib</code> .
<code>xerbla = <error handler></code>	Specifies the name of the object file <code><user_xerbla>.obj</code> that contains the user's error handler. The makefile adds this error handler to the library for use instead of the default Intel MKL error handler <code>xerbla</code> . If you omit this parameter, the native Intel MKL <code>xerbla</code> is used. See the description of the <code>xerbla</code> function in the Intel MKL Reference Manual on how to develop your own error handler. For the IA-32 architecture, the object file should be in the interface defined by the interface macro (<code>cdecl</code> or <code>stdcall</code>).
<code>MKLROOT = <mkl directory></code>	Specifies the location of Intel MKL libraries used to build the custom DLL. By default, the builder uses the Intel MKL installation directory.
<code>buf_lib</code>	<p>Manages resolution of the <code>__security_cookie</code> external references in the custom DLL on systems based on the Intel® 64 architecture.</p> <p>By default, the makefile uses the <code>bufferoverflowu.lib</code> library of Microsoft SDK builds 1289 or higher. This library resolves the <code>__security_cookie</code> external references.</p> <p>To avoid using this library, set the empty value of this parameter. Therefore, if you are using an older SDK, set <code>buf_lib= .</code></p>
<div style="border: 1px solid black; padding: 5px;">  CAUTION Use the <code>buf_lib</code> parameter only with the empty value. Incorrect value of the parameter causes builder errors. </div>	
<code>crt = <c run-time library></code>	Specifies the name of the Microsoft C run-time library to be used to build the custom DLL. By default, the builder uses <code>msvcrt.lib</code> .
<code>manifest = {yes no embed}</code>	<p>Manages the creation of a Microsoft manifest for the custom DLL:</p> <ul style="list-style-type: none"> • If <code>manifest=yes</code>, the manifest file with the name defined by the name parameter above and the <code>manifest</code> extension will be created. • If <code>manifest=no</code>, the manifest file will not be created. • If <code>manifest=embed</code>, the manifest will be embedded into the DLL. <p>By default, the builder does not use the <code>manifest</code> parameter.</p>

All the above parameters are optional.

In the simplest case, the command line is `nmake ia32`, and the missing options have default values. This command creates the `mkl_custom.dll` and `mkl_custom.lib` libraries with the `cdecl` interface for processors using the IA-32 architecture. The command takes the list of functions from the `functions_list` file and uses the native Intel MKL error handler `xerbla`.

An example of a more complex case follows:

```
nmake ia32 interface=stdcall export=my_func_list.txt name=mkl_small
xerbla=my_xerbla.obj
```

In this case, the command creates the `mkl_small.dll` and `mkl_small.lib` libraries with the `stdcall` interface 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.obj`.

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

See Also

[Linking with System Libraries](#)

Composing a List of Functions

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

1. Link your application with installed Intel MKL libraries to make sure the application builds.
2. Remove all Intel MKL libraries from the link line and start linking.
Unresolved symbols indicate Intel MKL functions that your application uses.
3. Include these functions in the list.



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

See Also

[Specifying Function Names](#)

Specifying Function Names

In the file with the list of functions for your custom DLL, adjust function names to the required interface. For example, you can list the cdecl entry points as follows:

```
DGEMM
DTRSM
DDOT
DGETRF
DGETRS
cblas_dgemm
cblas_ddot
```

You can list the stdcall entry points as follows:

```
_DGEMM@60
_DDOT@20
_DGETRF@24
```

For more examples, see domain-specific lists of function names in the `<mkl_directory>\tools\builder` folder. This folder contains lists of function names for both cdecl or stdcall interfaces.



NOTE The lists of function names are provided in the `<mkl_directory>\tools\builder` folder merely as examples. See [Composing a List of Functions](#) for how to compose lists of functions for your custom DLL.



TIP Names of Fortran-style routines (BLAS, LAPACK, etc.) can be both upper-case or lower-case, with or without the trailing underscore. For example, these names are equivalent:

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

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

1. In the `mkl_service.h` include file, look up a `#define` directive for your function.
2. Take the function name from the replacement part of that directive.

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

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

For the names of the Fortran support functions, see the [tip](#).

Building a Custom Dynamic-link Library in the Visual Studio* Development System

You can build a custom dynamic-link library (DLL) in the Microsoft Visual Studio* Development System (VS*) . To do this, use projects available in the `tools\builder\MSVS_Projects` subdirectory of the Intel MKL directory. The directory contains the `VS2005`, `VS2008`, and `VS2010` subdirectories with projects for the respective versions of the Visual Studio Development System. For each version of VS two solutions are available:

- `libia32.sln` builds a custom DLL for the IA-32 architecture.
- `libintel64.sln` builds a custom DLL for the Intel® 64 architecture.

The builder uses the following default settings for the custom DLL:

Interface:	<code>cdecl</code> for the IA-32 architecture and <code>LP64</code> for the Intel 64 architecture
Error handler:	Native Intel MKL <code>xerbla</code>
Create Microsoft manifest:	<code>yes</code>
List of functions:	in the project's source file <code>examples.def</code>

To build a custom DLL:

1. Set the `MKLROOT` environment variable with the installation directory of the Intel MKL version you are going to use.
2. Open the `libia32.sln` or `libintel64.sln` solution depending on the architecture of your system.

The solution includes the following projects:

- `i_malloc_dll`
- `vml_dll_core`
- `cdecl_parallel` (in `libia32.sln`) or `lp64_parallel` (in `libintel64.sln`)
- `cdecl_sequential` (in `libia32.sln`) or `lp64_sequential` (in `libintel64.sln`)

3. [Optional] To change any of the default settings, select the project depending on whether the DLL will use Intel MKL functions in the sequential or multi-threaded mode:

- In the `libia32` solution, select the `cdecl_sequential` or `cdecl_parallel` project.
- In the `libintel64` solution, select the `lp64_sequential` or `lp64_parallel` project.

4. [Optional] To build the DLL that uses the `stdcall` interface for the IA-32 architecture or the `ILP64` interface for the Intel 64 architecture:

Select Project>Properties>Configuration Properties>Linker>Input>Additional Dependencies.

In the `libia32` solution, change `mkl_intel_c.lib` to `mkl_intel_s.lib`.

In the `libintel64` solution, change `mkl_intel_lp64.lib` to `mkl_intel_ilp64.lib`.

5. [Optional] To include your own error handler in the DLL:

Select Project>Properties>Configuration Properties>Linker>Input.

Add `<user_xerbla>.obj`

6. [Optional] To turn off creation of the manifest:

Select Project>Properties>Configuration Properties>Linker>Manifest File>Generate Manifest.

Select: no.

7. [Optional] To change the list of functions to be included in the DLL:

Select Source Files.

Edit the `examples.def` file. Refer to [Specifying Function Names](#) for how to specify entry points.

8. To build the library:

- In VS2005 - VS2008, select **Build>Project Only>Link Only** and link projects in this order: `i_malloc_dll`, `vml_dll_core`, `cdecl_sequential/lp64_sequential` or `cdecl_parallel/lp64_parallel`.
- In VS2010, select **Build>Build Solution**.

See Also

[Using the Custom Dynamic-link Library Builder in the Command-line Mode](#)

Distributing Your Custom Dynamic-link Library

To enable use of your custom DLL in a threaded mode, distribute `libiomp5md.dll` along with the custom DLL.

Managing Performance and Memory

5

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Using Parallelism of the Intel® Math Kernel Library

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

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

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

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

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

See Also

[Managing Multi-core Performance](#)

Threaded Functions and Problems

The following Intel MKL function domains are threaded:

- Direct sparse solver.
- LAPACK.
For the list of threaded routines, see [Threaded LAPACK Routines](#).
- Level1 and Level2 BLAS.
For the list of threaded routines, see [Threaded BLAS Level1 and Level2 Routines](#).
- All Level 3 BLAS and all Sparse BLAS routines except Level 2 Sparse Triangular solvers.
- All mathematical VML functions.
- FFT.
For the list of FFT transforms that can be threaded, see [Threaded FFT Problems](#).

Threaded LAPACK Routines

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

The following LAPACK routines are threaded:

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

A number of other LAPACK routines, which are based on threaded LAPACK or BLAS routines, make effective use of parallelism:

?gesv, ?posv, ?gels, ?gesvd, ?syev, ?heev, cgegs/zgegs, cgegv/zgegv, cgges/zgges, cggesx/zggesx, cggev/zggev, cggevx/zggev, and so on.

Threaded BLAS Level1 and Level2 Routines

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

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

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

Threaded FFT Problems

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

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

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

One-dimensional (1D) transforms

1D transforms are threaded in many cases.

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

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

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

Prime-size complex-to-complex 1D transforms are not threaded.

Multidimensional transforms

All multidimensional transforms on large-volume data are threaded.

Avoiding Conflicts in the Execution Environment

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

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

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

Threading model	Discussion
You thread the program using OS threads (Win32* threads on Windows* OS).	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 <code>OMP_NUM_THREADS</code> environment variable affects both the compiler's threading library and <code>libiomp5</code> . 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: <code>mkl_sequential.lib</code> or <code>mkl_sequential.dll</code> (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). Section Intel(R) Optimized MP LINPACK Benchmark for Clusters discusses another solution for a Hybrid (OpenMP* + MPI) mode.

See Also

[Using Additional Threading Control](#)

Linking with Compiler Run-time Libraries

Techniques to Set the Number of Threads

Use one of the following 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 controls because they are inspected first.
- 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`. See [Using Additional Threading Control](#) for more details.
- You cannot change run-time behavior in the course of the run using the environment variables because they are read only once at the first call to Intel MKL.

Setting the Number of Threads Using an OpenMP* Environment Variable

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

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

Some shells require the variable and its value to be exported:

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

You can alternatively assign value to the environment variable using Microsoft Windows* OS Control Panel.

Note that you will not benefit from setting this variable on Microsoft Windows* 98 or Windows* ME because multiprocessing is not supported.

See Also

[Using Additional Threading Control](#)

Changing the Number of Threads at Run Time

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

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

```
// ***** C language *****  
#include "omp.h"  
#include "mkl.h"  
#include <stdio.h>
```

```

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

```

```

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

ALPHA = 1.1
BETA = -1.2
DO I=1,N
DO J=1,N
A(I,J) = I+J

```

```

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

```

Using Additional Threading Control

Intel MKL-specific Environment Variables for Threading Control

Intel MKL provides optional threading controls, that is, the environment variables and support functions that are independent of OpenMP. They behave similar to their OpenMP equivalents, but take precedence over them in the meaning that the MKL-specific threading controls are inspected first. By using these controls along with OpenMP variables, you can thread the part of the application that does not call Intel MKL and the library independently 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 Sometimes Intel MKL does not have a choice on the number of threads for certain reasons, such as system resources.

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

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

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

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



NOTE The functions take precedence over the respective environment variables.

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

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

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

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

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

MKL_DYNAMIC

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

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

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

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

- If the requested number of threads exceeds the number of physical cores (perhaps because of using the Intel® Hyper-Threading Technology), and `MKL_DYNAMIC` is not changed from its default value of `TRUE`, Intel MKL will scale down the number of threads to the number of physical cores.
- If you are able to detect the presence of MPI, but cannot determine if it has been called in a thread-safe mode (it is impossible to detect this with MPICH 1.2.x, for instance), and `MKL_DYNAMIC` has not been changed from its default value of `TRUE`, Intel MKL will run one thread.

When `MKL_DYNAMIC` is `FALSE`, Intel MKL tries not to deviate from the number of threads the user requested. However, setting `MKL_DYNAMIC=FALSE` does not ensure that Intel MKL will use the number of threads that you request. The library may have no choice on this number for such reasons as system resources.

Additionally, the library may examine the problem and use a different number of threads than the value suggested. For example, if you attempt to do a size one matrix-matrix multiply across eight threads, the library may instead choose to use only one thread because it is impractical to use eight threads in this event.

