Petascale Computing with Accelerators

Michael Kistler[1], John Gunnels[2], Daniel Brokenshire[1], Brad Benton[1]

[1]IBM Corporation
11501 Burnet Road
Austin, TX 78758
{mkistler,brokensh,brad.benton}@us.ibm.com

[2]IBM Corporation
1101 Kitchawan Road
Yorktown Heights, NY 10598
gunnels@us.ibm.com

Abstract

A trend is developing in high performance computing in which commodity processors are coupled to various types of computational accelerators. Such systems are commonly called hybrid systems. In this paper, we describe our experience developing an implementation of the Linpack benchmark for a petascale hybrid system, the LANL Roadrunner cluster built by IBM for Los Alamos National Laboratory. This system combines traditional x86-64 host processors with IBM PowerXCell™ 8i accelerator processors. The implementation of Linpack we developed was the first to achieve a performance result in excess of 1.0 PFLOPS, and made Roadrunner the #1 system on the Top500 list in June 2008. We describe the design and implementation of hybrid Linpack, including the special optimizations we developed for this hybrid architecture. We then present actual results for single node and multi-node executions. From this work, we conclude that it is possible to achieve high performance for certain applications on hybrid architectures when careful attention is given to efficient use of memory bandwidth, scheduling of data movement between the host and accelerator memories, and proper distribution of work between the host and accelerator processors.

Categories and Subject Descriptors D.1.3 [**Processor Architectures**]: Heterogeneous (hybrid) systems; D.2.3 [**Software Engineering**]: Coding tools and techniques.

General Terms Algorithms, Performance, Design.

Keywords Accelerators, hybrid programming models.

1. Introduction

Over the past decade, commodity clusters have scaled in size and have become the predominant architecture in supercomputing. However, rather than continuing on the path of scaling to ever larger numbers of nodes in a cluster, a new trend is emerging in which the capability of each node in the cluster is extended through the addition of various types of computational accelerators. This has led to hybrid computing technologies which combine a mix of general and specific processing elements to provide increasingly more powerful systems.

In this paper, we describe an implementation of the Linpack benchmark we developed to run on a petascale hybrid system called Roadrunner, which combines traditional x86-64 host processors with IBM PowerXCell 8i accelerator processors. Our primary goal in this work was to develop a version of Linpack for this hybrid system that could achieve 10^15 floating point operations per second (1.0 PFLOPS). This result was important as a demonstration of the computational capability of the system. We also had the secondary goal of achieving this level of performance using the production level system software and a targeted set of optimizations to the existing benchmark.

Our implementation of Linpack for this hybrid system is based on the standard open-source implementation, High Performance Linpack (HPL) [28], which is designed for homogeneous clusters. We employed a combination of well-known optimizations for homogeneous systems and new techniques made possible by the hybrid nature of the system. We also developed a performance model for the Roadrunner system and mapped the Linpack benchmark onto this performance model. This allowed us to identify the key computational kernels that needed to be accelerated, develop a high level design for the application, evaluate potential optimizations and their interactions, and estimate the expected performance of our implementation on the full Roadrunner system.

We developed specialized computational kernels for the PowerXCell 8i accelerators that achieve very high computational efficiency while keeping demand for memory bandwidth low. We also developed offload functions for the host processors to redirect requests for these kernels to the accelerators. Data transfers between the host and accelerator are carefully scheduled to overlap with computation on host, accelerator, or both, to minimize communication costs. The remainder of the benchmark code is left unmodified and executes on the host processors.

The resulting implementation of Linpack for Roadrunner achieves 350 GFLOPS (1 GFLOPS is 1 billion floating point operations per second) on one Roadrunner compute node, 63.2 TFLOPS on a 180-node Connected Unit (CU), and 1.026 PFLOPS on the full 17-CU Roadrunner configuration. The performance result on the full configuration represents 74.6% of the peak double-precision compute capability of the system, which is comparable to the efficiency achieved by many of the homogeneous clusters that appear in the Top500 list. Since the aggregate compute capability of the x84-64 processors of the system is only 44.1 TFLOPS, it is clear that the PowerXCell 8i processors contribute the vast majority of the achieved performance. The performance per-node declines by less than 4% in scaling from a single node to the full system, which demonstrates that the system design is well-balanced and our implementation scales very well from small to large configurations.

It is not uncommon for systems to achieve efficiency in the 70%-80% range on the Linpack benchmark, but our results are

PPoPP'09 February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02…$5.00.

significant because they were achieved on a hybrid architecture and on a scale never before attempted. In the Roadrunner system, 97% of the computational capability resides in the PowerXCell 8i accelerator processors, and before the work we present here was undertaken, it was not known if a significant fraction of this capability could be effectively delivered to applications, even for a relatively regular and well-structured application such as the Linpack benchmark.

While our work focused only on the Roadrunner architecture and the Linpack benchmark, we believe that it provides valuable insights into the more general area of developing high performance applications for hybrid architectures. In particular, our work indicates that memory bandwidth is a key constraint to application performance in hybrid architectures, so careful attention must be given to reducing the memory bandwidth demands of computational kernels. Most hybrid systems have separate memory domains for the host and accelerator processors, and we found that the design of the data movement between these domains plays a significant role in achieved performance. Finally, our work shows that the host processor can often be used to perform small, poorly structured, or otherwise low efficiency computations, allowing the computations on the accelerator to achieve very high efficiency.

