Software Development Kit for Multicore Acceleration
Version 3.1

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

SC33-8426-01

Software Development Kit for Multicore Acceleration
Version 3.1

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

SC33-8426-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 75.

Edition notice

This edition applies to version 3, release 1, modification 0, 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.

This edition replaces SC33-8426-00.

Contents

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 71.

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`

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).

What is new

The following routines are new for this release.

PPE APIs

The following routines have been optimized to use SPEs and are available as PPE APIs:

BLAS routine level	Functionality
Level 1	DASUM DNRM2 DROT
Level 2	DGBMV DGER/SGER DSYMV DTBMV DSYR SGEMV (added support for all parameters) STRMV STRSV
Level 3	SSYRK (added support for all parameters) STRSM (added support for all parameters) SSYMM DSYR2K/SSYR2K STRMM

SPE APIs

The following new SPE interfaces have been added in this release:

BLAS routine level	Functionality
Level 1	• dscal_spu • dcopy_spu • daxpy_spu • ddot_spu • idamax_spu, idamax_edp_spu • dasum_spu • dnrm2_spu, dnrm2_edp_spu • drot_spu

BLAS routine level	Functionality
Level 2	• dgemv_spu • dtrsv_spu_lower / dtrsv_spu_upper / strsv_spu_lower / strsv_spu_upper • dger_spu / sger_spu / dger_op_spu / sger_op_spu • dsymv_spu_lower
Level 3	• strsm_spu_upper / dtrsm_spu_lower / dtrsm_spu_upper / dtrsm64x64_lower / dtrsm64x64_upper • dgemm_spu • ssyrk_spu / dsyrk_spu / dsyrk_64x64 • strmm_spu_upper_trans_left / strmm_spu_upper_left / dtrmm_spu_upper_trans_left / dtrmm_spu_upper_left / • ssyr2k_spu_lower / ssyr2k_64x64_lower / dsyr2k_spu_lower / dsyr2k_64x64_lower /

Part 1. Overview of BLAS

The BLAS library 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.

It 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.

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 both real and complex routines in single and double precision. Complex routines have been newly added in this version of BLAS. However none of the complex routines are optimized using SPEs. All 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 the real single precision (SP) and real double precision (DP) 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
  - DASUM
  - DNRM2
  - DROT

- Level 2:
  - SGEMV, DGEMV
  - STRMV, DTRMV
  - STRSV, DTRSV
  - DGBMV
  - SGER, DGER
  - DSYMV
  - DTBMV
  - DSYR
- Level 3:
  - SGEMM, DGEMM
  - SSYRK, DSYRK
  - STRSM, DTRSM
  - STRMM, DTRMM
  - SSYMM, DSYMM
  - SSYR2K, DSYR2K

Part 2. Installing the BLAS library

The following topics describe the BLAS library installation packages.

For information about how to install the BLAS library, refer to the *SDK Installation Guide*.

Chapter 1. Package descriptions

This topic describes the BLAS installation packages for each of the supported operating environments.

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

Table 2. BLAS library installation packages

Package	Purpose	Platform	Contents
blas-3.1-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-3.1-x.ppc.rpm	Installs supporting files such as header files 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 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 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. /usr/lib/libblas_xlf.a: Static library providing XLF compatible interfaces for CDOTU, ZDOTU, CDOTC and ZDOTC. Only used when compiling applications with XLF.
blas-3.1-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-3.1-x.ppc64.rpm	Installs supporting files such as header files 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 /usr/lib64/libblas_xlf.a: Static library providing XLF compatible interfaces for CDOTU, ZDOTU, CDOTU and ZDOTC. Only used when compiling applications with XLF.

