DISCLAIMER

Dense and Sparse Matrix Operations on the Cell Processor

Samuel Williams, John Shalf, Leonid Oliker, Parry Husbands, Katherine Yelick
Lawrence Berkeley National Laboratory
1 Cyclotron Road
Berkeley CA, 94720
{SWWilliams, JShalf, LOliker, PJRHusbands, KAYelick}@lbl.gov

Abstract

The slowing pace of commodity microprocessor performance improvements combined with ever-increasing chip power demands has become of utmost concern to computational scientists. Therefore, the high performance computing community is examining alternative architectures that address the limitations of modern superscalar designs. In this work, we examine STI's forthcoming Cell processor: a novel, low-power architecture that combines a PowerPC core with eight independent SIMD processing units coupled with a software-controlled memory to offer high FLOP/s/Watt. Since neither Cell hardware nor cycle-accurate simulators are currently publicly available, we develop an analytic framework to predict Cell performance on dense and sparse matrix operations, using a variety of algorithmic approaches. Results demonstrate Cell's potential to deliver more than an order of magnitude better GFLOP/s per watt performance, when compared with the Intel Itanium2 and Cray X1 processors.

Keywords

STI, Cell, GEMM, SpMV, sparse matrix, three level memory

Introduction

Over the last five years, a plethora of alternatives have been suggested for cache-based architectures including scratchpad memories, paged on-chip memories, and three level memory architectures. Such software controlled memories can potentially improve memory subsystem performance by supporting finely-controlled prefetching and more efficient cache-utilization policies that take advantage of application-level information. Intelligent prefetching and scheduling policies can significantly improve the utilization of off-chip bandwidth – addressing the increasing requirements for these limited resources as multi-core chip technology becomes more prevalent.

Scratchpad memories [15, 16, 17] segment the addressable memory space into a cacheable off-chip space, and an on-chip space. The on-chip space is typically no larger than a few kilobytes. For many applications, this can provide significant (50%) improvement in performance by avoiding cache misses. However, the typical benchmark is a small embedded suite and often can easily fit within the scratchpad.

An alternate approach is paged on-chip memory [13]. The contents of the larger off-chip memory are paged in and out of the on-chip memory much like how virtual memory pages are swapped in and out of physical memory to disk. Thus control is the responsibility of the OS rather than the hardware or the program. A cache can be placed between the on-chip memory and the processor for improved performance. For some large scientific applications, performance can improve by 50%. For others, like TPCC, performance can be severely inhibited. The highly efficient VIRAM [14] coupled paged on-chip memory with high performance vector processing to achieve very high performance.

Three level memories [11, 12] create a third (registers, on-chip, off-chip) address space for on-chip memory. Placement of data into this address space is controlled by the program. In addition to avoiding cache misses, it is possible to avoid TLB misses as well. Often it is advantageous to utilize this memory as a buffer. In addition to higher performance, this approach is far more scalable.

Until recently, few of these architectural concepts made it into mainstream processor designs, but the increasingly stringent power/performance requirements for embedded systems have resulted in a number of recent implementations that have adopted these concepts. Chips like the Sony Emotion Engine [8, 9, 10] and Intel's MXP5800 both saw high performance at low power by adopting the three level memory architecture. More recently, the STI Cell processor has adopted a similar approach.

This paper studies the applicability of the STI Cell processor to the most common of scientific kernels – dense matrix multiplication and sparse matrix vector multiplication. Cell is a high-performance mainstream implementation of software-controlled memory in conjunction with considerable floating point resources that are required for demanding numerical algorithms. The current implementation of Cell offers higher performance single-precision arithmetic, which is widely considered insufficient for the majority of scientific

applications. However, we adopted Cell as a testbed architecture with the understanding that future variants for scientific computing will eventually include a fully pipelined double precision floating point unit and support for large virtual pages. This paper explores the complexity of mapping scientific algorithms to this chip architecture and uses simulation methods and analytic models to predict the potential performance of dense and sparse matrix arithmetic algorithms on Cell.

We start with an overview of the Cell processor architecture. This is followed by a discussion of viable programming models. In the next section, the performance prediction methodology and simulation methods are introduced. Finally, we describe various methods for mapping dense and sparse matrix arithmetic to the Cell processor and predict their performance. The paper concludes with a glimpse into future explorations.

1. Cell Background

Cell [1, 2] was designed by a partnership of Sony, Toshiba, and IBM (STI) to be the heart of Sony's forthcoming PlayStation 3 gaming system. Cell takes a radical departure from conventional multiprocessor or multi-core architectures. Instead of using identical cooperating processors, it uses a conventional high performance PowerPC core that controls eight simple SIMD cores, or SPE's. An overview of Cell is provided in Figure 1.

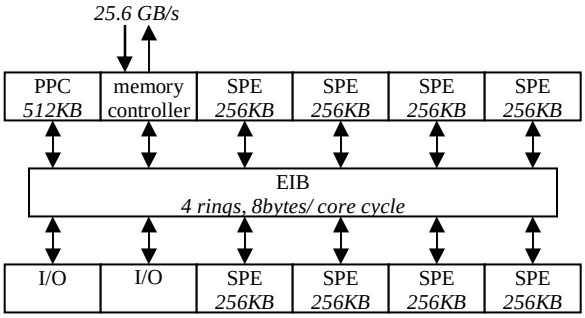


Figure 1

Cell Processor - Eight SPE's, one PowerPC core, one memory controller, and two I/O controllers are connected via four rings. Each ring is 128b wide and runs at half the core frequency. Each SPE has its own local memory from which it runs programs.