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

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

MKL_DOMAIN_NUM_THREADS

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

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

```

<MKL-env-string> ::= <MKL-domain-env-string> { <delimiter><MKL-domain-env-string> }
<delimiter> ::= [ <space-symbol>* ] ( <space-symbol> | <comma-symbol> | <semicolon-symbol> | <colon-symbol> ) [ <space-symbol>* ]
<MKL-domain-env-string> ::= <MKL-domain-env-name><uses><number-of-threads>
<MKL-domain-env-name> ::= MKL_DOMAIN_ALL | MKL_DOMAIN_BLAS | MKL_DOMAIN_FFT |
MKL_DOMAIN_VML | MKL_DOMAIN_PARDISO
<uses> ::= [ <space-symbol>* ] ( <space-symbol> | <equality-sign> | <comma-symbol> )
[ <space-symbol>* ]
<number-of-threads> ::= <positive-number>
<positive-number> ::= <decimal-positive-number> | <octal-number> | <hexadecimal-number>
    
```

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

<code>MKL_DOMAIN_ALL</code>	All function domains
<code>MKL_DOMAIN_BLAS</code>	BLAS Routines
<code>MKL_DOMAIN_FFT</code>	non-cluster Fourier Transform Functions
<code>MKL_DOMAIN_VML</code>	Vector Mathematical Functions
<code>MKL_DOMAIN_PARDISO</code>	PARDISO

For example,

```

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

The global variables `MKL_DOMAIN_ALL`, `MKL_DOMAIN_BLAS`, `MKL_DOMAIN_FFT`, `MKL_DOMAIN_VML`, and `MKL_DOMAIN_PARDISO`, as well as the interface for the Intel MKL threading control functions, can be found in the `mk1.h` header file.

The table below illustrates how values of `MKL_DOMAIN_NUM_THREADS` are interpreted.

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

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

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

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

Setting the Environment Variables for Threading Control

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

```
set <VARIABLE NAME>=<value>
```

For example,

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

Some shells require the variable and its value to be exported:

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

For example:

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

You can alternatively assign values to the environment variables using Microsoft Windows* OS Control Panel.

Tips and Techniques to Improve Performance

Coding Techniques

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

- Align arrays on 32-byte boundaries. See [Example of Data Alignment](#) for how to do it.
- Make sure leading dimension values ($n \times \text{element_size}$) of two-dimensional arrays are divisible by 32, where `element_size` is the size of an array element in bytes.
- For two-dimensional arrays, avoid leading dimension values divisible by 2048 bytes. For example, for a double-precision array, with `element_size = 8`, avoid leading dimensions 256, 512, 768, 1024, ... (elements).

LAPACK Packed Routines

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

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

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

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

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

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

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

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 ($n \times \text{element_size}$), of two-dimensional arrays should be divisible by cache line size, which equals:

- 32 bytes for the Intel® Pentium® III processors
- 64 bytes for the Intel® Pentium® 4 processors and processors using Intel® 64 architecture

Hardware Configuration Tips

Dual-Core Intel® Xeon® processor 5100 series systems

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

Intel® Hyper-Threading Technology

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

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

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

See Also

[Using Parallelism of the Intel® Math Kernel Library](#)

Managing Multi-core Performance

You can obtain best performance on systems with multi-core processors by requiring that threads do not migrate from core to core. To do this, bind threads to the CPU cores by setting an affinity mask to threads. Use one of the following options:

- OpenMP facilities (recommended, if available), for example, the `KMP_AFFINITY` environment variable using the Intel OpenMP library
- A system function, as explained below

Consider the following performance issue:

- The system has two sockets with two cores each, for a total of four cores (CPUs)
- Performance of the four -thread parallel application using the Intel MKL LAPACK is unstable

The following code example shows how to resolve this issue by setting an affinity mask by operating system means using the Intel compiler. The code calls the system function `SetThreadAffinityMask` to bind the threads to appropriate cores, thus preventing migration of the threads. Then the Intel MKL LAPACK routine is called:

```
// Set affinity mask
#include <windows.h>
#include <omp.h>
int main(void) {
#pragma omp parallel default(shared)
{
int tid = omp_get_thread_num();
// 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
DWORD_PTR mask = (1 << (tid == 0 ? 0 : 2));
SetThreadAffinityMask( GetCurrentThread(), mask );
}
// Call Intel MKL LAPACK routine
return 0;
}
```

Compile the application with the Intel compiler using the following command:

```
icl /Qopenmp test_application.c
```

where `test_application.c` is the filename for the application.

Build the application. Run it in four threads, for example, by using the environment variable to set the number of threads:

```
set OMP_NUM_THREADS=4
test_application.exe
```

See Windows API documentation at msdn.microsoft.com/ for the restrictions on the usage of Windows API routines and particulars of the `SetThreadAffinityMask` function used in the above example.

See also a similar example at en.wikipedia.org/wiki/Affinity_mask .

Operating on Denormals

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

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

FFT Optimized Radices

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

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

Using Memory Management

Intel MKL Memory Management Software

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

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

Redefining Memory Functions

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

Memory Renaming

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

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

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

How to Redefine Memory Functions

To redefine memory functions, use the following procedure:

If you are using the statically linked Intel MKL,

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

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

If you are using the dynamically linked Intel MKL,

1. Include the `i_malloc.h` header file in your code.
2. Redefine values of pointers `i_malloc_dll`, `i_free_dll`, `i_calloc_dll`, and `i_realloc_dll` prior to the first call to MKL functions, as shown in the following example:

```
#include "i_malloc.h"
. . .
i_malloc_dll = my_malloc;
i_calloc_dll = my_calloc;
i_realloc_dll = my_realloc;
i_free_dll   = my_free;
. . .
// Now you may call Intel MKL functions
```


Language-specific Usage Options

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

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Using Language-Specific Interfaces with Intel® Math Kernel Library

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

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

Interface Libraries and Modules

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

File name	Contains
Libraries, in Intel MKL architecture-specific directories	
<code>mkl_blas95.lib¹</code>	Fortran 95 wrappers for BLAS (BLAS95) for IA-32 architecture.
<code>mkl_blas95_ilp64.lib¹</code>	Fortran 95 wrappers for BLAS (BLAS95) supporting LP64 interface.
<code>mkl_blas95_lp64.lib¹</code>	Fortran 95 wrappers for BLAS (BLAS95) supporting ILP64 interface.
<code>mkl_lapack95.lib¹</code>	Fortran 95 wrappers for LAPACK (LAPACK95) for IA-32 architecture.
<code>mkl_lapack95_lp64.lib¹</code>	Fortran 95 wrappers for LAPACK (LAPACK95) supporting LP64 interface.
<code>mkl_lapack95_ilp64.lib¹</code>	Fortran 95 wrappers for LAPACK (LAPACK95) supporting ILP64 interface.

File name	Contains
<code>fftw2xc_intel.lib</code> ¹	Interfaces for FFTW version 2.x (C interface for Intel compilers) to call Intel MKL FFTs.
<code>fftw2xc_ms.lib</code>	Contains interfaces for FFTW version 2.x (C interface for Microsoft compilers) to call Intel MKL FFTs.
<code>fftw2xf_intel.lib</code>	Interfaces for FFTW version 2.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
<code>fftw3xc_intel.lib</code> ²	Interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs.
<code>fftw3xc_ms.lib</code>	Interfaces for FFTW version 3.x (C interface for Microsoft compilers) to call Intel MKL FFTs.
<code>fftw3xf_intel.lib</code> ²	Interfaces for FFTW version 3.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
<code>fftw2x_cdft_SINGLE.lib</code>	Single-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
<code>fftw2x_cdft_DOUBLE.lib</code>	Double-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
<code>fftw3x_cdft.lib</code>	Interfaces for MPI FFTW version 3.x (C interface) to call Intel MKL cluster FFTs.
<code>fftw3x_cdft_ilp64.lib</code>	Interfaces for MPI FFTW version 3.x (C interface) to call Intel MKL cluster FFTs supporting the ILP64 interface.

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>mkl95_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>mkl95_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>mkl95_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.
<code>mkl_service.mod</code> ¹	Fortran 95 interface module for Intel MKL support functions.

¹ Prebuilt for the Intel® Fortran compiler

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

See Also

[Fortran 95 Interfaces to LAPACK and BLAS](#)

Fortran 95 Interfaces to LAPACK and BLAS

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

1. Go to the respective directory `<mk1 directory>\interfaces\blas95` or `<mk1 directory>\interfaces\lapack95`
2. Type one of the following commands depending on your architecture:
 - For the IA-32 architecture,


```
nmake libia32 install_dir=<user dir>
```
 - For the Intel® 64 architecture,


```
nmake libintel64 [interface=lp64|ilp64] install_dir=<user dir>
```



Important The parameter `install_dir` is required.

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

By default, the `ifort` compiler is assumed. You may change the compiler with an additional parameter of `nmake`:

```
FC=<compiler>.
```

For example, the command

```
nmake libintel64 FC=f95 install_dir=<userf95 dir> interface=lp64
```

builds the required library and `.mod` files and installs them in subdirectories of `<userf95 dir>`.

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

- For the IA-32 architecture,


```
nmake cleania32 install_dir=<user dir>
```
- For the Intel® 64 architecture,


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


```
nmake clean install_dir=<user dir>
```



CAUTION Even if you have administrative rights, avoid setting `install_dir=..\..` or `install_dir=<mk1 directory>` in a build or clean command above because these settings replace or delete the Intel MKL prebuilt Fortran 95 library and modules.

Compiler-dependent Functions and Fortran 90 Modules

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

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

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

Using the stdcall Calling Convention in C/C++

Intel MKL supports stdcall calling convention for the following function domains:

- BLAS Routines
- Sparse BLAS Routines
- LAPACK Routines
- Vector Mathematical Functions
- Vector Statistical Functions
- PARDISO
- Direct Sparse Solvers
- RCI Iterative Solvers
- Support Functions

To use the stdcall calling convention in C/C++, follow the guidelines below:

- In your function calls, pass lengths of character strings to the functions. For example, compare the following calls to `dgemm`:

```
cdecl: dgemm("N", "N", &n, &m, &k, &alpha, b, &ldb, a, &lda, &beta, c, &ldc);
```

```
stdcall: dgemm("N", 1, "N", 1, &n, &m, &k, &alpha, b, &ldb, a, &lda, &beta, c, &ldc);
```

- Define the `MKL_STDCALL` macro using either of the following techniques:

- Define the macro in your source code before including Intel MKL header files:

```
...
#define MKL_STDCALL
#include "mkl.h"
...
```

- Pass the macro to the compiler. For example:

```
icl -DMKL_STDCALL foo.c
```

- Link your application with the following library:

- `mkl_intel_s.lib` for static linking
- `mkl_intel_s_dll.lib` for dynamic linking



See Also

[Using the cdecl and stdcall Interfaces](#)

[Compiling an Application that Calls the Intel® Math Kernel Library and Uses the CVF Calling Conventions](#)

[Include Files](#)

Compiling an Application that Calls the Intel® Math Kernel Library and Uses the CVF Calling Conventions

The IA-32 architecture implementation of Intel MKL supports the Compaq Visual Fortran* (CVF) calling convention by providing the stdcall interface.

Although the Intel MKL does not provide the CVF interface in its Intel® 64 architecture implementation, you can use the Intel® Visual Fortran Compiler to compile your Intel® 64 architecture application that calls Intel MKL and uses the CVF calling convention. To do this:

- Provide the following compiler options to enable compatibility with the CVF calling convention:
`/Gm` or `/iface:cvf`
- Additionally provide the following options to enable calling Intel MKL from your application:
`/iface:nomixed_str_len_arg`

See Also

[Using the cdecl and stdcall Interfaces Compiler Support](#)

Mixed-language Programming with the Intel Math Kernel Library

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

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

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



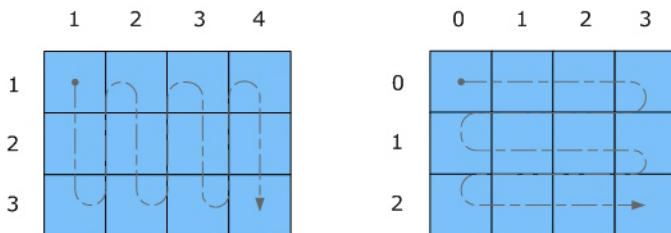
CAUTION Avoid calling BLAS 95/LAPACK 95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

LAPACK and BLAS

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

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

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



A: Column-major order (Fortran-style)

B: Row-major order (C-style)

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:

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

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

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

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

See [Example "Calling a Complex BLAS Level 1 Function from C++"](#) on how to call BLAS routines from C.

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

CBLAS

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

CBLAS is a C-style interface to the BLAS routines. You can call CBLAS routines using regular C-style calls. Use the `mk1.h` header file with the CBLAS interface. The header file specifies enumerated values and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. [Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"](#) illustrates the use of the CBLAS interface.

C Interface to LAPACK

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

The C interface to LAPACK is a C-style interface to the LAPACK routines. This interface supports matrices in row-major and column-major order, which you can define in the first function argument `matrix_order`. Use the `mk1_lapacke.h` header file with the C interface to LAPACK. The header file specifies constants and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. You can find examples of the C interface to LAPACK in the `examples\lapacke` subdirectory in the Intel MKL installation directory.

Using Complex Types in C/C++

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

Intel MKL provides complex types `MKL_Complex8` and `MKL_Complex16`, which are structures equivalent to the Fortran complex types `COMPLEX(4)` and `COMPLEX(8)`, respectively. The `MKL_Complex8` and `MKL_Complex16` types are defined in the `mk1_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 `mk1_types.h` header file. The only requirement is that the types must be compatible with the Fortran complex layout, that is, the complex type must be a pair of real numbers for the values of real and imaginary parts.

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

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

and

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

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

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

See Also

[Intel® Software Documentation Library](#)

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

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

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

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

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

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



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

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

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



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

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

- [Example "Calling a Complex BLAS Level 1 Function from C"](#)
- [Example "Calling a Complex BLAS Level 1 Function from C++"](#)
- [Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"](#)

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

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

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

```
#include "mkl.h"
#define N 5
int main()
{
    int n = N, inca = 1, incb = 1, i;
    MKL_Complex16 a[N], b[N], c;
    for( i = 0; i < n; i++ ){
```

```
a[i].real = (double)i; a[i].imag = (double)i * 2.0;
b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.real, c.imag );
return 0;
}
```

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

Below is the C++ implementation:

```
#include <complex>
#include <iostream>
#define MKL Complex16 std::complex<double>
#include "mkl.h"

#define N 5

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

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

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

This example uses CBLAS:

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

Support for Boost uBLAS Matrix-matrix Multiplication

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

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

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

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

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

- Include the header file `mk1_boost_ublas_matrix_prod.hpp` in your code (from the Intel MKL include directory)
- Add appropriate Intel MKL libraries to the link line.

The list of expressions that are substituted follows:

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

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

A code example provided in the `<mk1_directory>\examples\ublas\source\sylvester.cpp` file illustrates usage of the Intel MKL uBLAS header file for solving a special case of the Sylvester equation.

