DISCLAIMER

The Potential of the Cell Processor for Scientific Computing

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

The slowing pace of commodity microprocessor performance improvements combined with ever-increasing chip power demands has become of utmost concern to computational scientists. As a result, the high performance computing community is examining alternative architectures that address the limitations of modern cache-based designs. In this work, we examine the potential of the using the forthcoming STI Cell processor as a building block for future high-end computing systems. Our work contains several novel contributions. We are the first to present quantitative Cell performance data on scientific kernels and show direct comparisons against leading superscalar (AMD Opteron), VLIW (Intel Itanium2), and vector (Cray X1) architectures. Since neither Cell hardware nor cycle-accurate simulators are currently publicly available, we develop both analytical models and simulators to predict kernel performance. Our work also explores the complexity of mapping several important scientific algorithms onto the Cell's unique architecture. Additionally, we propose modest microarchitectural modifications that could significantly increase the efficiency of double-precision calculations. Overall results demonstrate the tremendous potential of the Cell architecture for scientific computations in terms of both raw performance and power efficiency.

Keywords
Cell, GEMM, SpMV, sparse matrix, FFT, Stencil, three level memory

1. Introduction

Over the last decade the HPC community has moved towards machines built on commodity microprocessors as a strategy for tracking the tremendous growth in processor performance in that market. As this pace slows, and the power requirements of these processors continues to grow, the HPC community is looking for alternative architectures that provide high performance on scientific applications, yet have a healthy market outside the scientific community. In this work, we examine the potential of the forthcoming STI Cell processor as a building block for future high-end computing systems, by investigating performance across several key scientific computing kernels: dense matrix multiply, sparse matrix vector multiply, stencil computations on regular grids, as well as 1D and 2D FFTs.

Cell is a high-performance implementation of software-controlled memory hierarchy in conjunction with the considerable floating point resources that are required for demanding numerical algorithms. Despite its radical departure from mainstream/commodity processor design, Cell is particularly compelling because it will be produced at such high volumes that it will be cost-competitive with commodity CPUs. The current implementation of Cell is most often noted for its extremely high performance single-precision (SP) arithmetic, which is widely considered insufficient for the majority of scientific applications. Although Cell's peak double precision performance is still impressive relative to its commodity peers *(~14.6GFLOP/s@3.2GHz)*, we explore how modest hardware changes could significantly improve performance for computationally intensive DP applications.

This paper presents several novel results. We present quantitative performance data for scientific kernels that compares Cell performance to leading superscalar (AMD Opteron), VLIW (Intel Itanium2), and vector (Cray X1) architectures. We believe this is the first published analysis of its kind. Since neither Cell hardware nor cycle-accurate simulators are currently publicly available, we develop both

analytical models and simulators to predict kernel performance. Our work also explores the complexity of mapping several important scientific algorithms onto the Cell's unique architecture in order to leverage the large number of available functional units and software controlled memory architecture. Additionally, we propose modest microarchitectural modifications that could increase the efficiency of double-precision arithmetic calculations, and demonstrate significant performance improvements compared with the current Cell implementation.

Overall results demonstrate the tremendous potential of the Cell architecture for scientific computations in terms of both raw performance and power efficiency. We also conclude that Cell's heterogeneous multi-core implementation is inherently better suited to the HPC environment than homogeneous commodity multi-cores.

2. Related Work

One of the key limiting factors for computational performance is off-chip memory bandwidth. Since increasing the off-chip bandwidth is prohibitively expensive, many architects are considering ways of using available bandwidth more efficiently. Examples include hardware multithreading or more efficient alternatives to conventional cache-based architectures such as software controlled memories. 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 – but do so with far less architectural complexity than conventional cache architectures. While placing data movement under explicit software control increases the complexity of the programming model, prior research has demonstrated that this approach can be more effective for hiding memory latencies (including cache misses and TLB misses) – requiring far smaller cache sizes to match the performance of conventional cache implementations [12, 13].

Over the last five years, a plethora of alternatives to conventional cache-based architectures have been suggested including scratchpad memories [15, 16, 17], paged on-chip memories [13, 14], and three level memory architectures [11, 12]. 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 achieved high performance at low power by adopting the three (registers, local memory, external DRAM) level memory architecture. More recently, the STI Cell processor has adopted a similar approach where data movement between these three address spaces is explicitly controlled by the application. This more aggressive approach to memory architecture was adopted to meet the demanding cost/performance requirements of Sony's upcoming video game console. However, to date, an in-depth study to evaluate the potential of utilizing the Cell architecture in the context of scientific computations does not appear in the literature.

3. Cell Background

Cell [1, 2] was designed by a partnership of Sony, Toshiba, and IBM (STI) to be the heart of Sony's forthcoming PlayStation3 gaming system. Cell takes a radical departure from conventional multiprocessor or multi-core architectures. Instead of using identical cooperating commodity processors, it uses a conventional high performance PowerPC core that controls eight simple SIMD cores, called synergistic processing elements (SPEs), where each SPE contains a synergistic processing unit (SPU) and a local memory. An overview of Cell is provided in Figure 1.

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 direct memory access (DMA) of SPE program and data from DRAM to the local store. Once the DMAs 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.

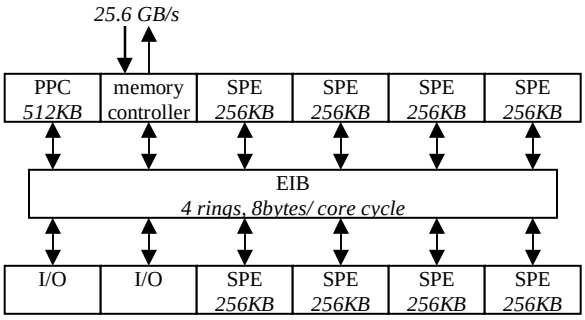


Figure 1 – Overview of the Cell processor. Eight SPEs, one PowerPC core, one memory controller, and two I/O controllers are connected via four rings. Each ring is 128b wide and runs at half the core frequency. Each SPE has its own local memory from which it runs programs.

