IBM Software Development Kit for Multicore Acceleration v3.0
Basic Linear Algebra Subprograms Programmer's Guide and API Reference

First Edition (November 2007)
This edition applies to the version 3, release 0 of the IBM Software Development Kit for Multicore Acceleration

Table of Contents

PREFACE ..... **iii**
About this publication iii
Intended audience iii
Conventions iii
Typographical conventions iii
Related information iii
Part I. Overview **1**
Chapter 1. BLAS Introduction 2
Part II. Configuring BLAS **4**
Chapter 2. Installing and Configuring BLAS 5
Description of Packages 5
Installation Sequence 7
Part III. Programming with BLAS **8**
Chapter 3. Programming 9
Basic Structure of BLAS Library 9
Usage of BLAS Library (PPE-Interface) 9
Usage of BLAS SPE-Library 10
Tuning BLAS library for performance 10
Programming tips to achieve maximum performance 12
Chapter 4. Programming for Cell BE 13
SPE Thread Creation 13
Support of User-specified SPE and Memory Callbacks 13
Debugging Tips 14
Part IV. BLAS API Reference **15**
PPE API 16
SPE API 17
sscal_spu / dscal_spu 18
scopy_spu / dcopy_spu 19
saxpy_spu / daxpy_spu 20
sdot_spu / ddot_spu 21
isamax_spu / idamax_spu 22
sgemv_spu / dgemv_spu 23
sgemm_spu / dgemm_spu 25
ssyrk_64x64 26
strsm_spu / dtrsm_spu 27
strsm_64x64 29
Additional APIs 31
SPE management API 31
Memory management API 39
Part V. Appendixes **42**
Appendix A. Accessibility 43
Appendix B. Notices 44

PREFACE

About this publication

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

Intended audience

The target audience for this document is application programmers using the IBM Software Development Kit (SDK) for Multicore Acceleration. The reader is expected to have a basic understanding of programming on the Cell platform and common terminology used with Cell BE.

Conventions

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

All the documents listed in this section will be available with IBM Software Development Kit (SDK) for Multicore Acceleration. Newer version of some of the documents can be found in the respective links given.

Cell BE

There is a set of tutorial and reference documentation for the Cell BE stored in the IBM online technical library at:
http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

Cell Broadband Engine Architecture

Cell Broadband Engine Programming Handbook

PPU & SPU C/C++ Language Extension Specification

Synergistic Processor Unit (SPU) Instruction Set Architecture

Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification

Cell BE programming using the SDK

IBM Software Development Kit (SDK) for Multicore Acceleration v3.0 Installation Guide

IBM Software Development Kit (SDK) for Multicore Acceleration v3.0 Programmer's Guide

Cell Broadband Engine Programming Tutorial

SIMD Math Library Specification for Cell Broadband Engine Architecture

IBM Full-System Simulator

IBM Full-System Simulator User's Guide

IBM Full-System Simulator Command Reference

Performance Analysis with the IBM Full-System Simulator

BLAS

Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard, August 2001 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,
http://www.netlib.org/blas/blas3-paper.ps

Basic Linear Algebra Subprograms – A quick reference guide, May 1997,
http://www.netlib.org/blas/blasqr.pdf

Part I. Overview

Chapter 1. BLAS Introduction

The BLAS (Basic Linear Algebra Subprograms) 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 API is available as standard ANSI C and standard FORTRAN 77/90 interfaces. BLAS implementations are also available in open-source (netlib.org). Based on its functionality, BLAS is divided into 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.

Each routine has 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 IBM Software Development Kit (SDK) for Multicore Acceleration supports only real single precision and real double precision versions (hereafter referred to as SP and DP respectively). 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 theses routines have been optimized using the Synergistic Processing Elements (SPEs) and these show a marked increase in performance in comparison to the corresponding versions implemented solely on the PPE.. These optimized routines have an SPE interface in addition to the PPE interface; 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; moreover, the single precision versions of these routines have been further optimized for maximum performance using various features of the SPE (e.g., SIMD, Dual Issue, etc.):

Level 1:

- SSCAL, DSCAL
- SCOPY, DCOPY
- ISAMAX, IDAMAX

- SAXPY, DAXPY
- SDOT, DDOT

Level 2:

- SGEMV, DGEMV (TRANS='No Transpose' and INCY=1)

Level 3:

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

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 which accepts input matrices stored in column-major order.