Table 2. BLAS library installation packages (continued)

Package	Purpose	Platform	Contents
blas-cross-devel-3.1-x.noarch.rpm where x is the build date	Installs the BLAS library and supporting files such as header files.	Other platforms, such as x86 series.	/opt/cell/sysroot/usr/include/blas.h: Contains prototypes of all BLAS Level 1, 2 and 3 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 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. /opt/cell/sysroot/usr/spu/lib/libblas.a: BLAS SPU library. /opt/cell/sysroot/usr/lib/libblas_xlf.a /opt/cell/sysroot/usr/lib64/libblas_xlf.a: Static library providing XLF compatible interfaces for CDOTU, ZDOTU, CDOTC and ZDOTC. Only used when compiling applications with XLF.
blas-examples-source-3.1-x.noarch.rpm	Installs BLAS sample applications.		/opt/cell/sdk/src/blas-examples-source.tar: Compressed file of BLAS samples.

Part 3. Programming

The topics in this section provide information about programming with the BLAS library.

The following topics are described:

Chapter 2. Basic structure of the BLAS library

The topics in this section describe the BLAS library components.

The BLAS Library has two components:

- Power Processing Element (PPE) interface
- Synergistic Processing Element (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 31.

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.

These are:

- 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, which means that the FORTRAN interface supports column-major storage only, whereas the C interface supports both row-major as well as column-major data storage.

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'.

Fortran applications must use the compiler options as shown in Table 3:

Table 3. Fortran compiler options

Compiler	Compilation mode	Compiler command
ppu-gfortran	32–bit	ppu-gfortran –m32 –ff2c myapp.f -lblas
	64–bit	ppu-gfortran –m64 –ff2c myapp.f –lblas
ppuxlf Application does not call complex routines cdotu, zdotu, cdotc, zdotc	32–bit	ppuxlf -q32 –qextname myapp.f -lblas
	64–bit	ppuxlf –q64 –qextname myapp.f -lblas
ppuxlf Application calls complex routines cdotu, zdotu, cdotc, zdotc	32–bit	ppuxlf -q32 –qextname myapp.f –lblas_xlf –lblas (-lblas_xlf must be placed before –lblas)
	64–bit	ppuxlf –q64 –qextname myapp.f –lblas_xlf –lblas (-lblas_xlf must be placed before –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.

The alignment is as follows:

- 4-byte aligned for real single precision
- 8-byte aligned for real double precision and complex single precision
- 16-byte aligned for complex double precision

The library does not support cases where this is not satisfied.

Programming samples

The following sample applications demonstrate the usage of the BLAS-PPE library. The application programs invoke 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-- )
            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 following topics describe BLAS library 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

use your own SPE and memory callbacks (described in Chapter 10, "Additional APIs," on page 53). The following table lists the environment variables:

Table 4. Environment variables

Variable name	Purpose	Default value
BLAS_NUMSPES	Specifies the number of SPEs to be used per application. For multi threaded applications, the SPEs specified by BLAS_NUMSPEs are shared by all the application threads. 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_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

Note: The environment variable BLAS_NUMA_NODE is no longer supported. You can use the command line NUMA policy tool numactl to achieve the same functionality.

