Optimization of BLAS on the Cell Processor

Vaibhav Saxena[*], Prashant Agrawal[*], Yogish Sabharwal[*], Vijay K. Garg[*], Vimitha A. Kuruvilla[†], John A. Gunnels[‡]

Abstract

The unique architecture of the heterogeneous multi-core Cell processor offers great potential for high performance computing. It offers features such as high memory bandwidth using DMA, user managed local stores and SIMD architecture. In this paper, we present strategies for leveraging these features to develop a high performance BLAS library. We propose techniques to partition and distribute data across SPEs for handling DMA efficiently. We show that suitable pre-processing of data leads to significant performance improvements, particularly when data is unaligned. In addition, we use a combination of two kernels – a specialized high performance kernel for the more frequently occurring cases and a generic kernel for handling boundary cases – to obtain better performance. Using these techniques for double precision, we obtain up to 70-80% of peak performance for different memory bandwidth bound BLAS level 1 and 2 routines and up to 80-90% for computation bound BLAS level 3 routines.

1 Introduction

Recent trends in processor design exhibit a predominant shift toward multi-core architectures, primarily driven by increasing power consumption and the diminishing gains in processor performance from increasing operating frequency. The Cell Broadband Engine, also referred to as the Cell processor, is a multi-core processor jointly developed by Sony, Toshiba and IBM. The Cell is a radical departure from conventional multi-core architectures – combining a conventional high-power PowerPC core (PPE) with eight simple Single-Instruction, Multiple-Data (SIMD) cores, called Synergistic Processing Element (SPE) in a heterogeneous multi-core offering. It offers extremely high compute-power on a single chip combined with a power-efficient software-controlled memory hierarchy. The theoretical peak performance of each SPE for single precision floating point operations is 25.6 GFLOPS leading to an aggregate performance of 204.8 GFLOPS for 8 SPEs. The theoretical peak performance for double precision is 12.8 GFLOPS per SPE and 102.4 GFLOPS aggregate. Each SPE has 256 KB of Local Store for code and data. An SPE cannot directly access

[*] *IBM India Research Lab, New Delhi 110070. Email: {vaibhavsaxena, prashant_agrawal, ysabharwal, vijgarg1}@in.ibm.com*

[†] *IBM India STG Engineering Labs, Bangalore 560071. Email: vimitha.k@in.ibm.com*

[‡] *IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. Email: gunnels@us.ibm.com*

the data stored in an off-chip main memory and explicitly issues Direct Memory Access (DMA) requests to transfer the data between the main memory and its local store. Access to the external memory is handled via a 25.6 GB/s Rambus extreme data rate (XDR) memory controller. The PPE, eight SPEs, memory controller and input/output controllers are connected via the high bandwidth Element Interconnect Bus (EIB) [20], as shown in Fig. 1.

Distinctive features of the Cell such as the XDR memory subsystem, coherent EIB interconnect, SPEs, etc. make it suitable for computation and data intensive applications. There has been a considerable amount of work that has demonstrated the computational power of the Cell processor for a variety of applications, such as dense matrix multiply [32], sparse matrix-vector multiply [32], fast Fourier transforms [3, 8, 16, 32], sorting [14], ray tracing [6] and many others. The Cell processor is commercially available in various platforms such as Sony Playstation 3 gaming console and IBM BladeCenter (with 2 Cell processors connected in a NUMA configuration).

Numerical linear algebra is fundamental to scientific computing, financial engineering, image and signal processing, data mining, bioinformatics, and many other applications. It is often the most computationally intensive part of these applications. Basic Linear Algebra Subprograms (BLAS) is a widely accepted standard for linear algebra interface specifications in high-performance computing and scientific domains and forms the basis for high quality linear algebra packages such as LAPACK [1] and LINPACK [10]. BLAS routines are categorized into three classes – level 1 routines (vector and scalar operations), level 2 routines (vector-matrix operations) and level 3 routines (matrix-matrix operations). Level 2 and level 3 routines were motivated by the advent of vector machines, hierarchichal memory machines and shared memory parallel machines. There exists a rich set of documentation [26] that discusses general optimization strategies for BLAS routines as well as optimization for specific architectures.

BLAS has been tuned and optimized for many platforms to deliver good performance, e.g. ESSL on IBM pSeries and Blue Gene [23], MKL for Intel [24], GotoBLAS on a variety of platforms [15], etc. Successful efforts have also been made towards automatic tuning of linear algebra software (ATLAS) [2] to provide portable performance across different platforms using empirical techniques. Some of these portable libraries give good performance when executed on the Cell PPE. However, given the unique architecture of the Cell processor and the SPE feature set, specialized code needs to be designed and developed for obtaining high performance BLAS for the Cell processor. Williams et al. [32] have discussed optimization strategies for the general matrix-multiply routine on the Cell processor, obtaining near-peak performance [18, 21, 26, 27, 29, 31]. However, existing literature and optimization strategies of linear algebra routines on the Cell make simplified assumptions regarding the input data related to their alignment, size, etc. A BLAS library needs to address many issues for completeness, such as different alignments of the input vectors/ matrices, unsuitable vector/ matrix dimension sizes, vector strides, etc. that can have significant impact on the performance. Moreover, there are many combinations of input parameters in BLAS routines, such as transpose or non-transpose, upper or lower triangular, diagonal or non-diagonal, stride or non-stride vector access, positive or negative stride vector access, etc. This leads to a large number of combinations of the operations to be performed. Hence productivity is a very important factor to be considered while optimizing the BLAS library.