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

```
nmake libia32 boost_root = <your_path>\boost_1_37_0
```

Intel MKL ublas examples on default Boost uBLAS configuration support only:

- Microsoft Visual C++* Compiler versions 2005 and higher
- Intel C++ Compiler versions 11.1 and higher with Microsoft Visual Studio IDE versions 2005 and higher

See Also

Using Code Examples

Invoking Intel MKL Functions from Java* Applications

Intel MKL Java* Examples

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

`<mkl_directory>\examples\java.`

The examples are provided for the following MKL functions:

- `?gemm`, `?gemv`, and `?dot` families from CBLAS
- The complete set of 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:

`<mkl_directory>\examples\java\examples.`

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

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

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

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

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

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

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

- `docs`
- `include`
- `classes`
- `bin`
- `_results`

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

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

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

- `com.intel.mkl.VSL`

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

```
<mkl directory>\examples\java\docs\index.html.
```

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

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

```
<mkl directory>\examples\java\wrappers.
```

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

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

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

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

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

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

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

¹ IBM Engineering Scientific Subroutine Library (ESSL*).

See Also

[Running the Java* Examples](#)

Running the Java* Examples

The Java examples support all the C and C++ compilers that Intel MKL does. The makefile intended to run the examples also needs the `nmake` utility, which is typically provided with the C/C++ compiler package.

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

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

- J2SE* SDK 1.4.2, JDK 5.0 and 6.0 from Sun Microsystems, Inc. (<http://sun.com/>).
- JRockit* JDK 1.4.2 and 5.0 from Oracle Corporation (<http://oracle.com/>).

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

- java
- javac
- javah
- javadoc

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

```
SET JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
SET PATH=%JAVA_HOME%\bin;%PATH%
```

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

```
SET JDK_HOME=
```

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

```
nmake {dllia32|dllintel64|libia32|libintel64} [function=...] [compiler=...]
```

If you type the make command and omit the target (for example, `dllia32`), the makefile prints the help info, which explains the targets and parameters.

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

Known Limitations of the Java* Examples

This section explains limitations of Java examples.

Functionality

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

Performance

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

Known bugs

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

Coding Tips

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

Aligning Data for Consistent Results

Routines in Intel MKL may return different results from run to run on the same system. This is usually due to a change in the order in which floating-point operations are performed. The two most influential factors are array alignment and parallelism. Array alignment can determine how internal loops order floating-point operations. Non-deterministic parallelism may change the order in which computational tasks are executed. While these results may differ, they should still fall within acceptable computational error bounds. To better ensure identical results from run to run, do the following:

- Align all arrays on 32-byte boundaries
- Run Intel MKL in the sequential mode

To align an array on 32-byte boundaries, use `mkl_malloc()` in place of system provided memory allocators, as shown in the code example below. Sequential mode of Intel MKL removes the influence of non-deterministic parallelism.

Aligning Addresses on 32-byte Boundaries

```
// ***** C language *****
...
#include <stdlib.h>
#include <mkl.h>
...
void *darray;
int workspace;
// Set value of alignment
int alignment=32;
...
// Allocate aligned workspace
darray = mkl_malloc( sizeof(double)*workspace, alignment );
...
// call the program using MKL
mkl_app( darray );
...
// Free workspace
mkl_free( darray );
```

```
! ***** Fortran language *****
...
! Set value of alignment
integer    alignment
parameter (alignment=32)
...
! Declare Intel MKL routines
#ifdef _IA32
integer mkl_malloc
#else
integer*8 mkl_malloc
#endif
external mkl_malloc, mkl_free, mkl_app
...
double precision darray
pointer (p_wrk,darray(1))
integer workspace
```

```

...
! Allocate aligned workspace
p_wrk = mkl_malloc( %val(8*workspace), %val(alignment) )
...
! call the program using Intel MKL
call mkl_app( darray )
...
! Free workspace
call mkl_free(p_wrk)

```

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

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

The following preprocessor symbols are available:

Predefined Preprocessor Symbol	Description
<code>__INTEL_MKL__</code>	Intel MKL major version
<code>__INTEL_MKL_MINOR__</code>	Intel MKL minor version
<code>__INTEL_MKL_UPDATE__</code>	Intel MKL update number
<code>INTEL_MKL_VERSION</code>	Intel MKL full version in the following format: <code>INTEL_MKL_VERSION =</code> <code>(__INTEL_MKL__ *100+ __INTEL_MKL_MINOR__) *100+ __I</code> <code>NTEL_MKL_UPDATE__</code>

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

To perform conditional compilation:

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

Example

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

C/C++:

```

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

```

Fortran:

```

include "mkl.fi"
!DEC$IF DEFINED INTEL_MKL_VERSION
!DEC$IF INTEL_MKL_VERSION .EQ. 100304

```

```
* Code to be conditionally compiled  
!DEC$ENDIF  
!DEC$ENDIF
```


Working with the Intel® Math Kernel Library Cluster Software

8

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

MPI Support

Intel MKL ScaLAPACK and Cluster FFTs support MPI implementations identified in the *Intel® Math Kernel Library (Intel® MKL) Release Notes*.

To link applications with ScaLAPACK or Cluster FFTs, you need to configure your system depending on your message-passing interface (MPI) implementation as explained below.

If you are using MPICH2, do the following:

1. Add `mpich2\include` to the include path (assuming the default MPICH2 installation).
2. Add `mpich2\lib` to the library path.
3. Add `mpi.lib` to your link command.
4. Add `fmpich2.lib` to your Fortran link command.
5. Add `cxx.lib` to your Release target link command and `cxxd.lib` to your Debug target link command for C++ programs.

If you are using the Microsoft MPI, do the following:

1. Add `Microsoft Compute Cluster Pack\include` to the include path (assuming the default installation of the Microsoft MPI).
2. Add `Microsoft Compute Cluster Pack\Lib\AMD64` to the library path.
3. Add `msmpi.lib` to your link command.

If you are using the Intel® MPI, do the following:

1. Add the following string to the include path: `%ProgramFiles%\Intel\MPI\, where <ver> is the directory for a particular MPI version and <arch> is ia32 or intel64, for example, %ProgramFiles%\Intel\MPI\3.1\intel64\include.`
2. Add the following string to the library path: `%ProgramFiles%\Intel\MPI\, for example, %ProgramFiles%\Intel\MPI\3.1\intel64\lib.`
3. Add `impi.lib` and `impicxx.lib` to your link command.

Check the documentation that comes with your MPI implementation for implementation-specific details of linking.

Linking with ScaLAPACK and Cluster FFTs

To link with Intel MKL ScaLAPACK and/or Cluster FFTs, use the following commands :

```
set lib =<path to MKL libraries>;<path to MPI libraries>;%lib%
<linker> <files to link> <MKL cluster library> <BLACS> <MKL core libraries> <MPI
libraries>
```

where the placeholders stand for paths and libraries as explained in the following table:

<path to MKL libraries>	<mk1 directory>\lib\{ia32 intel64}, depending on your architecture. If you performed the Setting Environment Variables step of the Getting Started process, you do not need to add this directory to the lib environment variable.
<path to MPI libraries>	Typically the lib subdirectory in the MPI installation directory. For example, C:\Program Files (x86)\Intel\MPI\3.2.0.005\ia32\lib for a default installation of Intel MPI 3.2.
<linker>	One of icl, ifort, xilink.
<MKL cluster library>	One of ScaLAPACK or Cluster FFT libraries for the appropriate architecture, which are listed in Directory Structure in Detail . For example, for the IA-32 architecture, it is one of mkl_scalapack_core.lib or mkl_cdft_core.lib.
<BLACS>	The BLACS library corresponding to your architecture, programming interface (LP64 or ILP64), and MPI version. These libraries are listed in Directory Structure in Detail . For example, for the IA-32 architecture, choose one of mkl_blacs_mpich2.lib or mkl_blacs_intelmpi.lib in case of static linking or mkl_blacs_dll.lib in case of dynamic linking; specifically, for MPICH2, choose mkl_blacs_mpich2.lib in case of static linking.
<MKL core libraries>	Intel MKL libraries other than ScaLAPACK or Cluster FFTs libraries.



TIP Use the [Link-line Advisor](#) to quickly choose the appropriate set of <MKL cluster Library>, <BLACS>, and <MKL core libraries>.

Intel MPI provides prepackaged scripts for its linkers to help you link using the respective linker. Therefore, if you are using Intel MPI, the best way to link is to use the following commands:

```
<path to Intel MPI binaries>\mpivars.bat
set lib = <path to MKL libraries>;%lib%
<mpilinker> <files to link> <MKL cluster Library> <BLACS> <MKL core libraries>
```

where the placeholders that are not yet defined are explained in the following table:

<path to MPI binaries>	By default, the bin subdirectory in the MPI installation directory. For example, C:\Program Files (x86)\Intel\MPI\3.2.0.005\ia32\lib for a default installation of Intel MPI 3.2;
------------------------	---

<MPI linker> mpicl or mpiifort

See Also

[Linking Your Application with the Intel® Math Kernel Library](#)
[Examples for Linking with ScaLAPACK and Cluster FFT](#)

Determining the Number of Threads

The OpenMP* software responds to the environment variable `OMP_NUM_THREADS`. Intel MKL also has other mechanisms to set the number of threads, such as the `MKL_NUM_THREADS` or `MKL_DOMAIN_NUM_THREADS` environment variables (see [Using Additional Threading Control](#)).

Make sure that the relevant environment variables have the same and correct values on all the nodes. Intel MKL versions 10.0 and higher no longer set the default number of threads to one, but depend on the OpenMP libraries used with the compiler to set the default number. For the threading layer based on the Intel compiler (`mkl_intel_thread.lib`), this value is the number of CPUs according to the OS.



CAUTION Avoid over-prescribing the number of threads, which may occur, for instance, when the number of MPI ranks per node and the number of threads per node are both greater than one. The product of MPI ranks per node and the number of threads per node should not exceed the number of physical cores per node.

The `OMP_NUM_THREADS` environment variable is assumed in the discussion below.

Set `OMP_NUM_THREADS` so that the product of its value and the number of MPI ranks per node equals the number of real processors or cores of a node. If the Intel® Hyper-Threading Technology is enabled on the node, use only half number of the processors that are visible on Windows OS.

See Also

[Setting Environment Variables on a Cluster](#)

Using DLLs

All the needed DLLs must be visible on all the nodes at run time, and you should install Intel® Math Kernel Library (Intel® MKL) on each node of the cluster. You can use Remote Installation Services (RIS) provided by Microsoft to remotely install the library on each of the nodes that are part of your cluster. The best way to make the DLLs visible is to point to these libraries in the `PATH` environment variable. See [Setting Environment Variables on a Cluster](#) on how to set the value of the `PATH` environment variable.

The ScaLAPACK DLLs for the IA-32 and Intel® 64 architectures (in the `<Composer XE directory>\redist\ia32\mkl` and `<Composer XE directory>\redist\intel64\mkl` directories, respectively) use the MPI dispatching mechanism. MPI dispatching is based on the `MKL_BLACS_MPI` environment variable. The BLACS DLL uses `MKL_BLACS_MPI` for choosing the needed MPI libraries. The table below lists possible values of the variable.

Value	Comment
MPICH2	Default value. MPICH2 1.0.x for Windows* OS is used for message passing
INTELMPI	Intel MPI is used for message passing
MSMPI	Microsoft MPI is used for message passing

If you are using a non-default MPI, assign the same appropriate value to `MKL_BLACS_MPI` on all nodes.

See Also

Setting Environment Variables on a Cluster

Setting Environment Variables on a Cluster

If you are using MPICH2 or Intel MPI, to set an environment variable on the cluster, use `-env`, `-genv`, `-genvlist` keys of `mpiexec`.

See the following MPICH2 examples on how to set the value of `OMP_NUM_THREADS`:

```
mpiexec -genv OMP_NUM_THREADS 2 ....
```

```
mpiexec -genvlist OMP_NUM_THREADS ....
```

```
mpiexec -n 1 -host first -env OMP_NUM_THREADS 2 test.exe : -n 1 -host second -env OMP_NUM_THREADS 3 test.exe ....
```

See the following Intel MPI examples on how to set the value of `MKL_BLACS_MPI`:

```
mpiexec -genv MKL_BLACS_MPI INTELMPI ....
```

```
mpiexec -genvlist MKL_BLACS_MPI ....
```

```
mpiexec -n 1 -host first -env MKL_BLACS_MPI INTELMPI test.exe : -n 1 -host second -env MKL_BLACS_MPI INTELMPI test.exe.
```

When using MPICH2, you may have problems with getting the global environment, such as `MKL_BLACS_MPI`, by the `-genvlist` key. In this case, set up user or system environments on each node as follows: From the **Start** menu, select **Settings > Control Panel > System > Advanced > Environment Variables**.

If you are using Microsoft MPI, the above ways of setting environment variables are also applicable if the Microsoft Single Program Multiple Data (SPMD) process managers are running in a debug mode on all nodes of the cluster. However, the best way to set environment variables is using the Job Scheduler with the Microsoft Management Console (MMC) and/or the Command Line Interface (CLI) to submit a job and pass environment variables. For more information about MMC and CLI, see the Microsoft Help and Support page at the Microsoft Web site (<http://www.microsoft.com/>).

Building ScaLAPACK Tests

To build ScaLAPACK tests:

- For the IA-32 architecture, add `mkl_scalapack_core.lib` to your link command.
- For the Intel® 64 architecture, add `mkl_scalapack_lp64.lib` or `mkl_scalapack_ilp64.lib`, depending on the desired interface.

Examples for Linking with ScaLAPACK and Cluster FFT

This section provides examples of linking with ScaLAPACK and Cluster FFT.

Note that a binary linked with ScaLAPACK runs the same way as any other MPI application (refer to the documentation that comes with your MPI implementation).

For further linking examples, see the support website for Intel products at <http://www.intel.com/software/products/support/>.

See Also

Directory Structure in Detail

Examples for Linking a C Application

These examples illustrate linking of an application whose main module is in C under the following conditions:

- MPICH2 1.0.x is installed in `c:\mpich2x64`.
- You use the Intel® C++ Compiler 10.0 or higher.

