Scientific Programming 17 (2009) 43–57
DOI 10.3233/SPR-2009-0278
IOS Press

Programming the Linpack benchmark for the IBM PowerXCell 8i processor

Michael Kistler, John Gunnels, Daniel Brokenshire and Brad Benton
IBM Corporation
E-mails: {mkistler, gunnels, brokensh, brad.benton}@us.ibm.com

Abstract. In this paper we present the design and implementation of the Linpack benchmark for the IBM BladeCenter QS22, which incorporates two IBM PowerXCell 8i[1] processors. The PowerXCell 8i is a new implementation of the Cell Broadband Engine™[2] architecture and contains a set of special-purpose processing cores known as Synergistic Processing Elements (SPEs). The SPEs can be used as computational accelerators to augment the main PowerPC processor. The added computational capability of the SPEs results in a peak double precision floating point capability of 108.8 GFLOPS. We explain how we modified the standard open source implementation of Linpack to accelerate key computational kernels using the SPEs of the PowerXCell 8i processors. We describe in detail the implementation and performance of the computational kernels and also explain how we employed the SPEs for high-speed data movement and reformatting. The result of these modifications is a Linpack benchmark optimized for the IBM PowerXCell 8i processor that achieves 170.7 GFLOPS on a BladeCenter QS22 with 32 GB of DDR2 SDRAM memory. Our implementation of Linpack also supports clusters of QS22s, and was used to achieve a result of 11.1 TFLOPS on a cluster of 84 QS22 blades. We compare our results on a single BladeCenter QS22 with the base Linpack implementation without SPE acceleration to illustrate the benefits of our optimizations.

Keywords: Accelerators, hybrid programming

1. Introduction

A recent trend in processor architecture is the design and implementation of multi-core processors. Often, this takes the form of a replication of the single-core architecture to yield multiple cores on a die, all with the same functionality as the single-core predecessor. IBM, Intel and AMD, all have multi-core products such as these in the market. A concomitant trend in high performance computing is the use of specialized components to provide acceleration of computational tasks. This can take the form of exploitation of GPUs or the addition of a dedicated floating point accelerator. These two trends are now starting to merge with the advent of multi-core technology targeted for specialized tasks, and not just the replication of a general purpose processor. The IBM PowerXCell 8i processor is an example of this move toward heterogeneous multi-core processors. The PowerXCell 8i is a new implementation of the Cell Broadband Engine architecture (CBEA). In this processor, a standard 64-bit PowerPC core is augmented by 8 specialized computational units known as Synergistic Processing Elements (SPEs) that can function as both computational accelerators and high-speed data manipulation units. The combined peak double-precision computational capability of the PowerPC core and 8 SPEs is 108.8 GFLOPS (one GFLOPS is 10^9 floating point operations per second). This technology has been packaged in the IBM BladeCenter QS22. A QS22 blade contains two IBM PowerXCell 8i processors and supports up to 32 GB of DDR2 memory. To demonstrate the capability of this architecture, we developed a version of the Linpack benchmark to take advantage of the computational acceleration provided by the SPEs of the PowerXCell 8i processor.

In this paper we describe the design and implementation of the Linpack benchmark we developed for the IBM PowerXCell 8i processor. Our implementation of Linpack is based on the standard open-source implementation, High Performance Linpack (HPL) [23]. We modified this code to accelerate key computational kernels using the SPEs of the PowerXCell 8i

[1] PowerXCell 8i is a trademark of the International Business Machines Corporation, in the United States, other countries, or both.

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

1058-9244/09/$17.00 © 2009 – IOS Press and the authors. All rights reserved

processors. The rest of the application code runs on the PowerPC core and uses the `libspe2` services of the IBM SDK for Multicore Acceleration to initiate the computational kernels on the SPEs. We have verified correct operation of our Linpack implementation and have measured its performance on an IBM QS22 blade system with 32 GB of DDR2 SDRAM memory. Using a matrix size $N = 48{,}895$, we have achieved 170.7 GFLOPS, which is 78% of the peak double precision computational performance of the system.

Our implementation of Linpack also supports clusters of QS22s using the Message Passing Interface (MPI) for internode communication. Support for cluster configurations is a feature largely inherited from the base HPL implementation. A team in IBM Germany exploited this capability to produce a Linpack result of 11.1 TFLOPS on a cluster of 84 QS22 blades with 8 GB of memory per blade interconnected with 4X single-data-rate Infiniband. This system was ranked at number 324 in the June 2008 Top500 list of the 500 most powerful computer systems [25]. In addition, the performance of Linpack on this system, in combination with the very high energy efficiency of the QS22, placed the system in the first position on the Green 500 list of most energy-efficient supercomputers with an energy-efficiency rating of 488 MFLOPS/W [24].

The remainder of this paper is organized as follows. Section 2 describes the PowerXCell 8i processor and the QS22 blade system we used to develop our Linpack implementation. Section 3 gives an introduction to the Linpack benchmark and describes the design modifications we made to adapt the benchmark to the PowerXCell 8i processor. Section 4 describes the design and implementation of our acceleration library. We present performance results in Section 5, and review related work in Section 6. Section 7 concludes the paper.

2. The IBM PowerXCell 8i processor

The IBM PowerXCell 8i is a new implementation of the Cell Broadband Engine Architecture (CBEA) [16], which is a fully compatible extension to the IBM 64-bit PowerPC Architecture. The CBEA proscribes a processor design that contains one (or more) PowerPC Processor Elements (PPEs) and a set of Synergistic Processor Elements (SPEs), which implement a completely new instruction set architecture designed specifically for high-performance numerical computations. The CBEA also specifies mechanisms and interfaces for explicit control of the memory hierarchy and data movement within the processor.

The IBM PowerXCell 8i consists of one PPE and eight SPEs. The PPE is intended to be used as the "control" core, where the operating system and general control functions for an application are executed. The PPE is a dual-issue, in-order 64-bit PowerPC processor core with two-way simultaneous multithreading, 32 kB level 1 instruction and data caches, and a 512 kB level 2 cache. Each SPE has a Synergistic Processor Unit (SPU), 256 kB local store (LS), and corresponding Memory Flow Controller (MFC). The SPU is a streaming processor with a 128-bit single-instruction, multiple-data (SIMD) instruction set architecture that operates only on data within the SPE local store. The MFC is used to control transfers between local store and system memory or to another SPE's local store. The SPU issues direct memory access (DMA) commands to the MFC to get data from main memory into local store or put data from local store into main memory. DMA commands are performed concurrently with SPU program execution, allowing very efficient overlap of computation with communication. The PPE and SPEs are connected to each other and to an on-chip memory controller and I/O controller through the Element Interconnect Bus (EIB), which delivers a peak bandwidth of 204.8 GB/s. The on-chip memory controller can support up to 25.6 GB/s of bandwidth to off-chip memory.

Each SPU has 128 128-bit SIMD registers. The large number of architected registers facilitates highly efficient instruction scheduling and enables important optimization techniques such as loop unrolling. All SPU instructions are inherently SIMD operations that operate on all 128 bits of the operand registers. The SPU is an in-order processor with two instruction pipelines, referred to as pipeline 0 and pipeline 1. Each SPU can issue and complete up to two instructions per cycle – one per pipeline. The fixed- and floating-point units are on pipeline 0, and the remaining functional units, including the load/store unit, are on pipeline 1. In the PowerXCell 8i processor, all fixed- and floating-point operations are fully pipelined and can be issued at the full SPU clock rate.

The first implementation of the CBEA is the Cell/B.E. processor, which is the processor used in Sony's Playstation3 game console. The IBM PowerXCell 8i differs from the Cell/B.E. in that it includes an enhanced double precision unit on the SPEs, giving each SPE a peak computational capability of 12.8 GFLOPS in double precision. In addition, the IBM PowerXCell 8i supports industry-standard DDR2 SDRAM memory, enabling cost-effective system designs with large memory capacities. Figure 1 shows a die photo of the IBM PowerXCell 8i processor that identifies the new features of this processor.