Since the 1.0 PFLOPS Linpack run, a number of impressive results have been achieved on the Roadrunner system with a variety of applications. Researchers at Los Alamos National Laboratory (LANL) were able to demonstrate a molecular dynamics code, SPaSM, that achieved 361 TFLOPS (double precision) on the full Roadrunner system [29]. A second application, VPIC, which employs the *particle in cell* method for plasma simulation, achieved 374 TFLOPS (single precision) on the full Roadrunner configuration, with almost linear scaling up to the full system level [4]. These codes are representative of the application workloads that LANL plans to execute on Roadrunner, demonstrating that the results we achieved with Linpack are not an isolated case, and that real applications are also able to exploit the computational capability of this hybrid architecture.

The remainder of this paper is organized as follows. Section 2 describes the hardware and system software of the Roadrunner system. Section 3 presents the design and Section 4 describes the implementation of our version of Linpack for Roadrunner. Performance results are presented in Section 5. Section 6 reviews related work and Section 7 concludes the paper.

2. The LANL Roadrunner System

2.1 System Overview

Roadrunner is a petascale hybrid supercomputer developed by IBM for the Los Alamos National Laboratory (LANL) under the LANL project name "Roadrunner" [3]. The Roadrunner project was named after the state bird of New Mexico. Roadrunner is a hybrid system combining traditional x86-64 host processors with PowerXCell 8i accelerator processors. The host processors are intended to serve as the basic execution engine for applications. The host processors also can offload compute-intensive operations to a companion PowerXCell 8i processor, which has a peak floating point capability of almost 30x that of the host processor.

An overview of the Roadrunner system is shown in **Figure 1**. The system consists of 17 Connected Units (CUs), where each CU contains 180 compute nodes and 12 I/O nodes. All the nodes in a CU are interconnected with a 288-port 4X double-data-rate (DDR) InfiniBand switch. The CUs are interconnected with a second level of 288-port 4X DDR InfiniBand switches. The network is configured in a reduced-bandwidth fat-tree topology, which means that simultaneous independent broadcasts within a CU will perform at full bandwidth, but broadcasts across CUs might see as little as 50% of full bandwidth due to link contention.

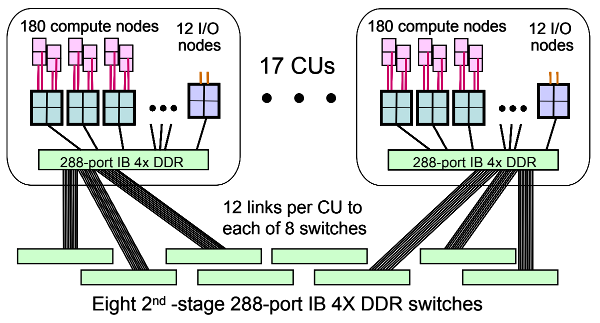

Figure 1. LANL's Roadrunner System

2.2 The Roadrunner Compute Node

The basic building block of Roadrunner is a hybrid compute node, which consists of a 4-core 1.8 GHz Opteron LS21 blade with 16 GB of memory, linked to two QS22 Cell blades, each with two IBM PowerXCell 8i processors and 8 GB of memory. The memory on each blade is physically local to the blade, meaning that only the processors on that blade have direct access to the memory. Each host processor can communicate with its associated accelerator over a dedicated PCI Express x8 link, which can provide bandwidths of up to 2.0 GB/s in each direction. Each compute node has a single 4X DDR InfiniBand adapter for application traffic and a 1 gigabit Ethernet connection used for management functions. For reliability and ease of software distribution, all the compute nodes are diskless. The compute nodes are called triblades since they combine one Opteron blade with two QS22 Cell blades, but the physical package is actually 4-wide, since one additional slot is used for the interconnect cables and bridge components that link the Opteron and PowerXCell 8i processors. **Figure 2** is a diagram of a triblade compute node of Roadrunner.

The IBM PowerXCell 8i processors contained in the QS22 blades are a new implementation of the Cell Broadband Engine™ Architecture (CBEA) [18]. The first implementation of the CBEA is the Cell/B.E. processor, which was jointly developed by IBM, Sony, and Toshiba and is used in Sony's Playstation® 3 game console. The PowerXCell 8i consists of one PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs) which implement a completely new instruction set architecture designed specifically for high-performance numerical computations. The PPE can be thought of as the "host" or "control" core, where the operating system and general control functions for an application are executed. Each SPE has a Synergistic Processor Unit (SPU), 256KB local store, and corresponding Memory Flow Controller (MFC). The SPU is an in-order streaming processor with a 128-bit 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 computa-

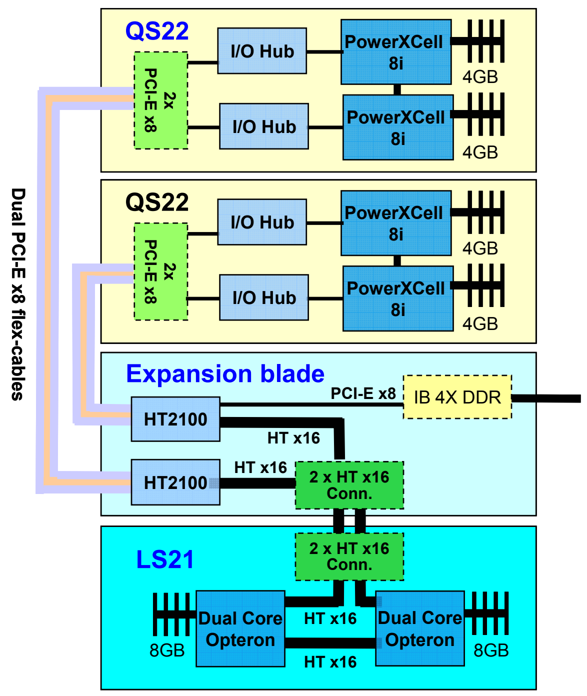

Figure 2. Diagram of the triblade Compute Node

tion 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/sec. The memory controller can support up to 25.6 GB/s of bandwidth to off-chip memory.