To link with ScaLAPACK using LP64 interface for a cluster of Intel® 64 architecture based systems, set the environment variable and use the link line as follows:

```
set lib=c:\mpich2x64\lib;<mkl_directory>\lib\intel64;%lib%
icl <user files to link> mkl_scalapack_lp64.lib mkl_blacs_mpich2_lp64.lib
mkl_intel_lp64.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib mpi.lib cxx.lib
bufferoverflowu.lib
```

To link with Cluster FFT using LP64 interface for a cluster of Intel® 64 architecture based systems, set the environment variable and use the link line as follows:

```
set lib=c:\mpich2x64\lib;<mkl_directory>\lib\intel64;%lib%
icl <user files to link> mkl_cdft_core.lib mkl_blacs_mpich2_lp64.lib mkl_intel_lp64.lib
mkl_intel_thread.lib mkl_core.lib libiomp5md.lib mpi.lib cxx.lib bufferoverflowu.lib
```

See Also

[Linking with ScaLAPACK and Cluster FFTs](#)

[Linking with System Libraries](#)

Examples for Linking a Fortran Application

These examples illustrate linking of an application whose main module is in Fortran under the following conditions:

- Microsoft Windows Compute Cluster Pack SDK is installed in `c:\MS CCP SDK`.
- You use the Intel® Fortran Compiler 10.0 or higher.

To link with ScaLAPACK using LP64 interface for a cluster of Intel® 64 architecture based systems, set the environment variable and use the link line as follows:

```
set lib="c:\MS CCP SDK\Lib\AMD64";<mkl_directory>\lib\intel64;%lib%
ifort <user files to link> mkl_scalapack_lp64.lib mkl_blacs_mpich2_lp64.lib
mkl_intel_lp64.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib msmpi.lib
bufferoverflowu.lib
```

To link with Cluster FFTs using LP64 interface for a cluster of Intel® 64 architecture based systems, set the environment variable and use the link line as follows:

```
set lib="c:\MS CCP SDK\Lib\AMD64";<mkl_directory>\lib\intel64;%lib%
ifort <user files to link> mkl_cdft_core.lib mkl_blacs_mpich2_lp64.lib
mkl_intel_lp64.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib msmpi.lib
bufferoverflowu.lib
```

See Also

[Linking with ScaLAPACK and Cluster FFTs](#)

[Linking with System Libraries](#)

Programming with Intel® Math Kernel Library in Integrated Development Environments (IDE)



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

Configuring the Microsoft Visual C/C++ Development System to Link with Intel® MKL

Steps for configuring Microsoft Visual C/C++ development system for linking with Intel® Math Kernel Library (Intel® MKL) depend on whether you installed the C++ Integration(s) in Microsoft Visual Studio* component of the Intel® Composer XE:

- If you installed the integration component, see [Automatically Linking Your Microsoft Visual C/C++ Project with Intel MKL](#).
- If you did not install the integration component or need more control over Intel MKL libraries to link, you can configure the Microsoft Visual C++ development system by performing the following steps. Though some versions of the Visual C++ development system may vary slightly in the menu items mentioned below, the fundamental configuring steps are applicable to all these versions.
 1. In Solution Explorer, right-click your project and click **Properties** (for Visual C++ 2010) or select **Tools > Options** (for Visual C++ 2005 and Visual C++ 2008)
 2. Select **Configuration Properties > VC++ Directories** (for Visual C++ 2010) or select **Projects and Solutions > VC++ Directories** (for Visual C++ 2005 and Visual C++ 2008)
 3. Select **Include Directories**. Add the directory for the Intel MKL include files, that is, `<mk1 directory>\include`
 4. Select **Library Directories**. Add architecture-specific directories for Intel MKL and OpenMP* libraries, for example: `<mk1 directory>\lib\ia32` and `<Composer XE directory>\compiler\lib\ia32`
 5. Select **Executable Directories**. Add architecture-specific directories with dynamic-link libraries:
 - For OpenMP* support, for example: `<Composer XE directory>\redist\ia32\compiler`
 - For Intel MKL (only if you link dynamically), for example: `<Composer XE directory>\redist\ia32\mk1`
 6. Select **Configuration Properties > Custom Build Setup > Additional Dependencies**. Add the libraries required, for example, `mk1_intel_c.lib` `mk1_intel_thread.lib` `mk1_core.lib` `libiomp5md.lib`

See Also

[Intel® Software Documentation Library Linking in Detail](#)

Configuring Intel® Visual Fortran to Link with Intel MKL

Steps for configuring Intel® Visual Fortran for linking with Intel® Math Kernel Library (Intel® MKL) depend on whether you installed the Visual Fortran Integration(s) in Microsoft Visual Studio* component of the Intel® Composer XE:

- If you installed the integration component, see [Automatically Linking Your Intel® Visual Fortran Project with Intel® MKL](#).

- If you did not install the integration component or need more control over Intel MKL libraries to link, you can configure your project as follows:
 1. Select **Project > Properties > Linker > General > Additional Library Directories**. Add architecture-specific directories for Intel MKL and OpenMP* libraries, for example: `<mkl_directory>\lib\ia32` and `<Composer XE directory>\compiler\lib\ia32`
 2. Select **Project > Properties > Linker > Input > Additional Dependencies**. Insert names of the required libraries, for example: `mkl_intel_c.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib`
 3. Select **Project > Properties > Debugging > Environment**. Add architecture-specific paths to dynamic-link libraries:
 - For OpenMP* support; for example: enter `PATH=%PATH%;<Composer XE directory>\redist\ia32\compiler`
 - For Intel MKL (only if you link dynamically); for example: enter `PATH=%PATH%;<Composer XE directory>\redist\ia32\mkl`

See Also

[Intel® Software Documentation Library](#)

Running an Intel MKL Example in the Visual Studio* 2008 IDE

This section explains how to create and configure projects with the Intel® Math Kernel Library (Intel® MKL) examples in Microsoft Visual Studio* 2008. For Intel MKL examples where the instructions below do not work, see [Known Limitations](#).

To run the Intel MKL C examples in Microsoft Visual Studio 2008:

1. Do either of the following:
 - Install Intel® C/C++ Compiler and integrate it into Visual Studio (recommended).
 - Use the Microsoft Visual C++* 2008 Compiler integrated into Visual Studio*.
2. [Create, configure, and run the Intel C/C++ and/or Microsoft Visual C++* 2008](#).

To run the Intel MKL Fortran examples in Microsoft Visual Studio 2008:

1. Install Intel® Visual Fortran Compiler and integrate it into Visual Studio. The default installation of the Intel Visual Fortran Compiler performs this integration. For more information, see the *Intel Visual Fortran Compiler documentation*.
2. [Create, configure, and run the Intel Visual Fortran project](#).

Creating, Configuring, and Running the Intel® C/C++ and/or Visual C++* 2008 Project

This section demonstrates how to create a Visual C/C++ project using an Intel® Math Kernel Library (Intel® MKL) example in Microsoft Visual Studio 2008.

The instructions below create a Win32/Debug project running one Intel MKL example in a Console window. For details on creation of different kinds of Microsoft Visual Studio projects, refer to MSDN Visual Studio documentation at <http://www.microsoft.com>.

To create and configure the Win32/Debug project running an Intel MKL C example with the Intel® C/C++ Compiler integrated into Visual Studio and/or Microsoft Visual C++* 2008, perform the following steps:

1. Create a C Project:
 - Open Visual Studio 2008.
 - On the main menu, select **File > New > Project** to open the **New Project** window.
 - Select **Project Types > Visual C++ > Win32**, then select **Templates > Win32 Console Application**. In the **Name** field, type `<project name>`, for example, `MKL_CBLAS_CAXPYIX`, and click **OK**.
 - The **New Project** window closes, and the **Win32 Application Wizard - <project name>** window opens.
 - Select **Next**, then select **Application Settings**, check **Additional options > Empty project**, and click **Finish**.

The **Win32 Application Wizard - <project name>** window closes.

The next steps are performed inside the **Solution Explorer** window. To open it, select **View > Solution Explorer** from the main menu.

2. (optional) To switch to the Intel C/C++ project, right-click **<project name>** and from the drop-down menu, select **Convert** to use Intel® C++ Project System. (The menu item is available if the Intel® C/C++ Compiler is integrated into Visual Studio.)

3. Add sources of the Intel MKL example to the project:

Right-click the **Source Files** folder under **<project name>** and select **Add > Existing Item...** from the drop-down menu.

The **Add Existing Item - <project name>** window opens.

Browse to the Intel MKL example directory, for example, `<mkl_directory>\examples\cblas\source`. Select the example file and supporting files with extension ".c" (C sources), for example, select files `cblas_caxpyix.c` and `common_func.c`. For the list of supporting files in each example directory, see [Support Files for Intel MKL Examples](#). Click **Add**.

The **Add Existing Item - <project name>** window closes, and selected files appear in the **Source Files** folder in Solution Explorer.

The next steps adjust the properties of the project.

4. Select **<project name>**.
5. On the main menu, select **Project > Properties** to open the **<project name> Property Pages** window.
6. Set Intel MKL Include dependencies:

Select **Configuration Properties > C/C++ > General**. In the right-hand part of the window, select **Additional Include Directories > ...** (the browse button).

The **Additional Include Directories** window opens.

Click the **New Line** button (the first button in the uppermost row). When the new line appears in the window, click the browse button.

The **Select Directory** window opens.

Browse to the `<mkl_directory>\include` directory and click **OK**.

The **Select Directory** window closes, and full path to the Intel MKL include directory appears in the **Additional Include Directories** window.

Click **OK** to close the window.

7. Set library dependencies:

Select **Configuration Properties > Linker > General**. In the right-hand part of the window, select **Additional Library Directories > ...** (the browse button).

The **Additional Library Directories** window opens.

Click the **New Line** button (the first button in the uppermost row). When the new line appears in the window, click the browse button.

The **Select Directory** window opens.

Browse to the directory with the Intel MKL libraries `<mkl_directory>\lib\<architecture>`, where `<architecture>` is one of `{ia32, intel64}`, for example: `<mkl_directory>\lib\ia32`. (For most laptop and desktop computers, `<architecture>` is `ia32`.) Click **OK**.

The **Select Directory** window closes, and the full path to the Intel MKL libraries appears in the **Additional Library Directories** window.

Click the **New Line** button again. When the new line appears in the window, click the browse button.

The **Select Directory** window opens.

Browse to the `<Composer XE directory>compiler\lib\<architecture>`, where `<architecture>` is one of `{ia32, intel64}`, for example: `<Composer XE directory>\compiler\lib\ia32`. Click **OK**.

The **Select Directory** window closes, and the specified full path appears in the **Additional Library Directories** window.

Click **OK** to close the **Additional Library Directories** window.

Select **Configuration Properties > Linker > Input**. In the right-hand part of the window, select **Additional Dependencies > ...** (the browse button).

The **Additional Dependencies** window opens.

Type the libraries required, for example, if `<architecture> = ia32`, type `mkl_intel_c.lib` `mkl_intel_thread.lib` `mkl_core.lib` `libiomp5md.lib`. For more details, see [Linking in Detail](#).

Click **OK** to close the **Additional Dependencies** window.

If the Intel MKL example directory does not contain a data directory, skip the next step.

8. Set data dependencies for the Intel MKL example:
 8. Select **Configuration Properties > Debugging**. In the right-hand part of the window, select **Command Arguments > [dropdown] > <Edit...>**. The **Command Arguments** window opens.
 9. Type the path to the proper data file in quotes. The name of the data file is the same as the name of the example file, with a ".d" extension, for example, "`<mkl_directory>\examples\cblas\data\cblas_caxpyix.d`".
 10. Click **OK** to close the **Command Arguments** window.
9. Click **OK** to close the **<project name> Property Pages** window.
10. Certain examples do not pause before the end of execution. To see the results printed in the Console window, set a breakpoint at the very last `'return 0;'` statement or add a call to `'getchar();'` before the last `'return 0'` statement.
11. To build the solution, select **Build > Build Solution**.



NOTE You may see warnings about unsafe functions and variables. To get rid of these warnings, go to **Project > Properties**, and when the **<project name> Property Pages** window opens, go to **Configuration Properties > C/C++ > Preprocessor**. In the right-hand part of the window, select **Preprocessor Definitions**, add `_CRT_SECURE_NO_WARNINGS`, and click **OK**.

12. To run the example, select **Debug > Start Debugging**. The Console window opens.
13. You can see the results of the example in the Console window. If you used the `'getchar();'` statement to pause execution of the program, press **Enter** to complete the run. If you used a breakpoint to pause execution of the program, select **Debug > Continue**. The Console window closes.

See Also

[Running an Intel MKL Example in the Visual Studio* 2008 IDE](#)

Creating, Configuring, and Running the Intel Visual Fortran Project

This section demonstrates how to create an Intel Visual Fortran project running an Intel MKL example in Microsoft Visual Studio 2008.

The instructions below create a Win32/Debug project running one Intel MKL example in a Console window. For details on creation of different kinds of Microsoft Visual Studio projects, refer to MSDN Visual Studio documentation at <http://www.microsoft.com>.

To create and configure a Win32/Debug project running the Intel MKL Fortran example with the Intel Visual Fortran Compiler integrated into Visual Studio, perform the following steps:

1. Create a Visual Fortran Project:
 1. Open Visual Studio 2008.
 2. On the main menu, select **File > New > Project** to open the **New Project** window.
 3. Select **Project Types > Intel® Fortran > Console Application**, then select **Templates > Empty Project**. When done, in the **Name** field, type `<project name>` for example, `MKL_PDETF_D_TRIG_TRANSFORM_BVP`, and click **OK**. The **New Project** window closes.