The IBM QS22 blade system contains two IBM PowerXCell 8i processors and can accommodate up to 32 GB of DDR2 SDRAM memory. Figure 2 is a diagram of the key components of the QS22. Note that system memory is attached directly to the processors through their on-chip memory controller. The input-output interface (IOIF) connection between the processors allows either processor to coherently access memory physically attached to the other processor. This makes the QS22 a non-uniform memory architecture (NUMA) system, because the access time to memory varies depending on whether the target of a memory access is attached to the processor performing the access or the other processor of the system. The QS22 also has two gigabit Ethernet interfaces and can optionally be equipped with up to two 4X Single Data Rate Infiniband interfaces, each of which can provide up to 1 GB/s of network bandwidth. The QS22 supports a software stack based on the Red Hat Enterprise Linux distribution. The IBM Software Kit for Multicore Acceleration (IBM SDK) [15] installs on top of RHEL Linux and provides additional tools, libraries, and documentation for utilizing the special capabilities of the PowerXCell 8i processor.

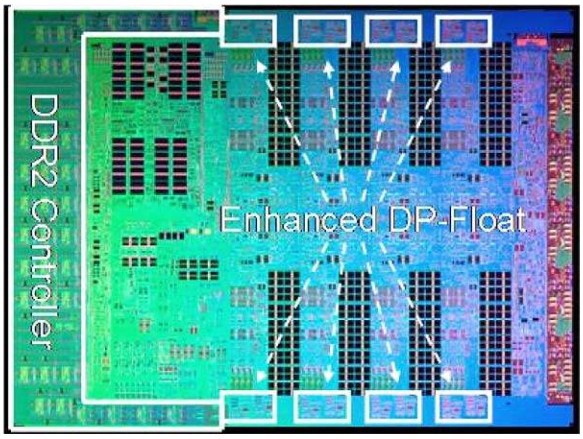


Fig. 1. Die photo of IBM PowerXCell 8i processor.

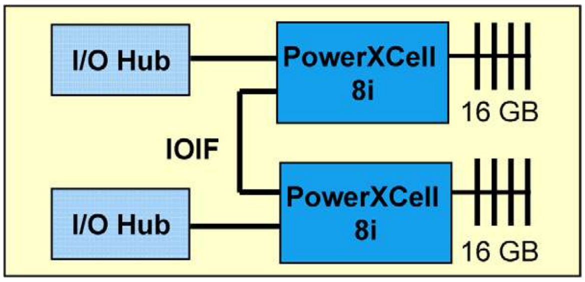


Fig. 2. A QS22 blade system.

3. Design of Linpack for PowerXCell 8i

The Linpack benchmark has become an industry standard benchmark for measuring the performance of computer systems. The benchmark solves a system of linear equations by performing LU factorization with partial pivoting on a dense matrix, and then solving the resulting triangular system of equations. All calculations are performed in double-precision. The bulk of the benchmark computation is spent in the factorization of the input matrix A into a lower-triangular matrix L and an upper-triangular matrix U. For a matrix of dimension N, the LU factorization requires $(2/3) \cdot N^3$ floating point operations while the triangular solve requires only N^2 floating point operations. The size of the problem is a key factor in the achievable performance.

The LU decomposition is typically performed using a blocked, right-looking algorithm, in which each iteration produces a portion of the final L and U matrices and leaves a reduced region of the matrix (the trailing sub-matrix) to be solved by the next iteration. This approach allows much of the computation to be performed using matrix–matrix (BLAS3) operations, which are much more efficient than vector–vector (BLAS1) or matrix–vector (BLAS2) operations on modern computer systems with deep memory hierarchies [5,19]. The high-level flow of the benchmark is as follows:

```
Allocate and initialize matrix
Iterate over block columns:
  Panel factorization – factor current block column
  Forward Pivot trailing sub-matrix
  Compute block row of final $U$ matrix (DTRSM)
  Update trailing sub-matrix (DGEMM)
Compute solution of the given system
Check the result
```

First storage for the matrix is allocated and initialized with random values. Then the benchmark enters the main loop of the LU factorization. The first step of this loop is panel factorization, which performs LU factorization with pivoting on the left-most column of blocks in the trailing sub-matrix. This produces one block column of the final L matrix, called the L-panel. Pivoting rearranges rows of the matrix to avoid division by small values during the factorization, which could lead to numerical instability. Pivoting is performed within the panel during panel factorization. The sequence of pivot operations is saved and then applied to the trailing sub-matrix in the forward pivoting step. The implementation of LU factorization in Linpack continuously updates the right-hand-side vector with the newly computed portion of the L-matrix. Consequently, the previously computed regions of L are not pivoted since they are no longer needed to compute the final solution. The benchmark rules allow this behavior and since pivoting L would only decrease the achieved performance, it is typically omitted. Following the forward pivot, a triangular solve with multiple right-hand-sides (DTRSM) is performed on the top block row of the trailing sub-matrix, producing one block row of the final U matrix, called the U-panel. In the final step of the main loop, the product of the L-panel and U-panel is subtracted from the remainder of the trailing sub-matrix. Following the LU factorization loop, the final solution is computed using a triangular solve, and then the benchmark checks this solution for correctness. Computation time is dominated by the matrix update step which is a form of matrix–matrix multiply (DGEMM), which is an O(N^3) operation. The panel factorization, DTRSM, and triangular solve operations are all O(N^2) operations. When the benchmark is parallelized to run on a cluster of nodes, the panel factorization and forward pivot steps of the benchmark involve inter-node communication.

For our implementation, we began with the publicly-available implementation, High Performance Linpack (HPL) [23], and then replaced the compute-intensive kernels with accelerated versions that utilize the SPEs of the PowerXCell 8i processor. We chose this approach primarily to demonstrate that applications initially developed for homogeneous architectures could be successfully adapted to the heterogeneous Cell/B.E. architecture. This approach also allowed us to leverage optimizations and tuning parameters in HPL that are useful in boosting performance. In particular, the technique of pipelining iterations of the main computation, or look ahead [18], is a key optimization we wanted to leverage.

In addition to accelerating certain compute kernels, we employ the well-known optimization of reformatting the input matrix into a hierarchically blocked organization [11]. However, we implement this in a novel manner by using the SPEs to perform the reformatting, which significantly reduces the cost of this operation. We also modified the memory allocation routines to obtain the matrix storage from huge pages, which reduces memory translation overheads. The data organization we chose for the matrix is a “blocked-row” format, where blocks of 64 columns are stored in row-major format. Reformatting the matrix allowed us to focus on developing highly specialized versions of our DGEMM and DTRSM kernels and improves the performance of the memory hierarchy in the pivoting operations. An advantage of the blocked-row format over a strictly row-ordered layout is that a column-ordered matrix (which is the format of the input matrix to the benchmark) can be transformed into blocked-row with just one extra block column of storage. Our implementation performs this reformatting at the beginning of the LU factorization, and then converts block-columns back to column major just before the panel factorization for this block column. In this way, the entire matrix is converted back to column-major format by the end of the factorization, which allows us to use the existing triangular solve code from the benchmark to compute the final solution.

In prior work [17], we developed an implementation of the Linpack benchmark for the Roadrunner system at Los Alamos National Laboratory, which is a hybrid system that contains both x86-64 and PowerXCell 8i processors. We expected that the basic design of Linpack for Roadrunner could be used in our implementation for the QS22 by viewing the PPE as the *host* processor and the SPEs as the *accelerators*. However, a number of modifications were needed to this design to achieve good performance on the QS22, mainly because the ratio of compute capability between the host and accelerator is different between these two architectures. In particular, the performance of panel factorization, using the PPE only, declined to the point where it became the bottleneck in overall performance. One solution to this problem would be to implement an SPE-based panel factorization routine, but this would require a significant new development effort. Instead, we chose to accelerate only the DGEMM operations performed as part of the recursive panel factorization algorithm. This allowed us to significantly increase the performance of panel factorization with a very modest additional development effort.