Unlike a typical coprocessor, each SPE has its own local memory from which it fetches code and reads and writes data. The PowerPC core, in addition to virtual to physical address translation, is responsible for the management of the contents of each SPE's 256KB of non-cache coherent local store. Thus to load and run a program on an SPE, the PowerPC core initiates the DMA's of SPE program and data from DRAM to the local store. Once the DMA's complete, the PowerPC core starts the SPE. For predictable data access patterns the local store approach is highly advantageous as it can be very efficiently utilized through explicit software-controlled scheduling. Improved bandwidth utilization through deep pipelining of memory requests, requires less power, and has a faster access time than a large cache due in part to its lower complexity. If however, the data access pattern lacks predictability, then the advantages of software managed memory are lost.

Access to external memory is handled via a 25.6GB/s XDR memory controller. The PowerPC core, the eight SPE's, the DRAM controller, and I/O controllers are all connected via 4 data rings, collectively known as the EIB. The ring interface within each unit allows 8 bytes/cycle to be read or written. Simultaneous transfers on the same ring are possible. All transfers are orchestrated by the PowerPC core.

Each SPE includes four single precision 6-cycle pipelined FMA datapaths and one double precision half-pumped 9-cycle pipelined FMA datapath [20]. As more information becomes publicly available, these may need to be refined as sources are terse. Thus for computationally intense algorithms like GEMM, we expect single precision implementations to run near peak whereas double precision would drop to approximately one tenth the peak single-precision flop rate according to IBM's design specifications [21]. Similarly, for bandwidth intensive applications – SpMV, we expect single precision to be between 50% and four times as fast depending on density and uniformity.

2. Programming Models

Although the presence of eight independent processing elements offers huge benefits for peak performance, it presents a very challenging programming model. In this section, we explore some of the options available for mapping scientific algorithms onto this complex new architectural model.

The data-parallel programming model is very common in the sciences and offers the simplest and most direct method of decomposing the problem. The programming model is very similar to loop-level parallelization afforded by OpenMP or the vector-like multistreaming on the Cray X1 and the Hitachi SR-8000. Although this decomposition offers the simplest programming model, the restrictions on program structure and the fine-grained synchronization mean that it may not be the fastest or the most efficient approach.

If we adopt a more loosely coupled model, the same methods that are applied to partitioning problems for distributed memory machines can be applied to this processor as if it were a "cluster on chip." The DMA engines on the SPE's can be used to mimic a message-passing interface to coordinate action on the SPE's. However, for some categories of algorithms, this model can become significantly more complicated than a data-parallel approach.

A pipelined execution model was also envisioned by the Cell designers as a way to support streaming algorithms such as video codecs by partitioning stages of the process among the SPEs. However, we do not consider this programming model in this paper because early analysis indicated it did not present

significant benefits in easer of programming or performance over the other programming models.

In each model, it is the responsibility of the PowerPC core to manage the flow of data in, out, and between the SPE's. This management is very computationally light. Thus the PowerPC core, and most likely its L2 cache will be available for system functions without undermining computational performance or consuming much power.

Each SPE is capable of simultaneously servicing DMA's and executing code. Thus instead of loading data, computing on it, then returning the results before moving on to the next block – something analogous to bulk synchronous, the SPE's should load data for the next operation, store data for the previous operation, and compute the current operation simultaneously. This of course necessitates double buffering inputs and outputs, and will incur a startup and finish overhead penalty like any pipelining approach. Nevertheless, in a data parallel approach, it should allow high utilization of either memory bandwidth or computational assets.

3. Simulation Methodology

In this paper, performance estimation is broken into two steps commensurate with the two phase double buffered computational model. This provides a high level understanding of the performance limitations of the Cell processor on various algorithms. Once we gain access to a cycle accurate simulator, we will verify our results and gain understanding of the processor features that limit performance.

In the first step, pencil and paper estimations were performed for operations such as gather, SAXPY, dot product, and register blocked matrix multiply. The resulting relations were quantified into models that took into account the SIMD nature of the SPE's – i.e. 4 FLOPs/cycle vs. 1 FLOP/cycle at four times the frequency.

In the second step, we construct a model that accounts for the time required to load, via DMA, various objects, such as vectors, matrix cache blocks, or a stream of nonzeros, into the local store of an SPE. The model must accurately reflect the constraints imposed by resource conflicts. For instance, a sequence of DMA's issued to multiple SPE's must be serialized, as there is only a single DRAM controller. The model also presumes a fixed DMA initiation latency of 1000 cycles based on existing design documents available regarding the Cell implementation. The model also presumes a broadcast mechanism that is similar to classic token ring network implementations where tokens are dispatched from the memory controller, travel along the ring as their data is copied by each SPE they pass, and are automatically removed once they have looped back to the memory controller.

The width of the SIMD units, the number of SPE's, and the local store size and bandwidth are encoded directly into the model as constants. External control is provided for parameters such as: the time for a SIMD reduction, the steady state time (per element) for a gather, a dot product or a SAXPY, the time to perform an inter SPE vector reduction, and the DMA initiation latency.

Our simulation framework is essentially a memory trace simulator – the difference being the complexity of the concurrent memory and computation operations that it must simulate. Instead of explicitly simulating computation using a cycle-accurate model of the functional units, we simulate the flow of data through the machine, and annotate the flow with execution time. The execution unit bandwidths and pipeline latencies are modeled as part of the data flow simulation. Therefore, our simulation is more sophisticated than a typical memory-trace simulator; however, it does not actually perform the computation. Additionally, instead of storing the matrix in a particular storage format and writing a program for each format, we store the key parameters for the matrix, and the nonzeros of the sparse matrix in an internal format, and write a performance annotator for each simulated storage format. This facilitated our work and allowed for detailed understanding of various characteristics of the sparse matrices.