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 numactl --cpunodebind=0 --membind=0
BLAS_SWAP_SIZE=12288 ./test_name
```

Programming tips to achieve maximum performance

You can use the tips described here 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. 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 15, 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.
- On an IBM BladeCenter QS21 or QS22, which has 16 SPEs, `BLAS_NUMSPES` can be set to 16 to get better performance for compute bound routines.

Chapter 5. Debugging tips

You can use the steps described in this topic 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 15.
- 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 15) 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.

Part 4. SPE and memory management

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.

Chapter 6. Creating SPE threads

When a prebuilt 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 15.

Chapter 7. 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.

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-examples-source.tar**), which is installed with the blas-examples-source RPM.

Part 5. BLAS API reference

The BLAS library provides two sets of interfaces.

These are:

- Chapter 8, "PPE APIs," on page 29
- Chapter 9, "SPE APIs," on page 31

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

Chapter 8. PPE APIs

The PPE APIs are available for all standard 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`

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. The following table provides a list of routines that are new for SDK 3.1:

BLAS routine level	Functionality
Level 1	• dscal_spu • dcopy_spu • daxpy_spu • ddot_spu • idamax_spu, idamax_edp_spu • dasum_spu • dnrm2_spu, dnrm2_edp_spu • drot_spu
Level 2	• dgemv_spu • dtrsv_spu_lower / dtrsv_spu_upper / strsv_spu_lower / strsv_spu_upper • dger_spu / sger_spu / dger_op_spu / sger_op_spu • dsymv_spu_lower
Level 3	• strsm_spu / strsm_spu_upper / dtrsm_spu_lower / dtrsm_spu_upper / dtrsm64x64_lower / dtrsm64x64_upper • dgemm_spu • ssyrk_spu / dsyrk_spu /dsyrk_64x64 • strmm_spu_upper_trans_left / strmm_spu_upper_left / dtrmm_spu_upper_trans_left / dtrmm_spu_upper_left • ssyr2k_spu_lower / ssyr2k_64x64_lower / dsyr2k_spu_lower / dsyr2k_64x64_lower

sscal_spu / dscal_spu

NAME

sscal_spu / dscal_spu - Scales a vector by a constant.

SYNOPSIS

void sscal_spu (float *sx, float sa, int n)
void dscal_spu (double *dx, double da, int n)

Parameters

sx/dx	Pointer to vector of floats/doubles to scale.
sa/da	Float/double constant to scale vector elements with.
n	Integer storing number of vector elements to scale. (Must be a multiple of 32 for SP and 16 for DP)

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 16 for DP and 32 for SP. 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 / dcopy_spu

NAME

scopy_spu / dcopy_spu - Copies a vector from source to destination.

SYNOPSIS

void scopy_spu (float *sx, float *sy, int n)
void dcopy_spu (double *dx, double *dy, int n)

Parameters

sx/dx	Pointer to source vector of floats/doubles
sy/dy	Pointer to destination vector of floats/doubles
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 / daxpy_spu

NAME

saxpy_spu / daxpy_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)
void daxpy_spu (double *dx, double *dy, double da, int, n)

Parameters

sx/dx	Pointer to source vector (x) of floats/doubles
sy/dy	Pointer to destination vector (y) of floats/doubles
sa/da	Float/double 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. The value of n should be a multiple of 32 for DP and 64 for SP. 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 / ddot_spu

NAME

sdot_spu / ddot_spu - Performs dot product of two vectors.

SYNOPSIS

float sdot_spu (float *sx, float *sy, int n)
double ddot_spu (double *dx, double *dy, int n)

Parameters

sx/dx	Pointer to first vector (*x*) of floats/doubles
sy/dy	Pointer to second vector (*y*) of floats/doubles
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 16 for DP and 32 for SP. The *x* and *y* vector must be aligned at a 16 byte boundary.

RETURN VALUE

float/double	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 / idamax_spu / idamax_edp_spu

NAME

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

SYNOPSIS

int isamax_spu (float *sx, int n)
int idamax_spu (double *dx, int n)
int idamax_edp_spu (double *dx, int n)

Parameters

sx/dx	Pointer to vector (x) of floats/doubles
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 32 for both SP and DP. The *x* vector must be aligned at a 16 byte boundary.

idamax_edp_spu is optimized exclusively for PowerXCell 8i processor with enhanced double precision support (eDP) (for example, in an IBM BladeCenter QS22) and cannot be used with previous Cell/B.E. processors without eDP support (for example, in an IBM BladeCenter QS21). The **idamax_spu** routine is compatible with both the processors.

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 ;
}
```

dasum_spu

NAME

dasum_spu - Returns the sum of the absolute elements in a vector.

SYNOPSIS

double dasum_spu (double *dx, int n)

Parameters

dx	Pointer to vector of doubles to be summed up
n	Integer storing number of vector elements to be summed up (must be a multiple of 64)

DESCRIPTION

This BLAS 1 routine returns the sum of the absolute elements in a vector.

The routine performs:

result $\leftarrow \sum|x_i|$

The SPE routine is designed for stride 1 only, wherein *n* consecutive elements, starting with first element, get summed up. The *x* vector must be aligned at a 16 byte boundary.

RETURN VALUE

The absolute sum of elements in the vector.

dnrm2_spu / dnrm2_edp_spu

NAME

dnrm2_spu / dnrm2_edp_spu - Returns the euclidean norm of a vector.