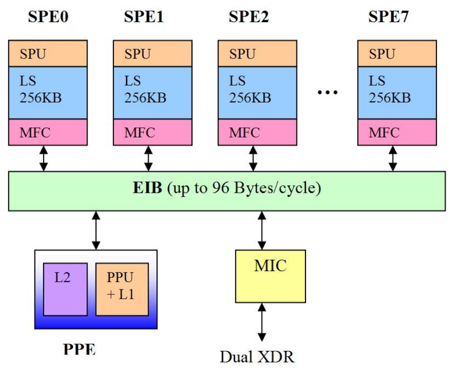


Figure 1: Cell Broadband Engine Overview

For example, a commonly used technique is the reuse of a highly optimized GEMM core routine in developing other BLAS level 3 routines [25].

In this paper, we discuss the challenges and opportunities involved in optimizing BLAS for the Cell processor. We focus on the optimization strategies used for producing the high performance BLAS library that is shipped with the Cell Software Development Kit (SDK). The library consists of single and double precision routines ported to the PPE; a selected subset of these routines have been optimized using the SPEs. The routines conform to the standard BLAS interface at the PPE level. This effort, of offering a high performance BLAS library, is the first of its kind for the Cell. We propose techniques to partition and distribute data across SPEs for handling DMA efficiently. We show that suitable pre-processing of data leads to significant performance improvements, particularly when data is unaligned. In addition, we use a combination of two kernels – a specialized high performance kernel for the more frequently occurring cases and a generic kernel for handling boundary cases – to obtain better performance.

The rest of the paper is organized is as follows. In Section 2 we discuss the challenges and opportunities that the Cell offers with respect to BLAS. In Section 3, we discuss the optimization strategies followed by performance results in Section 4. We conclude in Section 5 with a brief discussion of the ongoing and future planned work.

2 Challenges and Opportunities

The non-traditional architecture of the Cell demands more than a simple re-compilation of the application to achieve high performance. Therefore, a good design of the BLAS library needs to address the challenges posed by the unique architecture of the Cell and efficiently leverage the opportunities that it has to offer.

The most important factor in developing a high performance library such as BLAS for any multi-core architecture is the task/data partitioning and distribution, i.e. breaking down the routines into smaller operations that can be mapped to multiple cores. Most BLAS routines are highly data-parallel [9, 13] – they can be broken down into smaller but similar set of operations that can be scheduled on separate cores. Even for routines that have inherent dependencies, a careful data partitioning and distribution strategy can lead to high dataparallelism for a significant part of the computation. The data partitioning and distribution strategy depends on various factors such as input data sizes, vector or matrix operations, load balancing considerations, etc.

There are other architecture specific features that need to be considered for obtaining good performance of BLAS on the Cell. These features are described in the subsequent sections.

2.1 Memory Hierarchy and DMA

One of the major factors limiting the processor performance is the widening gap between the processor frequency and the memory latency. Memory hierarchy is a popular technique used in processor architecture to reduce the impact of high memory latencies.

On the Cell processor, the memory hierarchy of the PPE is similar to conventional processors whereas SPEs have a distinctive three-level hierarchy: (a) 128×128-bit unified SIMD register file, (b) 256 KB of local store memory, and (c) shared off-chip main memory. Each SPE works only on the code and data stored in its local store memory and uses DMA transfers to move data between its local store and the main memory (or the local stores of other SPEs).

These DMA transfers are asynchronous and enable the SPEs to overlap computation with data transfers. Although the theoretical peak memory bandwidth is 25.6 GB/s, the effective bandwidth obtained may be considerably lower if the DMA transfers are not setup properly. This can degrade performance of BLAS routines, particularly level 1 and level 2 routines, which are typically memory bandwidth bound. Therefore, it is important to take into account the factors that influence the DMA performance in order to develop a high performance BLAS library. Some of these factors are discussed here.

Memory Alignment: DMA performance is best when both source and destination buffers are 128-byte (one cache line) aligned and the size of the transfer is a multiple of 128 bytes. This involves transfer of full cache lines between main memory and local store. If the source and destination are not 128-byte aligned, then DMA performance is best when both have the same quadword offset within a cache line. This affects the data partitioning strategy. Typically, an SPE works on blocks of the input data by iteratively fetching them from main memory to its local store, performing required operation on these blocks and finally storing back the computed data blocks to main memory. Therefore, it is important to partition the input data in a manner such that the blocks are properly aligned so that their DMA transfers are efficient.

Transfer of unaligned data may result in the use of DMA lists. However, direct (contiguous) DMA transfers generally lead to better bandwidth utilization in comparison to DMA list accesses. To illustrate this, consider a block of size $16x + 12$ bytes starting at a 128 byte aligned address. DMA transfers can be done in units of 1, 2, 4, 8 and multiple of 16 bytes starting at memory addresses that are 1, 2, 4, 8 and 16 byte aligned, respectively. One way of transferring this block is to construct a DMA list that has 3 list elements, one each for (1) the $16x$ byte aligned part, (2) the 8 byte part and (3) the 4 byte part. As each transfer consumes 128 bytes worth of bandwidth, there may be close to 128 bytes worth of bandwidth loss for each of the transfers. A better strategy is to transfer some extra bytes at the tail, making the transfer size a multiple of 16 bytes so that a direct DMA transfer can be used. When fetching data, the extra fetched bytes can be discarded by the SPE. However this strategy cannot be used for writing data back to main memory as it can lead to memory inconsistencies. Hence, DMA lists need to be used for writing back unaligned data.