Each algorithm was broken into a number of phases in which communication for the current objects and computation for the previous objects can take place simultaneously. Of course, for each phase, it was necessary to convert cycles into actual time and FLOP rates. For simplicity we chose to model a 3.2GHz, 8 SPE version of Cell with 25.6GB/s of memory bandwidth which is likely to be used in the first release of the Sony PlayStation 3 [19]. This ensured that both the EIB and DRAM controller could deliver two single precision words per cycle. The maximum flop rate of such a machine would be 204.8GFLOP/s, with a computational intensity of 32 FLOPs/word. It is unlikely that any version of Cell would have less memory bandwidth or run at a lower frequency.

For comparison, we use Cray's X1 MSP and Intel's Itanium2. The key characteristics of the processors are detailed in Table 1.

	CELL		X1(MSP)	Itanium2
	SPE	Chip		
Architecture	SIMD	multi-core	multi-chip Vector	VLIW
Frequency	3.2GHz	3.2GHz	800MHz	900MHz
DRAM BW	-	25.6GB/s	34GB/s	6.4GB/s
GFLOP/s (single)	25.6	204.8	25.6	3.6
GFLOP/s (double)	2.6	20.5	12.8	3.6
Local Store	256KB	2MB	-	-
L2 Cache	-	512KB	2MB	256KB
L3 Cache	-	-	-	1.5MB
Power	3W [1]	~30W	100W	130W

Table 1

Summary of architectural features of IBM's Cell [21], the Cray X1 MSP, and Intel's Itanium2 including single and double precision peak performance. A local store is not part of a cache hierarchy. Total Cell power is based on the active SPE's/idle PowerPC programming model.

4. Dense Matrix Matrix Multiplication

The first benchmark run, SGEMM, is a single precision dense matrix matrix multiply. With its extremely high computational intensity, one should expect Cell to achieve a high percentage of peak flop rate.

Two storage formats were explored. The default is a column major format for all three matrices. The second format, block data layout, or BDL, organizes matrix sub-blocks into contiguous blocks of memory [7]. This can be particularly advantageous as it not only minimizes the number of DMA's required, but also minimizes the number of pages touched when loading a sub-block. Although a matrix might not be stored in BDL, it can quickly be converted on the fly. Figure 2 shows a matrix stored in the two formats.

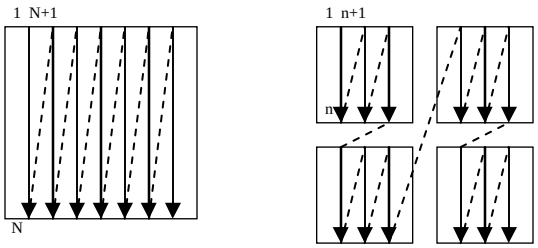


Figure 2

Left: column major layout. *Right*: BDL. Within each n x n block, values are stored in column major order

4.1 Algorithm Considerations

Each matrix was broken into square "cache" blocks. Technically they aren't cache blocks as the SPE's have no caches, but for clarity we will continue to use the terminology. A column layout will require a number of short (cache block dimension) DMA's equal to the dimension of the cache block – e.g. 64 DMA's of length 64. BDL will require a single DMA of length 16KB. The assumed 1000 cycles of DMA latency can result in poor memory bandwidth usage for small cache blocks.

The local store, being only 256KB, can't store more than about 56K words of data because the program and stack must also fit in the local store. Cache blocks, if double buffered, require $6*n^2$ words of local store (one cache block from each matrix). Thus it is impossible, with the current version of Cell, to utilize square cache blocks larger than 96x96. Additionally, in column layout, there is additional pressure on the maximum cache block size for large matrices as each column within a cache block will be on a different page. Upward pressure is derived from the computational intensity of matrix multiplication and the FLOPs to word ratio of the processor. In the middle, there is a cache block size which delivers peak performance.

The loop order in the column major storage format was chosen to minimize the average number of pages touched per phase. In BDL, as TLB misses are not nearly the problem, the loop order was chosen to minimize memory bandwidth.

A data parallel approach was chosen to disseminate work between the SPE's. Thus SPE cache blocks of size n x n are aggregated into Cell cache blocks of size 8n x n.

An alternate approach considered, although not presented here would be to adapt Cannon's algorithm [6] for parallel machines to a parallel machine on-chip. Although this could reduce the DRAM bandwidth requirements by transferring blocks via the EIB, for a column major layout, it could significantly increase the number of pages touched. Simulation based analysis will have to wait until a detailed model of the PowerPC TLB is available.

For small matrix sizes, it is most likely advantageous to choose a model that minimizes the number of DMA's. For example, broadcast both matrices, in their entirety, to all SPE's.

4.2 SGEMM Results

Figure 3 shows SGEMM performance for various matrix dimensions, cache block sizes, and storage formats. Clearly shown is that 32x32 cache blocks are far too small to achieve a computational intensity high enough to fully utilize the processor. The choice of loop order and the resulting increase in memory traffic prevents column major 64x64 blocks from achieving 90% of peak. One doesn't expect matrix multiplication to be memory bound, but for small cache blocks, it can be. Only 96x96 blocks provide enough computational intensity to overcome the additional block loads and stores.