Access to external memory is handled via a 25.6GB/s XDR memory controller. The PowerPC core, the eight SPEs, 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 (SP) 6-cycle pipelined FMA datapaths and one double precision (DP) half-pumped (one SIMD instruction every other cycle) 9-cycle pipelined FMA datapath with 4 cycles of overhead for data movement [20]. Cell has a 7 cycle in-order execution pipeline and forwarding network [1]. IBM appears to have solved the problem of inserting a 13 (9+4) cycle DP pipeline into a 7 stage in-order machine by choosing the minimum effort/performance/power solution of simply stalling for 6 cycles after issuing a DP instruction. The SPU's DP throughput [29] of one DP instruction every 7 (1 issue + 6 stall) cycles coincides perfectly with this reasoning.

Thus for computationally intense algorithms like dense matrix multiply (GEMM), we expect SP implementations to run near peak whereas DP versions would drop to approximately one fourteenth the peak SP flop rate[21]. Similarly, for bandwidth intensive applications such as sparse matrix vector multiplication (SpMV) we expect SP versions to be between 1.5x and 4x as fast as DP, depending on density and uniformity.

With respect to the memory subsystem, all loads and stores issued from the SPE can only access the SPE's local memory. The limited scope of loads and stores allows one to view the SPE as having a two-level register file. The first level is a 128 x 128b single cycle register file, where the second is a 16K x 128b six cycle register file. Data must be moved into the first level before it can be operated on by instructions.

The Cell processor depends on explicit DMA operations to move data from main memory to the local store of the SPE. Whereas scalar processors have byte, word, half and doubleword loads, the SPEs have selectable length DMAs that run in parallel (via DMA engines) with the SIMD code. Thus an SPE has the capability of mitigating memory latency overhead via double-buffered DMA loads and stores. At the SPE level, most algorithms are programmed much the same way as they are on traditional architectures. The SPE's DMAs are simply a condensed version of a stream of scalar loads. In fact, they are much like a traditional unit stride vector load – the major difference being that they do not suffer the performance issues associated with a hardware encoded vector length. We exploit these similarities to existing HPC platforms to select programming models that are both familiar and tractable for scientific application developers.

4. Programming Models

Moving from a hardware managed memory hierarchy to one controlled explicitly by the application significantly complicates the programming model. Our goal is to select the programming paradigm that offers the simplest possible expression of an algorithm while being capable of fully utilizing the hardware resources of the Cell processor.

The candidate programming models for Cell can be divided into three categories: independent SPEs, data pipelined, and lock step data parallel. Heterogeneous programming of heterogeneous elements is atypical of applications

in the scientific community so the independent SPE model was not pursued vigorously. Data pipelining, where large blocks of data are streamed from one SPE to the next, may be suitable for certain classes of algorithms and will be the focus of future investigation. The data-parallel programming model is well established in the computational sciences and offers the simplest and most direct method of decomposing the problem, and thus is the focus of our investigation.

Data-parallel programming is quite 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 and thus slight variations were employed.

5. 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 and/or Cell hardware, we will verify our results and gain understanding of the processor features that limit performance.

In the first step, pencil and paper calculations were performed to estimate the computational and memory requirements for the kernels of the four benchmarks. The kernels were segmented into code-snippets that operate only on data present in the local store of the SPE. The performance estimation was further refined by hand-coding SPE assembly instructions for the code snippets and calculating the execution times for those snippets – taking into account the latency of each operation, and the operand alignment requirements given the SIMD/quadword nature of the SPE execution pipeline. The execution times for these snippets are parameters for the execution component of the performance estimator.

In the second step, we construct a model that tabulates the time required for DMA loads and stores of the operands required by the code snippets. The model must accurately reflect the constraints imposed by resource conflicts in the memory subsystem. For instance, a sequence of DMAs issued to multiple SPEs 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 either a snooping or a broadcast mechanism was implemented in the EIB either in hardware or via software emulation.

The number of SPEs, the local store size and the bandwidths are encoded directly into the model as constants. External control is provided for parameters such as 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. Therefore, our simulation is more sophisticated than a typical memory-trace simulator; however, although it should accurately model execution time, it does not actually perform the computation.

	CELL		X1(MSP)	Opteron	Itanium2
	SPE	Chip			
Architecture	SIMD	multi-core SIMD	multi-chip Vector	Super Scalar	VLIW
Frequency	3.2GHz	3.2GHz	800MHz	2.2GHz	900MHz
DRAM BW	-	25.6GB/s	34GB/s	6.4GB/s	6.4GB/s
GFlop/s(single)	25.6	204.8	25.6	8.8	3.6
GFlop/s(double)	1.83	14.63	12.8	4.4	3.6
Local Store	256KB	2MB	-	-	-
L2 Cache	-	512KB	2MB	1MB	256KB
L3 Cache	-	-	-	-	1.5MB
Power	3W [1]	~40W	100W	89W	130W

Table 1 - Architectural overview of STI Cell [21], Cray X1 MSP, AMD Opteron, and Intel Itanium2. Total Cell power and peak GFlop/s are based on the active SPEs/idle PowerPC programming model.

