Software Development Kit for Multicore Acceleration
Version 3.0



Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference

SC33-8426-00

Software Development Kit for Multicore Acceleration
Version 3.0



Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference

SC33-8426-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 63.

Edition notice

This edition applies to version 3.0, release 0, modification 0, fix pack 3 of the IBM Software Development Kit for Multicore Acceleration (Product number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2007, 2008. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this publication v
How to send your comments vi

Part 1. Introduction to BLAS. 1

Part 2. Installing and configuring the BLAS library 3

Chapter 1. Package descriptions 5

Part 3. Programming 7

Chapter 2. Basic structure of the BLAS library 9

Chapter 3. Using the BLAS library (PPE interface) 11
Input requirements 11
Programming samples 11

Chapter 4. Tuning the BLAS library for performance 13
Programming tips to achieve maximum performance 14

Part 4. Programming Cell/B.E. applications 17

Chapter 5. Creating SPE threads . . . 19

Chapter 6. Support of user-specified SPE and memory callbacks 21

Chapter 7. Debugging tips 23

Part 5. BLAS API reference 25

Chapter 8. PPE APIs 27

Chapter 9. SPE APIs 29
sscal_spu 30
scopy_spu 31
saxpy_spu 32
sdot_spu 33
isamax_spu 34
sgemv_spu 35
sgemm_spu 36
ssyrk_64x64 37
strsm_spu 38
strsm_64x64 39

Chapter 10. Additional APIs 41
SPE management APIs 41
spes_info_handle_t 42
spe_info_handle_t 43
BLAS_NUM_SPES_CB 44
BLAS_GET_SPE_INFO_CB 45
BLAS_SPE_SCHEDULE_CB 46
BLAS_SPE_WAIT_CB 47
BLAS_REGISTER_SPE 48
SPE management with multi-threaded applications 51
Memory management APIs 51
BLAS_Malloc_CB 52
BLAS_Free_CB 53
BLAS_REGISTER_MEM 54

Part 6. Appendixes. 57

Appendix A. Related documentation . . 59

Appendix B. Accessibility features . . . 61

Notices 63
Trademarks 65
Terms and conditions 65

Glossary 67

Index 69

© Copyright IBM Corp. 2007, 2008

About this publication

This publication describes in detail how to configure the Basic Linear Algebra Subprograms (BLAS) library and how to program applications using it on the IBM Software Development Kit for Multicore Acceleration (SDK). It contains detailed reference information about the APIs for the library as well as sample applications showing usage of these APIs.

Who should use this book

The target audience for this document is application programmers using the SDK. You are expected to have a basic understanding of programming on the Cell Broadband Engine™ (Cell/B.E.) platform and common terminology used with the Cell/B.E. platform.

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
Bold	Lowercase commands, library functions.	**void sscal_spu (float *sx, float sa, int n)**
Italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	The following example shows how a test program, *test_name* can be run
`Monospace`	Examples of program code or command strings.	`int main()`

Related information

For a list of SDK documentation, see Appendix A, "Related documentation," on page 59.

In addition the following documents about BLAS are available from the World Wide Web:

- Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard, August 2001, see

  `http://www.netlib.org/blas/blast-forum/blas-report.pdf`
- A Set of Level 3 Basic Linear Algebra Subprograms, Jack Dongarra, Jeremy Du Croz, Iain Duff, Sven Hammarling, August 1998, see

  `http://www.netlib.org/blas/blas3-paper.ps`
- Basic Linear Algebra Subprograms – A quick reference guide, May 1997, see

  `http://www.netlib.org/blas/blasqr.pdf`

© Copyright IBM Corp. 2007, 2008

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this publication, send your comments using IBM Resource Link™ at http://www.ibm.com/servers/resourcelink. Click **Feedback** on the navigation pane. Be sure to include the name of the book, the form number of the book, and the specific location of the text you are commenting on (for example, a page number or table number).

Part 1. Introduction to BLAS

The BLAS library is based upon a published standard interface, see the BLAS Technical Forum Standard document available at

`http://www.netlib.org/blas/blast-forum/blas-report.pdf`

for commonly-used linear algebra operations in high-performance computing (HPC) and other scientific domains.

It is widely used as the basis for other high quality linear algebra software, for example LAPACK and ScaLAPACK. The Linpack (HPL) benchmark largely depends on a single BLAS routine (DGEMM) for good performance.

The BLAS APIs are available as standard ANSI C and standard FORTRAN 77/90 interfaces. BLAS implementations are also available in open-source (netlib.org). Based on their functionality, BLAS routines are categorized into the following three levels:

- Level 1 routines are for scalar and vector operations
- Level 2 routines are for matrix-vector operations
- Level 3 routines are for matrix-matrix operations

BLAS routines can have up to four versions – real single precision, real double precision, complex single precision and complex double precision, represented by prefixing S, D, C and Z respectively to the routine name.

The BLAS library in the SDK supports only real single precision (SP) and real double precision (DP) versions. All SP and DP routines in the three levels of standard BLAS are supported on the Power Processing Element (PPE). These are available as PPE APIs and conform to the standard BLAS interface. (Refer to `http://www.netlib.org/blas/blasqr.pdf`)

Some of these routines have been optimized using the Synergistic Processing Elements (SPEs) and these exhibit substantially better performance in comparison to the corresponding versions implemented solely on the PPE. An SPE interface in addition to the PPE interface is provided for some of these routines; however, the SPE interface does not conform to the standard BLAS interface and provides a restricted version of the standard BLAS interface.

The following routines have been optimized to use the SPEs:

- Level 1:
  - SSCAL, DSCAL
  - SCOPY, DCOPY
  - ISAMAX, IDAMAX
  - SAXPY, DAXPY
  - SDOT, DDOT
- Level 2:
  - SGEMV (TRANS='No Transpose' and INCY=1)
  - DGEMV
  - DTRMV

© Copyright IBM Corp. 2007, 2008

  - DTRSV
- Level 3:
  - SGEMM, DGEMM
  - SSYRK, (Only for UPLO='Lower' and TRANS='No transpose')
  - DSYRK
  - STRSM (Only for SIDE='Right', UPLO='Lower', TRANS='Transpose' and DIAG='Non-Unit')
  - DTRSM
  - DTRMM
  - DSYMM

These routines support implementations of BLAS-3-based Cholesky and BLAS-1 based LU factorization.

The parameter restrictions, as mentioned above for some of the optimized BLAS level 2 and 3 routines, apply to the FORTRAN compatible C interface for these routines. The FORTRAN compatible C interface accepts input matrices stored in column-major order.

Part 2. Installing and configuring the BLAS library

The following sections describe how to install and configure the BLAS library.

• Chapter 1, “Package descriptions,” on page 5

© Copyright IBM Corp. 2007, 2008

Chapter 1. Package descriptions

The BLAS library can be installed on various platforms using the following packages:

Table 2. BLAS library installation packages

Package	Purpose	Platform	Contents
blas-3.0-x.ppc.rpm where x is the build date	Installs BLAS library **libblas.so.x.y** where **x** and **y** are the major and minor version numbers respectively:	IBM PowerPC Architecture™ and IBM BladeCenter® QS20, IBM BladeCenter QS21, IBM BladeCenter QS22.	/usr/lib/libblas.so.x.y : BLAS library. /usr/lib/libblas.so.x: Soft link to libblas.so.x.y
blas-devel-3.0-x.ppc.rpm	Installs supporting files such as header files and sample applications of BLAS for developing applications using the BLAS library.	PowerPC Architecture and IBM BladeCenter QS20, IBM BladeCenter QS21, IBM BladeCenter QS22.	/usr/include/blas.h: Contains prototypes of all BLAS Level 1, 2 and 3 (SP and DP) functions that have PPE APIs in the library. PPE APIs refer to Standard BLAS APIs on the PPE. /usr/include/blas_callback.h: Contains prototypes of functions that can be used to register user-specified SPE thread creation and memory allocation callbacks. /usr/include/cblas.h: Contains prototypes of all the C-interface versions of BLAS Level 1, 2 and 3 (SP and DP) functions that have a PPE API in the library. /usr/lib/libblas.so: Soft link to the soft link libblas.so.x /usr/spu/include/blas_s.h: Contains prototypes of selected functions of BLAS Level 1, 2 and 3 that have an SPE API in the library. These functions have limited functionality and are not as generic as the PPE APIs. /usr/spu/lib/libblas.a: BLAS SPE library. /opt/cell/sdk/src/blas.tar: Compressed file of BLAS samples.
blas-3.0-x.ppc64.rpm where x is the build date	Installs the BLAS library **libblas.so.x.y** where x and y are the major and minor version numbers respectively	PowerPC Architecture-64 bit and IBM BladeCenter QS20, IBM BladeCenter QS21, IBM BladeCenter QS22.	/usr/lib64/libblas.so.x.y : BLAS library. /usr/lib64/libblas.so.x: Soft link to libblas.so.x.y
blas-devel-3.0-x.ppc64.rpm	Installs supporting files such as header files and sample applications of BLAS for developing applications using the BLAS library.	PowerPC Architecture-64 bit and IBM BladeCenter QS20, IBM BladeCenter QS21, IBM BladeCenter QS22.	/usr/lib64/libblas.so: Soft link to the soft link libblas.so.x