The IBM PowerXCell 8i differs from the Cell/B.E. in that it includes an enhanced double precision unit on the SPEs, giving it a peak computational capability of 108.8 GFLOPS in double precision (102.4 GFLOPS in the SPEs and 6.4 GFLOPS in the PPE). In addition, the IBM PowerXCell 8i supports industry-standard DDR2 SDRAM memory, enabling system designs with large memory capacities. The PowerXCell 8i processor is implemented in a 65nm SOI process with a die size of 212 square mm and runs at frequencies up to 3.2 GHz. **Figure 3** shows a die photo of the IBM PowerXCell 8i processor that identifies the new features of this processor.

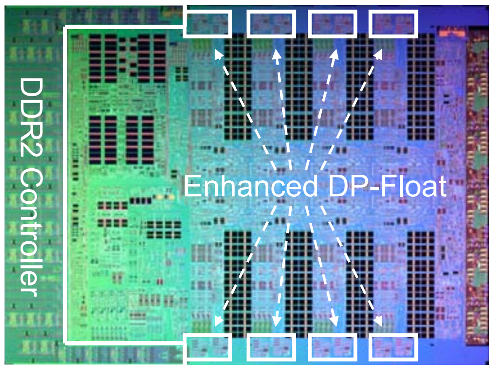

Figure 3. Die photo of IBM PowerXCell 8i processor

2.3 Systems Software

Each compute node in the Roadrunner system runs three separate OS images, one for the Opteron blade, and one for each of the QS22 Cell blades; all are based on the Fedora distribution of Linux. In addition to the standard Linux packages, each node has specialized components for application development on the hybrid architecture provided in the IBM Software Kit for Multicore Acceleration (IBM SDK) [15]. This combination of open-source software and the SDK provides compilers, debuggers, integrated development environment, performance analysis tools, and optimized libraries. A good review of programming methods and tooling for Cell/B.E. is available elsewhere [5].

A key component of the IBM SDK used in our implementation of Linpack is the Data Communication and Synchronization (DaCS) services, which provide communication and control mechanisms between the host and accelerator processors in heterogeneous multi-core systems. These mechanisms include standard two-sided send/receive message passing, one-sided remote DMA services, lightweight synchronization and mailbox facilities, process management, topology services, and error handling. In Roadrunner, DaCS uses special device drivers to route host-to-accelerator communications over the PCI Express link between the Opteron and QS22 blades, and the DaCS configuration associates a dedicated IBM PowerXCell 8i processor to each of the host Opteron cores.

For inter-node communication, Roadrunner uses the Message Passing Interface (MPI) [23][24], which has become the *de facto* standard for scientific parallel computing. MPI provides a rich set of semantics for distributed-memory programming where explicit data movement is required by the application developer. The MPI library chosen for use on Roadrunner is Open MPI, an open source MPI implementation from the Open MPI Project [11]. The Open MPI library allows for selection (and restriction) of components at both configuration time and run time, and its numerous tuning parameters allow us to carefully tune Open MPI for the Roadrunner environment.