Algorithms that employed double-buffering were 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. This version of Cell is likely to be used in the first release of the Sony PlayStation3 [19]. The lower frequency had the simplifying benefit that both the EIB and DRAM controller could deliver two SP 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 examine performance on several leading processor designs: the vector Cray X1 MSP, superscalar AMD Opteron 248 and VLIW Intel Itanium2. The key architectural characteristics are detailed in Table 1.

5.1 Cell+ Architectural Exploration

In order to explore the limitations of Cell's DP issue bandwidth, we propose an alternate design with a longer forwarding network. In this hypothetical implementation, called Cell+, each SPE would still have the single DP datapath, but would be able to dispatch one DP SIMD instruction every other cycle instead of one every 7 cycles. The Cell+ design would achieve 3.5x the DP throughput of the Cell (51.2 GFlop/s) by fully utilizing the existing DP datapath; however, it would maintain the same SP throughput, frequency, bandwidth, and power as the Cell. Based on our experience designing the VIRAM vector processor-in-memory chip [14], we believe the Cell+ design modifications are modest. The Cell+ design would require a tiny increase in transistor count, while having a potentially significant impact on DP application performance.

6. Dense Matrix-Matrix Multiply

We begin by examining the performance of dense matrix-matrix multiplication, or GEMM. This kernel is characterized by high computational intensity and regular memory access patterns, making it a extremely well suited for the Cell architecture. We explored two storage formats: column major and block data layout [7] (BDL). BDL is a two-stage addressing scheme (block row/column, element sub row/column) detailed in appendix A.

6.1 Algorithm Considerations

For GEMM, we adopt what is in essence an outer loop parallelization approach. Each matrix is broken into 8n x n Cell "cache" blocks, which in turn are split into eight n x n SPE "cache" blocks. Technically they aren't cache blocks as the SPEs have no caches, but for clarity we will continue to use the terminology. For the column layout, the matrix will be accessed via a number of short DMAs equal to the dimension of the cache block – e.g. 64 DMAs of length 64. BDL, on the other hand, will require a single long DMA of length 16KB.

Since the local store is only 256KB, and must contain both the program and stack, program data in the local store is limited to about 56K words. The cache blocks, when double buffered, require $6n^2$ words of local store (one from each matrix) – thus making 96^2 the maximum square cache block in SP. Additionally, in column layout, there is added pressure on the maximum cache block size for large matrices, as each column within a cache block will be on a different page resulting in TLB misses. The minimum size of a cache block is determined by the FLOPs to word ratio of the processor. In the middle, there is a cache block "sweet spot" that delivers peak performance.

The loop order was therefore chosen to minimize the average number of pages touched per phase for a column major storage format. The BDL approach, as TLB misses are of little concern, allows us to structure the loop order such that memory bandwidth is minimized.

A possible alternate approach is to adapt Cannon's algorithm [6] for parallel machines. Although this strategy 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. This will be the subject of future work.

Note that for small matrix sizes, it is most likely advantageous to choose a model that minimizes the number of DMAs. One such solution would be to broadcast a copy of the first matrix to all SPEs.

6.2 Single Precision GEMM Results

Cell GEMM performance for large matrices is presented in Table 2 (comprehensive

SGEMM results for various matrix and cache block sizes are available in Appendix B). SGEMM simulation data show that 32^2 blocks do not achieve sufficient computational intensity to fully utilize the processor. The choice of loop order and the resulting increase in memory traffic prevents column major 64^2 blocks from achieving a large fraction of peak (over 90%) for large matrices. Only 96^2 block sizes provide enough computational intensity to overcome the additional block loads and stores, and thus achieving near-peak performance - over 200GFlop/s. For BDL, however, 64^2 blocks effectively achieve peak performance. Whereas we assume a 1000 cycle DMA startup latency in our simulations, if the DMA latency were only 100 cycles, then the 64^2 column major performance would reach parity with BDL.

At 3.2GHz, each SPE requires about 3W [1]. Thus with a nearly idle PPC and L2, Cell achieves over 200GFlop/s for approximately 40W of power – nearly 5GFlop/s/Watt. Clearly for well-suited applications, Cell is extremely power efficient.

6.3 Double Precision GEMM Results

A similar set of strategies and simulations were performed for the DGEMM. Cache blocks are now limited to 64^2 due to the limited size of local store. However, unlike SGEMM, this block size does not limit performance. Although the time to load a DP 64^2 block is twice that of the SP version, the time required to compute on a 64^2 DP block is about 14x as long as the SP counterpart (due to the limitations of the DP issue logic). Thus it is far easier for DP to reach its peak performance. – a mere 14.6 GFlop/s. However, when using our proposed Cell+ hardware variant, DGEMM performance jumps to an impressive 51 GFlop/s.

6.4 Performance Comparison

Table 2 shows a performance comparison of GEMM between Cell and the set of modern processors evaluated in our study. Note the impressive performance characteristics of the Cell processors, achieving 57x, 27x, and 12.5x speed up for SGEMM compared with the Itanium2, Opteron, and X1 respectively. For DGEMM, the default Cell processor is 4.2x, 3.7x, and 1.3x faster than the Itanium2, Opteron, and X1. In terms of power, the Cell performance is even more impressive, achieving nearly 200x the efficiency of the Itanium2 for SGEMM!

Our Cell+ exploration architecture is capable, for large cache blocks, of fully exploiting the DP pipeline and achieving over 50 GFLOP/s. However, for smaller blocks (e.g. 32^2) performance will drop to under 35 GFLOP/s. In DP, the Cell+ architecture would be nearly 15 times faster than the Itanium2 and nearly 50 times more power efficient. Additionally, traditional micros (Itanium2, Opteron, etc) in multi-core configurations would require either enormous power saving innovations or dramatic reductions in performance, and thus would show even poorer performance/power compared with the Cell technology. Compared to the X1, Cell+ would be 3 times faster and 11 times more power efficient.