Data Access Pattern: Input vectors may be accessed contiguously or non-contiguously depending upon stride value. When the data is stored contiguously, a direct DMA transfer can be used to move the data. However for non-contiguous access, a DMA list needs to be created and used to scatter/ gather the required elements. Every DMA list element consumes at least 128 bytes worth of bandwidth, independent of the size of the transfer. This opens the possibility of rearranging data in certain cases (e.g. vectors in BLAS 2 routines) that may be reused multiple times resulting in improved overall performance.

Data Block Size: Selection of an appropriate block size is critical for high performance. Large data block size not only improves DMA efficiency but also results in sufficient computations to hide overlapped DMA latencies. DMA efficiency improves as it leads to larger transfers per DMA request which results in higher memory bandwidth. However, SPE local store size limitation of 256 KB restricts the data block sizes that can be handled and places demanding constraints on the size of the SPE code.

2.2 SPE Architecture

Each SPE is a dual issue SIMD processor. It supports in-order execution of the instructions that operate on multiple elements in the 128-bit registers in parallel. SPE instructions typically have a latency of 2-9 cycles with single cycle throughput in most cases. Each SPE has two pipelines with which it can simultaneously issue two independent instructions per cycle – one to each of the pipelines. The availability of a large number of registers allows compilers to efficiently schedule instructions by loop unrolling and other techniques to resolve code dependencies and hide instruction latencies.

An efficient BLAS library on the Cell requires design and development of highly optimized SPE computation kernels that make effective use of the above mentioned SPE features to perform specialized BLAS-like operations on the data present in the local store.

3 Optimization Strategies

Algorithmic and architecture specific optimizations of linear algebra libraries such as BLAS and LAPACK, have been well studied [1, 2, 15, 24, 26]. However, several factors have to be taken into consideration when applying these proven strategies on the Cell processor. Besides, new techniques are required for enabling high performance of routines like BLAS on the Cell, as discussed in Section 2. In this section we discuss the different strategies used for optimizing BLAS on the Cell. It should be noted that these strategies are targeted for a single Cell processor, large data sets, column-major matrices and huge memory pages (16 MB).

3.1 Data Partitioning and Distribution

Data partitioning and distribution are a critical part of designing linear algebra subprograms on multi-cores. The proposed strategy for data partitioning and distribution differs across the three categories of the BLAS routines. For the memory bandwidth bound level 1 and level 2 routines, data partitioning is carried out with an objective to get close to the peak memory-bandwidth, whereas for the computation bound level 3 routines the objective is to get close to the peak computation rate.

BLAS Level 1 Routines: BLAS level 1 routines typically operate on one or two vector(s) and produce as output a vector or a scalar. The goal is to partition the data into equal sized blocks that can be distributed to the SPEs with each SPE getting roughly an equal number of blocks. When the output is a vector, the output (e.g. in DCOPY) or I/O[1] (e.g. in DSCAL) vector is divided into blocks that are 128-byte aligned, are multiple of 128 bytes and large enough (16 KB – the maximum transfer size for a single DMA operation). These blocks are then divided (almost) equally, among the SPEs, with each SPE getting a contiguous set of blocks.

In the case of two vectors – an input and an I/O vector (e.g. in DAXPY), both the vectors have to be partitioned such that an SPE receives the same range of elements of each vector. One of the two vectors is divided into blocks taking into consideration the 128-byte alignment of the blocks, as described above. The other vector is divided with respect to the former vector, without considering the memory alignment, such that its blocks have the same range of elements as the former vector. In our strategy, we partition and distribute the I/O vector based on 128-byte alignment considerations and DMA in the corresponding part of the other vector by pulling in 128 bytes extra (if required). This ensures that the DMA writes from local store to main memory can be performed without the need for DMA lists. In cases where the output is a scalar (e.g. in DDOT), partitioning with memory alignment considerations can be carried out for any of the vectors.

When the vector being partitioned does not start or end on a 128-byte boundary, there may be small parts of the vector at the start (head) and the end (tail) that do not satisfy the alignment and size criteria mentioned above. These are handled directly on the PPE.

In case where access for one or more vectors is strided, the size of each block is restricted to 2048 elements (the maximum number of DMA transfers that can be specified in a single DMA list operation).

BLAS Level 2 Routines: BLAS level 2 routines perform matrix-vector operations and their output can either be a vector or a matrix. The complexity of these routines is determined by the memory bandwidth requirements for fetching/storing the matrix. Thus, data partitioning and distribution for these routines is done keeping in mind efficient DMA considerations for the matrix. The column-major matrix is divided into rectangular blocks which are distributed among the SPEs. The SPEs typically operate on one block in an iteration. A block is fetched using a DMA list where each list element transfers one column of the block. To improve the efficiency of the DMA, column sizes of the block should be large and multiples of 128 bytes. In case where column start addresses are not 128-byte aligned,

[1]Data that is both read and updated

up to 128 extra bytes are fetched for each column in order to ensure that each column can be transferred using a single DMA list element. The block dimensions are appropriately chosen depending on the number of vectors used and SPE local store size.