3. The Design of Hybrid Linpack

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

LU factorization is typically performed using a blocked, right-looking algorithm, where each iteration produces a portion of the final L and U matrices and leaves a reduced region of the matrix, the *trailing submatrix*, to be solved by the remaining iterations. This approach allows much of the computation to be performed using matrix-matrix (BLAS3) operations [8], which are much more efficient than vector-vector (BLAS1) or matrix-vector (BLAS2) operations [22] on modern computer systems with deep memory hierarchies. The high-level flow of the benchmark is as follows:

```
Allocate and Initialize Matrix
Iterating over block columns:
    Panel Factorization - factor current block column
    Forward Pivot trailing submatrix
    Compute block row of final U matrix (DTRSM)
    Update trailing submatrix      (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 on the left-most column of blocks in the trailing submatrix. This produces one block column of the final L matrix, called the *L-panel*. Pivoting is performed within the panel during panel factorization to ensure numerical stability. The sequence of pivot operations is saved and then applied to the trailing submatrix in the forward pivoting step. Then a triangular solve with multiple right-hand-sides (DTRSM) is performed on the top block row of the trailing submatrix, 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 submatrix. On termination of the main loop, the LU factorization is complete, and the orginal system of equations Ax=b has been transformed into Ux=y, where y = Lb. At this point the final solution, x, 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), an $O(N^3)$ operation. The panel factorization, DTRSM, and triangular solve operations are all $O(N^2)$ operations.

Parallel LU factorization builds on the standard blocked right-looking approach by distributing the blocks of the matrix across nodes in a block-cyclic fashion. Nodes are organized into a two-dimensional grid (rows and columns). Each node performs updates to its own blocks during the execution of the algorithm using data supplied by other nodes, and in turn supplies its blocks to other nodes when needed. Inter-node communication is required for pivoting in both the panel factorization and forward pivot steps.

We developed a performance model for the Linpack benchmark to explore the potential performance we could achieve on the Roadrunner system. The model allowed us to identify the key computational kernels that needed to be accelerated, to develop a high level design for the application, evaluate potential optimizations and their interactions, and to estimate the expected performance of our implementation on the full Roadrunner system. Details of the performance model are described elsewhere [20], so only the key results are included here. Clearly, the matrix update (DGEMM) operation should be assigned to the accelerators, since this is where the bulk of the computation occurs. The performance model indicated that it would also be beneficial to perform the DTRSM computations on the accelerators. The remainder of the benchmark, including the panel factorization and pivoting code, would remain on the host. In particular, the panel factorization and final triangular solve step are performed by the host processors, in large part because of the inter-node communication required in performing these operations.

The decision to perform the matrix update on the accelerators drove another important design choice - where to store the matrix data? During any given iteration, the vast majority of the data located on a node does not need to be communicated to other nodes, but may be updated by the matrix update operation. This

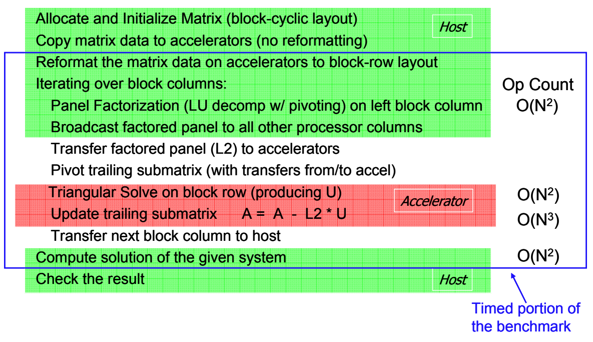
Allocate and Initialize Matrix (block-cyclic layout)
Host
Copy matrix data to accelerators (no reformatting)
Reformat the matrix data on accelerators to block-row layout
Iterating over block columns:
Op Count
Panel Factorization (LU decomp w/ pivoting) on left block column
O(N²)
Broadcast factored panel to all other processor columns
Transfer factored panel (L2) to accelerators
Pivot trailing submatrix (with transfers from/to accel)
Triangular Solve on block row (producing U)
Accelerator
O(N²)
Update trailing submatrix A = A - L2 * U
O(N³)
Transfer next block column to host
Compute solution of the given system
O(N²)
Check the result
Host
Timed portion of the benchmark

Figure 4. High Level Design of Hybrid Linpack

means that it is generally unnecessary to transfer the entire results of the matrix update of one iteration back to the host, since it much of this data would simply have to be transferred right back to the accelerator in the next iteration. As a result, we chose to store the matrix data in accelerator memory. These basic design decisions led us to the high-level design for hybrid Linpack shown in **Figure 4**.

For Roadrunner, Linpack is configured so that each node runs 4 MPI tasks, one for each host core, for a system-wide total of 17*180*4 = 12240 MPI tasks. Our performance model indicates that an aspect ratio for the task grid between 1:1 and 1:8 will give the best performance. Therefore, we chose a rectangular grid of 68 x 180 tasks to both stay within the optimal aspect ratio and to keep all tasks in any given row mapped to a single CU to leverage higher network bandwidth within CUs for row broadcasts. MPI is configured to assign a consecutive set of ranks to the four MPI tasks of each node, and the task grid is created in row order, so that row communications can take advantage of shared memory communications on 3/4 of the links.

Preliminary experiments indicated that roughly 14GB per node can be used to hold matrix data. This translates to global matrix dimensions of 2,470,000 x 2,470,000. The expected run time for a matrix of this size, at a performance level of 1.0 PFLOPS, is 2 hours, 48 minutes inside the timed portion of the benchmark, plus some additional time for initialization and verification of the result. The expected mean time between failures (MTBF) for the full Roadrunner system is 22 hours, which gives a comfortable margin for completing a 1.0 PFLOPS run without the need for integrated checkpointing.

4. Implementation Details

4.1 Base Implementation and Optimizations