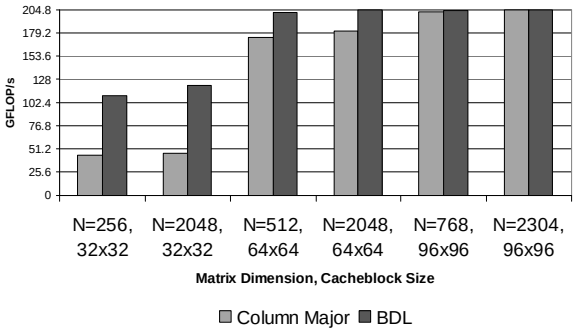


Figure 3

SGEMM on Cell. Even with the minimum overhead of BDL, the lack of computational intensity prevents 32x32 cache blocks from attaining 60% of peak. The inefficiency of column major layout prevents it from reaching peak performance without very large cache blocks.

It should be noted that if the DMA latency were 100 cycles, then the 64x64 column major performance would reach parity with BDL. This is certainly a motivation for user controlled DMA. Alternatively, if 128x64 blocks were used, then performance would also rival BDL.

Higher frequencies (e.g. 4GHz) will not help if peak performance is not being achieved due to high memory traffic, DMA latency, or TLB misses. Similarly,

higher bandwidth will go unused if the configuration is computationally bound.

Larger local stores, and thus larger cache blocks are not necessarily helpful in the column major layout as that would necessitate more pages being touched, and the likelihood of far more TLB misses. This is not an obstacle in the BDL approach.

4.3 DGEMM Results

A similar set of strategies and simulations were performed for DGEMM. Cache blocks are now limited to be no larger than 64x64 – finite local store. This is not the performance limitation it was in SGEMM. Although the time to load a double precision 64x64 cache block is twice that of a single precision version, the time required to compute on a 64x64 double precision cache block is about ten times (no more fully pipelined SIMD) as long as the single precision counterpart. Thus it is far easier for double precision to reach its peak performance. Of course peak double precision performance is one tenth of single precision – a mere 20 GFLOP/s.

4.4 Comparison

At 3.2GHz, each SPE requires about 3W [1]. Thus with a nearly idle PPC and L2, Cell achieves over 200GFLOP/s for around 30W – nearly 7GFLOP/s/W. Clearly Cell is highly efficient at large matrix multiplication

DGEMM and SGEMM were also run on Cray's X1, and a 900MHz Itanium2. The results are detailed in the Table 2.

	Double (GFLOP/s)			Single (GFLOP/s)		
	Cell	**X1**	**Itanium2**	**Cell**	**X1**	**Itanium2**
Peak	20.4	11.2	3.5	204.7	16.4	3.6

Table 2

Peak GEMM performance (in GFLOP/s) for large square matrices on Cell, X1, and the Itanium2. With about ¼ the power, Cell is nearly 6 times faster in double precision and more than 50 times faster in single precision than the Itanium2.

The 900MHz Itanium2 is capable of running the Intel MKL DGEMM at 3.5GFLOP/s. Although this number is an impressively high percentage of peak, the architecture is not power efficient, and scaling to multiprocessors exacerbates the problem. For example a 4 processor 1.5GHz Itanium2 system will consume well over 500W, and yet only deliver about 22GFLOP/s. Contrast this with a single Cell processor which consumes less than $1/15^{th}$ the power and provides about the same performance in double precision – by no means Cell's forte. The Itanium2 does not have SIMD support. Thus, in single precision, Cell is more than 100 times more power efficient. Similarly, in double precision, Cell is about twice as fast as the X1, and at least 6 times more power efficient. In single precision, Cell is nearly 40 times more power efficient.

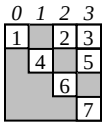
The primary focus for matrix multiplication on Cell is the choice of data storage to minimize the number of DMA's and TLB misses while maximizing computational intensity. Secondary is the choice of programming model. The decoupling of main memory data access from the computational kernel guarantees constant memory access latency since there will be no cache misses, and all TLB accesses are resolved in the communication phase.

Matrix multiplication is perhaps the best benchmark to demonstrate Cell's computational capabilities as it achieves high performance by buffering large cache blocks on chip before computing on them

5. Sparse Matrix Vector Multiply

Naïvely, SpMV would seem to be the worst application to run on Cell since the SPE's have neither caches nor gather/scatter support. Furthermore, SpMV has O(1) computational intensity. However, these are perhaps less important than the low functional units and local store latency (<2ns), the task parallelism afforded by the SPE's, the eight independent load store units, and ability to stream nonzeros via DMA's.

Three storage formats were examined: compressed sparse row (CSR), compressed sparse column (CSC), and blocked compressed sparse row (BCSR). CSR collects the nonzeros from one row at a time and appends three arrays: the values, the corresponding columns for the values, and the locations in the first two arrays where the row starts. BCSR behaves in much the same way as CSR. The difference is that CSR operates on what are in effect 1x1 blocks, and BCSR operates on r x c blocks. Thus the values array is grouped into r*c segments which include zeros. CSC is organized around columns rather than rows.



CSR
Values = {1,2,3,4,5,6,7}
Columns = {0,2,3,1,3,2,3}
RowStart = {0,3,5,6,7}

BCSR (2x2)
Values = {1,0,0,4, 2,3,0,5, 6,0,0,7}
Columns = {0, 2, 2 }
RowStart = {0.8.12}

Figure 4

A 4x4 matrix with columns numbered from 0 to 3 is shown stored in 1x1 BCSR (CSR), and 2x2 BCSR. CSC would look similar to CSR except that it is organized along columns rather than rows.

All three storage formats provide regular access patterns to the nonzeros. However, CSR and CSC force a very irregular access pattern to the source and destination vectors respectively. For SIMD sized granularities BCSR provides regular access within a block, but requires irregular accesses outside. BCSR also has the pitfall that zeros are both loaded and computed on. Only the 2x2 BCSR data will be shown as the 4x4 blocks showed poor

performance. Figure 4 provides an example matrix and the corresponding data structures used in CSR and BCSR.