If the output is a vector and there are two vectors – an input and a I/O vector (e.g. in DGEMV), the I/O vector is divided into blocks by taking memory alignment into consideration, as it is done for level 1 routines. The I/O vector blocks are divided among the SPEs uniformly with each SPE getting a set of contiguous blocks. Each SPE fetches an I/O vector block, iteratively fetches the blocks of the matrix and the input vector required for the computation, carries out the computation and writes back the I/O vector block to the main memory.

If there is only one I/O vector (e.g. in DTRMV), a block of elements cannot be updated until all the computations involving it are completed. To resolve this dependency, a copy of the vector is created and is used as the input vector. The SPEs can then independently update the blocks of the output vector.

BLAS Level 3 Routines: BLAS level 3 routines perform matrix-matrix operations and are computationally intensive. Thus, the key consideration in data partitioning and distribution for these routines is computational efficiency. The matrices are partitioned into square blocks (to maximize computations in order to hide DMA latencies) instead of rectangular blocks (which are more DMA efficient) as in the case of level 2 routines . The blocking factor of the matrices is decided based on factors such as SPE local store size and the number of input and output matrices being operated upon. Another important factor influencing the blocking factor is that when up to 16 SPEs are used on multi-Cell processor platforms, such as the IBM BladeCenter, the blocksize should result in sufficient computations so that the routine does not become memory bandwidth bound. Taking all these constraints into consideration, we have determined that a blocking factor of 64×64 can be used with the given memory constraints and is sufficient to keep the BLAS level 3 routines computation bound, even with 16 SPEs.

When there are no dependencies in the computation of the output matrix blocks (e.g. in DGEMM), these blocks are distributed across the SPEs and each SPE determines at runtime the output matrix block to process. This dynamic distribution of the blocks ensures a better load balancing across the SPEs. An SPE fetches an output matrix block, iteratively fetches the input matrices blocks required for the computation of the output block, carries out the computation and stores back the computed block to main memory. Since input matrix blocks are used multiple times in the computation of different output matrix blocks, the input matrices are reformatted before the computation (see Section 3.2 for more details) to improve the DMA efficiency for the transfer of these blocks.

In case where there are dependencies in the computation of the output matrix blocks, the blocks are distributed across the SPEs such that the computation across these sets are independent as much as possible. The order of computation of the blocks within a set is routine specific, e.g. in case of TRSM while computing $B \leftarrow A^{-1} \cdot B$, where A is a lower triangular matrix, dependencies exists in the computation of the elements along a column but there is no dependency among elements in different columns. Therefore the columnsets[2]

[2]A set of blocks along the column of the matrix; columnset i refers to the set of all the i^{th} blocks in each

can be computed independently. Thus for TRSM, the columnsets of the output matrix are distributed across the SPEs and the blocks are processed in the top-down order within a columnset. Similar distribution can be used for other input parameter combinations as well. The SPEs determine at runtime the sets they should process.

For particular combinations of input parameters, we carry out the complete computation on the PPE if it is more beneficial – for instance when the matrix/vector dimensions are so small that SPE launching overheads exceed computation time.

3.2 Efficient DMA Handling

Efficient DMA is critical for high performance of BLAS level 1 and level 2 routines since they are memory bandwidth bound. Even though BLAS level 3 routines are computation bound, the blocks of the matrices are fetched multiple times. Therefore, unless careful attention is given to DMA related aspects, especially alignment related issues, there can be significant performance degradation in the form of creation of DMA lists, packing/unpacking of data in the SPE local store, etc. We discuss some of the DMA related optimizations for BLAS in this section.

Pre-Processing of Input Matrices for BLAS Level 3 Routines: Pre-processing of input data such as data layout transformation, padding, etc. have been used previously for improving performance of BLAS level 3 like operations for various reasons [17, 30, 33]. We demonstrate that such techniques are useful in improving the performance of BLAS routines on the Cell as they improve efficiency of the underlying DMA operations. We also show how we can apply simple operations during the pre-processing step in order to reduce computations during the actual processing, thereby improving performance and increasing productivity. For BLAS level 3 routines, we rearrange the column-major input matrices into block-layout form, using blocksize of 64×64, before performing the operation, so that the columns of a block are stored contiguously starting at 128-byte aligned addresses. The advantages of pre-preprocessing are:

- *Transfer of Blocks Using Direct DMA*: The block columns are not contiguous in memory, and therefore fetching the blocks requires a DMA list of 64 elements where each list element transfers 64 matrix elements. This DMA list has to be created everytime a block is transferred between main memory and local store. When a column does not begin at a 128-byte aligned address, this can lead to significant bandwidth loss. Though this may not impact performance when few SPEs are in service, it can significantly deteriorate performance when there are 16 SPEs – pushing the memory bandwidth to its limits. With pre-processing, each block can be fetched using direct DMA.

- *Reduction in the Number of SPE Kernels*: Several transformations can be applied to input matrices during the pre-processing phase itself. These transformations enhance productivity by reducing the number of different kernels required for different combinations of input parameters such as transpose, triangularity (upper or lower), side (left or right), unit or non-unit triangular, etc. For example, the GEMM operation $C = \alpha A^T B + \beta C$

row of the matrix.

can be performed using the same kernel as the one used for $C = \alpha AB + \beta C$ by simply transposing the matrix A during the pre-processing phase. For the DTRSM routine, we implemented only 2 kernels to cater to 8 different input parameter combinations by applying such transformations. Similar reductions in kernel implementations were achieved for other routines.