We started with the publicly available implementation, High Performance Linpack (HPL) [28], and extended it to support the hybrid architecture of the Roadrunner system. There were a number of good reasons for taking this approach. One is that submissions to the Top500 list must use certain components of HPL in order to be considered valid. Another reason is that HPL already contains a number of optimizations and tuning parameters that are useful in boosting performance. In particular, the technique of pipelining iterations of the main loop of the LU factorization, or *look-ahead* [21], is a key optimization we wanted to leverage. Finally, we wanted to demonstrate that applications initially developed for homogeneous clusters could be successfully adapted to hybrid systems without a complete redesign. We provide details of the implementation effort required in Section 4.4.

We began by implementing certain well-know optimizations for Linpack. In particular, we reformat the matrix into a hierarchically blocked organization. This is a well-known technique for the optimization of dense matrix kernels [13], and is well suited to a hybrid implementation since the matrix data already must be transferred between the host and accelerator. To ensure compliance with the benchmark rules, we could not actually reformat the data during transfer from the host to accelerator. However, we expect this technique will be beneficial for many hybrid applications. The data organization we chose for the matrix is *blocked-row* format, where blocks of 64 columns are stored in row-major format. Reformatting the matrix in this way 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. We also added special memory allocation functions to obtain storage for the matrix and auxiliary buffers in huge page memory, which significantly reduces memory management overheads.

We developed a new implementation of the forward pivot operation that employs the MPI collectives MPI_Scatterv and MPI_Allgatherv. Rows in the top block row are transferred into the body of the trailing submatrix with MPI_Scatterv, and the rows that will replace these are collected into the top block row with MPI_Allgatherv. Each row of processors receives a copy of this top block row to use as input to the subsequent trailing matrix update operation. There were two motivations for this work. The first was that we believed that the MPI collectives could actually outperform the existing pivot implementations in the standard HPL source base. We later determined that the performance benefit afforded by the collective communications routines was minimal. The second motivation was that this allowed us to centralize the code required to transfer matrix data to/from the accelerators during the pivot step. HPL contains several alternative pivoting operations, but the code to implement these is distributed across a collection of more than a dozen source files. By implementing our own pivot operation, we could limit our source changes to the addition of just three files.

4.2 Leveraging the Accelerators

We then added support to utilize the accelerators of the hybrid system. We did this by replacing the compute-intensive kernels identified by our performance modeling with remote invocations of accelerated versions of these kernels that utilize the PowerXCell 8i processors.

Since Linpack performance is dominated by the performance of the DGEMM operation, many of the system design decisions were dictated by the DGEMM design. A block size of 64x64 was chosen as the largest power of two that allowed the SPE local storage to accommodate double buffered blocks of the three matrices. The matrix multiplication kernel that computes $C^T = C^T - A \times B^T$ was developed in assembly using techniques similar to those described by Alvaro [2], though our work pre-dates this publication. The resulting kernel of 7664 bytes achieves 99.87% of the available 12.8 GFLOPS per SPE.

This very high computational performance in DGEMM can place very high demands for bandwidth on the memory subsystem. In a typical implementation using 64x64 element blocks, DGEMM could require 17.9 GB/s of sustained memory bandwidth, with occasional bursts of nearly 23.8 GB/s. Since the maximum memory bandwidth supported by the memory controller of the PowerXCell 8i is 25.6 GB/s, DGEMM could practically consume all of the available memory bandwidth. This was problematic since we expected additional demands on the memory subsystem from computations performed by the PPE and communications with the host processor. To address this issue, we chose to increase the block size used by the Linpack benchmark to 128 with the accelerators operating on 64x64 sub-blocks, and we developed a novel access pattern that serpentines through the matrix. These two optimizations decrease the required sustained memory bandwidth of DGEMM to less that 12 GB/s without occasional bursts of extra demand, leaving ample memory bandwidth to support concurrent PPE computation and host communication.

An optimized DTRSM kernel that performs the triangular solve was also developed. This kernel operates on 128x16 sub blocks such that the 8 SPEs in aggregate asynchronously operate on 128x128 blocks. This kernel is 6000 bytes of instruction text with a compute efficiency of 98.75%.

The DGEMM and DTRSM kernels are the foundation of a PPE callable acceleration library that parallelizes the computations across 8 SPEs. Integral multiples of the kernel block size are offloaded to the SPEs while the remaining partial rows and columns are handled by the PPE in parallel with the SPE request. The PPE initiates a request by storing the request parameters to a parameter block in memory and sending each SPE a message containing a command opcode and the effective address of the parameter block. Each SPE indicates that it has completed its portion of the operation by updating a shared completion variable in main storage which the PPE polls until all SPEs have finished.

Matrix components are stored in varied formats in accelerator memory to improve locality of reference and data transfer speeds. The matrix is stored in a blocked-row organization, described above, in big-endian format. U panels are row ordered, big-endian because they contain rows received from other nodes during the pivoting phase. Since L panels are the result of a panel factorization performed by the host processors, they are stored column-ordered, little-endian. The acceleration library provides data reformatting functions to convert the data between the three different formats using the SPEs memory flow controller (MFC).