© Copyright IBM Corp. 2007, 2008

Table 2. BLAS library installation packages (continued)

Package	Purpose	Platform	Contents
blas-cross-devel-3.0-x.noarch.rpm where x is the build date	Installs the BLAS library, supporting files such as header files and sample applications of the BLAS library.	Other platforms, such as x86 series.	/opt/cell/sysroot/usr/include/blas.h: Contains prototypes of all BLAS Level 1, 2 and 3 (SP and DP) functions supported in the library with PPE APIs. /opt/cell/sysroot/usr/include/blas_callback.h: Contains prototypes of functions that can be used to register user-specified SPE thread creation and memory allocation callbacks. /opt/cell/sysroot/usr/include/cblas.h: Contains prototypes of all C-interface versions of BLAS level 1, 2 and 3 (SP and DP) functions supported in the library with the PPE APIs. /opt/cell/sysroot/usr/lib/libblas.so: Soft link to the soft link libblas.so.x /opt/cell/sysroot/usr/lib/libblas.so.x: Soft link to libblas.so.x.y /opt/cell/sysroot/usr/lib/libblas.so.x.y: BLAS library. /opt/cell/sysroot/usr/lib64/libblas.so: Soft link to the soft link libblas.so.x (64 bit). /opt/cell/sysroot/usr/lib64/libblas.so.x: Soft link to the soft link libblas.so.x.y (64 bit). /opt/cell/sysroot/usr/lib64/libblas.so.x.y: BLAS library (64 bit). /opt/cell/sysroot/usr/spu/include/blas_s.h: Contains prototypes of selected functions of BLAS Level 1, 2 and 3 that have an SPE API in the library. These functions have limited functionality and are not as generic as the PPE APIs. /usr/spu/lib/libblas.a: BLAS SPU library. /opt/cell/sdk/src/blas.tar: Compressed file of BLAS samples.

Part 3. Programming

The following sections provide information about programming with the BLAS library:

- Chapter 2, "Basic structure of the BLAS library," on page 9
- Chapter 3, "Using the BLAS library (PPE interface)," on page 11
- Chapter 4, "Tuning the BLAS library for performance," on page 13

© Copyright IBM Corp. 2007, 2008

Chapter 2. Basic structure of the BLAS library

The BLAS Library has two components:

- PPE interface
- SPE interface

PPE applications can use the standard BLAS PPE APIs (defined by BLAS Technical Forum Standard, see documents in "Related information" on page v) and the SPE programs can directly use the SPE APIs.

A detailed description of the SPE interface is provided in Chapter 9, "SPE APIs," on page 29.

© Copyright IBM Corp. 2007, 2008

Chapter 3. Using the BLAS library (PPE interface)

At the PPE level, the BLAS APIs support two different set of C interfaces to the BLAS routines:

- C interface to the legacy BLAS as set out by the BLAS Technical Forum, with prefix `cblas_` appended to the routine name, for example, `cblas_dgemm` for DGEMM routine
- FORTRAN-callable C interface with underscore ('_') suffixed to the routine name, for example, `dgemm_` for DGEMM routine

A PPE application can either use the C interface or the FORTRAN-compatible (callable) C interface on the PPE provided by the BLAS library. Both these interfaces conform to the standard BLAS interface. The C interface is built on top of FORTRAN-compatible C interface. The PPE application must include the appropriate header file (blas.h or cblas.h depending on the interface used) and must be linked with '-lblas'.

The following topics describe the input requirements and a sample application.

Input requirements

The BLAS library requires all the matrices and vectors to be naturally aligned, that is 4-byte aligned for single precision and 8-byte aligned for double precision. Cases where this is not satisfied are not supported by the library.

Programming samples

The following sample applications demonstrate the usage of the BLAS-PPE library. This application program invokes the **scopy** and **sdot** routines, using the BLAS-PPE library.

Example: Using the FORTRAN-compatible C interface

```
#include <blas.h>
#define BUF_SIZE 32

/********************** MAIN ROUTINE **********************/
int main()
{
    int  i,j ;
    int entries_x, entries_y ;
    float sa=0.1;
    float *sx, *sy ;
    int incx=1, incy=2;
    int n = BUF_SIZE;
    double result;

    entries_x = n * incx ;
    entries_y = n * incy ;

    sx = (float *) _malloc_align( entries_x * sizeof( float ), 7 ) ;
    sy = (float *) _malloc_align( entries_y * sizeof( float ), 7 ) ;

    for( i = 0 ; i < entries_x ; i++ )
         sx[i] = (float) (i) ;
    j = entries_y - 1 ;
    for( i = 0 ; i < entries_y ; i++,j-- )
```

© Copyright IBM Corp. 2007, 2008

```
            sy[i] = (float) (j) ;

    scopy_( &n, sx, &incx, sy, &incy ) ;
    result = sdot_( &n, sx, &incx, sy, &incy ) ;

    return 0;
}
```

Example: Using the C interface (cblas_*)

```
#include <cblas.h>
#define BUF_SIZE 32

/********************* MAIN ROUTINE *********************/
int main()
{
    int  i,j ;
    int entries_x, entries_y ;
    float sa=0.1;
    float *sx, *sy ;
    int incx=1, incy=2;
    int n = BUF_SIZE;
    double result;

    entries_x = n * incx ;
    entries_y = n * incy ;

    sx = (float *) _malloc_align( entries_x * sizeof( float ), 7 ) ;
    sy = (float *) _malloc_align( entries_y * sizeof( float ), 7 ) ;

    for( i = 0 ; i < entries_x ; i++ )
         sx[i] = (float) (i) ;
    j = entries_y - 1 ;
    for( i = 0 ; i < entries_y ; i++,j-- )
            sy[i] = (float) (j) ;

    cblas_scopy( n, sx, incx, sy, incy ) ;
    result = cblas_sdot( n, sx, incx, sy, incy ) ;

    return 0;
}
```

Chapter 4. Tuning the BLAS library for performance

The BLAS library provides additional features for customizing the library. You can use these features to effectively use the available resources and potentially achieve higher performance.

Swap space

The optimized BLAS level 3 routines use extra space to suitably reorganize the matrices. It is advisable to use huge pages for storing the input/output matrices as well as for storing the reorganized matrices in BLAS level 3. To achieve better performance, it is also beneficial to reuse the allocated space across multiple BLAS calls, rather than allocate fresh memory space with every call to the routine. This reuse of allocated space becomes especially useful when operating on small matrices. To overcome the overhead required for small matrices, a pre-allocated space, called *swap space*, is created only once with huge pages (and touched on the PPE). You can specify the size of swap space with the environment variable `BLAS_SWAP_SIZE`. By default no swap space is created.

When any optimized BLAS3 routine is called and if the extra space required for reorganizing the input matrices is less than the pre-allocated swap space, this swap space is used by the routine to reorganize the input matrices (instead of allocating new space).

The idea is to use swap space up to 16 MB (single huge page size), this takes care of extra space requirement for small matrices. You can achieve considerable performance improvement for small matrices through the use of swap space.

Memory bandwidth-bound and compute-bound routines

BLAS Level 1 and Level 2 routines are memory bandwidth bound in general on the Cell/B.E. processor. When the data to be processed by these routines is on the same Cell/B.E. node, the best performance is generally achieved with four or less SPEs. The performance of these routines is not expected to improve further by using more SPEs. The BLAS library internally uses the optimal number of SPEs for level 1 and 2 routines to achieve the best performance for these routines, even if more SPEs are available for its use. However, level 3 routines are generally computation-bound on the Cell/B.E. processor. The performance of these routines is expected to scale with the number of SPEs used.