Part II. Configuring BLAS

Chapter 2. Installing and Configuring BLAS

The following sections describe the installation and configuration of BLAS library.

Description of Packages

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

1. blas-3.0-x.ppc.rpm

Purpose: This rpm will install BLAS library 'libblas.so.x'.

Platform: PowerPC Architecture™ and IBM BladeCenter® QS21.

Contents:

/usr/lib/libblas.so.x : BLAS library

2. blas-devel-3.0-x.ppc.rpm

Purpose: This rpm installs supporting files such as header files and sample applications of BLAS required by the library.

Platform: PowerPC Architecture™ and IBM BladeCenter® QS21.

Contents:

/usr/include/blas.h: Contains prototypes of all BLAS Level 1, 2 and 3 (SP and DP) functions that have a PPE API in the library. PPE API refers to Standard BLAS API 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 libblas.so.x

/usr/spu/include/blas_s.h: Contains prototypes of selected functions of BLAS Level 1, 2 and 3 (SP and DP) that have an SPE API in the library. These functions have limited functionality and are not as generic as the PPE API.

/usr/spu/lib/libblas.a: BLAS SPE library.

/opt/cell/sdk/src/blas.tar: Compressed file of BLAS samples .

3. blas-3.0-x.ppc64.rpm

Purpose: This rpm installs the BLAS library libblas.so.x.

Platform: PowerPC Architecture™-64 bit and IBM BladeCenter® QS21.

Contents:

/usr/lib64/libblas.so.x : BLAS library

4. blas-devel-3.0-x.ppc64.rpm

Purpose: This rpm installs supporting files such as header files and sample applications of BLAS for developing sample applications using BLAS library.

Platform: PowerPC Architecture™-64 bit and IBM BladeCenter® QS21.

Contents:

/usr/lib64/libblas.so: Soft link to libblas.so.x

5. blas-cross-devel-3.0-x.noarch.rpm

Purpose: This rpm installs the BLAS library and supporting files such as header files and sample applications of BLAS required by the library.

Platform: Other platforms, such as x86 series.

Contents:

/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 API.

/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 PPE API.

/opt/cell/sysroot/usr/lib/libblas.so: Soft link to libblas.so.x

/opt/cell/sysroot/usr/lib/libblas.so.x: BLAS library

/opt/cell/sysroot/usr/lib64/libblas.so: Soft link to libblas.so.x (64 bit)

/opt/cell/sysroot/usr/lib64/libblas.so.x: 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 (SP and DP) that have an SPE API in the library. These functions have limited functionality and are not as generic as the PPE API.

/usr/spu/lib/libblas.a: BLAS SPU library.

/opt/cell/sdk/src/blas.tar: Compressed file of BLAS samples.

Installation Sequence

Use the following sections to install the BLAS library.

Platform: PowerPC Architecture™ and IBM BladeCenter® QS21.

Install the following packages in order, using "rpm –ivh":

blas-3.0-x.ppc.rpm

blas-devel-3.0-x.ppc.rpm

Platform: PowerPC Architecture™-64 bit and IBM BladeCenter® QS21.

Install the following packages in order, using "rpm –ivh":

blas-3.0-x.ppc64.rpm

blas-devel-3.0-x.ppc64.rpm

Platform: Other platforms, such as x86 series.

To install on other platforms such as x86 series. install the following package using "rpm –ivh":

blas-cross-devel-3.0-x.noarch.rpm

Note: Make sure to install blas-3.0-x.xxx.rpm before you install blas-devel-3.0-x.xxx.rpm.

Part III. Programming with BLAS

Chapter 3. Programming

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

Basic Structure of BLAS Library

The BLAS Library has two components: PPE interface and SPE interface.
PPE applications can use the standard BLAS PPE API (set forth by BLAS Technical Forum Standard, see documents listed under "BLAS" in the "Related Information" section) and the SPE programs can directly use the SPE API. A detailed description of the SPE interface is provided in the API reference section.

Usage of BLAS Library (PPE-Interface)

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

```
#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;
}
```

Usage of BLAS SPE-Library

The following sample application demonstrates the usage of the BLAS-SPE library. This application program invokes **sdot**, using the BLAS-SPE library.