The next steps are performed inside the Solution Explorer window. To open it, select **View>Solution Explorer** from the main menu.
2. Add sources of Intel MKL example to the project:
 1. Right-click the **Source Files** folder under **<project name>** and select **Add > Existing Item...** from the drop-down menu. The **Add Existing Item - <project name>** window opens.
 2. Browse to the Intel MKL example directory, for example, `<mkl_directory>\examples\pdttf\source`. Select the example file and supporting files with extension ".f" or ".f90" (Fortran sources). For example, select the `d_trig_tforms_bvp.f90` file. For the list of supporting files in each example directory, see [Support Files for Intel MKL Examples](#). Click **Add**.

The **Add Existing Item - <project name>** window closes, and the selected files appear in the **Source Files** folder in Solution Explorer. Some examples with the "use" statements require the next two steps.

Right-click the **Header Files** folder under **<project name>** and select **Add > Existing Item...** from the drop-down menu.

The **Add Existing Item - <project name>** window opens.

Browse to the `<mkl_directory>\include` directory. Select the header files that appear in the "use" statements. For example, select the `mkl_dfti.f90` and `mkl_trig_transforms.f90` files. Click **Add**.

The **Add Existing Item - <project name>** window closes, and the selected files to appear in the **Header Files** folder in Solution Explorer.

The next steps adjust the properties of the project:

3. Select the **<project name>**.
4. On the main menu, select **Project > Properties** to open the **<project name> Property Pages** window.
5. Set the Intel MKL include dependencies:
 - Select **Configuration Properties > Fortran > General**. In the right-hand part of the window, select **Additional Include Directories > ▼ > <Edit...>**. The **Additional Include Directories** window opens.
 - Type the Intel MKL include directory in quotes: "`<mkl_directory>\include`". Click **OK** to close the window.
6. Select **Configuration Properties > Fortran > Preprocessor**. In the right-hand part of the window, select **Preprocess Source File > Yes** (default is **No**). This step is recommended because some examples require preprocessing.
7. Set library dependencies:
 - Select **Configuration Properties > Linker > General**. In the right-hand part of the window, select **Additional Library Directories > ▼ > <Edit...>**. The **Additional Library Directories** window opens.
 - Type the directory with the Intel MKL libraries in quotes, that is, "`<mkl_directory>\lib\<architecture>`", where `<architecture>` is one of { `ia32`, `intel64` }, for example: "`<mkl_directory>\lib\ia32`". (For most laptop and desktop computers `<architecture>` is `ia32`.) Click **OK** to close the window.
 - Select **Configuration Properties > Linker > Input**. In the right-hand part of the window, select **Additional Dependencies** and type the libraries required, for example, if `<architecture> =ia32`, type `mkl_intel_c.lib mkl_intel_thread.lib mkl_core.lib libiomp5md.lib`.
8. In the **<project name> Property Pages** window, click **OK** to close the window.
9. Some examples do not pause before the end of execution. To see the results printed in the Console window, set a breakpoint at the very end of the program or add the 'pause' statement before the last 'end' statement.
10. To build the solution, select **Build > Build Solution**.
11. To run the example, select **Debug > Start Debugging**. The Console window opens.
12. You can see the results of the example in the Console window. If you used 'pause' statement to pause execution of the program, press **Enter** to complete the run. If you used a breakpoint to pause execution of the program, select **Debug > Continue**. The Console window closes.

Support Files for Intel® Math Kernel Library Examples

Below is the list of support files that have to be added to the project for respective examples:

```
examples\cblas\source: common_func.c
```

```
examples\dftc\source: dfti_example_status_print.c dfti_example_support.c
```

Known Limitations of the Project Creation Procedure

You cannot create a Visual Studio* project using the instructions from [Creating, Configuring, and Running the Intel® C/C++ and/or Visual C++* 2008 Project](#) or [Creating, Configuring, and Running the Intel® Visual Fortran Project](#) for examples from the following directories:

```
examples\blas
examples\blas95
examples\cdftc
examples\cdftf
examples\dftf
examples\fftw2x_cdf
examples\fftw2xc
examples\fftw2xf
examples\fftw3xc
examples\fftw3xf
examples\java
examples\lapack
examples\lapack95
```

Getting Assistance for Programming in the Microsoft Visual Studio* IDE

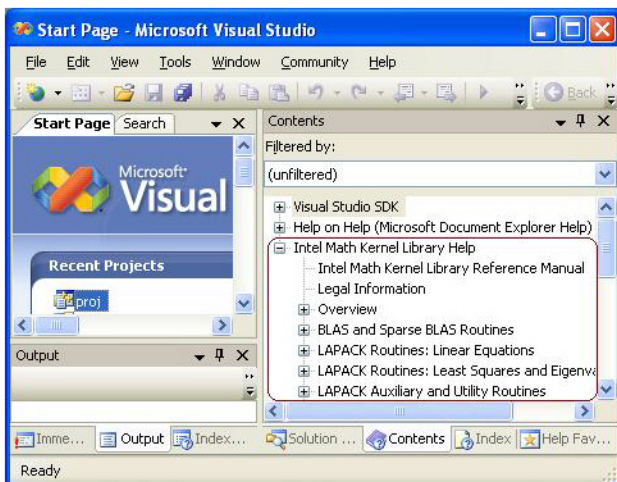
Viewing Intel MKL Documentation in Visual Studio* IDE

Viewing Intel MKL Documentation in Document Explorer (Visual Studio* 2005/2008 IDE)

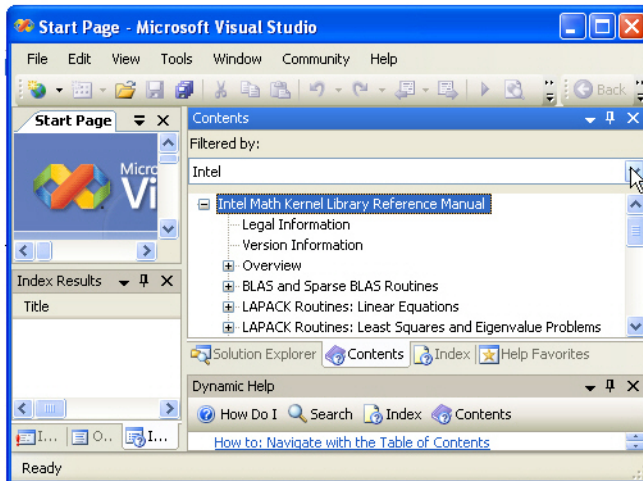
Intel MKL documentation is integrated in the Visual Studio IDE (VS) help collection. To open Intel MKL help,

1. Select **Help > Contents** from the menu. This displays the list of **VS Help** collections.
2. Click Intel **Math Kernel Library Help**.
3. In the help tree that expands, click **Intel MKL Reference Manual**.

To open the help index, select **Help > Index** from the menu. To search in the help, select **Help > Search** from the menu and enter a search string.



You can filter Visual Studio Help collections to show only content related to installed Intel tools. To do this, select "Intel" from the Filtered by list. This hides the contents and index entries for all collections that do not refer to Intel.



Accessing Intel MKL Documentation in Visual Studio* 2010 IDE

To access the Intel MKL documentation in Visual Studio* 2010 IDE:

- Configure the IDE to use local help (once). To do this,
 - Go to **Help > Manage Help Settings** and check **I want to use online help**
- Use the **Help > View Help** menu item to view a list of available help collections and open the Intel MKL documentation.

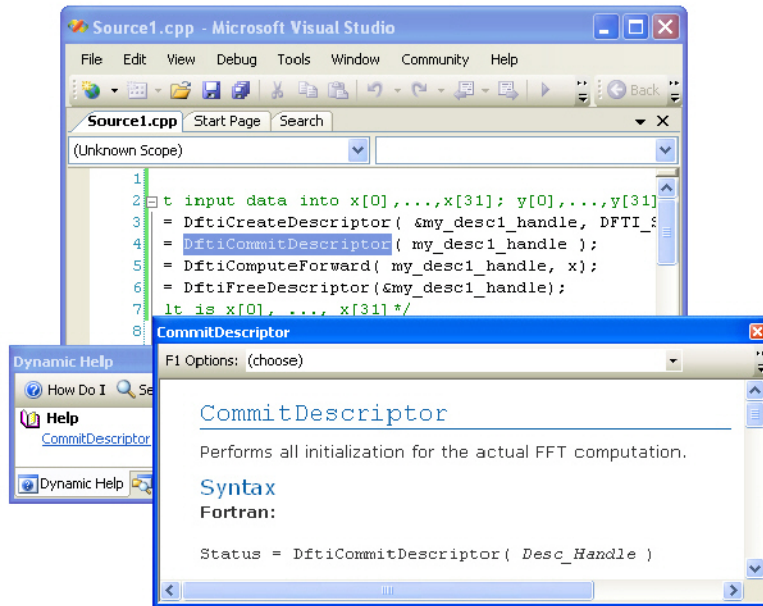
Using Context-Sensitive Help

When typing your code in the Visual Studio* (VS) IDE Code Editor, you can get context-sensitive help using the F1 Help and Dynamic Help features.

F1 Help

To open the help topic relevant to the current selection, press F1.

In particular, to open the help topic describing an Intel MKL function called in your code, select the function name and press F1. The topic with the function description opens in the window that displays search results:



Dynamic Help

Dynamic Help also provides access to topics relevant to the current selection or to the text being typed. Links to all relevant topics are displayed in the Dynamic Help window.

To get the list of relevant topics each time you select the Intel MKL function name or as you type it in your code, open the Dynamic Help window by selecting **Help > Dynamic Help** from the menu.

To open a topic from the list, click the appropriate link in the Dynamic Help window, shown in the above figure. Typically only one link corresponds to each Intel MKL function.

Using the IntelliSense* Capability

IntelliSense is a set of native Visual Studio*(VS) IDE features that make language references easily accessible.

The user programming with Intel MKL in the VS Code Editor can employ two IntelliSense features: Parameter Info and Complete Word.

Both features use header files. Therefore, to benefit from IntelliSense, make sure the path to the include files is specified in the VS or solution settings. For example, see [Configuring the Microsoft Visual C/C++* Development System to Link with Intel® MKL](#) on how to do this.

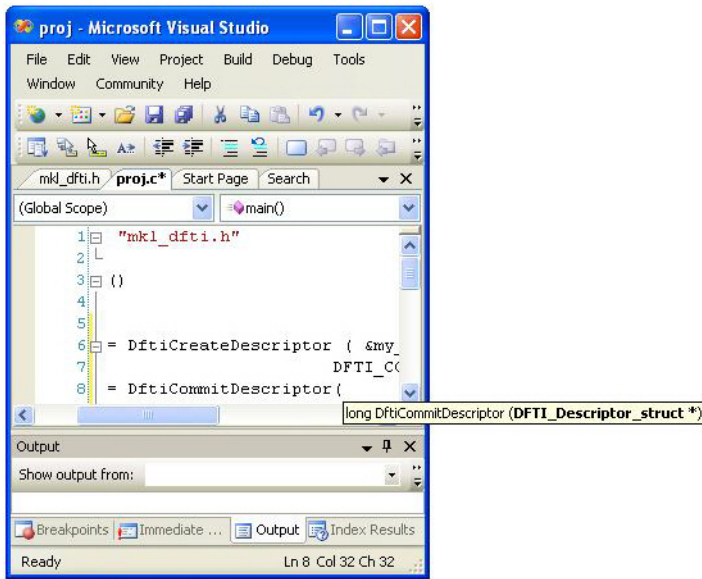
Parameter Info

The Parameter Info feature displays the parameter list for a function to give information on the number and types of parameters. This feature requires adding the `include` statement with the appropriate Intel MKL header file to your code.

To get the list of parameters of a function specified in the header file,

1. Type the function name.
2. Type the opening parenthesis.

This brings up the tooltip with the list of the function parameters:

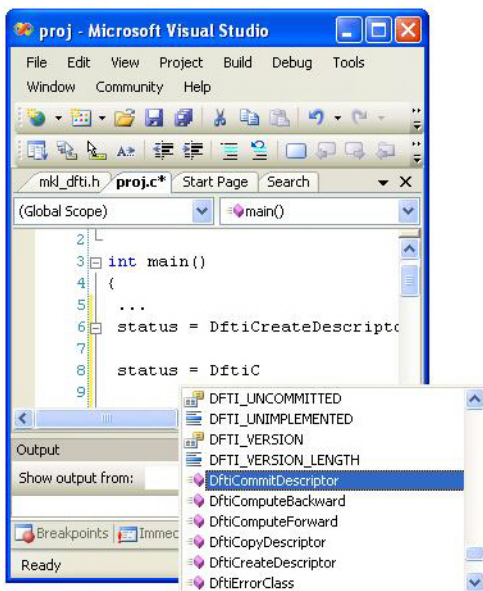


Complete Word

For a software library, the Complete Word feature types or prompts for the rest of the name defined in the header file once you type the first few characters of the name in your code. This feature requires adding the `include` statement with the appropriate Intel MKL header file to your code.

To complete the name of the function or named constant specified in the header file,

1. Type the first few characters of the name.
2. Press **Alt+RIGHT ARROW** or **Ctrl+SPACEBAR**.
If you have typed enough characters to disambiguate the name, the rest of the name is typed automatically. Otherwise, a pop-up list appears with the names specified in the header file
3. Select the name from the list, if needed.



LINPACK and MP LINPACK Benchmarks

10

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Intel® Optimized LINPACK Benchmark for Windows* OS

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

Do not use this benchmark to report LINPACK 100 performance because that is a compiled-code only benchmark. This is a shared-memory (SMP) implementation which runs on a single platform. Do not confuse this benchmark with:

- MP LINPACK, which is a distributed memory version of the same benchmark.
- LINPACK, the library, which has been expanded upon by the LAPACK library.