A CSR/BCSR pseudocode overview can be illustrative. In CSR, *Y[r]*, values*[i]*, and *X[columns[i]]* are all scalars. In BCSR, *Y[r]*, and *X[columns[i]]* now are segments of the vectors, and the *values[i]* are blocks. The *X[columns[i]]* statement is referred to as a gather operation. CSR performs a dot product for each row.

```
for all rows r
  for all elements i in row r
    Y[r] = Y[r] + values[i]*X[columns[i]]
```

For completeness, the following is pseudo code for CSC.

```
for all columns c
  for all elements i in column c
    Y[rows[i]] = Y[rows[i]] + values[i]*X[c]
```

CSC performs a SAXPY for each column. The write to Y is a scatter operation. Thus there is a dependency from the gather to the scatter, and there is a potential dependency from the scatter for one column to the gather on the next.

5.1 Algorithm Considerations

As there are no gather/scatter DMA operations in Cell, cache blocking must be utilized. Once again, to be clear, the term cache blocking, when applied to Cell, implies that blocks of data, in this case the vectors, will be loaded in the SPE's local stores. For simplicity all benchmarks were run using square cache blocks. The data structure required to store the entire matrix is a 2D array of cache blocks. Each cache block stores its nonzeros and row pointers as if it were an entire matrix. This results in more row pointers being loaded and substantial overhead. Cache blocks were not double buffered as this would require more local store, or more precisely smaller cache blocks. This is an area for future exploration. Collectively the cache blocks were chosen to be no larger than ~36K words (half that in double precision).

It should be noted that since the local store is not a write back cache, it is possible to overwrite its contents without fear of either consuming DRAM bandwidth or corrupting the actual arrays. The objects in the local store are more appropriately copies of the objects in DRAM. In CSR it is possible to decouple the gather operations from the dot products because there is only a read after read hazard from the gather operation on one row to the gather on the next row. The contents of the index array, or more appropriately, the copy of the index array, can be overwritten with their corresponding source vector values. This decoupled gather operation can be fully unrolled and software pipelined, thereby being performed in close to 1.5 cycles/element. If the gather remained coupled to the dot product, then without unrolling, the local store latency would be exposed and performance may suffer.

The gather/scatter operations in CSC cannot be decoupled from the SAXPY's since there is a potential read after write dependency from the scatter of one column to the gather in the next column. This decoupled approach provides CSR with a significant advantage over CSC. As a result, we expect that CSC will never outperform CSR; at best it will provide the same performance (memory bound). Thus CSC results will not be presented in this paper.

It should be noted that it is possible to unroll the longer dot products or SAXPY's. This will provide higher performance only if SpMV is not memory bound. Beyond SIMD, this optimization was not explored in this paper.

As the nonzeros are stored in arrays, it is easy to stream them in via DMA. Here it is essential to double buffer to ensure that the SPE's always have work to do. Buffering 16KB was sufficient. Thus for CSR, this is 2K values and 2K indices. For BCSR, this is at about 1K tiles. For each phase, a load of nonzeros and indices, there is the omnipresent 1000 cycle DMA latency.

The biggest question was how to partition work between SPE's. By allowing all SPE's work on the same cache block, it is possible to broadcast the cache blocked source vector and row pointers to minimize memory traffic. One approach to divide work within a cache block would be to more or less evenly divide the nonzeros between the SPE's. Of course this not only necessitates each SPE have a private copy of the destination vector, but that an inter-SPE reduction be performed at the end of the blocked row. We call this approach PrivateY. The alternate method, which we call PartitionedY, partitions the destination vector evenly among the SPE's. By reducing the size of the destination vector within each SPE, one can double the size of the source vector "cached" within the local store. However there is no longer any guarantee that the SPE's computations will remain balanced. The most loaded SPE determines the execution time for the entire cache block. Thus for balanced cache blocks, the PartitionedY usually wins out. However for unbalanced cache blocks, not an uncommon occurrence, the PrivateY approach wins.

The algorithm proceeds through the matrix one blocked row at a time. The height of the blocked row is equal to the cache block height. Thus it is possible to save the write to the destination vector, and a possible reduction until the end of the blocked row. The blocked row is divided into cache blocks of equal height and width. For each cache block the source vector and row pointers are loaded. Once that is completed, blocks of nonzeros are streamed to the SPE's and partitioned according to the partitioning strategy. Each block is processed once it has been received, and at the same time as the next block is being sent. Thus for each cache block's execution, there is a startup penalty (to transfer the first block of nonzeros) where no processing takes place, and a finish penalty (to operate on the last block of nonzeros), where no data transfer is taking place. Usually these overheads, as well as the source vector load time,

destination vector reduction and destination store time, are small compared to the total time.

It should be noted that there could be some benefit by writing a kernel optimized for a symmetric matrix. The nonzero memory traffic does not increase, but the number of operations can double. One must, however, cache block two blocks at the same time. Thus the symmetric kernel divides memory allocated for cache blocking the vectors evenly among the two sub-matrices, and for each row in the lower triangle, performs a dot product and a SAXPY.

5.2 Evaluation Matrices

In order to evaluate SpMV performance, six synthetic matrices, four unsymmetric matrices from the SPARSITY matrix suite [3, 5], and six symmetric matrices, also from the SPARSITY suite were run. Their characteristics are summarized in the Table 3.