The need for data reformatting was further reduced by incorporating the byte swapping of the L panel data into the DGEMM and DTRSM kernels. This allowed us to handle the byte ordering differences between the Opterons and PowerXCell 8i processors with no loss of performance. In addition, both the DTRSM and DGEMM SPE off-load functions were extended to support certain inputs and outputs in alternate formats, further reducing data reformatting costs. For example, the DTRSM SPE off-load function can produce its result either back into the row ordered U panel or directly into the matrix in blocked row format. Storing the U panel directly into the matrix saved an extra copy step that would have been required on the processor row that owned this portion of the matrix. This also drove the requirement for two variants of the DGEMM off-load function - one that takes the U panel input from a row-ordered buffer and another that can take the U panel directly from matrix storage. Supporting these alternate formats further reduced memory bandwidth requirements by avoiding data copying / reformatting, with no effect on the computational efficiency of these kernels.

4.3 Advanced Optimizations

We further optimized our implementation by taking advantage of the parallelism available between the host and accelerator processors. Since the host offloads the trailing matrix update step to the accelerators, it is then free to perform other computations or communications. We found two ways in which this could be exploited.

First, we modified the benchmark to overlap panel factorization performed on the host with the accelerator DGEMM update of some number of block columns. There is a tradeoff involved in determining the number of block columns to update, or overlap value, during panel factorization. A small overlap value could allow the DGEMM work to complete before panel factorization, leaving the PowerXCell 8i processors idle. A large overlap value could delay the work following panel factorization that needs to employ the accelerators. Currently we have determined a good overlap value through experimentation, but with some additional effort this overlap point could be determined dynamically, which would allow it to vary over the course of the execution, with the potential to further improve performance.

Our second optimization is to overlap the broadcast of the L panel with DGEMM updates on the accelerators. This optimization is significant because all MPI tasks participate in L panel broadcast, and thus will benefit from this optimization. In contrast, only one column of processors is involved in panel factorization in a given iteration of the main loop, which limits the benefits of our first optimization.

Finally, we identified a problem in MPI scaling as we started ramping execution to larger configurations, specifically in the Open MPI implementation of MPI_Allgatherv. This function is used in the pivoting step, where rows in the body of the matrix are gathered to all processor rows to be used as input to the matrix update step. We determined that the performance problem stemmed from the fact that the number of matrix rows gathered from each participating task could vary significantly, and MPI_Allgatherv implementation in Open MPI was really designed for the case of equally sized data coming from all tasks [6]. To resolve this performance problem, we developed a two-step implementation of Allgatherv, where the first step uses point-to-point messages to distribute the rows equally across the tasks, and the second step issues MPI_Allgatherv. Our new Allgatherv implementation is also completely general and can be employed on any architecture. In fact, we plan to offer this implementation as a contribution to the Open MPI code base.

4.4 Implementation Effort

The base for our implementation, High Performance Linpack, contains approximately 16K lines of code. Our implementation of the forward pivot using MPI collectives added about 500 lines to the code base, and new functions to allocate matrix and buffer storage from huge page regions added another 250. Both of these optimizations are implemented in a general way and are not specific to the Roadrunner architecture.

Our computational kernels for the PowerXCell 8i contain about 4000 lines of C-language code and approximately 6000 lines of SPU assembly code. To access these functions from the host, we wrote an additional 2000 lines of C code, split about evenly between the host and PPE, for function offload and data communication between the two processors. The implementation of the offload functions is relatively straightforward, involving marshalling of parameters into messages and demarshalling on the receiver. While we wrote these functions by hand, tools or frameworks could be developed to automatically generate or significantly reduce the size of these functions.

Overlapping panel factorization and panel broadcast on the host with DGEMM operations on the accelerators required some code restructuring but very little in the way of additional code. Our new Allgatherv implementation required about 200 lines of additional C code.

In total, we added about 7000 lines of C code and 6000 lines of SPU assembly to the HPL code base. However, the majority of this new code is in the specialized kernel functions we developed for the PowerXCell 8i accelerators. Recent versions of the IBM SDK for Multicore Acceleration [15] include versions of the BLAS and LAPACK libraries for the PowerXCell 8i, so typical applications should be able to use existing libraries and avoid this effort. The remaining 3000 lines include our basic and advance optimizations and the interface functions that connect the host and accelerator processes. Excluding the specialized kernels, only 3000 lines of code, or about 20% of the original code size, were needed to create a hybrid implementation of Linpack.

5. Results

5.1 Single Node Results

We executed our hybrid implementation of Linpack on production-level versions of the Roadrunner compute nodes and verified that it produces correct solutions. **Figure 5** presents the performance of our hybrid Linpack implementation on a single Roadrunner compute node, along with the measured performance of the standard HPL implementation linked with the AMD Core Math Library (ACML) [1]. For these experiments, Linpack is configured to use a 1x4 grid of processes and both the host-only and accelerated version use our huge page optimization and MPI collectives pivoting implementation. The x-axis of the graph is the size of the matrix (N), and the y-axis indicates the achieved performance of the benchmark in GFLOPS. On a single compute node, our implementation of Linpack achieves 350 GFLOPS for a matrix of size N=42943. This performance result is 77.8% of the peak compute capability of a Roadrunner compute node (counting all the flops available in the Opteron, PowerPC, and SPU cores), which is equal to or better than the Linpack efficiency achieved by many conventional systems. These results also show that our implementation of Linpack utilizing the IBM PowerXCell 8i-based accelerators can outperform the host-only implementation by a factor of 28 at large problem sizes.