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

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

Contents of the Intel® Optimized LINPACK Benchmark

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

File in <code>benchmarks\linpack\</code>	Description
<code>linpack_xeon32.exe</code>	The 32-bit program executable for a system based on Intel® Xeon® processor or Intel® Xeon® processor MP with or without Streaming SIMD Extensions 3 (SSE3).
<code>linpack_xeon64.exe</code>	The 64-bit program executable for a system with Intel® Xeon® processor using Intel® 64 architecture.
<code>runme_xeon32.bat</code>	A sample shell script for executing a pre-determined problem set for <code>linpack_xeon32.exe</code> . <code>OMP_NUM_THREADS</code> set to 2 processors.
<code>runme_xeon64.bat</code>	A sample shell script for executing a pre-determined problem set for <code>linpack_xeon64.exe</code> . <code>OMP_NUM_THREADS</code> set to 4 processors.

File in benchmarks \linpack\	Description
lininput_xeon32	Input file for pre-determined problem for the <code>runme_xeon32</code> script.
lininput_xeon64	Input file for pre-determined problem for the <code>runme_xeon64</code> script.
win_xeon32.txt	Result of the <code>runme_xeon32</code> script execution.
win_xeon64.txt	Result of the <code>runme_xeon64</code> script execution.
help.lpk	Simple help file.
xhelp.lpk	Extended help file.

See Also

[High-level Directory Structure](#)

Running the Software

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

```
runme_xeon32.bat
runme_xeon64.bat
```

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

```
linpack_xeon32.exe -e
linpack_xeon64.exe -e
```

The pre-defined data input files `lininput_xeon32` and `lininput_xeon64` are provided merely as examples. Different systems have different numbers of processors or amounts of memory and thus require new input files. The extended help can be used for insight into proper ways to change the sample input files.

Each input file requires at least the following amount of memory:

```
lininput_xeon32    2 GB
lininput_xeon64   16 GB
```

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

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Known Limitations of the Intel® Optimized LINPACK Benchmark

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

- Intel Optimized LINPACK Benchmark is threaded to effectively use multiple processors. So, in multi-processor systems, best performance will be obtained with the Intel® Hyper-Threading Technology turned off, which ensures that the operating system assigns threads to physical processors only.
- 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.

Intel® Optimized MP LINPACK Benchmark for Clusters

Overview of the Intel® Optimized MP LINPACK Benchmark for Clusters

The Intel® Optimized MP LINPACK Benchmark for Clusters is based on modifications and additions to HPL 2.0 from Innovative Computing Laboratories (ICL) at the University of Tennessee, Knoxville (UTK). The Intel Optimized MP LINPACK Benchmark for Clusters can be used for Top 500 runs (see <http://www.top500.org>). To use the benchmark you need be intimately familiar with the HPL distribution and usage. The Intel Optimized MP LINPACK Benchmark for Clusters provides some additional enhancements and bug fixes designed to make the HPL usage more convenient, as well as explain Intel® Message-Passing Interface (MPI) settings that may enhance performance. The `.\benchmarks\mp_linpack` directory adds techniques to minimize search times frequently associated with long runs.

The Intel® Optimized MP LINPACK Benchmark for Clusters is an implementation of the Massively Parallel MP LINPACK benchmark by means of HPL code. It solves a random 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. You can solve any size (N) system of equations that fit into memory. The benchmark uses full row pivoting to ensure the accuracy of the results.

Use the Intel Optimized MP LINPACK Benchmark for Clusters on a distributed memory machine. On a shared memory machine, use the Intel Optimized LINPACK Benchmark.

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 HPL benchmark. Use the Intel Optimized MP LINPACK Benchmark to benchmark your cluster. The prebuilt binaries require that you first install Intel® MPI 3.x be installed on the cluster. The run-time version of Intel MPI is free and can be downloaded from www.intel.com/software/products/.

The Intel package includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories and neither the University nor ICL endorse or promote this product. Although HPL 2.0 is redistributable under certain conditions, this particular package is subject to the Intel MKL license.

Intel MKL has introduced a new functionality into MP LINPACK, which is called a *hybrid* build, while continuing to support the older version. The term *hybrid* refers to special optimizations added to take advantage of mixed OpenMP*/MPI parallelism.

If you want to use one MPI process per node and to achieve further parallelism by means of OpenMP, use the hybrid build. In general, the hybrid build is useful when the number of MPI processes per core is less than one. If you want to rely exclusively on MPI for parallelism and use one MPI per core, use the non-hybrid build.

In addition to supplying certain hybrid prebuilt binaries, Intel MKL supplies some hybrid prebuilt libraries for Intel® MPI to take advantage of the additional OpenMP* optimizations.

If you wish to use an MPI version other than Intel MPI, you can do so by using the MP LINPACK source provided. You can use the source to build a non-hybrid version that may be used in a hybrid mode, but it would be missing some of the optimizations added to the hybrid version.

Non-hybrid builds are the default of the source code makefiles provided. In some cases, the use of the hybrid mode is required for external reasons. If there is a choice, the non-hybrid code may be faster. To use the non-hybrid code in a hybrid mode, use the threaded version of Intel MKL BLAS, link with a thread-safe MPI, and call function `MPI_init_thread()` so as to indicate a need for MPI to be thread-safe.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Contents of the Intel® Optimized MP LINPACK Benchmark for Clusters

The Intel Optimized MP LINPACK Benchmark for Clusters (MP LINPACK Benchmark) includes the HPL 2.0 distribution in its entirety, as well as the modifications delivered in the files listed in the table below and located in the `benchmarks\mp_linpack\` subdirectory of the Intel MKL directory.



NOTE Because MP LINPACK Benchmark includes the *entire* HPL 2.0 distribution, which provides a configuration for Linux* OS only, some Linux OS files remain in the directory.

Directory/File in benchmarks \mp_linpack\ 	Contents
testing\ptest\HPL_pctest.c	HPL 2.0 code modified to display captured DGEMM information in ASYUOGO2_DISPLAY if it was captured (for details, see New Features).
src\blas\HPL_dgemm.c	HPL 2.0 code modified to capture DGEMM information, if desired, from ASYUOGO2_DISPLAY.
src\grid\HPL_grid_init.c	HPL 2.0 code modified to do additional grid experiments originally not in HPL 2.0.
src\pgesv\HPL_pdgesvK2.c	HPL 2.0 code modified to do ASYUOGO and ENDEARLY modifications.
src\pgesv\HPL_pdgesv0.c	HPL 2.0 code modified to do ASYUOGO, ASYUOGO2, and ENDEARLY modifications.
testing\ptest\HPL.dat	HPL 2.0 sample HPL.dat modified.
makes	All the makefiles in this directory have been rebuilt in the Windows OS distribution.
testing\ptimer\ 	Some files in here have been modified in the Windows OS distribution.
testing\timer\ 	Some files in here have been modified in the Windows OS distribution.
Make	(New) Sample architecture makefile for nmake utility to be used on processors based on the IA-32 and Intel® 64 architectures and Windows OS.
bin_intel\ia32\XHPL_ia32.exe	(New) Prebuilt binary for the IA-32 architecture, Windows OS, and Intel® MPI.
bin_intel \intel64\XHPL_intel64.exe	(New) Prebuilt binary for the Intel® 64 architecture, Windows OS, and Intel MPI.

Directory/File in benchmarks	Contents
\mp_linpack\ lib_hybrid \ia32\libhpl_hybrid.lib	(New) Prebuilt library with the hybrid version of MP LINPACK for the IA-32 architecture and Intel MPI.
lib_hybrid \intel64\libhpl_hybrid.lib	(New) Prebuilt library with the hybrid version of MP LINPACK for the Intel® 64 architecture and Intel MPI.
bin_intel \ia32\xhpl_hybrid_ia32.exe	(New) Prebuilt hybrid binary for the IA-32 architecture, Windows OS, and Intel MPI.
bin_intel \intel64\xhpl_hybrid_intel64.exe	(New) Prebuilt hybrid binary for the Intel® 64 architecture, Windows OS, and Intel MPI.
nodeperf.c	(New) Sample utility that tests the DGEMM speed across the cluster.

See Also

[High-level Directory Structure](#)

Building the MP LINPACK

The MP LINPACK Benchmark contains a few sample architecture makefiles. You can edit them to fit your specific configuration. Specifically:

- Set `TOPdir` to the directory that MP LINPACK is being built in.
- Set MPI variables, that is, `MPdir`, `MPinc`, and `MPLib`.
- Specify the location Intel MKL and of files to be used (`LAdir`, `LAinc`, `LALib`).
- Adjust compiler and compiler/linker options.
- Specify the version of MP LINPACK you are going to build (hybrid or non-hybrid) by setting the version parameter for the `nmake` command. For example:

```
nmake arch=intel64 mpi=intelmpi version=hybrid install
```

For some sample cases, the makefiles contain values that must be common. However, you need to be familiar with building an HPL and picking appropriate values for these variables.

New Features of Intel® Optimized MP LINPACK Benchmark

The toolset is basically identical with the HPL 2.0 distribution. There are a few changes that are optionally compiled in and disabled until you specifically request them. These new features are:

ASYOUGO: Provides non-intrusive performance information while runs proceed. There are only a few outputs and this information does not impact performance. This is especially useful because many runs can go for hours without any information.

ASYOUGO2: Provides slightly intrusive additional performance information by intercepting every `DGEMM` call.

ASYOUGO2_DISPLAY: Displays the performance of all the significant `DGEMMs` inside the run.

ENDEARLY: Displays a few performance hints and then terminates the run early.

FASTSWAP: Inserts the LAPACK-optimized `DLASWP` into HPL's code. You can experiment with this to determine best results.

HYBRID: Establishes the Hybrid OpenMP/MPI mode of MP LINPACK, providing the possibility to use threaded Intel MKL and prebuilt MP LINPACK hybrid libraries.



CAUTION Use this option only with an Intel compiler and the Intel® MPI library version 3.1 or higher. You are also recommended to use the compiler version 10.0 or higher.

Benchmarking a Cluster

To benchmark a cluster, follow the sequence of steps below (some of them are optional). Pay special attention to the iterative steps 3 and 4. They make a loop that searches for HPL parameters (specified in `HPL.dat`) that enable you to reach the top performance of your cluster.

1. Install HPL and make sure HPL is functional on all the nodes.
2. You may run `nodeperf.c` (included in the distribution) to see the performance of `DGEMM` on all the nodes.

Compile `nodeperf.c` with your MPI and Intel MKL. For example:

```
icl /Za /O3 /w /D_WIN_ /I"<Home directory of MPI>\include" "<Home directory of MPI libraries>\<MPI library>"
"<mkl directory>\lib\intel64\mkl_core.lib"
"<Composer XE directory>\lib\intel64\libiomp5md.lib" nodeperf.c
```

where `<MPI library>` is `msmpi.lib` in the case of Microsoft* MPI and `mpi.lib` in the case of MPICH.

Launching `nodeperf.c` on all the nodes is especially helpful in a very large cluster. `nodeperf` enables quick identification of the potential problem spot without numerous small MP LINPACK runs around the cluster in search of the bad node. It goes through all the nodes, one at a time, and reports the performance of `DGEMM` followed by some host identifier. Therefore, the higher the `DGEMM` performance, the faster that node was performing.

3. Edit `HPL.dat` to fit your cluster needs. Read through the HPL documentation for ideas on this. Note, however, that you should use at least 4 nodes.
4. Make an HPL run, using compile options such as `ASYOUGO`, `ASYOUGO2`, or `ENDEARLY` to aid in your search. These options enable you to gain insight into the performance sooner than HPL would normally give this insight.

When doing so, follow these recommendations:

- Use MP LINPACK, which is a patched version of HPL, to save time in the search.

All performance intrusive features are compile-optional in MP LINPACK. That is, if you do not use the new options to reduce search time, these features are disabled. The primary purpose of the additions is to assist you in finding solutions.

HPL requires a long time to search for many different parameters. In MP LINPACK, the goal is to get the best possible number.

Given that the input is not fixed, there is a large parameter space you must search over. An exhaustive search of all possible inputs is improbably large even for a powerful cluster. MP LINPACK optionally prints information on performance as it proceeds. You can also terminate early.

- Save time by compiling with `-DENDEARLY -DASYOUGO2` and using a negative threshold (do not use a negative threshold on the final run that you intend to submit as a Top500 entry). Set the threshold in line 13 of the HPL 2.0 input file `HPL.dat`
 - If you are going to run a problem to completion, do it with `-DASYOUGO`.
5. Using the quick performance feedback, return to step 3 and iterate until you are sure that the performance is as good as possible.

See Also

[Options to Reduce Search Time](#)

Options to Reduce Search Time

Running large problems to completion on large numbers of nodes can take many hours. The search space for MP LINPACK is also large: not only can you run any size problem, but over a number of block sizes, grid layouts, lookahead steps, using different factorization methods, and so on. It can be a large waste of time to run a large problem to completion only to discover it ran 0.01% slower than your previous best problem.

Use the following options to reduce the search time:

- `-DASYOUGO`
- `-DENDEARLY`
- `-DASYOUGO2`

Use `-DASYOUGO2` cautiously because it does have a marginal performance impact. To see `DGEMM` internal performance, compile with `-DASYOUGO2` and `-DASYOUGO2_DISPLAY`. These options provide a lot of useful `DGEMM` performance information at the cost of around 0.2% performance loss.

If you want to use the old HPL, simply omit these options and recompile from scratch. To do this, try `"nmake arch=<arch> clean_arch_all"`.