SYNOPSIS

double dnrm2_spu (int n, double *dx)
double dnrm2_edp_spu (int n, double *dx)

Parameters

n	Pointer to integer storing number of vector elements to be operated on (must be a multiple of 32).
dx	Pointer to vector x to be normalised.

DESCRIPTION

This BLAS 1 routine returns the euclidean norm of a vector.

The routine performs:

```
result ← √(x.x')
```

where x and x' is the vector and its transpose.

The SPE routine is designed for stride 1 only, wherein n consecutive elements, starting with first element, get operated on. The x vector must be aligned at a 16–byte boundary.

dnrm2_edp_spu is optimised exclusively for PowerXCell 8i processor with enhanced double precision support (eDP) (for example, in an IBM BladeCenter QS22) and cannot be used with previous Cell/B.E. processors without eDP support (for example, in an IBM BladeCenter QS21). The **dnrm2_spu** routine is compatible with both the processors.

drot_spu

NAME

drot_spu - Applies a real plane rotation to real vectors.

SYNOPSIS

void drot_spu (int n, double *dx, double *dy, double c, double s)

Parameters

n	Integer storing number of vector elements to be rotated (must be a multiple of 16)
dx	Pointer to vector x to be rotated.
dy	Pointer to vector y to be rotated.
c	Double storing cosine of the angle of rotation.
s	Double storing sine of the angle of rotation.

DESCRIPTION

This BLAS 1 routine applies a real plane rotation to real vectors. The plane rotation is applied to *n* points, where the points to be rotated are contained in vectors *x* and *y*, and where the cosine and sine of the angle of rotation are *c* and *s*, respectively. The operation is as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

where x_i and y_i is the index of each element to be operated on.

The SPE routine is designed for stride 1 only, wherein *n* consecutive elements, starting with first element, get operated on. The *x* and *y* vectors must be aligned at a 16 byte boundary.

sgemv_spu / dgemv_spu

NAME

sgemv_spu / dgemv_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)
void dgemv_spu (int m, int n, double alpha, double *a, double *x, double *y)

Parameters

m	Integer specifying number of rows in matrix A
n	Integer specifying number of columns in matrix A
alpha	Float/double storing constant to scale the matrix 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 routines **sgemv_spu** and **dgemv_spu** perform the following operation:

$y \leftarrow \alpha A x + y$

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 for both SP and DP. n must be a multiple of 8 for both SP and DP. All the input vectors and matrix must be 16-byte aligned. Matrix A must be stored in 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;
                    }

```

dtrsv_spu_lower / dtrsv_spu_upper / strsv_spu_lower / strsv_spu_upper

NAME

dtrsv_spu_lower / dtrsv_spu_upper / strsv_spu_lower / strsv_spu_upper - Solves systems of triangular equations involving a triangular matrix and a vector.

SYNOPSIS

void dtrsv_spu_lower (unsigned int n, double *a, int lda, double *x)
void dtrsv_spu_upper (unsigned int n, double *a, int lda, double *x)
void strsv_spu_lower (unsigned int n, float *a, int lda, float *x)
void strsv_spu_upper (unsigned int n, float *a, int lda, float *x)

Parameters

n	Integer specifying order of matrix A (must be a multiple of 16)
a	Pointer to matrix A
lda	Integer specifying leading dimension of the first dimension of matrix A, must be greater than or equal to max(1, n). It must be a multiple of 2 for double precision and a multiple of 4 for single precision (so that the next column of matrix A starts from a 16 byte aligned address).
x	Pointer to vector x (only stride/increment of one supported for vector x)

DESCRIPTION

This BLAS 2 routine solves systems of triangular equations involving a triangular matrix and a vector.

The routine performs:

$x = A^{-1} x$

where for:

dtrsv_spu_lower/strsv_spu_lower
A is a lower triangular matrix and *x* is a vector

dtrsv_spu_upper/strsv_spu_upper
A is an upper triangular matrix and *x* is a vector

The input vectors and matrices must be 16-byte aligned. Matrix A must be stored in column major order. The triangular part of the matrix A must not contain any NaN or Infinity values.

dger_spu / sger_spu / dger_op_spu / sger_op_spu

NAME

dger_spu / sger_spu / dger_op_spu / sger_op_spu - Computes the outer product of two vectors with suitable scaling.

SYNOPSIS

void dger_spu (unsigned int P, unsigned int Q, double alpha,
double *x, double *y, double *A, unsigned int lda, double beta) void sger_spu (unsigned
float *x, float *y, float *A, unsigned int lda, float beta)
void dger_op_spu (unsigned int P, unsigned int Q, double alpha,
double *x, double *y, double *A, unsigned int lda)
void sger_op_spu (unsigned int P, unsigned int Q, float alpha,
float *x, float *y, float *A, unsigned int lda)

Parameters

P	Unsigned integer specifying number of rows of matrix A
Q	Unsigned integer specifying number of columns of matrix A
alpha	Scalar constant
x	Pointer to a block of vector X
y	Pointer to a block of vector Y
A	Pointer to a block of matrix A
lda	Unsigned integer specifying leading dimension of the first dimension of the block of matrix A. lda must be >= max(1,P)
beta	Scalar constant

DESCRIPTION

These routines compute the outer product of two vectors with suitable scaling. Routines **dger_spu** and **sger_spu** performs the following operation

$A \leftarrow \alpha.x.y^T + \beta.A$

where A is a column-major regular matrix of dimension PxQ and leading dimension of lda. x and y are vectors of size P and Q, respectively and stride 1. α and β are scalar constants. Vectors x and y and matrix A are 16-byte aligned.

Routines **dger_op_spu** and **sger_op_spu** performs the following operation where it does not fetches matrix A

$A \leftarrow \alpha.x.y^T$

Unlike equivalent PPE interface, the SPE interface for this routine only supports stride (increment) of one for vectors x and y. P, Q and lda must be a multiple of 4 for SP and 2 for DP. lda must also be >= max(1,P). All the input vectors and matrix must be 16-byte aligned and matrix A must be stored in column major order.

dsymv_spu_lower

NAME

dsymv_spu_lower - Multiplies a symmetric matrix and a vector, adding the result to a resultant vector with suitable scaling.

SYNOPSIS

void dsymv_spu_lower (unsigned int n, double alpha, double *a, int lda, double *x, double *y)

Parameters

`n`	Integer specifying order of matrix A (must be a multiple of 16).
`alpha`	Double storing constant to scale the matrix-vector product A*x*.
`a`	Pointer to matrix A.
`lda`	Integer specifying leading dimension of the first dimension of matrix A (must be >= max(1,n) and multiple of 2 (so that the next column of matrix A starts from a 16-byte aligned address).
`x`	Pointer to vector X.
`y`	Pointer to vector Y.

DESCRIPTION

The routine **dsymv_spu_lower** performs the following:

$y \leftarrow \alpha A x + y$

Where *A* is symmetric matrix and only lower triangular elements of *A* need to be referenced. Vectors and matrices must be 16 byte aligned and matrices must be in column major order.

strsm_spu

NAME

**strsm_spu / strsm_spu_upper / strsm_64x64 /
dtrsm_spu_lower / dtrsm_spu_upper /
dtrsm64x64_lower / dtrsm64x64_upper -**

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)
void strsm_spu_upper (unsigned int m, unsigned int n, float *a, float *b, unsigned int lda, unsigned int ldb)
void strsm_64x64 (float *a, float *b)
void dtrsm_spu_lower (unsigned int m, unsigned int n, double *a, double *b, unsigned int lda, unsigned int ldb)
void dtrsm_spu_upper (unsigned int m, unsigned int n, double *a, double *b, unsigned int lda, unsigned int ldb)
void dtrsm64x64_lower (double *a, double *b)
void dtrsm64x64_upper (double *a, double *b)

Parameters

`m`	Integer specifying number of columns of matrix B in case of **strsm_spu** and number of rows in case of other routines.
`n`	Integer specifying number of rows of matrix B in case of **strsm_spu** and number of columns in case of other routines.
`a`	Pointer to matrix A
`b`	Pointer to matrix B
`lda`	Integer specifying leading dimension for matrix A. It must be greater than or equal to max(1, m), and a multiple of 2 for DP and 4 for SP
`ldb`	Integer specifying leading dimension for matrix B. It must be greater than or equal to max(1, n), and a multiple of 2 for DP and 4 for SP

DESCRIPTION

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

$B \leftarrow A^{-1}B$

where for:

strsm_spu
A is lower triangular *n* x *n* matrix and B is a *n* x *m* regular matrix. *m* must be a multiple of 8, *n* must be a multiple of 4.

strsm_64x64
A is lower triangular 64 x 64 matrix and B is a 64 x 64 regular matrix.

strsm_spu_upper / dtrsm_spu_upper
A is the upper triangular *m* x *m* matrix and B is a *m* x *n* regular matrix. Both *m* and *n* must be a multiple of 16.

dtrsm_spu_lower
A is the lower triangular *m* x *m* matrix and B is a *m* x *n* regular matrix. Both *m* and *n* must be a multiple of 16.

dtrsm64x64_lower
A is the lower 64 x 64 triangular matrix and B is a 64 x 64 regular matrix.

dtrsm64x64_upper
A is the upper 64 x 64 triangular matrix and B is a 64 x 64 regular matrix.

Matrices A and B must be aligned at a 16–byte boundary and must be stored in row-major order. The triangular part of the matrix A must not contain any NaN or Infinity values.

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;

}
```