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

float buf_x[BUF_SIZE] __attribute__ (( aligned (16) )) ;
float buf_y[BUF_SIZE] __attribute__ (( aligned (16) )) ;

/********************** MAIN ROUTINE **********************/
int main()
{
    int   size,    k ;
    float sum = 0.0 ;
    size = BUF_SIZE;

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

    sum = sdot_spu( buf_x, buf_y, size ) ;

    return 0 ;
}
```

Tuning BLAS library for performance

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

The optimized BLAS level 3 routines utilize extra space for suitably reorganizing 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. Moreover, for achieving better performance it is 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 this overhead 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. The user can achieve considerable performance improvement for small matrices through the use of swap space.

Environment Variables

There are many environment variables available to customize the launching of SPE and memory allocation in the BLAS library. However, for full control, you can register and use your own SPE and Memory callbacks (described in the following sections). The following table lists the environment variables:

Variable Name	Purpose
BLAS_NUMSPES	Specifies the number of SPEs to use. The default is 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. The default is to use huge pages. Set the variable to 0 to use heap instead.
BLAS_HUGE_PAGE_SIZE	Specifies the huge page size to use, in KB. The default value is 16384 KB (16 MB). The huge page size on the system can be found in the file /proc/meminfo.
BLAS_HUGE_FILE	Specifies the name of the file to be used for allocating new space using huge pages in BLAS3 routines. The default 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. The default NUMA node is -1 which indicates no NUMA binding.
BLAS_SWAP_SIZE	Specifies the size of swap space, in KB. The default is not to use swap space.
BLAS_SWAP_NUMA_NODE	Specifies the NUMA node on which swap space is allocated. The default

	NUMA node is -1 which indicates no NUMA binding.
BLAS_SWAP_HUGE_FILE	Specifies the name of the file that will be used to allocate swap space using huge pages. The default filename is /huge/blas_lib_swap.bin.

The following example shows how a test program, *test_name,* can be run with 5 SPEs, using binding on NUMA node 0 and using 12MB 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:

1) Make the matrices/vectors 128 byte aligned – Memory access is more efficient when the data is 128 byte aligned.
2) Use huge pages to store vectors and matrices. By default, the library uses this feature for memory allocation done within the library.
3) 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.
4) Use the swap space feature described in the earlier section, for matrices smaller than 1K with appropriate NUMA binding.
5) The library gives better performance while working on vectors and matrices of large sizes. Performance of optimized routines is better when the stride value is 1. Level 3 routines show good performance when the number of rows and columns are a multiple of 64 for Single Precision (SP) and 32 for Double Precision (DP).

Chapter 4. Programming for Cell BE

This chapter 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 as discussed previously. 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.

SPE Thread Creation

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 default SPE management provided by the BLAS library. The sample application in the "Usage of BLAS Library (PPE-Interface)" section in "Chapter 3 – Programming" 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 the "Environment Variables" section.

Support of User-specified SPE and Memory Callbacks

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/destroying SPE threads, SPE program loading or context creation) with the registration function **blas_register_spe()**[1] provided by the BLAS library.

As mentioned earlier, the optimized level 3 routines in the library utilize some extra space for suitably reorganizing the input matrices. The library uses default memory management routines to allocate/de-allocate 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/de-allocation routines for allocating new space for reorganizing the input matrices. To do this, use the registration function **blas_register_mem()**.

[1] See the "Additional APIs" section under "BLAS API Reference" for more details on the APIs provided by BLAS library to register user-specified SPE and Memory management callbacks.

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

Debugging Tips

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

1) For using huge pages, the library assumes that a filesystem of type hugetlbfs is mounted on /huge directory. In case hugetlbfs filesystem 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.

2) If you receive a bus error 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 to use heap for memory allocation instead of huge pages.

3) When you are using the SPE API, make sure the alignment and parameter constraints are met. The results can be unpredictable if these constraints are not satisfied.

Part IV. BLAS API Reference

The BLAS library provides two sets of interfaces - a PPE interface and an SPE interface. The PPE interface conforms to the Standard BLAS interface. The library also provides additional functions to customize the library.

PPE API

PPE API is available for all SP and DP standard BLAS routines. The current PPE API does not support complex single precision and complex double precision versions of BLAS routines. The PPE API conforms 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 of these routines is available at http://www.netlib.org/blas/blast-forum/blas-report.pdf. For further information regarding BLAS, refer to netlib documentation on www.netlib.org.

SPE API

The library provides an SPE API only for certain routines. This API does not conform to the existing BLAS standard. There are constraints on the functionality (range of strides, sizes, etc) 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 this API.

sscal_spu / dscal_spu

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, wherein 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.

Syntax

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)

Example