To determine the benefits of overlapping host processing with DGEMM updates on the accelerators, we performed experiments in which we selectively disabled the overlap of panel factorization with DGEMM and the overlap of panel broadcast with DGEMM. The graph in **Figure 6** presents the results of these experiments, using the same axes as **Figure 5**. These experiments indicate that both overlap techniques have significant value, but the overlap of the panel broadcast with DGEMM has a far greater impact on performance, even on this relatively small configuration. The likely reason for this is that every task participates in a broadcast in every iteration of the main loop, whereas only one column of

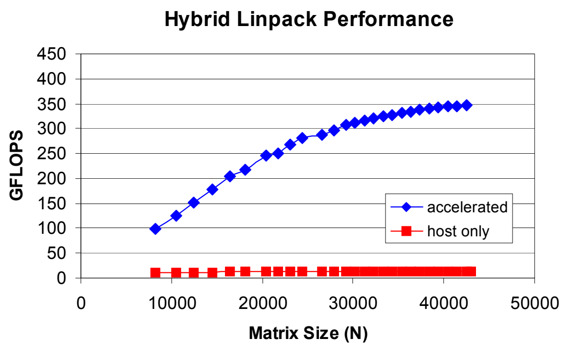

Figure 5. Hybrid Linpack Performance by Matrix Size

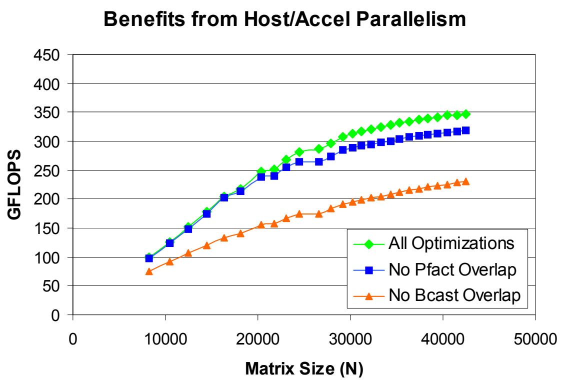

Figure 6. Performance of various overlap scenarios

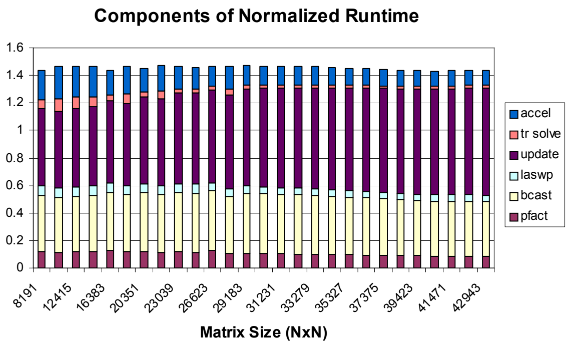

Figure 7. Breakdown of execution time

tasks (in our single node experiments, just one task) can overlap panel factorization with DGEMM.

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 *pfact*, which is the panel factorization step, *bcast*, which is the broadcast of the L panel across rows of processors in the grid, *laswp*, which is the local data transfers needed for pivoting, *update*, which is the DTRSM and DGEMM update of the trailing submatrix, *tr_solve*, which is the triangular solve performed during the backsolve step to obtain the final solution, and *accel*, which is a category we added to track the costs of matrix reformatting and data transfer to/from the accelerators.

Figure 7 shows the breakdown of execution time on a single compute node 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 exceeds the total runtime - this is because some of the optimizations we implemented allow the update function, being performed on the accelerators, to be performed concurrently with operations on the host, such as panel factorization and broadcast. There are some clear trends visible in this graph. The first trend is that *update* 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. The *bcast* component is a relatively constant fraction of runtime, which is somewhat unexpected. We believe this

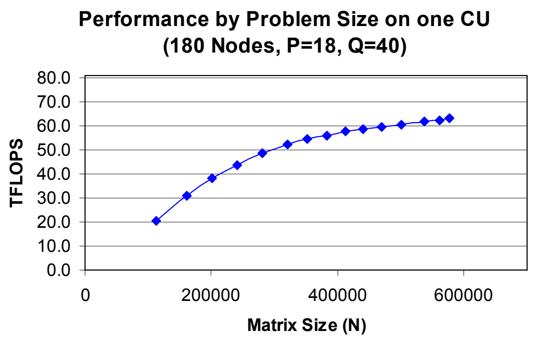

Figure 8. Performance on one Connected Unit

is caused by load imbalance, and does not accurately reflect the actual time required for this communication. 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). In particular, the *accel* overhead for the largest matrix is reduced to nearly half its value, indicating that host to cell communications are not a bottleneck in this design.

5.2 Multi-Node Results

To evaluate the scalability of our implementation, we ran a graduated series of problem sizes across all 180 nodes of a single CU. What we expected to see was a steady rise in performance that asymptotically approaches the performance achieved on a single node. The results of this experiment are shown in **Figure 8**. Here we see a nice rise up to a peak of 63.2 TFLOPS, which represents 78.1% of peak for one CU. It is a bit surprising that the achieved performance on a full CU actually exceeds the results achieved on a single node. The explanation for this phenomenon lies in the aspect ratio of the task grid used in these two configurations - the 18x40 aspect ratio is more favorable than a 1x4 aspect ratio for our implementation.

Finally, **Figure 9** shows performance scaling across multiple CUs of the Roadrunner system. Only a limited set of results could be obtained due to limitations on availability of larger sys-

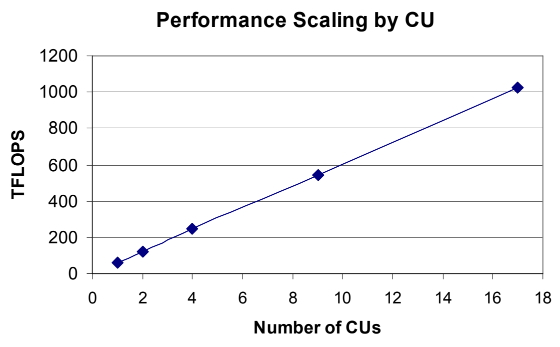

Figure 9. Performance by number of CUs

tem configurations. Nevertheless, our results show excellent scaling from a single CU all the way up to the full 17 CU configuration of Roadrunner. The achieved performance on 17 CUs was 1.026 PFLOPS, which is 74.6% of the peak of this configuration. This implies that scaling effects create no more than a 3.5% performance loss (measured as percent of peak) from 1 to 17 CUs.

6. Related Work

There is a large body of related work in the general area of exploiting accelerators to increase computational performance. Recent work in this area has focused on utilizing graphics processing units (GPUs) for accelerating a variety of dense linear algebra routines such as LU, QR, and Cholesky factorization [26][30]. Others have developed special-purpose processors intended for use as accelerators for technical computing applications, e.g. Clearspeed [7]. However, recent work has found that the benefits of accelerators for certain large scale applications can only be realized when the accelerator processors have a large computational capability in comparison to the host processors [19]. This suggests that the PowerXCell 8i could be quite beneficial as an accelerator for such applications.

Due to its position as the benchmark used in the Top500 rankings, the Linpack benchmark is another area that has been extensively studied. A detailed explanation of the benchmark computations and general performance model are given in [9]. A number of researchers have studied the performance implications of alternate data layouts [13][27]. Other techniques such as multithreading and one-sided communication have also been explored as a means to improve Linpack performance [14]. 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 [12].

Both AMD and Intel now offer quad-core processors with significantly greater peak performance than the 1.8 GHz Opteron host processors in a Roadrunner compute node. For example, a 3.0 GHz Intel "Harpertown" quad-core processor has a peak of 48 DP GFLOPS (12 DP GFLOPS per core), and Intel has published benchmark results for a two processor (8 core) Harpertown system that achieves 81 GFLOPS on Linpack [17]. However, the Roadrunner compute node still achieves more than 4x the performance of this system.