4. The HPL acceleration library

4.1. Overview

To accelerate the HPL benchmark we chose to develop a proprietary, specialized function, offload library instead of using an off-the-shelf, PowerXCell 8i optimized Basic Linear Algebra Subprograms (BLAS) library. The library was designed with the following objectives:

Efficient: Offload as much heavy computation onto the SPEs as possible while keeping memory bandwidth requirements low so that SPE accelerated computation, PPE host computation, and node to node communications can occur simultaneously without contention for memory bandwidth.

Simple: Instead of utilizing one of the many programming frameworks available to Cell programmers, we chose to implement directly to the low-level processor APIs and not incur the overhead, in both processing cycles and local storage, introduced by a generalized programming framework.

Flexible and extendible: The design needed to be flexible enough to easily add library functions, accommodate the use of overlays when local storage becomes constrained, or direct specific functions to a subset of the computing resources.

Asynchronous and pipelined: Provide facilities for issuing multiple library calls and getting asynchronous notification when each of them complete.

Robust: Provide acceleration of functions for arbitrary parameter sizes and alignments.

Similar: Provide a calling interface that is similar in style and semantics to the standard BLAS library so that code can be readily ported between the two libraries.

Verifiable: Ensure that the implementation performs correctly.

The acceleration library is designed to utilize the 8 SPEs of a single PowerXCell 8i processor. The primary rationale for this design is to allow management of the NUMA impacts of the QS22 system design at a higher layer in the application. In HPL, we execute two MPI tasks on the QS22, one per PowerXCell 8i processor. Each task creates its own instance of the HPL acceleration library, and uses NUMA memory binding to ensure that the majority of memory accesses are to the local region of system memory.

The HPL acceleration library architecture is shown in Fig. 3. The application (in this case HPL) calls one of the library functions. All functions include a "completion" parameter which is a pointer to a 64-bit variable that is used by the library to indicate asynchronous completion of the called function. NULL can be specified for functions that do not require completion noti-

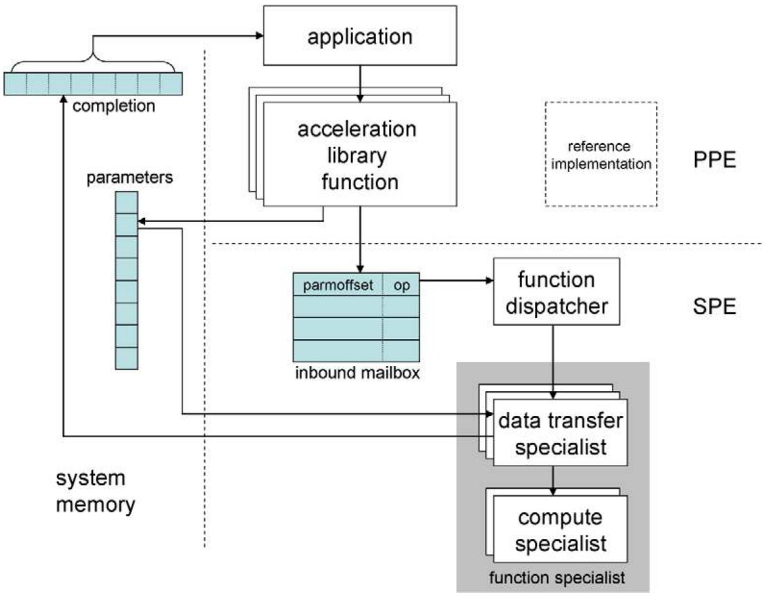


Fig. 3. PowerXCell 8i HPL acceleration library architecture.

fication. The acceleration library functions validate the parameters to ensure compliance with the constraints of the SPE acceleration software and then determine what portions of the computations can be efficiently performed on the SPEs. Computations involving properly aligned data regions that are large enough to be efficiently handled by the SPE's Memory Flow Controller (MFC) are offloaded to the SPEs. Remaining computations are performed by "cleanup code" on the PPE. For the HPL benchmark, special care is taken in memory allocation to ensure proper data alignments so that the majority of the request is offloaded to the SPEs.

To offload a function to the SPEs, the library function clears the completion variable, stores its parameters to a single cache-line parameter buffer in system memory, places a command request into the inbound mailbox of each SPE being recruited to complete the function, and returns control back to its caller. To ensure proper storage ordering, a sync instruction is executed after storing the parameters to system memory and before writing to the SPE's problem state inbound mailbox register. The mailbox command consists of a command opcode, stored in the 7 least significant bits, and a cache-line aligned offset to the system memory parameter buffer. The inbound mailbox of the SPE in the PowerXCell 8i is 4 entries deep, which provides more than adequate function buffering for the HPL benchmark.

All command requests are received by the function dispatcher running on the SPE, which decodes the command opcode to select a *function specialist* to process the command. Unless enlisted to perform acceleration requests, the SPEs block in a low power state waiting for an inbound mailbox message. Upon receipt of a command request, the SPEs immediately initiate a DMA of the parameters so the parameter transfer occurs while the opcode dispatch to the function specialist is performed. Partitioning of the kernel operation across SPEs is performed deterministically by the SPEs, which relieves the PPE from managing data partitioning and allows a single set of parameters to be passed to all SPEs involved in the kernel execution.

The function specialists are partitioned into two basic parts – data transfer and compute kernel. The data transfer part is responsible for the data flow of the function specialist. It determines the portion of the work to be performed on its SPE and utilizes multi-buffering techniques to overlap computation and data transfers. The compute kernel is responsible for performing the computation on a local store subset of the request.

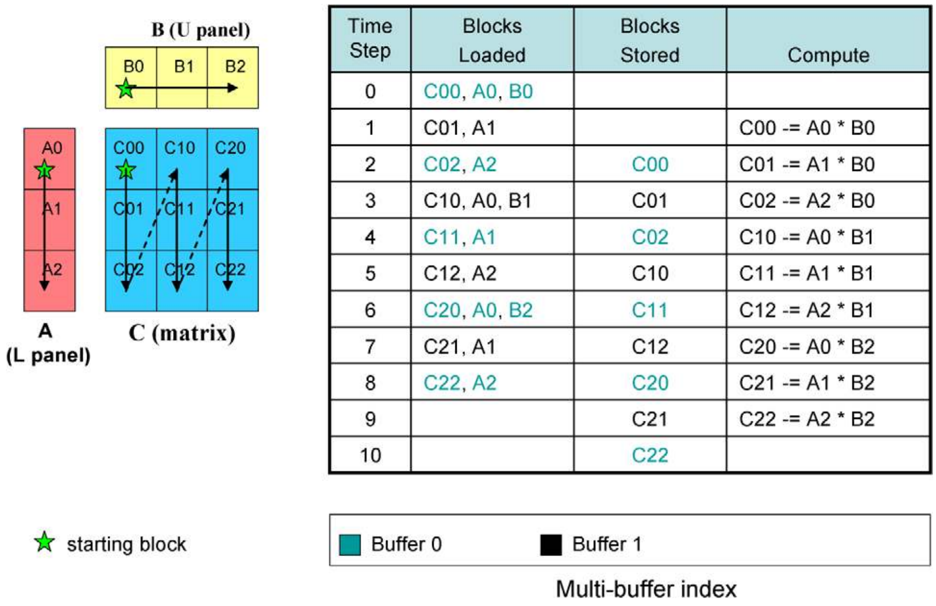
The acceleration library currently includes 12 different acceleration functions. However, not all of these functions are used by Linpack for QS22. The unused functions are present in the library because we maintain a common code base for the acceleration library between the Roadrunner and QS22 versions of Linpack. The full list of functions appears below. Functions that are not used by Linpack for QS22 are indicated by (*). Functions developed specifically for Linpack for QS22 are indicated by (**):