- *Simpler and More Efficient SPE Kernels*: The computational kernels on the SPEs are designed to handle matrix blocks which are properly aligned in the local store. This leads to design of simpler kernels that make effective use of the SIMD features of the SPEs without having to realign the vectors/ matrices based on their current alignment offsets. In the absence of pre-processing, either the vector/ matrix blocks would have to be realigned in memory before invoking the SPE kernels, leading to performance degradation, or more complex SPE kernels would have to be designed.

- *Reduction in Computation within SPE Kernels*: BLAS level 3 routines typically involve scaling of the input matrices. This scaling is carried out in the pre-processing stage itself. This eliminates the requirement of scaling being carried out by the SPE kernel thereby reducing its computation.

- *Reduction in Page Faults and Translation Lookaside Buffer (TLB) Misses*: In the absence of pre-processing, adjacent columns may be in different pages when smaller page sizes are used. Pre-processing can potentially reduce TLB misses under such circumstances [30].

We do not reformat the output matrices. This is because blocks of these matrices are typically updated only once (or few times in some cases) after a large number of computations. Therefore, the cost of fetching blocks of these matrices and reformatting them on the SPEs is fairly small and does not lead to significant performance loss.

Clearly, the suggested pre-processing techniques are feasible only for BLAS level 3 routines as BLAS level 2 routines have complexity comparable to the memory bandwidth requirements for fetching/ storing the matrix. However, it is feasible to similarly pre-process the vectors in case of BLAS level 2 routines. For instance, a strided-vector can be pre-processed and copied into contiguous locations so that parts of the vectors can be fetched using direct DMA instead of DMA lists.

All the pre-processing is carried out using SPEs since the SPEs together can attain better aggregate memory bandwidth compared to the PPE. The reformatting of the matrix blocks is independent and therefore lends itself naturally to parallel operations.

Use of Double Buffering: Double buffering is used across all the routines to overlap DMA transfers with computations. In some cases, statically assigning these buffers for all the matrices may not leave enough space in the SPE local store for code and other data structures. However not all the buffers are required at all times. Therefore, in our optimization strategy, we declare a pool of buffers from which buffers are fetched and returned back as and when required. The number of buffers initialized in this pool is the maximum number of buffers required at any point of time in the routine.

Reuse of DMA Lists: When DMA lists are used for data transfers, creation of the lists is an additional overhead. In the case of I/O data, lists are created both while fetching and storing the data. In our implementation, we minimize the overhead of creating the lists by retaining the list created while fetching the data and reusing it while storing it back.

3.3 Two-Kernel Approach for Level 3 Routines

Highly optimized and specialized SPE kernels are a key component of high performance BLAS routines, especially level 3 routines. As mentioned in Section 3.1, the matrices are partitioned into blocks of 64×64 elements. The SPE kernels which are optimized for 64×64 blocks are henceforth referred as *64×64 kernel*. When the dimension of the matrices is not a multiple of 64, blocks in the last rowset[3] and/or columnset of the matrices may not be of dimension 64×64. However, developing a kernel that handles all different block sizes is detrimental to performance – particularly if the kernel has to deal with matrix sizes that are not suitable for SIMD operations. A common and simple solution is to pad the last blocks to make them of a suitable size and use the same kernel. However, if the last blocks have very small dimension (1 in the worst case), this approach entails unnecessary computations.

We adopt a two-kernel strategy to cope with such scenarios. If the dimension of a matrix is not a multiple of 64, zeros are padded along that dimension to make it a multiple of 16. This results in blocks along the border of the matrix whose dimensions are a combination of 16, 32, 48 or 64. In this strategy, a set of two kernels is developed for each required combination – a *64×64 kernel* and a generic kernel which can process blocks of any dimension which is a multiple of 16 elements and is 16-byte aligned. This approach limits the maximum number of padded rows or columns to 15 in the worst case and at the same time ensures that the performance of the generic kernel is acceptable because it can still perform SIMD operations. The generic kernels typically show a degradation of less than 10% in comparison to the *64×64 kernel* performance, as shown in Fig. 2(a).

The use of two-kernel approach places significant demand on the memory requirements in the SPE local store. This is also the case when kernels such as DGEMM are reused for performing other BLAS level 3 operations. However, not all the kernels are required at all times. We use SPE overlays[4][22] to share the same region of memory across multiple kernels. Since one kernel routine is used for most computations (e.g. GEMM in BLAS level 3 routines), the amortized overheads of dynamic code reloading are small.

3.4 Efficient Use of Memory

BLAS routines allocate memory internally for pre-processing and rearranging input matrices/ vectors at runtime. There are overheads associated with allocation of memory and accessing

[3]A set of blocks along the row of the matrix; rowset i refers to the set of all the i^{th} blocks in each column of the matrix.

[4]Overlay is an SPE programming feature supported on the Cell to overcome the physical limitations on code and data size in the SPE. Depending on the frequency of use and interactions of the SPE routines, they are grouped into one or more segments which are further grouped into one or more overlay regions. The segments within the same region share the same space, replacing each other dynamically as and when required.

it for the first time due to page faults and TLB misses. To minimize this overhead across multiple BLAS calls, a small portion of memory, called *swap space*, can be allocated by the user (using environment variables) and retained across multiple calls of the BLAS routines. The *swap space* is allocated using huge pages. If the internal memory required by the BLAS routine is less than the size of the *swap space*, the routine uses the *swap space* else it allocates fresh memory. This leads to considerable improvement in the performance of the BLAS routines when the input data size is small, as shown for DGEMM in Fig. 2(b).