The primary focus for matrix multiplication on Cell should be the choice of data storage to minimize the number of DMAs and TLB misses while maximizing computational intensity. 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 blocks on chip before computing on them

Double Precision (GFlop/s)					Single Precision (GFlop/s)			
Cell+	Cell	X1	AMD64	IA64	Cell	X1	AMD64	IA64
51.1	14.6	11.2	3.9	3.5	204.7	16.4	7.5	3.6

Table 2 - GEMM performance (in GFlop/s) for large square matrices on Cell, X1, Opteron, and the Itanium2. Only the best performing numbers are shown. Cell demonstrates an impressive performance advantage with less than half the power of the other micros.

.

7. Sparse Matrix Vector Multiply

At first glance, SpMV would seem to be the worst application to run on Cell since the SPEs have neither caches nor word gather/scatter support. Furthermore, SpMV has O(1) computational intensity. However, these considerations are perhaps less important than the low functional unit and local store latency (<2ns), the task parallelism afforded by the SPEs, the

eight independent load store units, and ability to stream nonzeros via DMAs.

Two storage formats are presented in this paper: Compressed Sparse Row (CSR) and Blocked Compressed Sparse Row (BCSR) (*see Appendix C for details of these storage formats*). Only square BCSR was explored, and only 2x2 BCSR numbers will be presented here. Future Cell SpMV work will examine the entire BCSR space. Because of the quadword nature of the SPEs, all rows within a CSR cache block are padded to a multiple of 4. This greatly simplifies the programming model at the expense of increasing memory traffic. Note that this is very different than 1x4 BCSR.

7.1 Algorithmic Considerations

Without an accurate performance model of the MFC "get list" command, one must resort to cache blocking to provide a reasonable estimate for performance. 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 SPEs' 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, where each block stores its nonzeros and row pointers as if it were an entire matrix. This can result in more row pointer data being loaded and substantial overhead. We chose not to buffer the source and destination vector cache blocks as this would require more local store resources (or more precisely, result in a smaller block size). These tradeoffs will be examined in future work. Collectively the blocks are chosen to be no larger than ~36K words in SP (half that in DP).

The inner loop of CSR SpMV either requires significant software pipelining, hefty loop unrolling, or an approach algorithmically analogous to a segmented scan [30]. As there are no conditional stores in the SPU assembly language, we chose to partially implement a segmented scan, where the gather operations are decoupled from the dot products. This decoupled gather operation can be unrolled and software pipelined, thereby being performed in close to three cycles per element (the ISA is not particularly gather friendly). It is important to note 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. Future work will examine a full segmented scan via a software version of the conditional store.

As the nonzeros are stored contiguously in arrays, it is straightforward to stream them in via DMA. Here, unlike the source and destination vectors, it is essential to double buffer in order to maximize the SPEs computational throughput (remember the source and destination vectors are not double buffered). Using buffers of 16KB for SP, allows for 2K values and 2K indices for CSR, and 1K tiles for 2x2 BCSR. Note that for each phase – the loading nonzeros and indices – there is the omnipresent 1000 cycle DMA latency overhead in addition to the startup and finalize penalties (as in tradition pipelining).

To partition the work among the SPEs, we implemented a cooperative blocking model. By forcing all SPEs to work on the same block, it is possible to broadcast the blocked source vector and row pointers to minimize memory traffic. One approach, referred to as *PrivateY*, is to divide work among SPEs within a block by distributing the nonzeros as evenly as possible. This strategy necessitates that each SPE contains a private copy of the destination vector, and requires an inter-SPE reduction at the end of each blocked row. The alternate method, referred to as *PartitionedY*, partitions the destination vector evenly among the SPEs. 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 SPEs' computations will remain balanced, causing the execution time of the entire cache block to be limited by the most heavily loaded SPE. Thus for load balanced blocks, the PartitionedY approach is generally advantageous; however, for matrices exhibiting irregular (uneven) nonzero patterns, we expect higher performance using PrivateY.

Note that there is a potential performance benefit by writing a kernel specifically optimized for symmetric matrices. For these types of matrices, the number of operations can effectively double relative to the memory traffic. However, the algorithm must block two cache blocks at a time – thus the symmetric matrix kernel divides

memory allocated for blocking the vector evenly among the two submatrices, and performs a dot product and SAXPY for each row in the lower triangle.

7.2 Evaluation Matrices

In order to effectively evaluate SpMV performance, we examine six synthetic matrices, as well as ten real matrices used in numerical calculations from the Bebop SPARSITY suite [3,5] (four unsymmetric and six symmetric). Table 3 presents an overview of the evaluated matrices.

	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 – Suite of matrices used to evaluate SpMV performance. Matrix numbers as defined in the SPARSITY suite are shown in the first column.

7.3 Single Precision SpMV Results

Single and double precision SpMV results for the SPARSITY matrices are show in Tables 4 and 5. Surprisingly, given Cell's inherent SpMV limitations, the SPARSITY unsymmetric matrices average nearly 4GFlop/s, while the symmetric matrices average just over 6Gflop/s. Unfortunately, many of these matrices are so small that they utilize only a fraction of the default cache block size. Detailed results showing single precision SpMV performance on the Cell for our suite of matrices are shown in Appendix D.

Since it is clear that for this algorithm performance is almost entirely limited by the memory bandwidth, it is not possible for most unsymmetric matrices to attain the 6.4GFlop/s peak CSR performance, due to the substantial cache blocking and DMA overhead. As one might expect, large matrices with high densities show closer to peak performance, since the blocking overheads can be effectively amortized. Similarly, larger blocks yield higher performance for large matrices.