Startup costs

There is a one time startup cost due to initialization and setup of memory and SPEs within the BLAS library. This one time start-up cost is incurred only when an application invokes an optimized BLAS routine for the first time. Subsequent invocations of optimized BLAS routines by the same application do not incur this cost.

Environment variables

There are many environment variables available to customize SPE and memory management in the BLAS library. However, for full control, you can register and

© Copyright IBM Corp. 2007, 2008

use your own SPE and memory callbacks (described in Chapter 10, "Additional APIs," on page 41). The following table lists the environment variables:

Table 3. Environment variables

Variable name	Purpose	Default value
BLAS_NUMSPES	Specifies the number of SPEs to be used per application thread. For example, if BLAS_NUMSPES is set to 3 and a BLAS application has 4 threads, then the BLAS routines called by each application thread uses 3 SPEs. The value of this variable is read only once inside the BLAS library and then the same value is used throughout the application lifetime. Therefore, there is no effect if this variable is changed partway through an application during runtime.	8 (SPEs in a single node).
BLAS_USE_HUGEPAGE	Specifies if the library should use huge pages or heap for allocating new space for reorganizing input matrices in BLAS3 routines. Set the variable to 0 to use heap instead of the default.	Use huge pages.
BLAS_HUGE_PAGE_SIZE	Specifies the huge page size to use, in KB. The huge page size on the system can be found in the file /proc/meminfo.	16384 KB (16 MB).
BLAS_HUGE_FILE	Specifies the name of the file to be used for allocating new space using huge pages in BLAS3 routines.	The filename is /huge/blas_lib.bin
BLAS_NUMA_NODE	Specifies the NUMA node on which SPEs are launched by default and memory is allocated by default.	NUMA node is -1 which indicates no NUMA binding.
BLAS_SWAP_SIZE	Specifies the size of swap space, in KB.	Do not use swap space.
BLAS_SWAP_NUMA_NODE	Specifies the NUMA node on which swap space is allocated.	NUMA node is -1 which indicates no NUMA binding.
BLAS_SWAP_HUGE_FILE	Specifies the name of the file that is used to allocate swap space using huge pages.	The filename is /huge/blas_lib_swap.bin

The following example shows how a test program, *test_name*, can be run with five SPEs, using binding on NUMA node 0 and 12 MB of swap space on the same NUMA node:

```
env BLAS_NUMSPES=5 BLAS_NUMA_NODE=0
BLAS_SWAP_SIZE=12288 BLAS_SWAP_NUMA_NODE=0 ./test_name
```

Programming tips to achieve maximum performance

Use the following tips to leverage maximum performance from the BLAS library:

- Make the matrices/vectors 128 byte aligned, because memory access is more efficient when the data is 128 byte aligned.
- Use huge pages to store vectors and matrices. By default, the library uses this feature for memory allocation done within the library.
- Use NUMA binding for the application and the library. Set the BLAS_NUMA_NODE environment variable to enable this feature for the library. BLAS_NUMA_NODE can

be set to 0 or 1 for a dual node system. An application can enable NUMA binding either using the command line NUMA policy tool **numactl** or NUMA policy API **libnuma** provided on Linux.

- Use the swap space feature, described in Chapter 4, "Tuning the BLAS library for performance," on page 13, for matrices smaller than 1024 (1K), with appropriate NUMA binding.
- The library gives better performance when it processes vectors and matrices of large sizes. Performance of optimized routines is better when the stride value is 1. Routines that involve matrices show good performance when the leading dimension, number of rows and columns are a multiple of 64.
- `BLAS_NUMSPES` should be set appropriately for multi-threaded applications so that there is no contention among the threads. For example, if an application has four threads and `BLAS_NUMSPES` is set to 8, then the BLAS routines called by each of the four application threads try to use 8 SPEs. This can result in contention because the maximum available SPEs on a single blade is 16. You need to take this into account when you decide on the number of `BLAS_NUMSPES` so that optimal performance is obtained.

Part 4. Programming Cell/B.E. applications

This section describes the mechanisms available in the BLAS library that offer more control to advanced programmers for management of SPEs and system memory.

The default SPE and Memory management mechanism in the BLAS library can be partially customized by the use of environment variables. However for more control, an application can design its own mechanism for managing available SPE resources and system memory to be used by BLAS routines in the library.

© Copyright IBM Corp. 2007, 2008

Chapter 5. Creating SPE threads

When a pre-built BLAS application binary (executable) is run with the BLAS library, the library internally manages SPE resources available on the system using the default SPE management routines.

This is also true for the other BLAS applications that do not intend to manage the SPEs and want to use the default SPE management provided by the BLAS library. The sample application in the Chapter 3, "Using the BLAS library (PPE interface)," on page 11 is an example of this.

For such applications, you can partially control the behavior of BLAS library by using certain environment variables as described in "Environment variables" on page 13.

© Copyright IBM Corp. 2007, 2008

Chapter 6. Support of user-specified SPE and memory callbacks

The SPE and memory management mechanism used by the BLAS library can be customized with the help of user-specified callback routines.

Instead of using default SPE management functions defined in the BLAS library, a BLAS application can register its own SPE thread management routines (for example, for creating or destroying SPE threads or both, SPE program loading or context creation). This is done with the registration function BLAS_REGISTER_SPE provided by the BLAS library.

The optimized level 3 routines in the library use some extra space for suitably reorganizing the input matrices. The library uses default memory management routines to allocate and deallocate this extra space.

Similar to the user-specified SPE management routines, you can also specify custom memory management routines. Instead of using the default memory management functions defined in BLAS library, a BLAS application can register its own memory allocation and deallocation routines for allocating new space for reorganizing the input matrices. To do this, use the registration function BLAS_REGISTER_MEM.

Default SPE and memory management routines defined in the BLAS library are registered when you do not register any routines.

© Copyright IBM Corp. 2007, 2008

Chapter 7. Debugging tips

Use the following steps to debug common errors encountered in programming with the BLAS library:

- For using huge pages, the library assumes that a file system of type `hugetlbfs` is mounted on /huge directory. If the `hugetlbfs` file system is mounted on some other directory, you should change the name of the huge page files appropriately using the environment variables `BLAS_HUGE_FILE` and `BLAS_SWAP_HUGE_FILE`, see "Environment variables" on page 13.
- If the operating system kills the application process or a bus error is received, check that sufficient memory is available on the system. The optimized BLAS level 3 routines require additional space. This space is allocated with huge pages. If there are insufficient huge pages in the system, there is a possibility of receiving a bus error at the time of execution. You can set the environment variable `BLAS_USE_HUGEPAGE` to 0 (see "Environment variables" on page 13) to use heap for memory allocation instead of huge pages.
- When you use the SPE APIs, make sure the alignment and parameter constraints are met. The results can be unpredictable if these constraints are not satisfied.
- The BLAS library requires all the matrices and vectors to be naturally aligned, that is, 4–byte aligned for single precision and 8–byte aligned for double precision. Cases where this is not satisfied can give unpredictable results including a bus error.

© Copyright IBM Corp. 2007, 2008

Part 5. BLAS API reference

The BLAS library provides two sets of interfaces:

- Chapter 8, “PPE APIs,” on page 27
- Chapter 9, “SPE APIs,” on page 29

The PPE interface conforms to the Standard BLAS interface. The library also provides additional functions to customize the library.

© Copyright IBM Corp. 2007, 2008

Chapter 8. PPE APIs

The PPE APIs are available for all SP and DP standard BLAS routines.

The current PPE APIs do not support complex single precision and complex double precision versions of BLAS routines. The PPE APIs conform to the existing standard interface defined by the BLAS Technical Forum. The library offers both a C interface and a standard FORTRAN compatible C interface to BLAS routines at the PPE level. Prototypes of the routines in C interface can be found in **cblas.h** and FORTRAN compatible C interface in **blas.h**.

Detailed documentation for these routines is available at:
`http://www.netlib.org/blas/blast-forum/blas-report.pdf`

For further information about BLAS, refer to netlib documentation on:
`http://www.netlib.org`

© Copyright IBM Corp. 2007, 2008

Chapter 9. SPE APIs

The library provides SPE APIs only for certain routines.

These APIs do not conform to the existing BLAS standard. There are constraints on the functionality (range of strides, sizes, and so on) supported by these routines. Prototypes of these routines are listed in **blas_s.h**. The following sections provide detailed descriptions of the routines that are part of these APIs.