4 Performance Results

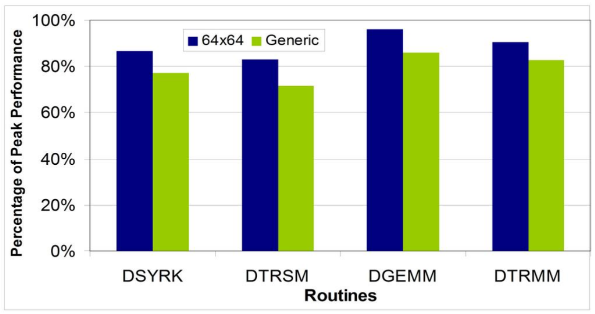
In this section, we report the performance of the BLAS routines obtained with our optimizations. The performance is profiled on IBM Cell Blade (QS22 with 8 GB RAM, RHEL 5.2, Cell SDK 3.0) with enhanced Double Precision pipeline using GCC 32-bit compiler. Huge pages are used by default. For level 1 and 2 routines, the performance is reported in units of GigaBytes per second (GB/s) since they are memory bandwidth bound and for level 3 routines the performance is reported in units of GigaFlops (GFLOPS).

Figure 2(c) shows the performance results for BLAS level 1 routines – IDAMAX, DSCAL, DCOPY, DDOT and DAXPY for ideal input data combinations, i.e. when the starting addresses are 128-byte aligned, stride is 1, dimensions are an exact multiple of their blocksizes. We achieve performance in the range of 70-85% of the peak performance (25.6 GB/s) depending on the routine – routines that largely perform unidirectional transfers (e.g. IDAMAX, DDOT) are observed to perform better than the routines that perform transfers in both directions. For BLAS level 1 routines, the performance for non-ideal cases, e.g. when vector start offsets are not 128-byte aligned, are almost the same and hence not reported.

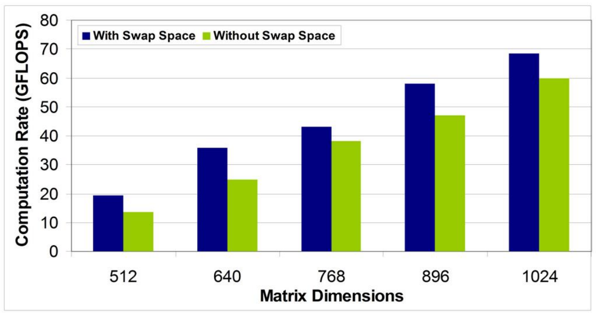
Figure 2(d) compares the performance of BLAS level 2 routines – DGEMV, DTRMV and DTRSV for ideal input cases. We achieve performance in the range of 75-80%% of the peak performance (25.6 GB/s) for BLAS level 2 routines as well. Performance for non-ideal cases (i.e., when data is not properly aligned or leading dimensions are not suitable multiples and vector strides are not 1) is expected to be worse for level 2 routines. This is because for each column within a matrix block, multiple list elements are used to transfer the entire column which results in several short DMAs instead of a single DMA used when the matrix is aligned. Figure 2(e) compares the performance of the DGEMV routine for ideal and non-ideal cases. Performance degrades by about 30% for the unaligned cases. As these routines are memory bandwidth-bound, it is not possible to pre-process the matrix for efficient DMA for unaligned matrices.

For BLAS level 1 and level 2 routines, performance is reported using 4 SPEs. This is because, typically 4 SPEs are enough to exhaust the memory bandwidth and we do not observe significant performance improvement using more SPEs.

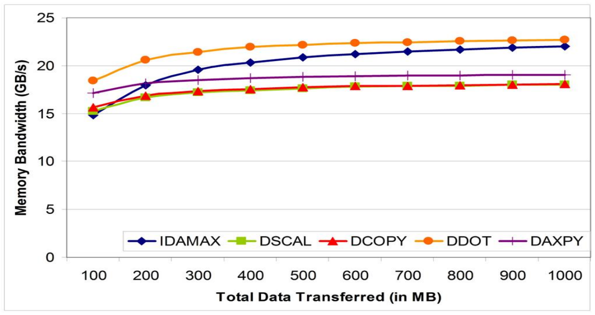
Figure 2(f) shows the performance results of BLAS level 3 routines – DGEMM, DSYMM, DSYRK, DTRSM and DTRMM for ideal input combinations (i.e. when matrix starting addresses are 128-byte aligned and dimensions are multiples of 64). We achieve up to 80-90% of the peak performance (102.4 GFLOPS). Figure 2(g) compares the performance of DGEMM and DTRSM routines for ideal and non-ideal cases. For the non-ideal cases, the



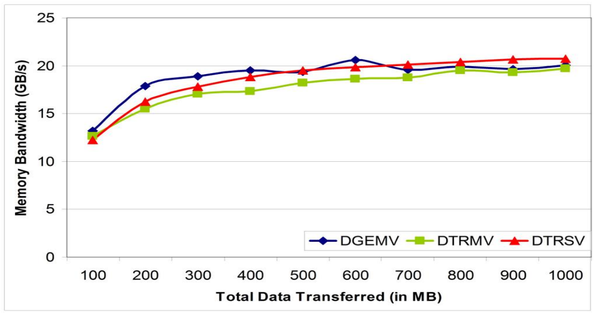
(a) Comparison of performance of 64x64 and generic SPE kernels for Level 3 routines.