	Name	N	NNZ	Comments
-	7pt_32	32K	227K	3D 7pt stencil on a 32^3 grid
-	Random	32K	512K	Totally random matrix
-	Random (symmetric)	32K	256K	Random Symmetric matrix – Total of 512K nonzeros
-	7pt_64	256K	1.8M	3D 7pt stencil on a 64^3 grid
-	Random	256K	4M	Totally random matrix
-	Random (symmetric)	256K	2M	Random Symmetric matrix – Total of 4M nonzeros
15	Vavasis	40K	1.6M	2D PDE Problem
17	FEM	22K	1M	Fluid Mechanics Problem
18	Memory	17K	125K	Memory Circuit from Motorola
36	CFD	75K	325K	Navier-Stokes, viscous flow, fully coupled
06	FEM Crystal	14K	490K	FEM Crystal free vibration stiffness matrix
09	3D Pressure	45K	1.6M	3D pressure Tube
25	Portfolio	74K	335K	Financial Portfolio - 512 Scenarios
27	NASA	36K	180K	PWT NASA Matrix with diagonal
28	Vibroacoustic	12K	177K	Flexible box, structure only
40	Linear Prog.	31K	1M	AA^T

Table 3 – Benchmark matrices used for SpMV
Number refers to the matrix number in the SPARSITY suite used by the Bebop group for benchmarking

5.3 Single Precision SpMV Results

The results of single precision performance estimation are detailed in Figures 5 and 6. Surprisingly, given Cell's SpMV limitations, every matrix achieves impressive multi-GFLOP performance for nearly every configuration. Unfortunately, many of the matrices are so small that they fully utilize only a fraction of a cache block.

It was clear that the performance is almost entirely determined by the memory bandwidth. It is not possible, for many matrices, to achieve the 6.4GFLOP/s CSR unsymmetric peak performance since there can be substantial cache blocking and DMA overhead. As one might expect, large matrices with high densities (to amortize the cache blocking overhead) show closer to peak performance. Similarly, larger cache blocks yield higher performance for large matrices.

Unlike the synthetic matrices, the library matrices, which contain dense sub-blocks, can exploit BCSR without wasting memory bandwidth on zeros. As memory traffic is key, storing BCSR blocks in a compressed format (the zeros are neither stored nor loaded) could allow significantly higher performance if there is sufficient support within the ISA to either decompress these blocks on the fly, or compute on compressed blocks. Since detailed knowledge of the permute datapath is currently unavailable, this exploration will have to wait.

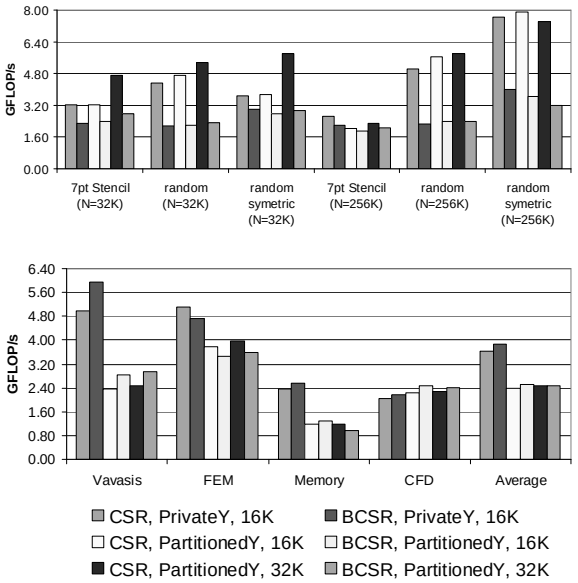


Figure 5
Top: Single precision SpMV using synthetic matrices – clear benefits from density and uniformity. Bottom: using SPARSITY unsymmetric matrices – PrivateY shows superior performance due to unbalance.

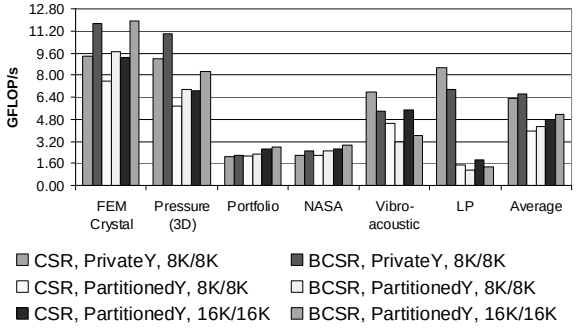


Figure 6
Single precision SpMV using SPARSITY symmetric matrices – Significant performance boost from minimization of nonzero traffic. Each of the cache blocks is half as big. Imbalance in PartitionedY strategy can generate serious performance degradation.

The choice of a partitioning strategy is pretty clear. PrivateY is almost invariably the better approach. Most likely, the matrices are sufficiently unbalanced that the uniform partitioning of the nonzeros coupled with a reduction requires less time.

Since the local store size is fixed, cache blocks in the symmetric kernels are in effect half the size of the space allocated. The symmetric kernel, when in the PartitionedY configuration, is extremely unbalanced for cache blocks along the diagonal. Thus, for small matrices, imbalance between SPE's, even if the matrix is uniform, can severely impair the performance. Figure 7, which assumes a Cell configuration with 4 SPE's, might help visualize the inherent flaw in the symmetric kernel as implemented. In fact, symmetric optimizations show only about 50% performance improvement (determined by running the symmetric matrices on the unsymmetric kernel).

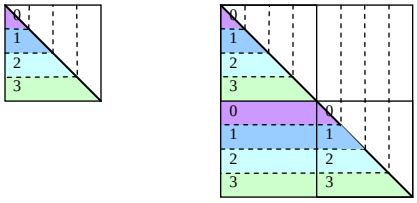


Figure 7

Left: small matrix with single cache block. If the matrix is uniform, and the PartitionedY strategy is used, then SPE3 performs 7 times as much work as SPE0. *Right*: larger matrix. Overall, SPE3 now performs a little over twice the work of SPE0. As on-diagonal cache blocks become the minority (large matrices), their inherent imbalance ceases to adversely affect performance.