This section describes the following APIs:

- "sscal_spu" on page 30
- "scopy_spu" on page 31
- "saxpy_spu" on page 32
- "sdot_spu" on page 33
- "isamax_spu" on page 34
- "sgemv_spu" on page 35
- "sgemm_spu" on page 36
- "ssyrk_64x64" on page 37
- "strsm_spu" on page 38
- "strsm_64x64" on page 39

© Copyright IBM Corp. 2007, 2008

sscal_spu

NAME

sscal_spu - Scales a vector by a constant.

SYNOPSIS

void sscal_spu (float *sx, float sa, int n)

Parameters

sx	Pointer to vector of floats to scale.
sa	Float constant to scale vector elements with.
n	Integer storing number of vector elements to scale. (Must be a multiple of 32)

DESCRIPTION

This BLAS 1 routine scales a vector by a constant. The following operation is performed in scaling:

$x \leftarrow \alpha x$

where x is a vector and α is a constant. Unlike the equivalent PPE API, the SPE interface is designed for stride 1 only, whereby n consecutive elements, starting with first element, get scaled. The routine has limitations on the n value and vector alignment. n value should be a multiple of 32. The x vector must be aligned at a 16 byte boundary.

EXAMPLES

```
#define len 1024
float buf_x[len] __attribute__ (( aligned (16) )) ;

int main()
{
   int   size=len, k ;

   float alpha = 0.6476 ;

   for(k=0;<ksize;k++)
   {
       buf_x[k] = (float)k ;
   }

   sscal_spu( buf_x, alpha, size ) ;

   return 0 ;
}
```

scopy_spu

NAME

scopy_spu - Copies a vector from source to destination.

SYNOPSIS

void scopy_spu (float *sx, float *sy, int n)

Parameters

sx	Pointer to source vector of floats
sy	Pointer to destination vector of floats
n	Integer storing number of vector elements to copy

DESCRIPTION

This BLAS 1 routine copies a vector from source to destination. The following operation is performed in copy:

$y \leftarrow x$

where x and y are vectors. Unlike the equivalent PPE API, this routine supports only stride 1, whereby n consecutive elements, starting with first element, get copied. The routine has no limitation on the value of n and vector alignments.

EXAMPLES

```
#define len 1000

int main()
{
   int size=len, k ;
   float buf_x[len] ;
   float buf_y[len] ;

   for(k=0;<ksize;k++)
   {
       buf_x[k] = (float)k ;
   }

   scopy_spu( buf_x, buf_y, size ) ;

   return 0 ;
}
```

saxpy_spu

NAME

saxpy_spu - Scales a source vector and element-wise adds it to the destination vector.

SYNOPSIS

void saxpy_spu (float *sx, float *sy, float sa, int n)

Parameters

sx	Pointer to source vector (x) of floats
sy	Pointer to destination vector (y) of floats
sa	Float constant to scale elements of vector x with
n	Integer storing number of vector elements to scale and add

DESCRIPTION

This BLAS 1 routine scales a source vector and element-wise adds it to the destination vector. The following operation is performed in scale and add:

$y \leftarrow \alpha x + y$

where x, y are vectors and α is a constant. Unlike the equivalent PPE API, the SPE interface is designed for stride 1 only, wherein n consecutive elements, starting with first element, get operated on. This routine has limitations on the n value and vector alignment supported. n value should be a multiple of 32. The x and y vectors must be aligned at a 16 byte boundary.

EXAMPLES

```
#define len 1024
float buf_x[len] __attribute__ (( aligned (16) )) ;
float buf_y[len] __attribute__ (( aligned (16) )) ;

int main()
{
   int   size=len, k ;
   float alpha = 0.6476 ;

   for(k=0; k<size; k++)
   {
       buf_x[k] = (float)k ;
       buf_y[k] = (float)(k * 0.23) ;
   }

   saxpy_spu( buf_x, buf_y, alpha, size ) ;
   return 0 ;
}
```

sdot_spu

NAME

sdot_spu - Performs dot product of two vectors.

SYNOPSIS

float sdot_spu (float *sx, float *sy, int n)

Parameters

sx	Pointer to first vector (x) of floats
sy	Pointer to second vector (y) of floats
n	Integer storing number of vector elements

DESCRIPTION

This BLAS 1 routine performs dot product of two vectors. The following operation is performed in dot product:

result ← x . y

where *x* and *y* are vectors. Unlike the equivalent PPE API, the SPE interface is designed for stride 1 only, whereby *n* consecutive elements, starting with first element, get operated on. This routine has limitations on the *n* value and vector alignment. *n* value should be a multiple of 32. The *x* and *y* vector must be aligned at a 16 byte boundary.

RETURN VALUE

float	Dot product of the two vectors

EXAMPLES

```
#define len 1024
float buf_x[len] __attribute__ (( aligned (16) )) ;
float buf_y[len] __attribute__ (( aligned (16) )) ;

int main()
{
   int size = len, k ;
   float sum = 0.0 ;

  for(k=0;<ksize;k++)
  {
    buf_x[k] = (float) k;
    buf_y[k] = buf_x[k];
  }

  sum = sdot_spu( buf_x, buf_y, size ) ;
  return 0 ;
}
```

isamax_spu

NAME

isamax_spu - Determines the (first occurring) index of the largest element in a vector.

SYNOPSIS

int isamax_spu (float *sx, int n)

Parameters

sx	Pointer to vector (x) of floats
n	Integer storing number of vector elements

DESCRIPTION

This BLAS 1 routine determines the (first occurring) index of the largest element in a vector. The following operation is performed in vector max index:

$result \leftarrow 1^{st} \;\; k \;\; s.t. \; x[k] = \max(x[i])$

where *x* is a vector. The routine is designed for stride 1 only, wherein n consecutive elements, starting with first element, get operated on. This routine has limitations on the *n* value and vector alignment. *n* value should be a multiple of 64. The *x* vector must be aligned at a 16 byte boundary.

RETURN VALUE

int	Index of (first occurring) largest element. (Indices start with 0.)

EXAMPLES

```
#define len 1024
float buf_x[len] __attribute__ (( aligned (16) )) ;

int main()
{
   int size=len, k ;

   int index ;

  for(k=0;<ksize;k++)
  {
    buf_x[k] = (float) k;
  }
  index = isamax_spu( buf_x, size ) ;
  return 0 ;
}
```

sgemv_spu

NAME

sgemv_spu - Multiplies a matrix and a vector, adding the result to a resultant vector.

SYNOPSIS

void sgemv_spu (int m, int n, float alpha, float *a, float *x, float *y)

Parameters

m	Integer specifying number of rows in matrix A
n	Integer specifying number of columns in matrix A
alpha	Float storing constant to scale the matrix-vector product AX
a	Pointer to matrix A
x	Pointer to vector X
y	Pointer to vector Y

DESCRIPTION

This BLAS 2 routine multiplies a matrix and a vector, adding the result to a resultant vector with suitable scaling. The following operation is performed:

$y \leftarrow \alpha A x + y$

where x and y are vectors, A is a matrix and α is a scalar.

Unlike equivalent PPE interface, the SPE interface for this routine only supports stride (increment) of one for vectors x and y. m must be a multiple of 32. n must be a multiple of 8. All the input vectors and matrix must be 16-byte aligned. Matrix A is assumed to be stored in the column major order.

EXAMPLES

```
#define M 512
#define N 32

float  Y[M] __attribute__ (( aligned (16) )) ;
float  A[M*N] __attribute__ (( aligned (16) )) ;
float  X[N] __attribute__ (( aligned (16) )) ;

int main()
{
    int k ;
    float alpha = 1.2;

    for(k = 0; k < M; k++)
        Y[k] = (float) k;

    for(k = 0; k < M*N; k++)
        A[k] = (float) k;

    for(k = 0; k < N; k++)
        X[k] = (float) k;

    sgemv_spu(M, N, alpha, A, X, Y);

    return 0;
}
```

sgemm_spu

NAME

sgemm_spu - Multiplies two matrices, A and B and adds the result to the resultant matrix C.