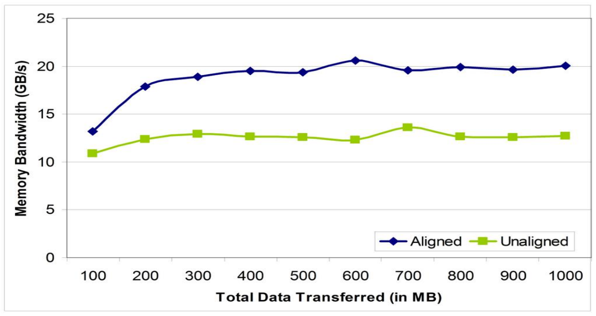
(b) Comparison of performance with and without swap space for DGEMM with 8 SPEs. Swap space size is 16MB.



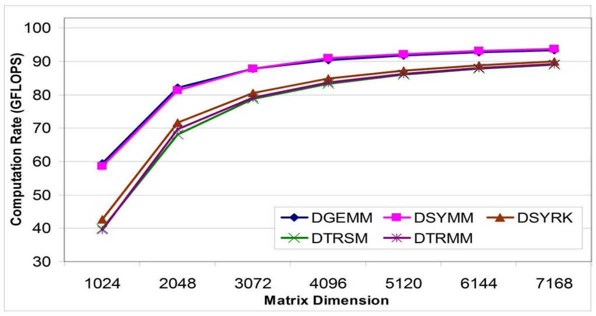
(c) Comparison of ideal case performance of all BLAS level 1 routines with 4 SPEs.



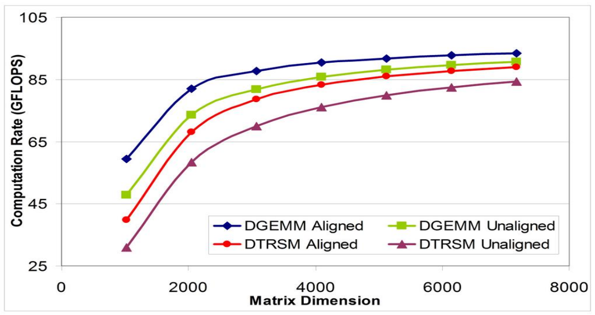
(d) Comparison of ideal case performance of all BLAS level 2 routines with 4 SPEs.



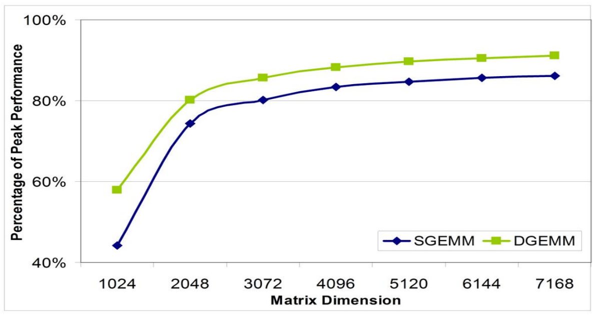
(e) Comparison of ideal and non-ideal case performance of DGEMV with 4 SPEs.



(f) Comparison of ideal case performance of all BLAS level 3 routines for 8 SPEs.



(g) Comparison of ideal and non-ideal case performance of DGEMM and DTRSM for 8 SPEs.



(h) Comparison of ideal case performance of SGEMM and DGEMM for 8 SPEs.

Figure 2: Performance Results

leading dimension is made not to be a multiple of 128 bytes. The performance difference for the non-ideal case is mostly within 10% of the ideal case, demonstrating to a large extent that the pre-processing restricts the performance loss for the non-ideal cases.

We performed additional experiments to determine the advantages of pre-processing. We found that the performance degradation is more than 25% without pre-processing in comparison to our approach. The drop in performance is due to the overhead associated in using DMA lists (bandwidth loss as described in Section 2) for fetching the blocks of the input matrices and pre-processing required in aligning the fetched blocks and/ or performing matrix related operation (e.g. transpose) before invoking the SPE kernel.

In Fig. 2(h), we compare the performance of SGEMM and DGEMM for ideal input combinations to give an idea of the difference in the performance of the single precision and double precision routines. It is observed that the performance of the single precision routines shows performance trends similar to the double precision routines.

5 Conclusions and Future Work

We have discussed the strategies used for optimizing and implementing the BLAS library on the Cell. We have proposed techniques for data partitioning and distribution to ensure optimal DMA and computational performance. We have shown that suitable pre-processing of data results in significant improvement in performance, especially for unaligned data. Besides these, we have proposed a combination of two kernels approach where a specialized high performance kernel is used for more frequently occurring cases and a generic kernel is used to handle boundary cases. Our experimental results for double precision show that the performance of level 1 routines is up to 70-85% of the theoretical peak (25.6 GB/s) for both ideal and non-ideal input combinations. The performance of level 2 routines is up to 75-80% of the theoretical peak (25.6 GB/s) for ideal input combinations. The performance of level 3 routines is up to 80-90% of the theoretical peak (102.4 GFLOPS) for ideal input combinations with less than 10% degradation in performance for non-ideal input combinations. These results show the effectiveness of our proposed strategies in producing a high performance BLAS library on the Cell.

The BLAS routines discussed in this paper have been optimized for a single Cell processor, large size data sets and huge memory pages. There is scope for optimizing these routines to optimally handle special input cases, normal memory pages and for multi-processor Cell platforms.