The TLB plays a much smaller role in this algorithm than in GEMM. Accesses are unit stride, and there is rarely more than 16 DMA's per phase. Each of these DMA's is of order a page size.

Once again DMA latency plays a relatively small role in this algorithm. In fact, reducing the DMA latency by a factor of ten results in only a 10% increase in performance. This is actually a good result. It means than the memory bandwidth is highly utilized. The majority of bus cycles are used for transferring data rather than stalls.

On the whole, clock frequency also plays a small part in the overall performance. Increasing the clock frequency by a factor of 2 (to 6.4GHz) provides less than a 5% increase in performance on the SPARSITY unsymmetric matrix suite. Similarly, cutting the frequency in half (to 1.6GHz) results in less than a 10% decrease in performance. One might wonder if the dot product times are too aggressive. Cutting the frequency in half could just as easily be interpreted as a doubling in dot product time – even if we were off by a factor of two, its only a 10% difference in overall performance. Simply put, for the common case, more time is used in transferring nonzeros and the vectors rather than computing on them. This means that solely doubling bandwidth will not necessarily double performance. In fact, in order to double performance, doubling bandwidth must be coupled with an amortization of DMA latency. Similarly, an increase in bandwidth efficiency – i.e. a reduction in meta data, is only useful if it is accompanied by an increase in parallelism.

5.4 Double Precision SpMV Results

Preliminary analysis indicates that single precision SpMV is nearly twice as fast as double precision on the Cell architecture. Initially this can be surprising since memory traffic should only increase by 50%. A single precision nonzero requires 8 bytes (a 32b value and a 32b index). In double precision, a nonzero requires 12 bytes – the index remains 32 bits. The biggest problem is the reduction in the number of values that are cache blocked. Twice as many cache blocks within a blocked row must be loaded and twice as many blocked rows are present. For example, consider 16K x 16K single precision cache blocks on a 128K x 128K matrix. The 512KB source vector must be loaded 8 times. In double precision, the cache blocks are only 8K x 8K. As a result, the 1MB source vector must be loaded 16 times. Thus far more memory bandwidth can be consumed on cache blocking.

5.5 Comparison

Results are compared with the SPARSITY suite, a highly tuned sparse matrix numerical library [3]. The previously documented optimum performance on a 900MHz Itanium2 processor, in addition to Cell's estimated performance behavior, is detailed in Tables 4 and 5.

	Double (GFLOP/s)		Single (GFLOP/s)	
Matrix	**Cell**	**Itanium2**	**Cell**	**Itanium2**
Vavasis	3.12	0.51	5.95	0.52
FEM	3.40	0.54	5.09	0.63
CFD	2.02	0.25	2.48	0.15
Average	**2.85**	**0.43**	**4.51**	**0.43**

Table 4

Peak SpMV performance (in GFLOP/s) of Cell and Itanium2 for both double and single precision on the SPARSITY unsymmetric matrix suite. Even in double precision, Cell is about seven times faster (with only four times the memory bandwidth).

	Double (GFLOP/s)		Single (GFLOP/s)	
Matrix	**Cell**	**Itanium2**	**Cell**	**Itanium2**
FEM Crystal	6.23	0.74	11.98	1.21
3D Pressure	5.97	0.72	11.02	1.24
Protfolio	1.63	0.23	2.77	0.19
NASA	1.76	0.27	2.91	0.22
Vibroacoustic	3.28	0.31	6.78	0.41
Linear Prog.	4.81	0.33	8.52	0.66
Average	**3.95**	**0.43**	**7.33**	**0.66**

Table 5

Peak SpMV performance (in GFLOP/s) of Cell and Itanium2 for both double and single precision on the SPARSITY symmetric matrix suite. Cell is about 10 times faster (with only four times the memory bandwidth).

With the Itanium2's 6.4GB/s bus, one would expect that a memory bound application like SpMV would perform only four times better on Cell, whose DRAM bandwidth is 25.6GB/s. Nevertheless, on average, Cell is more than seven times faster. In single precision, Cell is more than 10 times faster. Actually this is not that surprising. In order to achieve peak performance, Itanium2 must rely on BCSR and thus waste memory bandwidth loading zeros. For example, in matrix #17, Cell uses more than 50% of its bandwidth loading just the double precision nonzero values, while the Itanium2 utilizes only 33% of its bandwidth. The rest of Itanium's bandwidth is used for zeros and meta data. Cell's cache blocking is far more efficient in single precision since cache blocks double in size in both dimensions. It should be noted that where simulations on Cell involve a cold start to the local store, the Itanium2's have the additional advantage of a warm cache.

Cell's use of on-chip memory as a buffer is advantageous in both power and area than a cache. In fact, Cell is more than 20 times more power efficient than the Itanium2 on SpMV.

Comparing results with the X1, even with the permutation optimization (CSRP), an X1 MSP achieves only about 1 GFLOP/s on a 7pt stencil [4]. Standard CSR achieves less than 0.01 GFLOP/s. On a similar matrix, in double precision, Cell is able to achieve about 1.77GFLOP/s. Although the X1 has 50% more memory bandwidth, it is only a little better than half the performance of Cell.

Although it is true that cell achieves a dismally low percentage of peak flop rate (less than 5%), it is using a high percentage of memory bandwidth. For completion, a search of the entire BCSR space is required to find the true optimum performance levels.