-DASYOUGO

`-DASYOUGO` gives performance data as the run proceeds. The performance always starts off higher and then drops because this actually happens in LU decomposition (a decomposition of a matrix into a product of a lower (L) and upper (U) triangular matrices). The `ASYOUGO` performance estimate is usually an overestimate (because the LU decomposition slows down as it goes), but it gets more accurate as the problem proceeds. The greater the lookahead step, the less accurate the first number may be. `ASYOUGO` tries to estimate where one is in the LU decomposition that MP LINPACK performs and this is always an overestimate as compared to `ASYOUGO2`, which measures actually achieved `DGEMM` performance. Note that the `ASYOUGO` output is a subset of the information that `ASYOUGO2` provides. So, refer to the description of the `-DASYOUGO2` option below for the details of the output.

-DENDEARLY

`-DENDEARLY` terminates the problem after a few steps, so that you can set up 10 or 20 HPL runs without monitoring them, see how they all do, and then only run the fastest ones to completion. `-DENDEARLY` assumes `-DASYOUGO`. You do not need to define both, although it doesn't hurt. To avoid the residual check for a problem that terminates early, set the "threshold" parameter in `HPL.dat` to a negative number when testing `ENDEARLY`. It also sometimes gives a better picture to compile with `-DASYOUGO2` when using `-DENDEARLY`.

Usage notes on `-DENDEARLY` follow:

- `-DENDEARLY` stops the problem after a few iterations of `DGEMM` on the block size (the bigger the blocksize, the further it gets). It prints only 5 or 6 "updates", whereas `-DASYOUGO` prints about 46 or so output elements before the problem completes.
- Performance for `-DASYOUGO` and `-DENDEARLY` always starts off at one speed, slowly increases, and then slows down toward the end (because that is what LU does). `-DENDEARLY` is likely to terminate before it starts to slow down.
- `-DENDEARLY` terminates the problem early with an HPL Error exit. It means that you need to ignore the missing residual results, which are wrong because the problem never completed. However, you can get an idea what the initial performance was, and if it looks good, then run the problem to completion without `-DENDEARLY`. To avoid the error check, you can set HPL's threshold parameter in `HPL.dat` to a negative number.
- Though `-DENDEARLY` terminates early, HPL treats the problem as completed and computes Gflop rating as though the problem ran to completion. Ignore this erroneously high rating.
- The bigger the problem, the more accurately the last update that `-DENDEARLY` returns is close to what happens when the problem runs to completion. `-DENDEARLY` is a poor approximation for small problems. It is for this reason that you are suggested to use `ENDEARLY` in conjunction with `ASYOUGO2`, because `ASYOUGO2` reports actual `DGEMM` performance, which can be a closer approximation to problems just starting.

-DASYOUGO2

-DASYOUGO2 gives detailed single-node DGEMM performance information. It captures all DGEMM calls (if you use Fortran BLAS) and records their data. Because of this, the routine has a marginal intrusive overhead. Unlike -DASYOUGO, which is quite non-intrusive, -DASYOUGO2 interrupts every DGEMM call to monitor its performance. You should beware of this overhead, although for big problems, it is, less than 0.1%.

Here is a sample ASYOUGO2 output (the first 3 non-intrusive numbers can be found in ASYOUGO and ENDEARLY), so it suffices to describe these numbers here:

```
Col=001280 Fract=0.050 Mflops=42454.99 (DT=9.5 DF=34.1 DMF=38322.78).
```

The problem size was N=16000 with a block size of 128. After 10 blocks, that is, 1280 columns, an output was sent to the screen. Here, the fraction of columns completed is 1280/16000=0.08. Only up to 40 outputs are printed, at various places through the matrix decomposition: fractions

```
0.005 0.010 0.015 0.020 0.025 0.030 0.035 0.040 0.045 0.050 0.055 0.060 0.065 0.070
0.075 0.080 0.085 0.090 0.095 0.100 0.105 0.110 0.115 0.120 0.125 0.130 0.135 0.140
0.145 0.150 0.155 0.160 0.165 0.170 0.175 0.180 0.185 0.190 0.195 0.200 0.205 0.210
0.215 0.220 0.225 0.230 0.235 0.240 0.245 0.250 0.255 0.260 0.265 0.270 0.275 0.280
0.285 0.290 0.295 0.300 0.305 0.310 0.315 0.320 0.325 0.330 0.335 0.340 0.345 0.350
0.355 0.360 0.365 0.370 0.375 0.380 0.385 0.390 0.395 0.400 0.405 0.410 0.415 0.420
0.425 0.430 0.435 0.440 0.445 0.450 0.455 0.460 0.465 0.470 0.475 0.480 0.485 0.490
0.495 0.515 0.535 0.555 0.575 0.595 0.615 0.635 0.655 0.675 0.695 0.795 0.895.
```

However, this problem size is so small and the block size so big by comparison that as soon as it prints the value for 0.045, it was already through 0.08 fraction of the columns. On a really big problem, the fractional number will be more accurate. It never prints more than the 112 numbers above. So, smaller problems will have fewer than 112 updates, and the biggest problems will have precisely 112 updates.

Mflops is an estimate based on 1280 columns of LU being completed. However, with lookahead steps, sometimes that work is not actually completed when the output is made. Nevertheless, this is a good estimate for comparing identical runs.

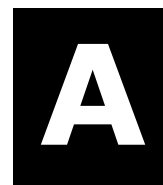
The 3 numbers in parenthesis are intrusive ASYOUGO2 addins. DT is the total time processor 0 has spent in DGEMM. DF is the number of billion operations that have been performed in DGEMM by one processor. Hence, the performance of processor 0 (in Gflops) in DGEMM is always DF/DT. Using the number of DGEMM flops as a basis instead of the number of LU flops, you get a lower bound on performance of the run by looking at DMF, which can be compared to Mflops above (It uses the global LU time, but the DGEMM flops are computed under the assumption that the problem is evenly distributed amongst the nodes, as only HPL's node (0,0) returns any output.)

Note that when using the above performance monitoring tools to compare different HPL.dat input data sets, you should be aware that the pattern of performance drop-off that LU experiences is sensitive to some input data. For instance, when you try very small problems, the performance drop-off from the initial values to end values is very rapid. The larger the problem, the less the drop-off, and it is probably safe to use the first few performance values to estimate the difference between a problem size 700000 and 701000, for instance. Another factor that influences the performance drop-off is the grid dimensions (P and Q). For big problems, the performance tends to fall off less from the first few steps when P and Q are roughly equal in value. You can make use of a large number of parameters, such as broadcast types, and change them so that the final performance is determined very closely by the first few steps.

Using these tools will greatly assist the amount of data you can test.

See Also
[Benchmarking a Cluster](#)

Intel® Math Kernel Library Language Interfaces Support



Language Interfaces Support, by Function Domain

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

Function Domain	FORTRAN 77 interface	Fortran 90/95 interface	C/C++ interface
Basic Linear Algebra Subprograms (BLAS)	Yes	Yes	via CBLAS
BLAS-like extension transposition routines	Yes		Yes
Sparse BLAS Level 1	Yes	Yes	via CBLAS
Sparse BLAS Level 2 and 3	Yes	Yes	Yes
LAPACK routines for solving systems of linear equations	Yes	Yes	Yes
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	Yes	Yes	Yes
Auxiliary and utility LAPACK routines	Yes		Yes
Parallel Basic Linear Algebra Subprograms (PBLAS)	Yes		
ScaLAPACK 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
Cluster FFT functions		Yes	Yes
Trigonometric Transform routines		Yes	Yes
Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines		Yes	Yes
Optimization (Trust-Region) Solver routines	Yes	Yes	Yes
Data Fitting functions	Yes	Yes	Yes
GMP* arithmetic functions ^{††}			Yes
Support functions (including memory allocation)	Yes	Yes	Yes

† Supported using a mixed language programming call. See [Intel® MKL Include Files](#) for the respective header file.

†† GMP Arithmetic Functions are deprecated and will be removed in a future release.

Include Files

Function domain	Fortran Include Files	C/C++ Include Files
All function domains	mkl.fi	mkl.h
BLAS Routines	blas.f90 mkl_blas.fi	mkl_blas.h
BLAS-like Extension Transposition Routines	mkl_trans.fi	mkl_trans.h
CBLAS Interface to BLAS		mkl_cblas.h
Sparse BLAS Routines	mkl_spblas.fi	mkl_spblas.h
LAPACK Routines	lapack.f90 mkl_lapack.fi	mkl_lapack.h
C Interface to LAPACK		mkl_lapacke.h
ScaLAPACK Routines		mkl_scalapack.h
All Sparse Solver Routines	mkl_solver.f90	mkl_solver.h
PARDISO	mkl_pardiso.f77 mkl_pardiso.f90	mkl_pardiso.h
DSS Interface	mkl_dss.f77 mkl_dss.f90	mkl_dss.h
RCI Iterative Solvers ILU Factorization	mkl_rci.fi	mkl_rci.h
Optimization Solver Routines	mkl_rci.fi	mkl_rci.h
Vector Mathematical Functions	mkl_vml.f77 mkl_vml.90	mkl_vml.h
Vector Statistical Functions	mkl_vsl.f77 mkl_vsl.f90	mkl_vsl_functions.h
Fourier Transform Functions	mkl_dfti.f90	mkl_dfti.h
Cluster Fourier Transform Functions	mkl_cdft.f90	mkl_cdft.h
Partial Differential Equations Support Routines		
Trigonometric Transforms	mkl_trig_transforms.f90	mkl_trig_transforms.h
Poisson Solvers	mkl_poisson.f90	mkl_poisson.h
Data Fitting functions	mkl_df.f77 mkl_df.f90	mkl_df.h
GMP interface [†]		mkl_gmp.h
Support functions	mkl_service.f90	mkl_service.h

Function domain	Fortran Include Files	C/C++ Include Files
	<code>mkl_service.fi</code>	
Memory allocation routines		<code>i_malloc.h</code>
Intel MKL examples interface		<code>mkl_example.h</code>

† GMP Arithmetic Functions are deprecated and will be removed in a future release.

See Also

[Language Interfaces Support, by Function Domain](#)

Support for Third-Party Interfaces

FFTW Interface Support

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

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



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

GMP* Functions

Intel® Math Kernel Library (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://software.intel.com/sites/products/documentation/hpc/mkl/gnump/index.htm>.



NOTE Intel MKL GMP Arithmetic Functions are deprecated and will be removed in a future release.

If you currently use the GMP* library, you need to modify `INCLUDE` statements in your programs to `mk1_gmp.h`.



Directory Structure in Detail

Tables in this section show contents of the Intel(R) Math Kernel Library (Intel(R) MKL) architecture-specific directories.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Detailed Structure of the IA-32 Architecture Directories

Static Libraries in the `lib\ia32` Directory

File	Contents
Interface layer	
<code>mkl_intel_c.lib</code>	cdecl interface library
<code>mkl_intel_s.lib</code>	CVF default interface library
<code>mkl_blas95.lib</code>	Fortran 95 interface library for BLAS. Supports the Intel® Fortran compiler
<code>mkl_lapack95.lib</code>	Fortran 95 interface library for LAPACK. Supports the Intel® Fortran compiler
Threading layer	
<code>mkl_intel_thread.lib</code>	Threading library for the Intel compilers
<code>mkl_pgi_thread.lib</code>	Threading library for the PGI* compiler
<code>mkl_sequential.lib</code>	Sequential library
Computational layer	
<code>mkl_core.lib</code>	Kernel library for IA-32 architecture
<code>mkl_solver.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_solver_sequential.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_scalapack_core.lib</code>	ScaLAPACK routines
<code>mkl_cdft_core.lib</code>	Cluster version of FFTs
Run-time Libraries (RTL)	

File	Contents
mkl_blacs_intelmpi.lib	BLACS routines supporting Intel MPI
mkl_blacs_mpich2.lib	BLACS routines supporting MPICH2

Dynamic Libraries in the `lib\ia32` Directory

File	Contents
mkl_rt.lib	Single Dynamic Library to be used for linking
Interface layer	
mkl_intel_c_dll.lib	cdecl interface library for dynamic linking
mkl_intel_s_dll.lib	CVF default interface library for dynamic linking
Threading layer	
mkl_intel_thread_dll.lib	Threading library for dynamic linking with the Intel compilers
mkl_pgi_thread_dll.lib	Threading library for dynamic linking with the PGI* compiler
mkl_sequential_dll.lib	Sequential library for dynamic linking
Computational layer	
mkl_core_dll.lib	Core library for dynamic linking
mkl_scalapack_core_dll.lib	ScaLAPACK routine library for dynamic linking
mkl_cdft_core_dll.lib	Cluster FFT library for dynamic linking
Run-time Libraries (RTL)	
mkl_blacs_dll.lib	BLACS interface library for dynamic linking

Contents of the `redist\ia32\mkl` Directory

File	Contents
mkl_rt.dll	Single Dynamic Library
Threading layer	
mkl_intel_thread.dll	Dynamic threading library for the Intel compilers
mkl_pgi_thread.dll	Dynamic threading library for the PGI* compiler
mkl_sequential.dll	Dynamic sequential library
Computational layer	
mkl_core.dll	Core library containing processor-independent code and a dispatcher for dynamic loading of processor-specific code
mkl_def.dll	Default kernel (Intel® Pentium®, Pentium® Pro, Pentium® II, and Pentium® III processors)

File	Contents
mk1_p4.dll	Pentium® 4 processor kernel
mk1_p4p.dll	Kernel for the Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), including Intel® Core™ Duo and Intel® Core™ Solo processors.
mk1_p4m.dll	Kernel for processors based on the Intel® Core™ microarchitecture (except Intel® Core™ Duo and Intel® Core™ Solo processors, for which mk1_p4p.dll is intended)
mk1_p4m3.dll	Kernel for the Intel® Core™ i7 processors
mk1_vml_def.dll	Vector Math Library (VML)/Vector Statistical Library (VSL)/Data Fitting (DF) part of default kernel for old Intel® Pentium® processors
mk1_vml_ia.dll	VML/VSL/DF default kernel for newer Intel® architecture processors
mk1_vml_p4.dll	VML/VSL/DF part of Pentium® 4 processor kernel
mk1_vml_p4p.dll	VML/VSL/DF for Pentium® 4 processor with Intel SSE3
mk1_vml_p4m.dll	VML/VSL/DF for processors based on the Intel® Core™ microarchitecture (except Intel® Core™ Duo and Intel® Core™ Solo processors, for which mk1_vml_p4p.dll is intended).
mk1_vml_p4m2.dll	VML/VSL/DF for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
mk1_vml_p4m3.dll	VML/VSL/DF for the Intel® Core™ i7 processors
mk1_vml_avx.dll	VML/VSL/DF optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
mk1_scalapack_core.dll	ScaLAPACK routines
mk1_cdft_core.dll	Cluster FFT dynamic library
libimalloc.dll	Dynamic library to support renaming of memory functions
Run-time Libraries (RTL)	
mk1_blacs.dll	BLACS routines
mk1_blacs_intelmpi.dll	BLACS routines supporting Intel MPI
mk1_blacs_mpich2.dll	BLACS routines supporting MPICH2
1033\mk1_msg.dll	Catalog of Intel® Math Kernel Library (Intel® MKL) messages in English
1041\mk1_msg.dll	Catalog of Intel MKL messages in Japanese. Available only if the Intel® C++ Composer XE or Intel® Visual Fortran Composer XE that includes Intel MKL provides Japanese localization. Please see the Release Notes for this information.