sgemm_spu / dgemm_spu / dgemm_64x64

NAME

sgemm_spu / dgemm_spu / dgemm_64x64 - 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)
void dgemm_spu (int m, int n, int k, double *a, double *b, double *c)
void dgemm_64x64 (double *c, double *a, double *b)

Parameters

m	Integer specifying number of rows in matrices A and C (must be a multiple of 4 for SP, and 16 for DP)
n	Integer specifying number of columns in matrices B and C (must be a multiple of 16 for both SP and DP)
k	Integer specifying number of columns in matrix A and rows in matrix B (must be a multiple of 4 for SP and 16 for DP)
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 ← A  B  + C
```

where A, B, and C are matrices. The matrices must be 16-byte aligned and stored in row major order. For **dgemm_64x64**, the matrices must be of dimensions 64x64

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_spu / dsyrk_spu / ssyrk_64x64 / dsyrk_64x64

NAME

ssyrk_spu / dsyrk_spu / ssyrk_64x64 / dsyrk_64x64 - Performs a rank-k update to a symmetric matrix A.

SYNOPSIS

void ssyrk_spu (float *blkA, float *blkC, float Alpha, unsigned int N, unsigned int K, unsigned int lda, unsigned int ldc)

void dsyrk_spu (double *blkA, double *blkC, double Alpha, unsigned int N, unsigned int K, unsigned int lda, unsigned int ldc)

void ssyrk_64x64(float *blkA, float *blkC, float *Alpha)
void dsyrk_64x64(double *blkA, double *blkC, double Alpha)

Parameters

N	Integer specifying order of matrix C (must be a multiple of 16)
K	Integer specifying number of columns in matrix A (must be a multiple of 16)
blkA	Pointer to matrix A
blkC	Pointer to matrix C
Alpha	Double scalar value to scale matrix product $A.A^T$
lda	Integer specifying leading dimension for matrix A (lda is greater than or equal to K and multiple of 2 for DP or 4 for SP)
ldc	Integer specifying leading dimension for matrix C (ldc is greater than or equal to N and multiple of 2 for DP or 4 for SP)

DESCRIPTION

The routine performs:

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

where only the lower triangular elements of matrix C are updated (the remaining elements remain unchanged). For **ssyrk_64x64** and **dsyrk_64x64**, the matrices must be of size 64 x 64.

The matrices must be 16-byte aligned and stored in row major order.

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 , &alpha) ;

                        return 0;
                    }

```

strmm_spu_upper_trans_left / strmm_spu_upper_left / dtrmm_spu_upper_trans_left / dtrmm_spu_upper_left

NAME

**strmm_spu_upper_trans_left / strmm_spu_upper_left /
dtrmm_spu_upper_trans_left / dtrmm_spu_upper_left
strmm_64x64_upper_trans_left / dtrmm_64x64_upper_trans_left
strmm_64x64_upper_left / dtrmm_64x64_upper_left -**
Computes the product of two matrices A and B where A is a triangular matrix.