6. Conclusions

The high performance computing community is exploring alternative architectural approaches to address the limitations of modern superscalar designs. This work presents the first attempt to explore the behavior of scientific kernels on the forthcoming Cell processor's novel architecture. Since neither Cell hardware nor cycle-accurate simulators are currently publicly available at this time, we develop an analytic framework to predict Cell performance on dense and sparse matrix operations, using a variety of algorithmic approaches. Results, compared with the Intel Itanium2 and Cray X1 processors, indicate the tremendous potential of the Cell architecture, both in terms of raw performance and power advantages.

Analysis shows that Cell's three level memory architecture, which completely decouples main memory load/store from computation, provides several advantages over mainstream cache-based architectures. First, kernel performance can be extremely predictable as the average load time from local store is also the worst case. Second, long block transfers can achieve a much higher percentage of memory bandwidth than individual loads in much the same way a prefetch engine, once engaged, can fully consume memory bandwidth. Finally, for predictable memory access patterns, communication and computation can be effectively overlapped. Increasing the size of the local store or reducing the DMA startup overhead on future Cell implementations may further enhance the scheduling efficiency in order to better overlap the communication and computation.

There are disadvantages to this architecture. Although GEMM is highly predictable, and inherently has a high computational intensity, SpMV, with its unpredictable access patterns and low computational intensity achieves a dismally low percentage of peak performance. Even memory bandwidth can be wasted since SpMV is constrained to use cache blocking to remove the unpredictable accesses to the source vector. The ability, however, to perform a decoupled gather, to stream nonzeros, and Cell's low functional unit latency, tends to hide this deficiency.

For dense matrix operations, it is essential to maximize computational intensity and thereby fully utilize the local store. However, if not done properly, this can result in TLB misses adversely affecting performance. BDL data storage, either created on the fly or before hand, can ensure that TLB misses remain a small issue as on-chip memories increase in size.

7. Future Work

A key component missing in this work is cycle-accurate simulation of the Cell architecture. We expect to work on validating the models that we have been using to predict Cell performance using a suite of high level architectural simulators that are due to be released by IBM Research in late July. We will report those results in this paper if the software release proceeds as scheduled. The simulation results will also be checked against runs on Cell-based workstations when they become available.

We are also actively expanding our study of the Cell architecture's applicability to science to include stencil-based computations, FFT's for spectral methods, and even pipelined mapping of large arithmetically intense loops.

Cell will not reach its true potential for scientific computing until an implementation that includes at least one (preferably two) fully pipelined double precision floating point unit becomes available. Until then, studies of Cell may provide insights into enhancements that may prove useful for mainstream desktop processor implementations or even a variant of the Cell processor that includes other HPC-oriented features.

References

[1] B. Flachs et al., A Streaming Processor Unit for a Cell Processor, *ISSCC Dig. Tech. Papers*, Paper 7.4, 134-135, February, 2005.

[2] D. Pham et al., The Design and Implementation of a First-Generation Cell Processor, *ISSCC Dig. Tech. Papers*, Paper 10.2, 184-185, February, 2005.

[3] R. W. Vuduc. Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley, 2003.

[4] E. F. D'Azevedo, M. R. Fahey, R. T. Mills. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. *ICCS*, 99-106, 2005

[5] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 2004.

[6] L. Cannon. A Cellular Computer to Implement the Kalman Filter Algorithm. PhD thesis, Montana State University, 1969.

[7] N. Park, B. Hong, and V. K. Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout, *International Conference on Parallel Processing (ICPP)*, August 2002.

[8] M. Oka, et al. Designing and programming the emotion engine. *Micro, IEEE*, Volume: 19, Issue: 6, Nov.-Dec. 1999

[9] A. Kunimatsu, et al. Vector Unit Architecture for Emotion Synthesis. *Micro, IEEE*, Volume: 20, Issue: 2, March-April 2000.

[10] M. Suzuoki, et al. A Microprocessor with a 128-Bit CPU, Ten Floating-Point MAC's, Four Floating-Point Dividers, and an MPEG-2 Decoder. *Solid-State Circuits, IEEE Journal*, Volume: 34, Issue: 11, November 1999.

[11] B. Khailany, et al. Imagine: Media Processing with Streams. *Micro, IEEE*, Volume: 21, Issue: 2, March-April 2001

[12] M. Kondo, et al. SCIMA: A Novel Processor Architecture for High Performance Computing. *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, Volume: 1, 14-17 May 2000.

[13] P. Keltcher, et al. An Equal Area Comparison of Embedded DRAM and SRAM Memory Architectures for a Chip Multiprocessor. HP Laboratories Palo Alto. April 2000.

[14] The Berkeley Intelligent RAM (IRAM) Project, Univ. of California, Berkeley, at http://iram.cs.berkeley.edu.

[15] S. Tomar, et al. Use of Local Memory for Efficient Java Execution. Computer Design, 2001. *ICCD*. Proceedings. 23-26 September 2001.

[16] M. Kandemir, et al. Dynamic Management of Scratch-Pad Memory Space. *Design Automation Conference*. Proceedings, 18-22 June 2001.

[17] P. Francesco, et al. An Integrated Hardware/Software Approach For Run-Time Scratchpad Management. *41st Design Automation Conference*. Proceedings, June 7-11, 2004.

[18] ORNL Cray X1 Evaluation. http://www.csm.ornl.gov/~dunigan/cray.

[19] Sony press release http://www.scei.co.jp/corporate/release/pdf/050517e.pdf

[20] S. Mueller, et al. The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor.. *17th IEEE annual Symposium on Computer Arithmetic*. June 27-29, 2005. (to appear)

[21] IBM Cell Specifications http://www.research.ibm.com/cell/home.html