SYNOPSIS

void sgemm_spu (int m, int n, int k, float *a, float *b, float *c)

Parameters

m	Integer specifying number of rows in matrices A and C
n	Integer specifying number of columns in matrices B and C
k	Integer specifying number of columns in matrix A and rows in matrix B
a	Pointer to matrix A
b	Pointer to matrix B
c	Pointer to matrix C

DESCRIPTION

This BLAS 3 routine multiplies two matrices, A and B and adds the result to the resultant matrix C, after suitable scaling. The following operation is performed:

$C \leftarrow A \ B + C$

where A, B and C are matrices. The matrices must be 16-byte aligned and stored in row major order. *m* must be multiple of 4. *n* must be multiple of 16. *k* must be multiple of 4.

EXAMPLES

```
#define M    64
#define N    16
#define K    32

float A[M * K] __attribute__( (aligned (16)) ) ;
float B[K * N] __attribute__( (aligned (16)) ) ;
float C[M * N] __attribute__( (aligned (16)) ) ;

int main()
{
    int i, j;

    for( i = 0 ; i < M ; i++ )
        for( j = 0; j < N ; j++ )
            C[ ( N * i ) + j ] = (float) i ;

    /* Similar code to fill in other
    matrix arrays */
    . . . .
    sgemm_spu( M, N, K, A, B, C) ;
    return 0;
}
```

ssyrk_64x64

NAME

ssyrk_64x64 - Multiplies matrix A with its transpose A^T and adds the result to the resultant matrix C.

SYNOPSIS

void ssyrk_64x64(float *blkA, float *blkC, float *Alpha)

Parameters

`blkA`	Pointer to input matrix A
`blkC`	Pointer to input matrix C; This matrix is updated with result
`Alpha`	Pointer to scalar value with which Matrix A is scaled

DESCRIPTION

This BLAS 3 routine multiplies matrix, A with its transpose A^T and adds the result to the resultant matrix C, after suitable scaling.

The following operation is performed:

$C \leftarrow \alpha \ A \ A^T + C$

where only the lower triangular elements of *C* matrix are updated (the remaining elements remain unchanged).

The matrices must be 16-byte aligned and stored in row major order. Also, the matrices must be of size 64x64.

EXAMPLES

```
#define MY_M    64
#define MY_N    64

float myA[ MY_M * MY_N ] __attribute__((aligned (16)));
float myC[ MY_M * MY_M ] __attribute__((aligned (16)));

int main()
{
    int i,j ;
    float alpha = 2.0;

    for( i = 0 ; i < MY_M ; i++ )
        for( j = 0; j < MY_N ; j++ )
            myA[ ( MY_N * i ) + j ] = (float)i ;

    for( i = 0 ; i < MY_M ; i++ )
        for( j = 0 ; j < MY_M ; j++ )
           myC[ ( MY_M * i ) + j ] = (float)i ;

    ssyrk_64x64( myA, myC , α) ;

    return 0;
}
```

strsm_spu

NAME

strsm_spu - Solves a system of equations involving a triangular matrix with multiple right hand sides.

SYNOPSIS

void strsm_spu (int m, int n, float *a, float *b)

Parameters

m	Integer specifying number of columns of matrix B
n	Integer specifying number of rows of matrix B
a	Pointer to matrix A
b	Pointer to matrix B

DESCRIPTION

This BLAS 3 routine solves a system of equations involving a triangular matrix with multiple right hand sides. The following equation is solved and the result is updated in matrix B:

$AX = B$

where A is lower triangular n x n matrix and B is a n x m regular matrix. This routine has certain limitations in the values supported for matrix sizes and alignment of the matrix. n must be a multiple of 4. m must be a multiple of 8. Matrices A and B must be aligned at a 16 byte boundary and must be stored in row-major.

EXAMPLES

```
#define MY_M    32
#define MY_N    32

float myA[ MY_N * MY_N ] __attribute__( (aligned (16)) ) ;
float myB[ MY_N * MY_M ] __attribute__( (aligned (16)) ) ;

int main()
{
    int i,j,k ;

    for( i = 0 ; i < MY_N ; i++ )
    {
        for( j = 0; j <= i ; j++ )
           myA[ ( MY_N * i ) + j ] = (float)(i + 1) ;
        for( j = i+1; j < MY_N ; j++ )
           myA[( MY_N * i ) + j ] = 0 ;
    }

    for( i = 0 ; i < MY_N ; i++ )
      for( j = 0 ; j < MY_M ; j++ )
         myB[ ( MY_M * i ) + j ] = (float)(i+1)*(j +1);

     strsm_spu( MY_M, MY_N, myA, myB ) ;

     return 0;

}
```

strsm_64x64

NAME

strsm_64x64 - Solves a system of equations involving a triangular matrix with multiple right hand sides.

SYNOPSIS

void strsm_64x64 (float *a, float *b)

Parameters

a	Pointer to matrix A
b	Pointer to matrix B

DESCRIPTION

This BLAS 3 routine solves a system of equations involving a triangular matrix with multiple right hand sides. The following equation is solved and the result is updated in matrix B:

AX = B

where A is lower triangular 64 x 64 matrix and B is a 64 x 64 regular matrix. This routine is similar in operation to **strsm_spu** but is designed specifically for matrix size of 64 x 64. Hence better performance is achieved for 64 x 64 matrices when this routine is used rather than the more generic **strsm_spu**. Matrices A and B must be aligned at a 16 byte boundary and must be stored in row-major.

EXAMPLES

```
#define MY_M    64
#define MY_N    64

float myA[ MY_N * MY_N ] __attribute__( (aligned (16)) ) ;
float myB[ MY_N * MY_M ] __attribute__( (aligned (16)) ) ;

int main()
{
    int i,j,k ;

    for( i = 0 ; i < MY_N ; i++ )
    {
        for( j = 0; j <= i ; j++ )
           myA[ ( MY_N * i ) + j ] = (float)(i + 1) ;
        for( j = i+1; j < MY_N ; j++ )
           myA[( MY_N * i ) + j ] = 0 ;
    }

    for( i = 0 ; i < MY_N ; i++ )
      for( j = 0 ; j < MY_M ; j++ )
         myB[ ( MY_M * i ) + j ] = (float)(i+1)*(j +1);

     strsm_64x64( myA, myB ) ;

     return 0;

}
```

SEE ALSO

strsm_spu

Chapter 10. Additional APIs

The default SPE and memory management mechanism in the BLAS library can be partially customized by the use of environment variables as discussed previously. However, for more control over the use of available SPE resources and memory allocation/de-allocation strategy, an application can design its own mechanism for managing available SPE resources and allocating memory to be used by BLAS routines in the library.

The library provides some additional APIs that can be used to customize the library. These additional APIs can be used for the registration of custom SPE and memory management callbacks. The additional APIs can be divided into two parts: SPE management APIs for customizing the use of SPE resources and memory management APIs for customizing memory allocation/de-allocation mechanism used in the BLAS library.

Data types and prototypes of functions provided by these APIs are listed in the **blas_callback.h** file, which is installed with the blas-devel RPM.

SPE management APIs

These APIs can be used to register user-defined SPE management routines. Registered routines are then used inside the BLAS library for creating SPE threads, loading and running SPE programs, destroying SPE threads and so on. These registered routines override the default SPE management mechanism inside the BLAS library.

The following data types and functions are provided as part of these APIs:

- "spes_info_handle_t" on page 42
- "spe_info_handle_t" on page 43
- "BLAS_NUM_SPES_CB" on page 44
- "BLAS_GET_SPE_INFO_CB" on page 45
- "BLAS_SPE_SCHEDULE_CB" on page 46
- "BLAS_SPE_WAIT_CB" on page 47
- "BLAS_REGISTER_SPE" on page 48

© Copyright IBM Corp. 2007, 2008

spes_info_handle_t

NAME

spes_info_handle_t - Handle to access information about all the SPEs that are used by the BLAS library.

DESCRIPTION

A simple typedef to void. Used as a handle to access information about all the SPEs that are used by BLAS library.

You provide a pointer to **spes_info_handle_t** when registering SPE callback routines. **spes_info_handle_t*** is used as a pointer to user-defined data structure that contains information about all the SPEs to be used in BLAS library. The BLAS library passes the provided **spes_info_handle_t*** to registered callback routines.