Acknowledgements

We would like to thank several people who were instrumental in the success of BLAS library development efforts on the Cell. We thank Mike Perks and Shakti Kapoor for supporting the BLAS project as part of the Cell SDK, Bhavesh D. Bhudhabhatti, Chakrapani Rayadurgam, Jyoti S. Panda, Pradeep K. Gupta, Lokesh K. Bichpuriya, Niraj K. Pandey, Amith R. Tudur, Ajay A. Kumar, and Sheshasayee V. Subramaniam for their implementation efforts, and Dan Brokenshire and Mike Kistler for sharing their Cell expertise and giving valuable suggestions.

Finally, we thank Manish Gupta and Ravi Kothari for their guidance and support to the entire team.

References

[1] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide (Third Edition)*, http://www.netlib.org/lapack/lug/index.html, August 1999.

[2] Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/.

[3] D.A. Bader, V. Agarwal, *FFTC: Fastest Fourier Transform on the IBM Cell Broadband Engine*, Proceedings of IEEE International Conference on High Performance Computing, pp. 172–184, December 2007.

[4] Basic Linear Algebra Subprograms: A Quick Reference Guide, http://www.netlib.org/blas, May 1997.

[5] Basic Linear Algebra Subroutine Technical (BLAST) Forum, *BLAST Forum Standard*, http://www.netlib.org/blas/blast-forum/, August 2001.

[6] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich, *Ray Tracing on the Cell Processor*, IEEE Symposium on Interactive Ray Tracing 2006, pp. 15–23, September 2006.

[7] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures*, LAPACK Working Note #190 UT-CS-07-600, University of Tennessee, September 2007.

[8] A.C. Chow, G.C. Fossum, and D.A. Brokenshire, *A Programming Example: Large FFT on the Cell Broadband Engine*, Proceedings of the Global Signal Processing Expo and Conference, 2005.

[9] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R.A. van de Geijn, *Parallel Implementation of BLAS: General Techniques for Level 3 BLAS*, Concurrency: Practise and Experience, 9(9), pp. 837–857, October 1995.

[10] J.J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, USA, 1979.

[11] J.J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16(1), pp. 1–17, March 1990.

[12] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, *An Extended Set of FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14(1), pp. 1–17, March 1988.

[13] J.J. Dongarra, and D.W. Walker, *The Design of Linear Algebra Libraries for High Performance Computers, Technical Report*, LAPACK Working Note #58 UT CS-93-192, University of Tennessee, August 1993.

[14] B. Gedik, R. R. Bordawekar, and P. S. Yu, *CellSort: High Performance Sorting on the Cell Processor*. Proceedings of International Conference on Very Large Data Bases, pp. 1286–1297, September 2007.

[15] GotoBLAS, http://www.tacc.utexas.edu/resources/software/#blas.

[16] J. Greene and R. Cooper, *A Parallel 64K Complex FFT Algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor*, Proceedings of the Global Signal Processing Expo and Conference, 2005.

[17] F.G. Gustavson, *High-Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices*, IBM Journal of Research and Development, 47(1), pp. 31–55, 2003.

[18] D. Hackenberg, *Fast Matrix Multiplication on Cell (SMP) System*, http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/, May 2007.

[19] High Performance Linpack Benchmark (HPL), http://www.netlib.org/benchmark/hpl/.

[20] IBM Corporation, *Cell Broadband Engine Programming Handbook v 1.1*, April 2007.

[21] IBM Corporation, *Cell BE SDK Code Sample*, /opt/cell/sdk/src/demos/matrix_mul.

[22] IBM Corporation, *SDK for Multicore Acceleration v3.0: Programmer's Guide*, October 2007.

[23] IBM Engineering Scientific Subroutine Library (ESSL), http://www-03.ibm.com/systems/p/software/essl/index.html.

[24] Intel Math Kernel Library (MKL), http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm.

[25] B. Kagstrom, P. Ling, and C. van Loan, *GEMM-based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark*, ACM Transactions on Mathematical Software, 24(3), pp. 268–302, September 1998.

[26] LAPACK Working Notes, http://www.netlib.org/lapack/lawns/index.html.

[27] F. Lauginiger, R. Cooper, J. Greene, M. Pepe, M. J. Prelle, *Performance of a Multicore Matrix Multiplication Library*, Proceedings of High Performance Embedded Computing, September 2007.

[28] C.L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN Usage*, ACM Transactions on Mathematical Software, 5(3), pp. 308–323, September 1979.

[29] Mercury Scientific Application Library (SAL), http://www.mc.com/products/productdetail.aspx?id=2836.

[30] N. Park, B. Hong, and V. K. Prasanna, *Tiling, Block Data Layout, and Memory Hierarchy Performance*, IEEE Transactions of Parallel and Distributed Systems, 14(7), pp. 640–654, July 2003.

[31] S. Williams, J. Shalf, L. Oliker, P. Husbands, and K. Yelick, *Dense and Sparse Matrix Operations on the Cell Processor*, http://repositories.cdlib.org/lbnl/LBNL-58253, May 2005.

[32] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, *The Potential of Cell Processor for Scientific Computing*, Proceedings of the 3rd Conference on Computing Frontiers, pp. 9–20, 2006.

[33] A.S. Zekri and S.G. Sedukhin, *Level-3 BLAS and LU Factorization on a Matrix Processor*, Information Processing Society of Japan Digital Courier, 4, pp. 151-166, 2008.