Unlike the synthetic matrices, the real matrices, which contain dense sub-blocks, can exploit BCSR without unnecessarily wasting memory bandwidth on zeros. As memory traffic is key, storing BCSR blocks in a compressed format (the zeros are neither stored nor loaded) would allow for significantly higher performance if there is sufficient support within the ISA to either decompress these blocks on the fly, or compute on compressed blocks. This is an area of future research.

Overall results show that the PrivateY approach is generally a superior partitioning strategy compared with PartitionedY. In most cases, the matrices are sufficiently unbalanced that the uniform partitioning of the nonzeros coupled with a reduction requires less time than the performing a load imbalanced calculation.

Since the local store size is fixed, blocks in the symmetric kernels are in effect half the size of the space allocated. When using the PartionedY approach, the symmetric kernel is extremely unbalanced for blocks along the diagonal. Thus, for matrices approximately the size of a single block, the imbalance between SPEs can severely impair the performance – even if the matrix is uniform. In fact, symmetric optimizations show only about 50% performance improvement when running the unsymmetric kernel on the symmetric matrices.

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 5% increase in performance. This is actually a good result. It means than the memory bandwidth is highly utilized and 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 only a 1% increase in performance on the SPARSITY unsymmetric matrix suite. Similarly, cutting the frequency in half (to

1.6GHz) results in only a 20% decrease in performance. Simply put, for the common case, more time is used in transferring nonzeros and the vectors rather than computing on them.

7.4 Double Precision SpMV Results

Results from our performance estimator show that single precision SPMV is almost twice as fast as double precision, even though the nonzero memory traffic only increases by 50%. This discrepancy is due to the reduction in the number of values contained in a cache block, where twice as many blocked rows are present. For example, when using $16K^2$ SP cache blocks on a $128K^2$ matrix, the 512KB source vector must be loaded 8 times. However, in DP, the cache blocks are only $8K^2$ – causing the 1MB source vector to be loaded 16 times, and thus resulting in a much higher volume of memory traffic. Future work will investigate caching mega blocks across SPEs to reduce total memory traffic.

Additionally, note that the extreme drop in floating point throughput (14x) between SP and DP, has relatively little impact on performance. This can also be seen in the difference between Cell and Cell+, where a 3.5x improvement in DP peak performance results in only a 5% speedup for SpMV.

	Double (GFlop/s)				Single (GFlop/s)	
Matrix	**Cell+**	**Cell**	**AMD64**	**IA64**	**Cell**	**IA64**
Vavasis	3.17	3.06	0.44	0.51	6.06	0.52
FEM	3.44	3.39	0.42	0.54	5.14	0.63
CFD	1.52	1.44	0.28	0.25	2.33	0.15
Average	**2.71**	**2.63**	**0.38**	**0.43**	**4.51**	**0.43**

Table 4 - SpMV performance (in GFlop/s) of Cell, Opteron and Itanium2 using single and double precision on the SPARSITY unsymmetric matrix suite. Even in double precision, Cell is about six times faster (with only four times the memory bandwidth).

	Double (GFlop/s)				Single (GFlop/s)	
Matrix	**Cell+**	**Cell**	**AMD64**	**IA64**	**Cell**	**IA64**
FEM	6.79	6.32	0.93	0.74	12.37	1.21
3D Tube	6.48	6.06	0.86	0.72	11.66	1.24
Portfolio	1.83	1.60	0.37	0.23	3.26	0.19
NASA	1.92	1.66	0.42	0.27	3.17	0.22
Vibro	3.90	3.47	0.57	0.31	7.08	0.41
LP	5.17	4.87	0.47	0.33	8.54	0.66
Average	**4.35**	**4.00**	**0.60**	**0.43**	**7.68**	**0.66**

Table 5 - SpMV performance (in GFlop/s) of Cell, Opteron and Itanium2 using single and double precision on the SPARSITY symmetric matrix suite. Cell is more than 9 times faster (with only four times the memory bandwidth).

7.5 Performance Comparison

Tables 4 and 5 compare Cell's estimated performance for SpMV with results from the Itanium2 and Opteron using the SPARSITY suite, a highly tuned sparse matrix numerical library. Considering that the Itanium2 and Opteron each have a 6.4GB/s bus compared to the Cell's 25.6GB/s DRAM bandwidth – one may expect that a memory bound application such as SpMV would perform only four times better on the Cell. Nonetheless, on average, Cell is more than 6x faster in DP and 10x faster in SP. This is because in order to achieve maximum performance, the Itanium2 must rely on the BCSR storage format, and thus waste memory bandwidth loading unnecessary zeros. However, the Cell's high FLOP to byte ratio ensures that the regularity of BCSR is unnecessary allowing it to avoid loading many of the superfluous zeros. For example, in matrix #17, Cell uses more than 50% of its bandwidth loading just the DP nonzero values, while the Itanium2 utilizes only 33% of its bandwidth. The rest of Itanium2's bandwidth is used for zeros and meta data. 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 compared with a traditional cache. In fact, Cell is nearly 20 times more power efficient than the Itanium2 and 15 times more efficient than the Opteron for SpMV. For a memory bound application such as this, multicore commodity processors will see little performance improvement unless they also scale memory bandwidth.

Comparing results with an X1 MSP, previously published work showed that a highly optimized permutation implementation (CSRP), achieves only 1 GFlop/s on a DP 7pt stencil matrix, while the standard CSR approach achieves less that 0.01 GFlop/s. On a similar matrix, Cell is able to achieve about 1.3GFlop/s. Thus, the Cell is nearly 50% faster, even though the X1 has 50% more memory bandwidth. The final paper version will contain the full set of SpMV results for both the Opteron and X1.