EXAMPLES

You can define the following structure to store the information about the SPEs:

```
/* Data structure to store information about a single SPE */
typedef struct {
  spe_context_ptr_t spe_ctxt ;
  pthread_t pts ;
  spe_program_handle_t *spe_ph ;
  unsigned int entry ;
  unsigned int runflags ;
  void *argp ;
  void *envp ;
} blas_spe_info ;

/* Data structure to store information about multiple SPEs */
typedef struct  {
  int num_spes ;
  blas_spe_info spe[16] ;
} blas_spes_info ;

/* Define a variable that will store information about all
   the SPEs to be used in BLAS library */
blas_spes_info si_user;

/* Get a pointer of type spes_info_handle_t* that can be
   used to access information about all the SPEs */
spes_info_handle_t *spes_info = (spes_info_handle_t*)(&si_user);

/* Using spes_info, get a pointer of type spe_info_handle_t*
   that can be used to access information about a single SPE
   with index spe_index in the list of all SPEs */
spe_info_handle_t *single_spe_info =
(spe_info_handle_t*)(&spes_info->spe[spe_index]);

/* spes_info will be passed to BLAS library when registering
   SPE callback routines */
blas_register_spe(spes_info, <SPE callback routines> );
```

SEE ALSO

"spe_info_handle_t" on page 43

spe_info_handle_t

NAME

spe_info_handle_t - Handle to access information about a single SPE.

DESCRIPTION

A simple typedef to void. Used as a handle to access information about a *single* SPE in the pool of multiple SPEs that is used by BLAS library.

EXAMPLES

You can define the following structure to store the information about the SPEs:

```
/* Data structure to store information about a single SPE */
typedef struct {
  spe_context_ptr_t spe_ctxt ;
  pthread_t pts ;
  spe_program_handle_t *spe_ph ;
  unsigned int entry ;
  unsigned int runflags ;
  void *argp ;
  void *envp ;
} blas_spe_info ;

/* Data structure to store information about multiple SPEs */
typedef struct  {
  int num_spes ;
  blas_spe_info spe[16] ;
} blas_spes_info ;

/* Define a variable that will store information about all
   the SPEs to be used in BLAS library */
blas_spes_info si_user;

/* Get a pointer of type spes_info_handle_t* that can be
   used to access information about all the SPEs */
spes_info_handle_t *spes_info = (spes_info_handle_t*)(&si_user);

/* Using spes_info, get a pointer of type spe_info_handle_t*
   that can be used to access information about a single SPE
   with index spe_index in the list of all SPEs */
spe_info_handle_t *single_spe_info =
(spe_info_handle_t*)(&spes_info->spe[spe_index]);

/* spes_info will be passed to BLAS library when registering
   SPE callback routines */
blas_register_spe(spes_info, <SPE callback routines> );
```

SEE ALSO

"spes_info_handle_t" on page 42

BLAS_NUM_SPES_CB

NAME

BLAS_NUM_SPES_CB - Obtains the maximum number of SPEs that are available to the BLAS library.

SYNOPSIS

int (*BLAS_NUM_SPES_CB) (spes_info_handle_t *spes_info);

Parameters

`spes_info` — A pointer passed to the BLAS library when this callback is registered. The BLAS library passes this pointer to the callback while invoking it.

DESCRIPTION

This is a callback function prototype that is registered to obtain the maximum number of SPEs that are available to the BLAS library.

RETURN VALUE

`int` — Maximum number of SPEs that are available to the BLAS library for use.

EXAMPLES

```
int get_num_spes_user(spes_info_handle_t* spes_ptr)
{
    blas_spes_info *spes = (blas_spes_info*) spes_ptr;
    return spes->num_spes;
}

/* Register user-defined callback function */
blas_register_spe(spes_info /* spes_info_handle_t* */,
                  get_num_spes_user,
                  <Other SPE callback routines>);
```

BLAS_GET_SPE_INFO_CB

NAME

BLAS_GET_SPE_INFO_CB - Obtains the information about a single SPE from the pool of SPEs used inside the BLAS library.

SYNOPSIS

spe_info_handle_t*
(*BLAS_GET_SPE_INFO_CB) (spes_info_handle_t *spes_info, int index);

Parameters

`spes_info`	A pointer passed to the BLAS library when this callback is registered. The BLAS library passes this pointer to the callback while invoking it. This pointer points to private user data containing information about all the SPEs that user wants to use in the BLAS library.
`index`	Index of the SPE that identifies a single SPE in the data pointed to by **spes_info**. The BLAS library first invokes the registered callback routine of type BLAS_NUM_SPES_CB to get the total number of SPEs (num_spes) and then pass index in the range of 0 to (num_spes-1) to this callback.

DESCRIPTION

This is a callback function prototype that is registered to obtain the information about a single SPE from the pool of SPEs used inside the BLAS library.

This single SPE information is used when loading and running the SPE program to this SPE.

RETURN VALUE

`spe_info_handle_t*`	Pointer to a private user data containing information about a single SPE.

EXAMPLES

```
spe_info_handle_t*
get_spe_info_user(spes_info_handle_t *spes_ptr, int index)
{
    blas_spes_info *spes = (blas_spes_info*) spes_ptr;
    return (spe_info_handle_t*) ( &spes->spe[index] );
}

/* Register user-defined callback function */
blas_register_spe(spes_info /* spes_info_handle_t* */,
                  get_spe_info_user,
                  <Other SPE callback routines>);
```

BLAS_SPE_SCHEDULE_CB

NAME

BLAS_SPE_SCHEDULE_CB - Schedules a given SPE main program to be loaded and run on a single SPE.

SYNOPSIS

```
void
(*BLAS_SPE_SCHEDULE_CB) (spe_info_handle_t *single_spe_info,
                         spe_program_handle_t *spe_program,
                         unsigned int runflags,
                         void *argp, void *envp);
```

Parameters

`single_spe_info`	Pointer to private user data containing information about a single SPE. The BLAS library obtains this pointer internally by invoking the registered callback routine of type BLAS_GET_SPE_INFO_CB. The returned pointer is then passed to this callback.
`spe_program`	A valid address of a mapped SPE main program. SPE program pointed to by **spe_program** is loaded to the local store of the SPE identified by **single_spe_info**.
`runflags`	A bitmask that can be used to request certain specific behavior while executing the **spe_program** on the SPE identified by **single_spe_info**. Zero is passed for this currently.
`argp`	A pointer to BLAS routine specific data.
`envp`	Pointer to environment specific data of SPE program. NULL is passed for this currently.

DESCRIPTION

This is a callback function prototype that is registered to schedule a given SPE main program to be loaded and run on a single SPE.

EXAMPLES

```
void spe_schedule_user( spe_info_handle_t* spe_ptr,
               spe_program_handle_t *spe_ph,
                        unsigned int runflags,
void *argp, void *envp )
{
    blas_spe_info *spe = (blas_spe_info*) spe_ptr;

    /* Code to launch SPEs with specified parameters */
}


/* Register user-defined callback function */
blas_register_spe(spes_info /* spes_info_handle_t* */,
                  spe_schedule_user,
                  <Other SPE callback routines>);
```

BLAS_SPE_WAIT_CB

NAME

BLAS_SPE_WAIT_CB - Waits for the completion of a running SPE program on a single SPE.

SYNOPSIS

void (*BLAS_SPE_WAIT_CB) (spe_info_handle_t *single_spe_info);

Parameters

`single_spe_info`	Pointer to a private user data containing information about a single SPE. The BLAS library obtains this pointer internally by invoking the registered callback routine of type BLAS_GET_SPE_INFO_CB. The returned pointer is then passed to this callback.

DESCRIPTION

This is a callback function prototype that is registered to wait for the completion of a running SPE program on a single SPE, that is, until the SPE is finished executing the SPE program and is available for reuse.

For a particular SPE, the BLAS routine first invokes callback of type BLAS_SPE_SCHEDULE_CB for scheduling an SPE program to be loaded and run, followed by invoking callback of type BLAS_SPE_WAIT_CB to wait until the SPE is done.

EXAMPLES

```
void spe_wait_job_user( spe_info_handle_t* spe_ptr )
{
    blas_spe_info *spe = (blas_spe_info*) spe_ptr;

    /* Code to wait until completion of SPE program
       is indicated.
    */
}


/* Register user-defined callback function */
blas_register_spe(spes_info /* spes_info_handle_t* */,
                  spe_wait_job_user,
                  <Other SPE callback routines>);
```

BLAS_REGISTER_SPE

NAME

BLAS_REGISTER_SPE - Registers the custom SPE management callback routines to manage SPEs instead of using default SPE management routines.