- DGEMM for a blocked-row C matrix
- DGEMM for a blocked-row C matrix, producing a column-ordered result (*)
- DGEMM for a column-ordered C matrix (**)
- DTRSM for column-ordered A and row-ordered B matrices
- DTRSM for column-ordered A and row-ordered B matrices, producing a blocked-row result
- DTRSM for column-ordered A and block-ordered B matrices (**)
- Reformat matrix from column-ordered to blocked-row
- Reformat a panel from blocked-row to column-ordered
- Reformat a panel from row-ordered to blocked-row
- Scatter rows from a row-ordered panel into a blocked-row matrix
- Gather rows from a blocked-row matrix into a row-ordered panel
- Swap rows within a blocked-row matrix (**)
- Copy rows with indirections from a row-ordered panel to another row-ordered panel (**)

The SPU program containing all these functions along with the dispatching mechanism and data buffers has a total size of 249,580 bytes, with 50,888 bytes used for instruction storage and 198,692 bytes used for data. The acceleration library does not perform heap allocations, so this leaves 12,564 bytes for the program stack, which is more than sufficient given the very shallow call depth of the library functions.

4.2. The DGEMM specialist

Since the HPL benchmark performance is dominated by the performance of the DGEMM operation ($C = C - AB$) applied during the update of the trailing sub-matrix during LU factorization, many of the system design decisions were dictated by the DGEMM design. A block size of 64×64 (32 kB/block) was cho-



Time Step	Blocks Loaded	Blocks Stored	Compute
0	C00, A0, B0		
1	C01, A1		C00 -= A0 * B0
2	C02, A2	C00	C01 -= A1 * B0
3	C10, A0, B1	C01	C02 -= A2 * B0
4	C11, A1	C02	C10 -= A0 * B1
5	C12, A2	C10	C11 -= A1 * B1
6	C20, A0, B2	C11	C12 -= A2 * B1
7	C21, A1	C12	C20 -= A0 * B2
8	C22, A2	C20	C21 -= A1 * B2
9		C21	C22 -= A2 * B2
10		C22	

Fig. 4. Traditional blocked DGEMM visitation order.

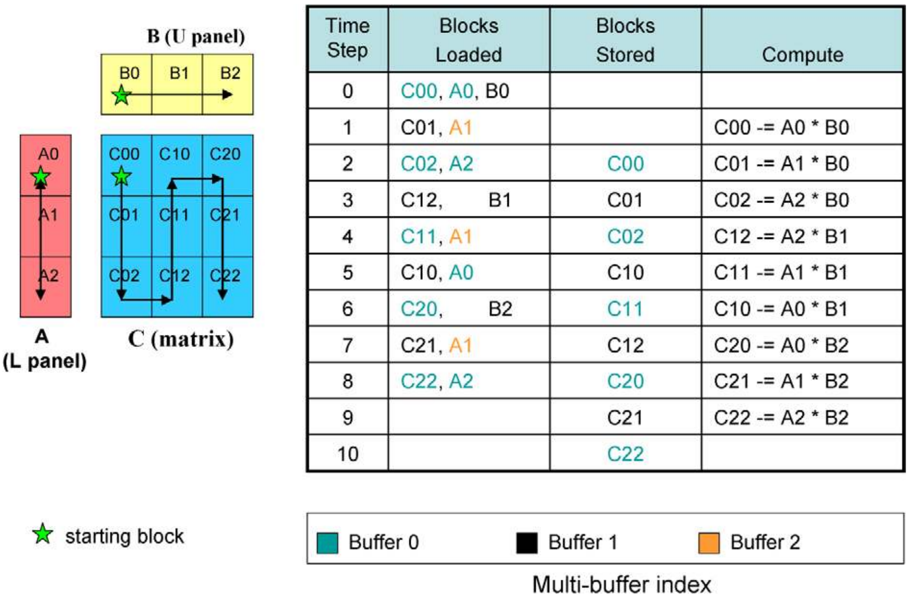
sen as the largest power of two that allows the SPE local storage to accommodate double buffered blocks of the three matrices. These 6 buffer blocks consume 192 kB of the available 256 kB of local storage. Most DGEMM implementations use a traditional block visitation order as shown in Fig. 4. In time steps 3 and 6, the number of 64 × 64 element blocks transferred is 4. These correspond to the "corner turns" of the matrix multiply. Assuming a highly efficient block multiply kernel, the aggregated data throughput demand of 8 SPEs is 17.9 GB/s with peaks up to 23.8 GB/s during corner turns. Taking into account the other activities placing demands on the memory subsystem and the loss of efficiency that results from read/write transitions, rank switches, and memory bank contention, we realized that an alternate solution was required.

Our first idea was to utilize "bounce" corner turns and compute result C matrix blocks in a serpentine order, see Fig. 5. Now the sustained number of blocks transferred during any one time step is 3, which reduces the memory demands of the DGEMM down to a sustained bandwidth of 17.9 GB/s. Furthermore, instead of fully double buffering the A and B matrix blocks, we can use a rotating set of three for A and B. This reduces our local storage requirements for data buffers, which thereby increases the space available for instruction storage and stack space, allowing more accelerated functions to be provided within a single library program file.

To further reduce our dependency on high memory throughput, we extended the bounce corner turn idea by using a larger LU factorization block size, 128 × 128, and sub-blocking them into 2 by 2, 64 × 64 element sub-blocks. The visitation order of the sub-blocks again is serpentine for the C matrix. The B matrix (U panel) is also traversed in a serpentine manner, whereas, the A matrix (L panel) is visited in a zigzag pattern, as shown in Fig. 6. This technique reduced the block data transfer needs from 3 blocks per time step to 2 blocks per time step, resulting in a reduced steady-state bandwidth demand of 11.9 GB/s.