An alternate approach to cache blocking is to employ the MFC's "get list" command. This would allow for a gather operation either from

main memory, or from all local stores – thus potentially eliminating the inefficiencies of the current cache blocking approach, and perhaps yielding higher overall results. Unfortunately, no accurate performance information is currently available for small granularities (word/double). Therefore, unlike cache blocking where large granularities can be used to amortize latency, it is not yet possible to accurately create a SpMV performance model for this approach. Future work will explore this approach as hardware, simulators, or detailed performance documentation become publicly available.

8. Stencil Computations

Stencil-based computations on regular grids are at the core of a wide range of important scientific applications. In these applications, each point in a multidimensional grid is updated with contributions from a subset of its neighbors. The numerical operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and block structured adaptive methods.

In this work we examine two flavors of stencil computations derived from the numerical kernels of the Chombo[24] and Cactus[25] toolkits. Chombo is a framework for computing solutions of partial differential equations (PDEs) using finite difference methods on adaptively refined meshes. Here we examine a stencil computation based on Chombo's demo application, *heattut*, which solves a simple heat equation without adaptivity. Cactus is modular open source framework for computational science, successfully used in many areas of astrophysics. Our work examines the stencil kernel of the Cactus demo, *WaveToy*, which solves a 3D hyperbolic PDE by finite differencing. The *heattut* and *WaveToy* equations are shown in Figure 2.

Notice that both kernels solve 7 point stencils in 3D for each point. However, the *heattut* equation only utilizes values from the current time step, while *WaveToy* requires values from the current state as well as the previous state. Additionally, *WaveToy* has a higher computational intensity, and can more readily exploit the FMA pipeline.

```
Xnext[i,j,k,t+1] =
        X[i-1,j,k,t] + X[i+1,j,k,t] +
        X[i,j-1,k,t] + X[i,j+1,k,t] +
        X[i,j,k-1,t] + X[i,j,k+1,t] +
        αX[i,j,k,t]


X[i,j,k,t+1] =
        dt²/dx²(X[i-1,j,k,t]+X[i+1,j,k,t])+
        dt²/dy²(X[i,j-1,k,t]+X[i,j+1,k,t])+
        dt²/dz²(X[i,j,k-1,t]+X[i,j,k+1,t])+
        αX[i,j,k,t] - X[i,j,k,t-1]
```

Figure 2 - Stencil kernels used in evaluation. *Top*: Chombo *heattut* equation requires only the current time step. *Bottom*: CACTUS *WaveToy* equation requires both the current and previous time steps.

8.1 Algorithmic considerations

The algorithm used on Cell is virtually identical to that used on traditional architectures except that the ISA forces main memory loads and stores to be explicit, rather than caused by cache misses and evictions. The basic algorithmic approach to update the 3D cubic data array is to sweep across the domain, updating one plane at a time. Since a stencil requires both the next and previous plane, a minimum of 4 planes must be present in the local stores: (z-1,t), (z,t), (z+1,t), and (z,t+1). Additionally, bus utilization can be maximized by double buffering the previous output plane (z-1,t+1) with the next input plane (z+2,t)

In order to parallelize across SPEs, each plane of the 3D domain is partitioned into eight overlapping blocks. Due to the finite size of the local store memory, a straightforward stencil calculation is limited to planes of 256^2 elements plus ghost regions. Thus each SPE updates the core 256x32 points from a 258x34 slab (as slabs also contain ghost regions too).

To improve performance of stencil computations on cache-based architectures, previous research has shown multiple time steps can be combined to increase performance. [26, 27, 28]. This concept of *time skewing* can also be effectively leveraged in our Cell implementation. By keeping multiple planes from multiple time steps in the SPE simultaneously, it is possible to double or triple the number of stencils performed with almost no increase in memory traffic; thus increasing computational intensity and improving overall performance. Figure 3 details a flow diagram for the heat equation, showing both the simple and time skewed implementations.

Note that the neighbor communication required by stencils is not well suited for the

aligned quadword load requirements of the SPU ISA – i.e. unaligned loads must be emulated with permute instructions. In fact, for SP stencils with extensive unrolling, after memory bandwidth, the permute datapath is the limiting factor in performance - not the FPU. This lack of support for unaligned accesses highlights a potential bottleneck of the Cell architecture; however we can partially obviate this problem for the stencil kernel via data padding.

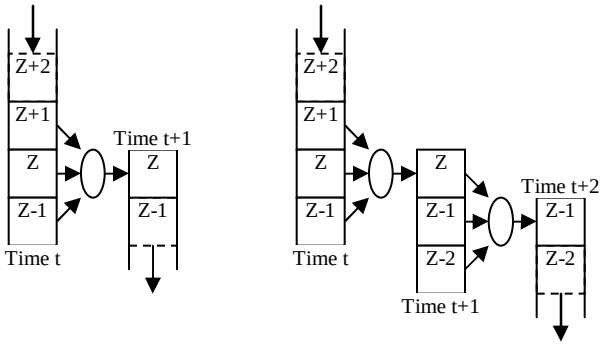


Figure 3 - Flow Diagram for Heat equation flow diagram. *Left*: Queues implemented within each SPE perform only one time step. *Right*: Time skewing version requires an additional circular queue to hold intermediate results.