SYNOPSIS

void strmm_spu_upper_trans_left (int m, int n, float *a, float *b)
void strmm_spu_upper_left (int m, int n, float *a, float *b)
void dtrmm_spu_upper_trans_left (int m, int n, double *a, double *b)
void dtrmm_spu_upper_left (int m, int n, double *a, double *b)
void strmm_64x64_upper_trans_left (float *a, float *b)
void strmm_64x64_upper_left (float *a, float *b)
void dtrmm_64x64_upper_trans_left (double *a, double *b)
void dtrmm_64x64_upper_left (double *a, double *b)

Parameters

m	Integer specifying number of rows of matrix B (must be a multiple of 16)
n	Integer specifying number of columns of matrix B (must be a multiple of 16)
a	Pointer to matrix A
b	Pointer to matrix B

DESCRIPTION

This BLAS 3 routine computes the product of two matrices A and B where A is a triangular matrix. The matrix B is updated with the result.

The routines **strmm_spu_upper_trans_left** and **dtrmm_spu_upper_trans_left** perform:

$B \leftarrow A^T B$

where A is an upper triangular matrix.

The routine **strmm_spu_upper_left** and **dtrmm_spu_upper_left** performs:

$B \leftarrow AB$

where A is an upper triangular matrix.

The routines **strmm_64x64_upper_trans_left** and **dtrmm_64x64_upper_trans_left** perform:

$B \leftarrow A^T B$

where A is an upper triangular 64 x 64 matrix.

The routine **strmm_64x64_upper_left** and **dtrmm_64x64_upper_left** performs:

$B \leftarrow AB$

where A is an upper triangular 64 x 64 matrix.

The matrices must be 16-byte aligned and stored in row major order.

ssyr2k_spu_lower / ssyr2k_64x64_lower / dsyr2k_spu_lower / dsyr2k_64x64_lower

NAME

ssyr2k_spu_lower / ssyrk2k_64x64_lower / dsyr2k_spu_lower / dsyr2k_64x64_lower - Performs one of the rank-2k updates to a symmetric matrix.

SYNOPSIS

void ssyr2k_spu_lower(unsigned int n, unsigned int k, float *A, float* B, float *C)
void ssyr2k_64x64_lower(float *A, float *B, float *C)
void dsyr2k_spu_lower(unsigned int n, unsigned int k, double *A, double* B, double *C)
void dsyr2k_64x64_lower(double *A, double* B, double *C)

Parameters

n	Integer specifying order of matrix C (must be a multiple of 16)
k	Integer specifying number of columns of matrices A and B (must be a multiple of 16)
A	Pointer to matrix A
B	Pointer to matrix B
C	Pointer to matrix C

DESCRIPTION

This BLAS 3 routine performs the following rank-2k updates to a symmetric matrix

```
C =  A B' +  B A' +  C
```

where C is a symmetric matrix and A, B are normal matrices.

The matrices must be 16–byte aligned and stored in row major order.

Chapter 10. Additional APIs

This topic describes the additional BLAS APIs that can be used to customize the library.

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

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 55

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 54

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-examples-source.tar**), which is installed with the **blas-examples-source** 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."

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 64
- "BLAS_Free_CB" on page 65
- "BLAS_REGISTER_MEM" on page 66

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-examples-source.tar**), which is installed with the **blas-examples-source** RPM.

Part 6. Appendixes

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*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*

- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *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

- *IBM PowerPC Architecture Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM 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.

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.

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.

program section

See *code section*.

ScaLAPACK

Scalable LAPACK. is a library of parallelized Linear Algebra routines. See “LAPACK” on page 79.

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

M

N

O

P

R

S

T

Printed in USA

SC33-8426-01