SYNOPSIS

```
void
blas_register_spe(spes_info_handle_t *spes_info,
                  BLAS_SPE_SCHEDULE_CB spe_schedule_function,
                  BLAS_SPE_WAIT_CB spe_wait_function,
                  BLAS_NUM_SPES_CB num_spes_function,
                  BLAS_GET_SPE_INFO_CB get_spe_info_function);
```

Parameters

`spes_info`	Pointer to a private user data containing information about a single SPE. The BLAS library obtains this pointer internally by invoking the registered callback routine of type BLAS_GET_SPE_INFO_CB. The returned pointer is then passed to this callback.
`spe_schedule_function`	A pointer to user-defined function for scheduling an SPE program to be loaded and run on a single SPE.
`spe_wait_function`	A pointer to user-defined function to be used for waiting on a single SPE to finish execution.
`num_spes_function`	A pointer to user-defined function to be used for obtaining number of SPEs that is used.
`get_spe_info_function`	A pointer to user-defined function to be used for getting the information about a single SPE.

DESCRIPTION

This function registers the user-specified SPE callback routines to be used by BLAS library for managing SPEs instead of using default SPE management routines.

None of the input parameters to this function can be NULL. If any of the input parameters is NULL, the function simply return without performing any registration. A warning is displayed to standard output in this case.

Call this function only once to register the custom SPE callback routines. In case SPE callback registration has already been done before, the function terminates the application by calling abort().

EXAMPLES

For an example of this function, see the sample application blas-examples/blas_thread/, contained in the BLAS examples compressed file (**blas.tar**), which is installed with the **blas-devel** RPM. The following code outlines the basic structure of this sample application:

```
#include <blas.h>
#include <blas_callback.h>

typedef struct {
  spe_context_ptr_t spe_ctxt ;
  pthread_t pts ;
  pthread_mutex_t m ;
  pthread_cond_t c ;
```

```
  spe_program_handle_t *spe_ph ;
  unsigned int entry ;
  unsigned int runflags ;
  void *argp ;
  void *envp ;
  spe_stop_info_t *stopinfo ;
  unsigned int scheduled ;
  unsigned int processed ;
} blas_spe_info ;

typedef struct  {
  int num_spes ;
  blas_spe_info spe[16] ;
} blas_spes_info ;

blas_spes_info si_user;

int init_spes_user()
{
    int i ;
    void *blas_thread( void * ) ;
    char *ns = getenv( "BLAS_NUMSPES" ) ;
    si_user.num_spes = (ns) ? atoi(ns) : MAX_SPES ;

    for ( i = 0 ; i < si_user.num_spes ; i++ )
    {
        si_user.spe[i].spe_ctxt = spe_context_create( 0, NULL ) ;

   /* Code to initialize other fields of
         si_user.spe[i]
   */

        pthread_create( &si_user.spe[i].pts, NULL,
                        blas_thread, &si_user.spe[i] ) ;
     }

    return 0 ;
}

int cleanup_spes_user()
{
    int i ;
    for ( i = 0 ; i < si_user.num_spes ; i++ )
    {
       /* Cleanup code */
       pthread_join( si_user.spe[i].pts, NULL ) ;
       /* Cleanup code */
    }

    return 0 ;
}

spes_info_handle_t* get_spes_info_user()
{
    return (spes_info_handle_t*) (&si_user) ;
}

spe_info_handle_t*
get_spe_info_user(spes_info_handle_t *spes_ptr, int index)
{
    blas_spes_info *spes = (blas_spes_info*) spes_ptr;
    return (spe_info_handle_t*) ( &spes->spe[index] );
}

int get_num_spes_user(spes_info_handle_t* spes_ptr)
{
    blas_spes_info *spes = (blas_spes_info*) spes_ptr;
```

```
    return spes->num_spes;
}

void *blas_thread( void *spe_ptr )
{
    blas_spe_info *spe = (blas_spe_info *) spe_ptr ;
    while(1)
    {
        /* Wait on condition until some SPE program
           is available for running.
        */

  /* Come out of the infinite while loop
     and exit if NULL spe program is passed.
  */

        spe_program_load( spe->spe_ctxt, spe->spe_ph ) ;
        spe_context_run( spe->spe_ctxt, &spe>entry,
                         spe->runflags,
                         spe->argp, spe->envp, NULL ) ;

  /* Code to indicate the completion of SPE
     program.
  */
}

    return NULL ;
}

void spe_wait_job_user( spe_info_handle_t* spe_ptr )
{
    blas_spe_info *spe = (blas_spe_info*) spe_ptr;

    /* Code to wait until completion of SPE program
       is indicated.
    */
}

void spe_schedule_user( spe_info_handle_t* spe_ptr,
               spe_program_handle_t *spe_ph,
                 unsigned int runflags,
                 void *argp, void *envp )
{
    blas_spe_info *spe = (blas_spe_info*) spe_ptr;

    /* Some code here */

    spe->entry = SPE_DEFAULT_ENTRY ;
    spe->spe_ph = spe_ph ;
    spe->runflags = runflags ;
    spe->argp = argp ;
    spe->envp = envp ;

    /* Code to Signal SPE thread indicating that an SPE
       program is available for running.
    */
}

int main()
{
/* Some code here */
    blas_register_spe(get_spes_info_user(), spe_schedule_user,
                      spe_wait_job_user, get_num_spes_user,
                      get_spe_info_user);

    init_spes_user();
```

```
    /* Invoke blas routines */
    scopy_(...);
    sgemm_(...);
    ...

    cleanup_spes_user();

    return 0;
}
```

SEE ALSO

See example code in "Memory management APIs."

SPE management with multi-threaded applications

The ability to register and use application-defined SPE management functions inside the BLAS library becomes particularly useful with multi-threaded BLAS applications where multiple application threads invoke BLAS routines. With the help of custom SPE management routines, the multi-threaded application can easily distribute SPE resources available on the system to multiple application threads in the desired manner.

An example of a multi-threaded BLAS application registering its own SPE management functions is available in the blas-examples/blas_thread/ directory contained in the BLAS examples compressed file (**blas.tar**), which is installed with the blas-devel RPM.

Memory management APIs

These APIs can be used to register user-specified custom memory management routines. Registered routines are then used inside the BLAS library for allocating and de-allocating memory overriding default memory management routines.

The following functions are provided with these APIs:

- "BLAS_Malloc_CB" on page 52
- "BLAS_Free_CB" on page 53
- "BLAS_REGISTER_MEM" on page 54

BLAS_Malloc_CB

NAME

BLAS_Malloc_CB - Allocates aligned memory space.

SYNOPSIS

void* (*BLAS_Malloc_CB) (size_t size);

Parameters

`size`	Memory size in bytes to be allocated

DESCRIPTION

This is a callback function prototype that can be registered to allocate aligned memory space.

RETURN VALUE

`void*`	Pointer to allocated aligned memory. Allocated memory space must be aligned to 128-byte boundary. This pointer must be NULL if request fails.

BLAS_Free_CB

NAME

BLAS_Free_CB - De-allocates memory space.

SYNOPSIS

void (*BLAS_Free_CB) (void* ptr);

Parameters

`ptr`	Pointer to a memory space that needs to be released. This pointer is returned by a previous call to memory allocation callback routine of type BLAS_Malloc_CB.

DESCRIPTION

This is a callback function prototype that can be registered to de-allocate memory.

BLAS_REGISTER_MEM

NAME

BLAS_REGISTER_MEM - Registers the user-specified memory callback routines.

SYNOPSIS

void blas_register_mem(BLAS_Malloc_CB malloc_function,
BLAS_Free_CB free_function);

Parameters

`malloc_function`	A pointer to user-defined function used to allocate 128-byte aligned memory.
`free_function`	A pointer to user-defined function used to de-allocate memory.

DESCRIPTION

This function registers the user-specified Memory callback routines to be used by the BLAS library for allocating and de-allocating memory instead of using the default memory management routines.

EXAMPLES

```
#include <stddef.h>
#include <stdint.h>
#include <blas.h>
#include <blas_callback.h>
/* For allocating aligned memory from heap */
#include <malloc_align.h>
#include <free_align.h>

/* User defined memory allocation routines. These routines
   MUST return 128-byte aligned memory.
*/
void* malloc_user(size_t size)
{
    return _malloc_align(size, 7);
}

void  free_user(void *ptr)
{
    _free_align(ptr);
}

int main()
{
    /* Some code here */
    blas_register_mem(malloc_user, free_user);

    /* Invoke blas routines.
       BLAS level 3 routines like sgemm will now use registered
       routines malloc_user/free_user for allocation/de-
       allocation of 128-byte aligned memory
    */
    sgemm_(...);
    sgemv_(...);
    ...
    return 0;
}
```

SEE ALSO

See the sample application blas-examples/blas_thread/ contained in the BLAS examples compressed file (**blas.tar**), which is installed with the blas-devel RPM.