8.2 Stencil Kernel Results

The performance estimation for the *heattut* and *WaveToy* stencil kernels is shown in Table 6 - detailed results are available in Appendix E. Results show that as the number of time steps increases, a corresponding decrease in the grid size is required due to the limited memory footprint of the local store. In SP, the heat equation on the Cell is effectively computationally bound with two steps of time skewing, resulting in over 41GFlop/s. More specifically, the permute unit becomes fully utilized as discussed in Section 8.1. In DP, however, the heat equation is truly computationally bound for only a single time step, achieving 8.2 GFlop/s. Analysis also shows that in the Cell+ approach, the heat equation is memory bound when using a single time step attaining 10.6 GFlop/s; for time skewing, performance of Cell+ DP jumps to over 21 GFlops/s.

We believe the temporal recurrence in the CACTUS *WaveToy* example will allow more time skewing in single precision at the expense of far more complicated code, and will be the subject of future investigation.

	Double Precision (GFlop/s)					Single Precision (GFlop/s)			
	Cell+ (2)	Cell+	Cell	AMD64	IA64	Cell (2)	Cell	AMD64	IA64
Heat	21.1	10.6	8.2	0.53	1.20	41.9	21.2	1.14	1.55
WaveToy	16.7	11.1	10.8	0.68	1.53	33.4	22.3	1.58	2.03

Table 6 - Performance for the heat equation and WaveToy stencils on Cell, Opteron, and Itanium2. Opteron and Itanium experiments use 128^3 and 256^3 grids. Cell uses the largest grid that would fit within the local store (similar sized, but varied with time skewing). The (2) versions denote a time skewed version where 2 time steps are computed.

8.3 Performance Comparison

Table 6 presents a performance comparison of the stencil computations across our evaluated set of leading processors. (The final paper version will contain X1 results.) Note that stencil performance has been optimized for the cache-based platforms as described in [22]

In single precision, for this memory bound computation, even without time skewing, Cell achieves and 11x and 14x speedup compared with the Itanium2 and Opteron respectively. Recall that the Cell has only four times the memory bandwidth of both of these platforms – indicating that Cell's potential to perform this class of computations in a much more efficient manner is due to the advantages of software controlled memory for algorithms exhibiting predictable memory accesses. Additionally, unlike the Opteron and Itanium2, simple time skewing has the potential to significantly increase performance in either SP (either version of Cell) or in DP on the Cell+ variant.

Finally, recall that in Section 7 we examined Cell SpMV performance using 7-point stencil matrices. We can now compare those results with the structured grid approach presented here, as the numerical computations is equivalent in both cases. Results show that for two time step calculations, the single precision structured grid approach achieves a 15x advantage compared with the sparse matrix method. This impressive speedup is attained through the regularity of memory accesses, reduction of memory traffic (constants are encoded in the equation rather than the matrix), and the ability to time skew (increased computational intensity). For double precision, the stencil algorithm advantage is diminished to approximately 6x, due mainly to the lack of time skewing

9. Fast Fourier Transforms

The FFT presents us with an interesting challenge: its computational intensity is much less than matrix-matrix multiplication and standard algorithms require a non-trivial amount of data movement. Extensive work has been performed on optimizing this kernel for both vector [31] and cache-based [23] machines. In addition, implementations for varying precisions appear in many embedded devices using both general and special purpose hardware. In this Section we evaluate the implementation of a standard FFT algorithm on the Cell processor.

9.1 Methods

We examine both the 1D FFT cooperatively executed across the SPEs, and a 2D FFT whose 1D FFTs are each run on a single SPE. In all cases the data appears in a single array of complex numbers. Internally (within the local stores) the data is unpacked into separate arrays, and a table lookup is used for the roots of unity so that no runtime computation of roots is required. As such, our results include the time needed to load this table. Additionally, all results are presented to the FFT algorithm and returned in natural order (i.e. a bit reversal was required to unwind the permutation process in all cases). Note that these requirements have the potential to severely impact performance.

For simplicity we evaluated a naive FFT algorithm (no double buffering and with barriers around computational segments) for the single 1D FFT. The data blocks are distributed cyclically to SPEs, 3 stages of local work are performed, the data is transposed (basically the reverse of the cyclic allocation), and then 9 to 13 stages of local computation is performed (depending on the FFT size). At that point the indices of the data on chip are bit-reversed to unwind the permutation process and the naturally ordered result copied back into main memory. Once again, we presume a large DMA initiation overhead of 1000 cycles. However, a Cell implementation where the DMA initiation overhead is smaller, would allow the possibility of much larger FFT calculations (including out of core FFTs) using smaller block transfers, with little or no slowdown using double buffering to hide the DMA latency.

Before exploring the 2D FFT, we briefly discuss simultaneous FFTs. For sufficiently small FFTs (<4K points in SP) it is possible to both double buffer and round robin allocate a large number of independent FFTs to the 8 SPEs. Although there is lower computational intensity, the sheer parallelism, and double buffering allow for extremely high performance (up to 76GFlop/s).

Simultaneous FFTs form the core of the 2D FFT. In order to ensure long DMAs, and thus validate our assumptions on effective memory bandwidth, we adopted an approach that requires two full element transposes. First, N 1D N-point FFTs are performed for the rows storing the data back to DRAM. Second, the data stored in DRAM is transposed (columns become rows) and stored back to DRAM. Third the 1D FFTs are performed on the columns, whose elements are now sequential (because of the transpose). Finally a second transpose is applied to the data to return it to its original layout. Instead of performing an N point bit reversal for every FFT, entire transformed rows (not the elements of the rows) are stored in bit-reversed order (in effect, bit reversing the elements of the columns). After the first transpose, a decimation in frequency FFT is applied to the columns. The columns are stored back in bit-reversed order - in doing so, the row elements are bit reversed. With a final transpose, the data is stored back to memory in natural order and layout in less time.