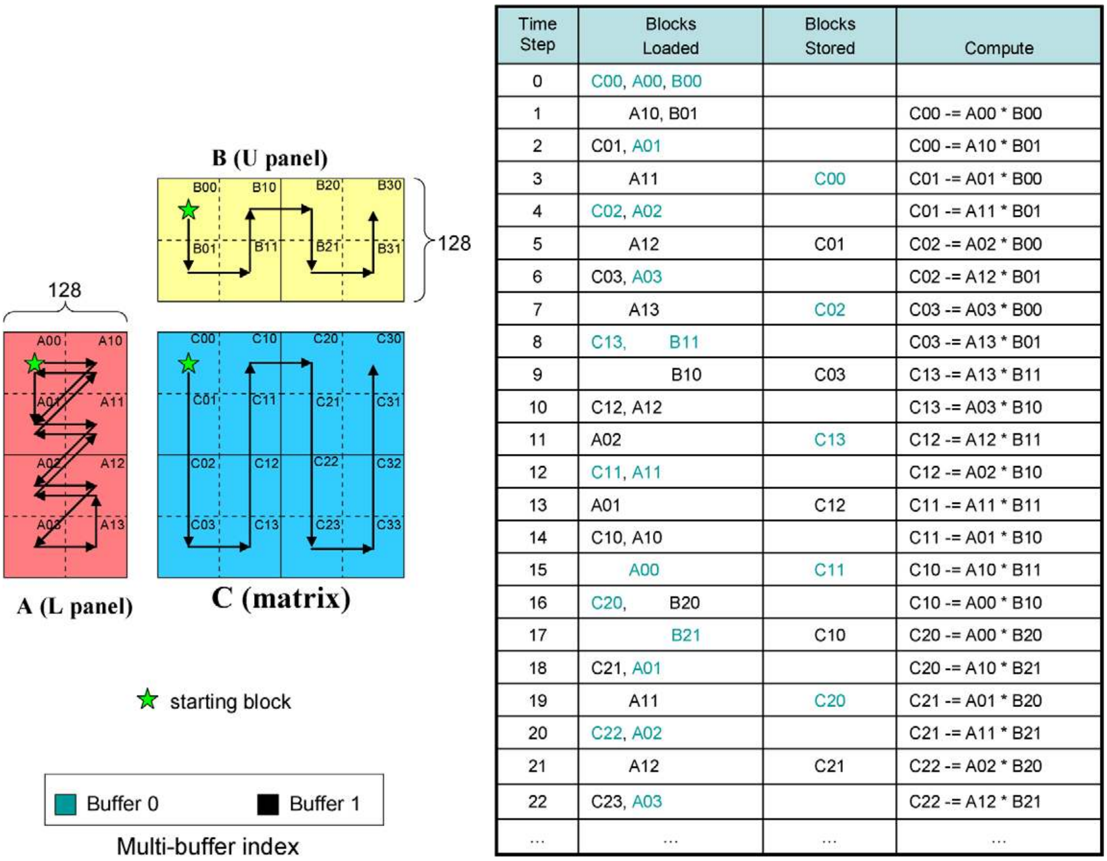
4.3. The DTRSM specialists

The acceleration library DTRSM functions perform a triangular solve for X of a matrix equation of the form $LX = B$. L is a column ordered 128 × 128 unit lower triangular matrix computed during panel factorization. B is a row ordered matrix of size 128 by N resulting from the forward pivot operation (a U panel). This configuration of operands corresponds to the standard BLAS DTRSM function with the left, lower, no-transpose, and unit options and an alpha of 1.0. The acceleration library DTRSM function is specialized for this operand configuration to simplify the implementation and enable configuration-specific optimizations.



Time Step	Blocks Loaded	Blocks Stored	Compute
0	C00, A0, B0		
1	C01, A1		C00 -= A0 * B0
2	C02, A2	C00	C01 -= A1 * B0
3	C12, B1	C01	C02 -= A2 * B0
4	C11, A1	C02	C12 -= A2 * B1
5	C10, A0	C10	C11 -= A1 * B1
6	C20, B2	C11	C10 -= A0 * B1
7	C21, A1	C12	C20 -= A0 * B2
8	C22, A2	C20	C21 -= A1 * B2
9		C21	C22 -= A2 * B2
10		C22	

Fig. 5. Bounce corner turn blocked DGEMM visitation order.



Time Step	Blocks Loaded	Blocks Stored	Compute
0	C00, A00, B00		
1	A10, B01		C00 -= A00 * B00
2	C01, A01		C00 -= A10 * B01
3	A11	C00	C01 -= A01 * B00
4	C02, A02		C01 -= A11 * B01
5	A12	C01	C02 -= A02 * B00
6	C03, A03		C02 -= A12 * B01
7	A13	C02	C03 -= A03 * B00
8	C13, B11		C03 -= A13 * B01
9	B10	C03	C13 -= A13 * B11
10	C12, A12		C13 -= A03 * B10
11	A02	C13	C12 -= A12 * B11
12	C11, A11		C12 -= A02 * B10
13	A01	C12	C11 -= A11 * B11
14	C10, A10		C11 -= A01 * B10
15	A00	C11	C10 -= A10 * B11
16	C20, B20		C10 -= A00 * B10
17	B21	C10	C20 -= A00 * B20
18	C21, A01		C20 -= A10 * B21
19	A11	C20	C21 -= A01 * B20
20	C22, A02		C21 -= A11 * B21
21	A12	C21	C22 -= A02 * B20
22	C23, A03		C22 -= A12 * B21
...	...	...	...

Fig. 6. Sub-blocked DGEMM visitation order with reduced memory demands.

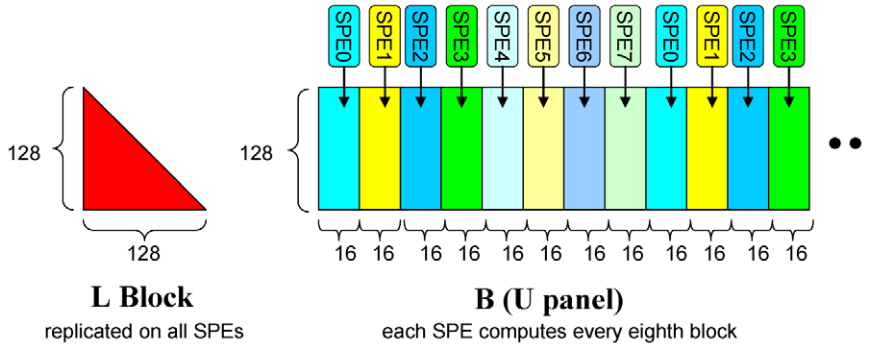


Fig. 7. Data partitioning scheme used in DTRSM kernel.

The computation of X is partitioned across the 8 SPEs in blocks of 128 × 16 elements and interleaved so that each SPE computes every eighth block. This approach was chosen to improve memory bank access uniformity for both block and row ordered B matrix outputs. This data partitioning scheme is illustrated in Fig. 7. We further improved the uniformity of our memory bank accesses by ensuring that the leading dimension of row ordered panels is never an integral multiple of 256 so that consecutive block rows begin in different memory banks. The 128 × 16 element blocks are triangularly solved 4 rows at a time using an integrated lazy update technique to reduce local store accesses of the upper panel matrix B. Aggressive loop transformations and software pipelining were used to ensure that the kernel is never load/store bound at any point in its execution [4].

4.4. Implementation details

4.4.1. Avoiding instruction runout

Contention on the SPU local store is an important consideration in the implementation of high-performance computational kernels for the SPU. The local store is a single-ported structure, meaning that it can service at most one data request each cycle. The local store services requests according to a priority scheme based on the source of the request, which could be (in priority order) an MMIO access, DMA, DMA list element fetch, ECC scrub, SPU load/store, hint fetch, or an inline instruction fetch [15]. The primary concern is instruction runout, a condition where inline instruction prefetch, the lowest priority request source, could be starved in periods of significantly high load/store sequences, which are common in computational kernels.

There are two methods of relieving pressure on the LS so that instruction runout does not occur in long sequences of load and stores – inserting an `lnop` (no-op on pipeline 1, which handles load/store operations) or inserting a special form of branch hint instruction called an `hbrp`. Inserting a `lnop` instruction will open an LS access slot from the perspective of the SPE instruction issue logic, but this slot may be consumed by a higher priority access, such as a DMA. Inserting a `hbrp` instruction will open an LS access slot for inline instruction prefetch, but could also create a multi-cycle stall if a higher priority access occurs at the same time. Our experience indicates that an `hbrp` could generate a stall of as much as 3 cycles, but can offer a guarantee of avoiding instruction runout, which stalls the SPU pipelines for approximately 18 cycles. The choice of `lnop` vs. `hbrp` is thus a tradeoff between the low fixed cost `lnops` with some occasional 18 cycle stalls for instruction fill vs. the variable cost `hbrp` instructions that can completely avoid instruction runout.

Both the gcc and xlc compilers use heuristics to insert `hbrps` into long sequences of load/store instructions to allow instruction prefetch accesses to the local store. However, we coded our computational kernels in assembly, for maximum performance, and thus we needed to develop our own approach to avoiding instruction runout. We have chosen to use `hbrps` in our kernels and found this strategy to be very effective. Here is the strategy we adopted:

(1) Align the function to a known line buffer boundary (128 bytes). This can be done with ".align 7". You can also align to half-line boundaries, but placement of `hbrp`'s should be based upon the line boundaries.