Part 6. Appendixes

© Copyright IBM Corp. 2007, 2008

Appendix A. Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the IBM developerWorks® Web site located at:

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*
- *Accelerated Library Framework for Hybrid-x86 Programmer's Guide and API Reference*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Hybrid-x86 Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*

© Copyright IBM Corp. 2007, 2008

- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Timer Library*

Installation

- *SDK for Multicore Acceleration Version 3.0 Installation Guide*

Tools

- *Getting Started - XL C/C++ Advanced Edition for Linux*
- *Compiler Reference - XL C/C++ Advanced Edition for Linux*
- *Language Reference - XL C/C++ Advanced Edition for Linux*
- *Programming Guide - XL C/C++ Advanced Edition for Linux*
- *Installation Guide - XL C/C++ Advanced Edition for Linux*
- *Getting Started - XL Fortran Advanced Edition for Linux*
- *Compiler Reference - XL Fortran Advanced Edition for Linux*
- *Language Reference - XL Fortran Advanced Edition for Linux*
- *Optimization and Programming Guide - XL Fortran Advanced Edition for Linux*
- *Installation Guide - XL Fortran Advanced Edition for Linux*
- *Using the single-source compiler*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC® Base

- *PowerPC Architecture Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Appendix B. Accessibility features

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

© Copyright IBM Corp. 2007, 2008

Notices

This information was developed for products and services offered in the U.S.A.

IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2007, 2008

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, Portable Document Format (PDF), and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of the manufacturer.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of the manufacturer.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any data, software or other intellectual property contained therein.

The manufacturer reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by the manufacturer, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

THE MANUFACTURER MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THESE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Glossary

API

Application Program Interface.

BLAS

Basic Linear Algebra Subprograms. A collection of subprograms for basic linear algebra operations, such as vector-vector operations (level 1 BLAS), matrix-vector operations (level 2 BLAS) and matrix-matrix operations (level 3 BLAS).

Broadband Engine

See *CBEA*.

Cholesky

The Cholesky factorization is named after André-Louis Cholesky. Cholesky found out that a symmetric positive-definite matrix can be decomposed into a lower triangular matrix and the transpose of the lower triangular matrix. The lower triangular matrix is the Cholesky triangle of the original, positive-definite matrix.

C++

C++ is an object-orientated programming language, derived from C.

CBEA

Cell Broadband Engine Architecture. A new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Cell/B.E. processor

The Cell/B.E. processor is a multi-core broadband processor based on IBM's Power Architecture.

Cell Broadband Engine processor

See *Cell/B.E* processor.

compiler

A programme that translates a high-level programming language, such as C++, into executable code.

FORTRAN

FORmula TRANslator). A high-level programming language for problems that can be expressed algebraically.

handle

A handle is an abstraction of a data object; usually a pointer to a structure.

LAPACK

Linear Algebra PACKage. These are routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.

LU factorization

In linear algebra, the LU factorization is a matrix decomposition, which writes a matrix as the product of a lower and upper triangular matrix.

PDF

Portable document format.

PPE

PowerPC Processor Element. The general-purpose processor in the Cell.

PPU

PowerPC Processor Unit. The part of the *PPE* that includes the execution units, memory-management unit, and L1 cache.

process

A process is a standard UNIX-type process with a separate address space.

© Copyright IBM Corp. 2007, 2008

program section

See *code section*.

ScaLAPACK

Scalable LAPACK. is a library of parallelized Linear Algebra routines. See “LAPACK” on page 67.

SDK

Software development toolkit for Multicore Acceleration. A complete package of tools for application development.

section

See *code section*.

SIMD

Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SPE

Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each cell processor.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

thread

A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously if each thread has its own architectural state (registers, program counter, flags, and other program-visible state). Each SPE can support only a single thread at any one time. Multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. It does this by duplicating the architectural state. A thread is typically created by the `pthreads` library.

vector

An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

Index

A

API 25, 41
  BLAS_Free_CB 53
  BLAS_GET_SPE_INFO_CB 45
  BLAS_Malloc_CB 52
  BLAS_NUM_SPES_CB 44
  BLAS_REGISTER_MEM 54
  BLAS_REGISTER_SPE 48
  BLAS_SPE_SCHEDULE_CB 46
  BLAS_SPE_WAIT_CB 47
  for managing SPEs 41
  isamax_spu 34
  memory management 51
  PPE 1
  saxpy_spu 32
  scopy_spu 31
  sdot_spu 33
  sgemm_spu 36
  sgemv_spu 35
  spe_info_handle_t 43
  spes_info_handle_t 42
  sscal_spu 30
  ssyrk_64x64 37
  strsm_64x64 39
  strsm_spu 38

B

bandwidth
  memory 13
BLAS documentation v
blas_callback.h 41
BLAS_Free_CB 53
BLAS_GET_SPE_INFO_CB 45
BLAS_HUGE_FILE 14, 23
BLAS_HUGE_PAGE_SIZE 14
BLAS_Malloc_CB 52
BLAS_NOMA_NODE
  performance 14
  tips 14
BLAS_NUM_SPES_CB 44
BLAS_NUMA_NODE 14
BLAS_NUMSPES 14
BLAS_REGISTER_MEM 54
BLAS_REGISTER_MEM() 21
BLAS_REGISTER_SPE 48
BLAS_REGISTER_SPE() 21
blas_s.h 29
BLAS_SPE_SCHEDULE_CB 46
BLAS_SPE_WAIT_CB 47
BLAS_SWAP_HUGE_FILE 14, 23
BLAS_SWAP_NUMA_NODE 14
BLAS_SWAP_SIZE 14
BLAS_USE_HUGEPAGE 14
blas-3.0-x.ppc.rpm 5
blas-3.0-x.ppc64.rpm 5
blas-devel RPM 41
blas-devel-3.0-x.ppc64.rpm 5
blas.h 27
blas.tar 51

C

C interface 11
  example application 11
cblas.h 27
Cell/B.E. applications 23
  debugging 23
  huge pages 23
  programming 17
Cholesky 1
compute-bound routine 13
customizing 13
  memory management 21

D

debugging
  Cell/B.E. applications 23
documentation 59
  BLAS-related v

E

environment variables 13
example
  C interface 11
  FORTRAN 11
  multi-threaded application 51
  PPE interface 11

F

FORTRAN
  C interface 11
  example application 11

H

header file 11
HPC 1
HPL 1
huge pages 13, 23
hugetlbfs file system 23

I

installing
  packages 5
isamax_spu 34

L

LAPACK 1
LAS_USE_HUGEPAGE 23
libnuma 14
library structure
  PPE interface 9
  SPE interface 9
Linpack benchmark 1

M

memory
  bandwidth 13
  callback 21
  custom management 21
  default management 21
  management API 51
multi-thread
  applications 51
  example 51
  SPE management 51

N

NUMA
  binding 14
  policy API libnuma 14
  policy tool numactl 14
numactl 14

O

optimizing 13
overview 1

P

packages
  blas-3.0-x.ppc.rpm 5
  blas-3.0-x.ppc64.rpm 5
  blas-devel-3.0-x.ppc64.rpm 5
performance
  considerations 13
  optimizing BLAS 14
  tips to improve 14
PPE
  API 1, 27
  BLAS library example 11
  C interfaces supported 11
  input requirements 11
  interface 9

R

restrictions 11
routine
  real double precision (DP) 1
  real single precision (SP) 1
RPM
  blas-devel 41

S

sample application 11
saxpy_spu 32
ScaLAPACK 1
scopy_spu 31
SDK documentation 59

© Copyright IBM Corp. 2007, 2008

sdot_spu 33
sgemm_spu 36
sgemv_spu 35
SP routine 1
SPE
  API 29
  APIs for managing 41
  creating threads 19
  interface 9
  multi-threaded applications 51
  thread management routine 21
spe_info_handle_t 43
spes_info_handle_t 42
sscal_spu 30
ssyrk_64x64 37
startup costs 13
strsm_64x64 39
strsm_spu 38
swap space 13

T

thread
  creating 19
  management routine for SPE 21
  SPE 19



Printed in USA

SC33-8426-00