Finally, in the area of programming for CBEA-compliant accelerators like the PowerXCell 8i processor, a number of efforts have focused on the development of BLAS and related dense linear algebra libraries to make the capabilities of the SPUs more accessible. The Cell SDK contains implementations of both the BLAS and LAPACK libraries that allow programs running on the PPE to transparently utilize the SPUs. Alvaro et al. [2] describe a systematic approach to developing computational kernels for the SPU that are both fast and small in terms of code size, an important consideration given the limited size of the local store (256 KB) of an SPU. In the area of compilers, IBM has developed an single source compiler that uses OpenMP [25] directives to identify regions of a program that can be offloaded to the SPUs [10].

7. Conclusions

We have described an implementation of the Linpack benchmark for a petascale hybrid system called Roadrunner, which combines traditional x86-64 host processors with IBM PowerXCell 8i accelerator processors. We used a combination of traditional optimization techniques, highly optimized computational kernels for the PowerXCell 8i processors, and novel techniques for exploiting additional parallelism available in hybrid architectures. We found that the key to high performance in this system lay not only in crafting highly efficient kernels, but also in carefully conserving memory bandwidth in computational kernels, overlapping computation with host-to-accelerator and host-to-host communications, and utilizing the host processor to perform low efficiency computations, allowing the computations on the accelerator to achieve very high efficiency.

Our version of Linpack achieves 350 GFLOPS on a single Roadrunner compute node, which is 77.8% of the peak double-precision compute capability. On multi-node configurations, Linpack demonstrates excellent scalability, achieving 63.2 TFLOPS on a 180-node Connected Unit (CU), and 1.026 PFLOPS on the full 17-CU Roadrunner configuration. These results are 78.1% and 74.6% of the peak efficiency for these configurations, respectively. These results are particularly significant because they are the first demonstration of a large scale hybrid system that achieves efficiency on the Linpack benchmark comparable to homogeneous systems.

References

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

[2] W. Alvaro, J. Kurzak, and J.J. Dongarra, Fast and Small Short Vector SIMD Matrix Multiplication Kernels for the CELL Processor. UT-CS-08-609, January 2008.

[3] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho. "Entering the Petaflop Era: The Architecture and Performance of Roadrunner", in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Nov 2008

[4] K.J. Bowers, B.J. Albright, B.K. Bergen, L. Yin, K.J. Barker, D.J. Kerbyson, 0.365 Pflop/s Trillion-particle Particle-in-cell Modeling of Laser Plasma Interactions on Roadrunner , in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Nov 2008

[5] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca, A Rough Guide to Scientific Computing on the PlayStation 3, Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee Knoxville, May 11, 2007.

[6] J. Chen, Y. Zhang, L. Zhang, W. Yuan, Performance Evaluation of Allgather Algorithms On Terascale Linux Cluster with Fast Ethernet, Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05), IEEE, 2005

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

[8] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1--17.

[9] J. J. Dongarra, R. A. van de Geijn, D. W. Walker, Scalability Issues Affecting the Design of a Dense Linear Algebra Library, Journal of Parallel and Distributed Computing, Vol 22, Number 3, pp 523--537, 1994

[10] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture, IBM Systems Journal, Vol 45, Number 1, 2006

[11] E. Gabriel, G. Fagg, G. Bosilca, et al, Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, Euro PVM/MPI, September, 2004.

[12] K. Goto and R. A. van de Geijn, Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software, to appear.

[13] F. G. Gustavson, High-performance linear algebra algorithms using new generalized data structures for matrices, IBM Journal of Research and Development, Vol. 47, Number 1, 2003, pp 31-55

[14] 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, Nov 2007

[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

[16] IBM, Data Communication and Synchronization Library for Hybrid-x86 Programmers Guide and API Reference, October 2007.

[17] Intel Corp, Intel® Xeon® Processor 5000 Sequence: HPC Benchmarks: Dense Floating-point, http://www.intel.com/performance/server/xeon/hpcapp.htm

[18] C. R. Johns and D. A. Brokenshire, Introduction to the Cell Broadband Engine Architecture, IBM Journal of Research and Development, Vol 51, Number 5, 2007, pp 503-520

[19] D. J. Kerbyson and A. Hoisie, Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems, Workshop on Unique Chips and Systems (UCAS2), IEEE Symposium on Performance Analysis of Systems and Software (ISPASS06), Austin, TX, Mar 2006

[20] M. Kistler, J. Gunnels, D. Brokenshire, B. Benton, Programming the Linpack Benchmark for Roadrunner, IBM Journal of Research and Development, to appear

[21] J. Kurzak and J. Dongarra, Implementing Linear Algebra Rutines on Multi-Core Processors with Pipelining and a Look Ahead, UT-CS-06-581, September 2006.

[22] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Soft., 5 (1979), pp. 308--323.

[23] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, June 1995. http://www.mpi-forum.org.

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

[25] OpenMP Specifications, http://www.openmp.org/drupal/node/view/8

[26] G. Quintana-Orti, F.D. Igual, E.S. Quintana-Orti, R. van de Geijn, Solving Dense Linear Algebra Problems on Platforms with Multiple Hardware Accelerators, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-22. May, 2008.

[27] Panziera J.-P. and Baron J. A Highly Efficient Linpack Implementation Based on Shared-Memory Parallelism. In Proceedings of the 2005 International Supercomputer Conference, 2005.

[28] A. Petitet, R. C. Whaley, J. 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

[29] S. Swaminarayan, K. Kadau, T.C. Germann, 350-450 Tflops Molecular Dynamics Simulations on the Roadrunner General-purpose Heterogeneous Supercomputer, in Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Nov 2008

[30] V. Volkov and J. Demmel, LU, QR, and Cholesky Factorizations using Vector Capabilities of GPUs, University of California at Berkeley Technical Report UCB/EECS-2008-49, May 2008