(2) For each branch target, align the target instruction sequence to either a line or half line boundary. If execution can fall through to the target too, then it might be best to align to a full line boundary so that the known line boundaries are consistent, regardless of the execution flow.

(3) For each line buffer of sequential instruction sequence, if the trailing end (5 dual issue cycles) of the previous buffer and the start (3 dual issue cycles) of the current buffer has no open slots for instruction prefetch, insert a `hbrp` in this area.

(4) If the sequence of instructions at a branch target starts with three dual-issue instruction pairs that each contain a load/store instruction, insert a `hbrp` at the start of this sequence.

4.4.2. 4 GB boundary crossings

The MFC DMA list commands are used to extract two-dimensional blocks of data from matrix buffers. Since all list elements of a DMA list command share a common high 32 bits of their effective address, a single command can only transfer data from within a single, aligned, 4 GB region. In order to support large problems and leverage the memory capacity of the 32 GB QS22 blade configurations, our software was forced to accommodate 4 GB boundary crossings.

To reduce the overhead of supporting 4 GB boundary crossings, we leveraged our ability to control the alignment of the system memory buffer allocations by ensuring that all L and U panel buffers never cross a 4 GB boundary and that the block formatted matrix only crosses 4 GB boundaries on a 64×64 element block boundary. This constraint fully eliminates ever having to break a DMA list within a list element and further reduces the number and location of a crossing. For example, when transferring a block formatted 128×16 DTRSM block, at most one crossing can occur and when it does, it will cross at the 64th list element. Instead of introducing additional branches in the code, we chose to always issue two DMA list commands, either of size 128 and 0 elements or 64 and 64 elements, depending on if a crossing occurred.

4.4.3. Checking for tag group completion

All accelerator library functions utilize a little known technique of reducing the cost associated with waiting on tag group completion when all DMAs within the tag group are complete (i.e., when not memory bound). To wait for tag group completion, most programmers use either the `mfc_read_tag_status_all` macro or the `spu_mfcstat` intrinsic. Both of these result in a write to the MFC Tag Update channel followed by a read of the MFC Tag Status channel. When issued consecutively, these two instructions have a minimum combined latency of 50 cycles because the write to the Tag Update channel restarts the state machine that updates the tag status, causing a stall on the read of the Tag Status channel. However, if the read of the Tag Status channel occurs at least 35 cycles after the write to the Tag Update channel, the stall is avoided, and the combined latency is reduced to just two cycles. In SPE programs that use double buffering, the write to the Tag Update channel can be issued immediately after the last DMA of the group, allowing the tag status state machine to complete its processing before the read of the Tag Status channel. The data transfer functions in the acceleration library employ this technique, thereby saving roughly 48 cycles per DMA completion wait.

4.4.4. Completion notification

When the SPE completes a request, it writes a single byte to the system memory completion variable using a fenced PUT command with the same tag identifier as the final buffer of computed results. Each of the 8 SPEs is assigned a unique byte within the completion double word so that an aggregated completion of all SPEs can be checked with a single 8-byte aligned memory access, which is performed atomically in the PowerPC architecture. The fence ensures that storage ordering is preserved and avoids the SPE waiting for the final buffer transfer to complete before writing back its completion notification. This allows the SPE to immediately fetch the next request from the inbound mailbox.

5. Results

5.1. Environment

We performed experiments using an IBM QS22 blade system with 32 GB of 800 MHz DDR2 SDRAM. The system is running Red Hat Enterprise Linux version 5.2 and version 3.0 of the IBM Software Kit for Multicore Acceleration (IBM SDK) [15]. The system was configured with 1280 huge (16 MB) pages, and all of the experiments were performed using memory allocated from huge pages.

As described above, we have implemented our own acceleration library to offload computations and data movement functions to the SPEs. However, panel factorization and the backsolve step are still performed on the PPE and thus require an optimized BLAS library for the PPE. We chose to use the Automatically Tuned Linear Algebra Software (ATLAS) [26] for this purpose. ATLAS is an open-source implementation of the BLAS that employs automated optimization techniques to generate a high performance BLAS library for the target platform.

The HPL implementation of Linpack uses the Message Passing Interface (MPI) [20,21] to enable execution on homogeneous clusters. We employ this capability to distribute execution to the two PowerXCell 8i processors of the QS22. This also allows our implementation to run on clusters of QS22 blade systems. The MPI implementation we use is Open MPI version 1.2.6, an open source MPI implementation from the Open MPI Project [9].

5.2. *Performance of the acceleration library*

Our acceleration library is designed to allow very lightweight initiation of the computational kernel functions. To quantify the costs of kernel initiation, we timed the execution of a DGEMM invocation that performs no actual computations on the SPUs. Thus, all the time for this invocation can be attributed to the overhead of initiation/completion notification mechanisms, which includes:

PPE – Application calls accelerator library.
PPE – Check the parameters.
PPE – Build a parameter buffer.
PPE – Sync the memory subsystem.
PPE – Store a command to 8 SPEs via the inbound mailbox.
SPE – Receive the command via the mailbox.
SPE – Fetch the parameters corresponding command request.
SPE – Dispatch the requested function specialist.
SPE – Write the notification to the system memory that the request is complete.
PPE – Detect that all the SPEs have completed the request.

Using the time-base register, a high precision timing mechanism in the PowerPC architecture, we measured this overhead to be approximately 11K cycles or 3.4 µs. This is significantly faster than the standard SPE program dispatch mechanism which requires creation of an SPE program context and loading of the SPU program into the SPE.

Next we measured the performance of the blocked implementations of DGEMM and DTRSM. We measure the time from initiation of the kernel on the PPE to the point where the PPE receives the completion notification. Thus, the performance includes the initiation costs described above and all DMA operations to transfer data/results between the SPEs and main storage, as well as the time for the actual computations. Figure 8 shows the performance of the DGEMM acceleration kernel. The graph shows performance of the DGEMM kernel performing the $C = C - AB$, where C is an $M \times M$ matrix in block row format, A is column ordered with size $M \times 128$, and B is row ordered of size $128 \times M$. The x-axis of this graph is the size of the matrix being updated (M) and the y-axis indicates the performance of the DGEMM kernel in GFLOPS. Note that the acceleration library will only use the PPE and 8 SPEs of one PowerXCell 8i processor, so the maximum performance achievable is 108.8 GFLOPS. The graph shows that performance increases very rapidly with matrix size, achieving 80% of the peak computation rate for $N = 512$, and exceeding 90% of the peak at $N = 896$. Since a QS22 with 32 GB of memory can easily run problems with N as large as 50,000, we should achieve a very high computation rate for the majority of the LU factorization.

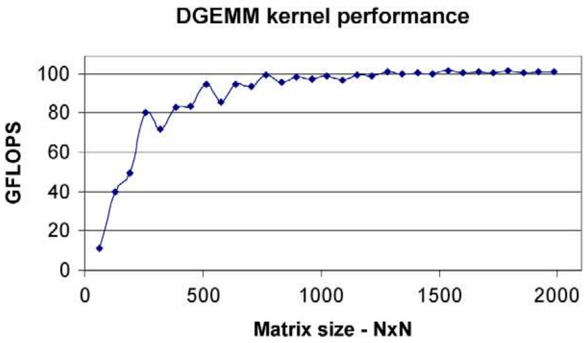


Fig. 8. Performance of DGEMM kernel.

Figure 9 shows a similar graph of measured performance of the DTRSM acceleration kernel. It is not uncommon for DTRSM to achieve lower performance than DGEMM, due to the slightly less regular pattern of computations and memory accesses. For small matrices, performance suffers because there is insufficient computation to amortize the startup cost of bringing in the lower triangle. Our DTRSM acceleration kernel achieves 60% of the peak computation rate at $N = 640$ and plateaus at roughly 77% of peak for $N > 1536$.