Detailed Structure of the Intel® 64 Architecture Directories

Static Libraries in the `lib\intel64` Directory

File	Contents
Interface layer	
<code>mkl_intel_lp64.lib</code>	LP64 interface library for the Intel compilers
<code>mkl_intel_ilp64.lib</code>	ILP64 interface library for the Intel compilers
<code>mkl_intel_sp2dp.a</code>	SP2DP interface library for the Intel compilers
<code>mkl_blas95_lp64.lib</code>	Fortran 95 interface library for BLAS. Supports the Intel® Fortran compiler and LP64 interface
<code>mkl_blas95_ilp64.lib</code>	Fortran 95 interface library for BLAS. Supports the Intel® Fortran compiler and ILP64 interface
<code>mkl_lapack95_lp64.lib</code>	Fortran 95 interface library for LAPACK. Supports the Intel® Fortran compiler and LP64 interface
<code>mkl_lapack95_ilp64.lib</code>	Fortran 95 interface library for LAPACK. Supports the Intel® Fortran compiler and ILP64 interface
Threading layer	
<code>mkl_intel_thread.lib</code>	Threading library for the Intel compilers
<code>mkl_pgi_thread.lib</code>	Threading library for the PGI* compiler
<code>mkl_sequential.lib</code>	Sequential library
Computational layer	
<code>mkl_core.lib</code>	Kernel library for the Intel® 64 architecture
<code>mkl_solver_lp64.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_solver_lp64_sequential.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_solver_ilp64.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_solver_ilp64_sequential.lib</code>	Deprecated. Empty library for backward compatibility
<code>mkl_scalapack_lp64.lib</code>	ScaLAPACK routine library supporting the LP64 interface
<code>mkl_scalapack_ilp64.lib</code>	ScaLAPACK routine library supporting the ILP64 interface
<code>mkl_cdft_core.lib</code>	Cluster version of FFTs
Run-time Libraries (RTL)	
<code>mkl_blacs_intelmpi_lp64.lib</code>	LP64 version of BLACS routines supporting Intel MPI
<code>mkl_blacs_intelmpi_ilp64.lib</code>	ILP64 version of BLACS routines supporting Intel MPI
<code>mkl_blacs_mpich2_lp64.lib</code>	LP64 version of BLACS routines supporting MPICH2
<code>mkl_blacs_mpich2_ilp64.lib</code>	ILP64 version of BLACS routines supporting MPICH2
<code>mkl_blacs_msmpi_lp64.lib</code>	LP64 version of BLACS routines supporting Microsoft* MPI
<code>mkl_blacs_msmpi_ilp64.lib</code>	ILP64 version of BLACS routines supporting Microsoft* MPI

Dynamic Libraries in the `lib\intel64` Directory

File	Contents
<code>mkl_rt.lib</code>	Single Dynamic Library to be used for linking
Interface layer	
<code>mkl_intel_lp64_dll.lib</code>	LP64 interface library for dynamic linking with the Intel compilers
<code>mkl_intel_ilp64_dll.lib</code>	ILP64 interface library for dynamic linking with the Intel compilers
Threading layer	
<code>mkl_intel_thread_dll.lib</code>	Threading library for dynamic linking with the Intel compilers
<code>mkl_pgi_thread_dll.lib</code>	Threading library for dynamic linking with the PGI* compiler
<code>mkl_sequential_dll.lib</code>	Sequential library for dynamic linking
Computational layer	
<code>mkl_core_dll.lib</code>	Core library for dynamic linking
<code>mkl_scalapack_lp64_dll.lib</code>	ScaLAPACK routine library for dynamic linking supporting the LP64 interface
<code>mkl_scalapack_ilp64_dll.lib</code>	ScaLAPACK routine library for dynamic linking supporting the ILP64 interface
<code>mkl_cdft_core_dll.lib</code>	Cluster FFT library for dynamic linking
Run-time Libraries (RTL)	
<code>mkl_blacs_lp64_dll.lib</code>	LP64 version of BLACS interface library for dynamic linking
<code>mkl_blacs_ilp64_dll.lib</code>	ILP64 version of BLACS interface library for dynamic linking

Contents of the `redist\intel64\mkl` Directory

File	Contents
<code>mkl_rt.dll</code>	Single Dynamic Library
Threading layer	
<code>mkl_intel_thread.dll</code>	Dynamic threading library for the Intel compilers
<code>mkl_pgi_thread.dll</code>	Dynamic threading library for the PGI* compiler
<code>mkl_sequential.dll</code>	Dynamic sequential library
Computational layer	

File	Contents
mkl_core.dll	Core library containing processor-independent code and a dispatcher for dynamic loading of processor-specific code
mkl_def.dll	Default kernel for the Intel® 64 architecture
mkl_p4n.dll	Kernel for the Intel® Xeon® processor using the Intel® 64 architecture
mkl_mc.dll	Kernel for processors based on the Intel® Core™ microarchitecture
mkl_mc3.dll	Kernel for the Intel® Core™ i7 processors
mkl_avx.dll	Kernel optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
mkl_vml_def.dll	Vector Math Library (VML)/Vector Statistical Library (VSL)/Data Fitting (DF) part of default kernel
mkl_vml_p4n.dll	VML/VSL/DF for the Intel® Xeon® processor using the Intel® 64 architecture
mkl_vml_mc.dll	VML/VSL/DF for processors based on the Intel® Core™ microarchitecture
mkl_vml_mc2.dll	VML/VSL/DF for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
mkl_vml_mc3.dll	VML/VSL/DF for the Intel® Core® i7 processors
mkl_vml_avx.dll	VML/VSL/DF optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
mkl_scalapack_lp64.dll	ScaLAPACK routine library supporting the LP64 interface
mkl_scalapack_ilp64.dll	ScaLAPACK routine library supporting the ILP64 interface
mkl_cdft_core.dll	Cluster FFT dynamic library
libimalloc.dll	Dynamic library to support renaming of memory functions
Run-time Libraries (RTL)	
mkl_blacs_lp64.dll	LP64 version of BLACS routines
mkl_blacs_ilp64.dll	ILP64 version of BLACS routines
mkl_blacs_intelmpi_lp64.dll	LP64 version of BLACS routines supporting Intel MPI
mkl_blacs_intelmpi_ilp64.dll	ILP64 version of BLACS routines supporting Intel MPI
mkl_blacs_mpich2_lp64.dll	LP64 version of BLACS routines supporting MPICH2
mkl_blacs_mpich2_ilp64.dll	ILP64 version of BLACS routines supporting MPICH2
mkl_blacs_msmapi_lp64.dll	LP64 version of BLACS routines supporting Microsoft* MPI
mkl_blacs_msmapi_ilp64.dll	ILP64 version of BLACS routines supporting Microsoft* MPI
1033\mkl_msg.dll	Catalog of Intel® Math Kernel Library (Intel® MKL) messages in English

File	Contents
1041\mk1_msg.dll	Catalog of Intel MKL messages in Japanese. Available only if the Intel® C++ Composer XE or Intel® Visual Fortran Composer XE that includes Intel MKL provides Japanese localization. Please see the Release Notes for this information.

Index

A

affinity mask 55
aligning data, example 71
architecture support 23

B

BLAS
 calling routines from C 63
 Fortran 95 interface to 61
 threaded routines 45
building a custom DLL
 in Visual Studio* IDE 42

C

C interface to LAPACK, use of 63
C, calling LAPACK, BLAS, CBLAS from 63
C/C++, Intel(R) MKL complex types 64
calling
 BLAS functions from C 65
 CBLAS interface from C 65
 complex BLAS Level 1 function from C 65
 complex BLAS Level 1 function from C++ 65
 Fortran-style routines from C 63
calling convention, cdecl and stdcall 19
CBLAS interface, use of 63
cdecl interface, use of 33
Cluster FFT, linking with 75
cluster software, Intel(R) MKL
cluster software, linking with
 commands 75
 linking examples 78
code examples, use of 19
coding
 data alignment
 techniques to improve performance 54
compilation, Intel(R) MKL version-dependent 72
compiler run-time libraries, linking with 38
compiler support 19
compiler-dependent function 61
complex types in C and C++, Intel(R) MKL 64
computation results, consistency 71
computational libraries, linking with 37
conditional compilation 72
configuring
 Intel(R) Visual Fortran 81
 Microsoft Visual* C/C++ 81
 project that runs Intel(R) MKL code example in Visual
 Studio* 2008 IDE 82
consistent results 71
context-sensitive Help, for Intel(R) MKL, in Visual Studio*
 IDE 87
conventions, notational 13
ctdcall interface, use of 33
custom DLL
 building 39
 composing list of functions 41
 specifying function names 41
CVF calling convention, use with Intel(R) MKL 62

D

data alignment, example 71

denormal number, performance 56
directory structure
 documentation 26
 high-level 23
 in-detail
documentation directories, contents 26

E

Enter index keyword 27
environment variables, setting 17
examples, linking
 for cluster software 78
 general 30

F

FFT interface
 data alignment 54
 optimised radices 56
 threaded problems 45
FFTW interface support 103
Fortran 95 interface libraries 36

G

GNU* Multiple Precision Arithmetic Library 103

H

header files, Intel(R) MKL 100
Help, for Intel(R) MKL in Visual Studio* IDE 86
HT technology, configuration tip 54
hybrid, version, of MP LINPACK 93

I

ILP64 programming, support for 34
include files, Intel(R) MKL 100
installation, checking 17
Intel(R) Hyper-Threading Technology, configuration tip 54
Intel(R) Visual* Fortran project, linking with Intel(R) MKL
 28
IntelliSense*, with Intel(R) MKL, in Visual Studio* IDE 88
interface
 cdecl and stdcall, use of 33
 Fortran 95, libraries 36
 LP64 and ILP64, use of 34
interface libraries and modules, Intel(R) MKL 59
interface libraries, linking with 33

J

Java* examples 68

L

language interfaces support 99
language-specific interfaces
 interface libraries and modules 59

LAPACK

- C interface to, use of 63
- calling routines from C 63
- Fortran 95 interface to 61
- performance of packed routines 54
- threaded routines 45

layers, Intel(R) MKL structure 25

libraries to link with

- computational 37
- interface 33
- run-time 38
- system libraries 39
- threading 36

link tool, command line 30

linking

- Intel(R) Visual* Fortran project with Intel(R) MKL 28
- Microsoft Visual* C/C++ project with Intel(R) MKL 28

linking examples

- cluster software 78
- general 30

linking with

- compiler run-time libraries 38
- computational libraries 37
- interface libraries 33
- system libraries 39
- threading libraries 36

linking, quick start 27

linking, Web-based advisor 29

LINPACK benchmark 91

M

memory functions, redefining 56

memory management 56

memory renaming 56

Microsoft Visual* C/C++ project, linking with Intel(R) MKL 28

mixed-language programming 63

module, Fortran 95 61

MP LINPACK benchmark 93

multi-core performance 55

N

notational conventions 13

number of threads

- changing at run time 48
- changing with OpenMP* environment variable 48
- Intel(R) MKL choice, particular cases 51
- setting for cluster 77
- techniques to set 48

P

parallel performance 47

parallelism, of Intel(R) MKL 45

performance

- multi-core 55
- with denormals 56
- with subnormals 56

S

ScaLAPACK, linking with 75

SDL 28, 32

sequential mode of Intel(R) MKL 36

Single Dynamic Library 28, 32

stdcall calling convention, use in C/C++ 62

structure

- high-level 23
- in-detail
- model 25

support, technical 11

supported architectures 23

system libraries, linking with 39

T

technical support 11

thread safety, of Intel(R) MKL 45

threaded functions 45

threaded problems 45

threading control, Intel(R) MKL-specific 50

threading libraries, linking with 36

U

uBLAS, matrix-matrix multiplication, substitution with Intel MKL functions 66

unstable output, getting rid of 71

usage information 15

V

Visual Studio* 2008 IDE, configuring a project that runs Intel(R) MKL code example 82

Visual Studio* IDE

- IntelliSense*, with Intel(R) MKL 88
- using Intel(R) MKL context-sensitive Help in 87
- Viewing Intel(R) MKL documentation in 86