```
#define len 1024
float buf_x[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 ;
   }

   sscal_spu( buf_x, alpha, size ) ;

   return 0 ;
}
```

scopy_spu / dcopy_spu

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, wherein *n* consecutive elements starting with first element get copied.

Syntax

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.

Example

```
#define len 1000

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

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

   scopy_spu( buf_x, buf_y, size ) ;

   return 0 ;
}
```

saxpy_spu / daxpy_spu

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 for DP and 64 for SP. The *x* and *y* vectors must be aligned at a 16 byte boundary.

Syntax

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.

Example

```
#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

Description

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

$result \leftarrow x \cdot y$

where *x* and *y* are vectors. 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. *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.

Syntax

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

Return Values

float/double	Dot product of the two vectors

Example

```
#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;k<size;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

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:

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

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 16 for DP and 64 for SP. The *x* vector must be aligned at a 16 byte boundary.

Syntax

int isamax_spu (float *sx, int n)

int idamax_spu (double *dx, int n)

Parameters

`sx/dx`	Pointer to vector (x) of floats/doubles
`n`	Integer storing number of vector elements

Return Values

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

Example

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

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

   int index ;

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

sgemv_spu / dgemv_spu

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 for SP and 16 for DP. *n* must be a multiple of 8 for SP and 4 for DP. All the input vectors and matrix must be 16-byte aligned.

Syntax

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

Example

```
#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 / dgemm_spu

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 for SP and 2 for DP. *n* must be multiple of 16 for SP and 4 for DP. *k* must be multiple of 4 for SP and 2 for DP.

Syntax

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)

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

Example

```
#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

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. Unlike the equivalent PPE API this routine supports only SP version (ssyrk). Also, the matrices must be of size 64x64.

Syntax

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.

Example

```
#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;
}
```

strsm_spu / dtrsm_spu

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 for SP and 2 for DP. *m* must be a multiple of 8 for SP and 4 for DP. Matrices A and B must be aligned at a 16 byte boundary and must be stored in row-major.

Syntax

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

void dtrsm_spu (int m, int n, double *a, double *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

Example

```
#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;

}
```

See Also

strsm_64x64

strsm_64x64

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 64x64. Hence better performance is got for 64x64 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.

Syntax

void strsm_64x64 (float *a, float *b)

Parameters

`a`	Pointer to matrix A
`b`	Pointer to matrix B

Example

```
#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

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 API for customizing the use of SPE resources and Memory management API 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 API

This API 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 etc. These registered routines override the default SPE management mechanism inside the BLAS library.

The following data types and functions are provided as part of this API:

spes_info_handle_t

spe_info_handle_t

These two data structures are a simple typedef to void.

spes_info_handle_t is used as a handle to access information about *all* the SPEs that will be used by BLAS library. *spe_info_handle_t* is used as a handle to access information about a *single* SPE in the pool of multiple SPEs that is used by BLAS library.

The user provides 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.

Example

For example a user can define 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> );
```

BLAS_NUM_SPES_CB

This is a callback function prototype that is registered to obtain the maximum number of SPEs to be used in the BLAS library.

Syntax

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.

Return Values

int	Number of SPEs that will be used in the BLAS library.

Example

```
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

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.

Syntax

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 will identify a single SPE in the data pointed to by **spes_info**. The BLAS library will first invoke 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.

Return Values

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

Example

```
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

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.

Syntax

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.

Example

```
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

This is a callback function prototype that is registered to wait for the completion of a running SPE program on a single SPE, i.e., 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.

Syntax

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.

Example

```
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

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 will 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().

Syntax

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	A pointer to user-defined data which contains information about all the SPEs to be used in the BLAS library. The BLAS library passes this pointer to registered callback routines while invoking these routines.
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.

Example

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 API section.

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 API

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

The following functions are provided with this API:

BLAS_Malloc_CB

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

Syntax

void* (*BLAS_Malloc_CB) (size_t size);

Parameters

size	Memory size in bytes to be allocated.

Return Values

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

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

Syntax

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.

BLAS_REGISTER_MEM

Description

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

Syntax

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.

Example

```
#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 V. Appendixes

Appendix A. Accessibility

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.

Appendix B. 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.

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 might 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.

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
2Z4A/101
11400 Burnet Road
Austin, TX 78758
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 document 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.

Trademarks