Our final set of kernel experiments examines the performance of the column-ordered DGEMM kernel used to boost the performance of panel factorization. This kernel is used by the recursive layer of panel factorization, and thus is always called to update a thin slice of the matrix, meaning that the row dimension is large but the number of columns is always 64 or less. Figure 10 presents the performance for the column-ordered DGEMM kernel for various matrix sizes. The x-axis is the row-dimension of the ma-

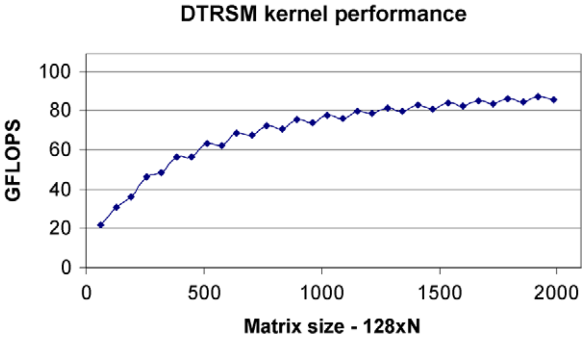


Fig. 9. Performance of DTRSM kernel.

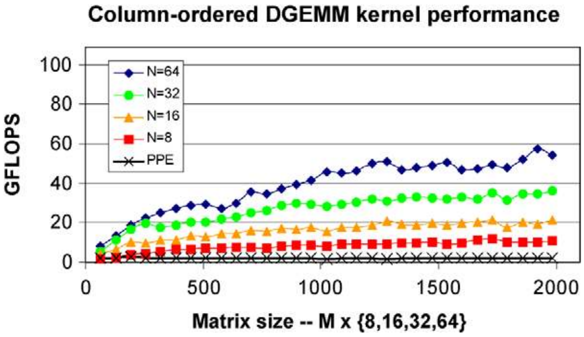


Fig. 10. Performance of column-ordered DGEMM kernel.

trix, and the top four curves in the graph indicate the performance of the column-ordered DGEMM with a column dimension of 64, 32, 16 and 8 columns. The final curve in the graph is the performance of the Atlas PPE-only DGEMM for the specified row dimension and a column dimension of 64. The y-axis is GFLOPS as in the previous graphs. Clearly this kernel achieves lower performance than the blocked kernels. This is because of several factors, including the matrix format, the matrix dimensions, and because the column-ordered DGEMM kernel is written in C rather than hand-optimized assembler. Nevertheless, the kernel still achieves substantially higher performance than a PPE-based DGEMM and thus provides a significant boost to the performance of panel factorization.

5.3. Overall benchmark performance

Figure 11 presents the performance of our Linpack implementation for the PowerXCell 8i using both processors of an IBM BladeCenter QS22. For comparison, this graph also includes results for the base Linpack implementation using the Atlas BLAS (PPE-only). The right-most data point on the graph represents an achieved performance of 170.7 GFLOPS using a matrix size $N = 48{,}895$, which is 78% of the peak double precision computational performance of the system. These results show that for large problem sizes, the SPEs of the IBM PowerXCell 8i can improve Linpack performance by more than a factor of 30.

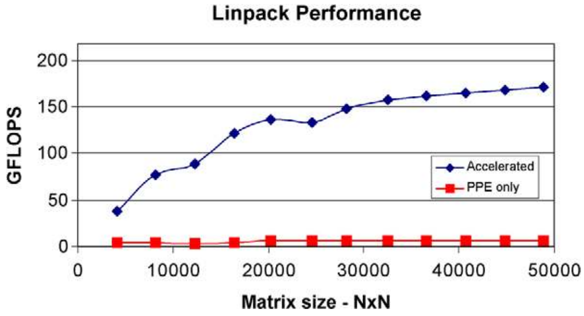


Fig. 11. Linpack performance of accelerated implementation and PPE-only implementation.

We used the detailed timing option of HPL to obtain a breakdown of the overall benchmark time into the major components of benchmark execution. The main components in this breakdown are *update*, which is the DTRSM and DGEMM update of the trailing sub-matrix, *pfact*, which is the panel factorization step, *la swap*, which covers the row swapping performed for the forward pivot operation, *tr solve*, which is the triangular solve performed during the backsolve step to obtain the final solution, and *accel ovhd*, which is a category we added to track the costs of matrix reformatting for the accelerators. Figure 12 shows the breakdown of execution time into these components, normalized to the runtime of the entire benchmark, for a range of matrix sizes. Note that in most cases the sum of the components slightly exceeds the total runtime – this is because the component time reported is the maximum time over all the MPI tasks used in the run (which is 2 in our case). There are some clear trends visible in this graph. The first trend is that the update time is the dominant component, consuming over 60% of the run time even at very small matrix sizes, and growing as a fraction of run time as the matrix size increases. All other components decrease as a fraction of run time as matrix size increases. These trends are expected, since the update step must perform O(N^3) operations, whereas the complexity of the other components is at most O(N^2). Nevertheless, the combined pfact and accel ovhd components account for over 13% of run time even at the largest matrix size. This suggests that these two areas are the best opportunities for finding additional performance improvements.

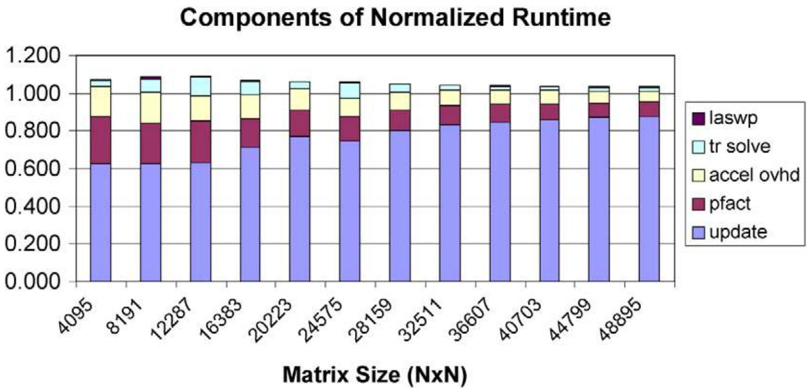


Fig. 12. Breakdown of normalized runtime into main components.

6. Related work

In prior work [17], we developed an implementation of the Linpack benchmark for the Roadrunner system, which is a petascale system built by IBM for Los Alamos National Laboratory. Roadrunner is a hybrid system that contains both x86-64 and PowerXCell 8i processors. Our Linpack implementation for the PowerXCell 8i reuses some of the design and implementation we developed for Roadrunner, but also required significant design changes and new acceleration kernels.

Because of its role as the benchmark used in the Top500 rankings, the Linpack benchmark has been extensively studied. A detailed explanation of the benchmark computations and general performance model are given in [8]. A number of researchers have studied the performance implications of alternate data layouts [11,22]. Other techniques such as multi-threading and one-sided communication have also been explored as a means to improve Linpack performance [13]. The matrix multiplication kernel at the heart of Linpack has also been carefully studied, and new techniques continue to be discovered to increase its efficiency [10].

The general topic of using specialized cores to accelerate application performance has been explored in many different forms. Recent work in this area has focused on utilizing graphics processing units (GPUs) for accelerating general purpose applications [3]. Others have developed special-purpose processors intended for use as accelerators for technical computing applications, e.g. Clearspeed [6].

Finally, in the area of programming for CBEA-compliant accelerators like the PowerXCell 8i processor, several SGEMM kernels have been previously developed for the Cell/B.E. processor [2,5,12]. To ensure compliance with the Linpack benchmark rules, all computations in our kernels are done in double precision. The primary differences affecting a double precision kernel implementation, as compared to single precision, are: (1) micro-blocking of the computation must be increased to cover 9 cycles of DP operation latency as compared to 6 cycles for SP operations, (2) the DP negative multiply subtract instruction is a destructive 3 operand instruction instead of a 4 operand instruction that preserves its inputs, and (3) the d-form load can only span 16 kB (-8 kB to $+8$ kB) of immediate offset [4]. A 64×64 DP block is 32 kB and therefore multiple pointers are needed in order to address the entire block using immediate offsets.

Alvaro et al. [2] does an in depth analysis in choosing the ideal $\{m, n, k\}$ triplet for an SGEMM kernel. For our kernel we chose a triplet of $\{4, 8, 4\}$ which results in 64 even pipeline double-floating-negative-multiply-and-subtract (dfnms) instructions and 40 odd pipeline instructions (8 loads for A, 16 loads of B and 16 shuffles of A) per tile. This left more than enough odd pipeline slots to do basic flow control, branch prediction, pointer unpacking, and register-to-register copies (using odd pipeline instructions like rotqbyi with a rotate of 0). The inner-most loop is fully unrolled such that each loop iteration updates a tile of the C matrix. The loop is further software pipelined so that several of the next loop inputs are prefetched and the most of the resultant stores are deferred until early in the next iteration. Buffer pointers and basic loop control are done through two levels of table lookup. The first table is 128 byte offsets to one of the quad-word entries in the second table. The second table contains 3 entries specifying the addend to be applied to the A buffer pointers, B buffer pointers and C buffer pointer. It also contains a pointer to the loop branch target so that every loop iteration (including the final iteration) is accurately predicted.

Unlike Alvaro's implementation, our block multiplier does not restrict the alignment of its buffers to a buffer-sized boundary. Instead we allow buffers to be aligned on any quad-word boundary at the cost of 1 even pipeline add per 1024 cycle loop.

7. Conclusion

We have described the approach we employed to modify an existing computationally intensive program to take advantage of the high-speed computational accelerators available on an IBM QS22 blade system. In particular, we have described the design and implementation of the Linpack benchmark for an IBM QS22 blade system which contains two IBM PowerXCell 8i processors. This implementation of Linpack achieves 170.7 GFLOPS on a BladeCenter QS22 with 32 GB of DDR2 SDRAM memory, which is 78% of the peak double precision computational performance of the system. This implementation was also used in a cluster of 84 BladeCeneter QS22s to achieve 11.1 TFLOPS, earning it a spot in the June 2008 Top500 list and the number one position in the corresponding Green 500 list of most energy efficient supercomputers. This work demonstrates that a high level of performance can be achieved by exploiting computational accelerators such as the SPEs of the IBM PowerCell 8i. We believe that many of the techniques we used to create our hybrid version of Linpack can be applied to other computationally intensive applications and result in significant performance improvements. As multi-core systems evolve and cores take on other specialized functions, we expect some of these techniques to be incorporated into new libraries and programming frameworks developed to exploit specialized functionality in multi-core systems.

References

[1] Advanced Micro Devices, AMD Core Math Library, http://www.amd.com/acml.

[2] W. Alvaro, J. Kurzak and J. Dongarra, Fast and small short vector SIMD matrix multiplication kernels for the CELL processor, UT-CS-08-609, January 2008.

[3] J. Bolz, I. Farmer E. Grinspun and P. Schroder, Sparse matrix solvers on the GPU: Conjugate gradients and multigrid, *ACM Transactions on Graphics (TOG)* **22**(3) (2003), 917–924.

[4] D. Brokenshire, Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance, IBM DeveloperWorks, June 2006.

[5] T. Chen, R. Raghaven, J. Dale and E. Iwata, Cell Broadband Engine Architecture and its first implementation, *IBM Journal of Research and Development* **51**(5) (2007), 559–572.

[6] ClearSpeed, Accelerated HPC Clusters, http://www.clearspeed.com/acceleration/accelhpcclusters/.

[7] J. Dongarra, J. Du Croz, I. Duff and S. Hammarling, A set of level 3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software* **16** (1990), 1–17.

[8] J. Dongarra, R. van de Geijn and D. Walker, Scalability issues affecting the design of a dense linear algebra library, *Journal of Parallel and Distributed Computing* **22**(3) (1994), 523–537.

[9] E. Gabriel, G. Fagg, G. Bosilca et al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: *11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[10] K. Goto and R. van de Geijn, Anatomy of high-performance matrix multiplication, *ACM Transactions on Mathematical Software* **34**(3) (2008), 1–25.

[11] F. Gustavson, High-performance linear algebra algorithms using new generalized data structures for matrices, *IBM Journal of Research and Development* **47**(1) (2003), 31–55.

[12] D. Hackenberg, Fast matrix multiplication on CELL systems, http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/matmul/, July 2007.

[13] P. Husbands and K. Yelick, Multi-threading and one-sided communication in parallel LU factorization, in: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Reno, NV, November 2007.

[14] IBM, Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor, Version 1.11, Section 3.1.1.3, May 2008.

[15] IBM, The IBM Software Kit for Multicore Acceleration Version 3.0 http://www.ibm.com/chips/techlib/techlib.nsf/products/IBM_SDK_for_Multicore_Acceleration, October 2007.

[16] C. Johns and D. Brokenshire, Introduction to the Cell Broadband Engine Architecture, *IBM Journal of Research and Development* **51**(5) (2007), 503–520.

[17] M. Kistler, J. Gunnels, D. Brokenshire and B. Benton, Petascale computing with accelerators, in: *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, February 2009.

[18] J. Kurzak and J. Dongarra, Implementing linear algebra routines on multi-core processors with pipelining and a look ahead, UT-CS-06-581, September 2006.

[19] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, Basic linear algebra subprograms for FORTRAN usage, *ACM Transactions on Mathematical Software* **5** (1979), 308–323.

[20] Message Passing Interface Forum, MPI: A message passing interface standard, http://www.mpi-forum.org, June 1995.

[21] Message Passing Interface Forum, MPI-2: Extensions to the message passing interface, http://www.mpi-forum.org, July 1997.

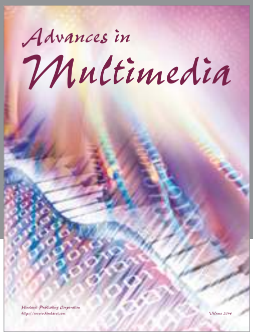
[22] J. Panziera and J. Baron, A highly efficient Linpack implementation based on shared-memory parallelism, in: *Proceedings of the 2005 International Supercomputer Conference*, Heidelberg, Germany, June 2005.

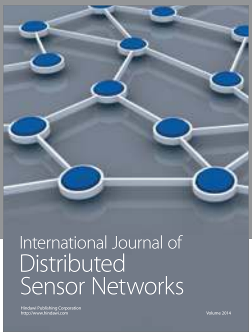
[23] A. Petitet, R. Whaley, J. Dongarra and A. Cleary, HPL – A portable implementation of the high-performance linpack benchmark for distributed memory computers, http://www.netlib.org/benchmark/hpl/, 2006.

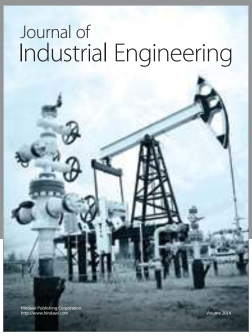
[24] The 3rd Edition of the Green 500 List, http://www.green500.org/lists/2008/06/list.php, June 2008.

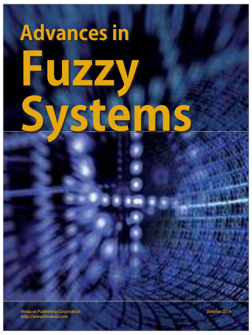
[25] TOP500 List, http://top500.org/list/2008/06, June 2008.

[26] R. Whaley and J. Dongarra, Automatically tuned linear algebra software, in: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, San Jose, CA, November 1998.









Hindawi

Submit your manuscripts at
http://www.hindawi.com