9.2 Single Precision FFT Performance

Table 7 presents performance results for the Cell 1D and 2D FFT. For the 1D case, more than half of the total time is spent just loading and storing points and roots of unity from DRAM. If completely memory bound, peak performance is approximately 3.2GHz * 5NlogN/3N cycles ~ 2.7logN GFlop/s. This means performance is limited to 64GFlop/s for a 4K point SP FFT regardless of CPU frequency. A clear area for future exploration is hiding computation within the communication and the minimization of the overhead involved with the loading of the roots of unity. Unfortunately the two full element transposes, used in the 2D FFT to guarantee long sequential accesses, consume nearly 50% of the time. Thus, although simultaneous FFTs achieve

76GFlop/s, the 2D FFT reaches only 46GFlop/s – an impressive figure nonetheless. Without the bit reversal approach, the performance would have further dropped to about 40GFlop/s.

9.3 Double Precision FFT Performance

When DP is employed, the balance between memory and computation is changed by a factor of 7. This pushes a slightly memory bound application strongly into the computationally bound domain. The SP simultaneous FFT is 10 times faster than the DP version. On the upside, the transposes required in the 2D FFT are now less than 20% of the total time, compared with 50% for the SP case. Cell+ finds a middle ground between the 4x reduction in computational throughput and the 2x increase in memory traffic – increasing performance by almost 2.5x compared with the Cell for all problem sizes.

9.4 Performance Comparison

The peak Cell FFT performance is compared to a number of other processors in the Table 7. These results are conservative given the naïve 1D FFT implementation we used on Cell whereas the other systems in the comparison used highly tuned FFTW [23] or vendor-tuned FFT implementations [18]. Nonetheless, in DP, Cell is 8x faster than the Itanium2, and Cell+ could be as much as 20x faster than the Itanium2 on a large 2D FFT. Cell+ more than doubles the DP FFT performance of Cell for all problem sizes. Cell performance is nearly at parity with the X1; however, we believe much headroom remains for more sophisticated Cell FFT implementations.

		Double Precision (GFlop/s)					Single Precision (GFlop/s)		
	N	Cell+	Cell	X1	AMD64	IA64	Cell	AMD64	IA64
1D	4K	12.6	5.6	2.6	2.1	2.7	29.9	3.8	2.8
	16K	14.2	6.1	5.8	1.6	2.2	37.4	2.6	2.7
	64K	-	-	8.8	1.2	1.5	41.8	1.9	2.4
2D	$1K^2$	15.9	6.6	-	1.1	0.8	35.9	1.5	1.6
	$2K^2$	16.5	6.7	-	-	-	40.5	-	-
	$4K^2$	-	-	-	-	-	44.9	-	-

Table 7 – Performance of 1D and 2D FFT on Cell, X1, Opteron, and Itanium2. For large FFTs, Cell is more than 10 times faster in SP than its competitors. Note: the Opteron used here is a 2GHz model.

Note that FFT performance on Cell performance improves as the number of points increases, so long as the points fit within the local store. In comparison, the performance on cache-based machines typically reach peak at a problem size that is far smaller than the on-chip cache-size, and then drop precipitously once the associativity of the cache is exhausted and cache lines start getting evicted due to aliasing. The evictions are unavoidable on cache-based architectures given the power-of-two problem sizes required by the FFT algorithm, but such evictions will not occur on Cell's software-managed local store. Furthermore, we believe that even for problems that are larger than local store, 1D FFTs will continue to scale much better on Cell than typical cache-based processors with set-associative caches since local store provides the same benefits as a fully associative cache. The FFT performance clearly underscores the advantages of software-controlled three-level memory architecture over conventional cache-based architectures.

10. Conclusions

The high performance computing community is exploring alternative architectural approaches to address the performance and power limitations of conventional processor designs. The Cell processor offers an innovative architectural approach that will be produced in large enough volumes to be cost-competitive with commodity CPUs. This work presents the first quantitative study Cell's performance on scientific kernels and directly compares its performance to tuned kernels running on leading superscalar (Opteron), VLIW (Itanium2), and vector (X1) architectures. 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, stencil computations, and 1D and 2D FFTs. While peak Cell DP throughput, required by most scientific applications, is far lower than SP, it still outperforms conventional processors on many kernels. Overall results demonstrate the tremendous potential of the Cell architecture for scientific computations in terms of both raw DP and SP performance and power efficiency.

Furthermore, we propose Cell+, a modest architectural variant to the Cell architecture designed to improve DP behavior. Results show that, aside from SpMV, the Cell+ significantly outperforms Cell for all of our evaluated kernels.

It is clear that if Cell is ever to play a leading role in scientific computing, DP must be promoted to a first class citizen within Cell.

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 hardware stream 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 also disadvantages to this architecture. For example, SpMV, with its unpredictable access patterns and low computational intensity achieves a dismally low percentage of Cell's peak performance. Even memory bandwidth may 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. Additionally, we see Stencil computations as an example of an algorithm with performance that is heavily influenced by the performance of the permute pipeline. Here, the lack of support for an unaligned load instruction is a more important performance bottleneck than either the SP execution rate or the memory bandwidth

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

Table 8 compares the advantage in DP of Cell and Cell+ in terms of performance and power efficiency for our suite of evaluated kernels and architectural platforms. (All missing performance data will appear in the final version.) Observe that the Cell+ approach greatly increases the already impressive performance characteristics of Cell – recall that both the Cell and Cell+ have just one DP floating-point unit, but the Cell+ can utilize it more effectively through modest enhancements to the execution pipeline.

It is important to consider these performance differences in the context of imminently prevalent multi-core commodity processors. The first generation of this technology will instantiate at most two cores per chip, and thus will deliver less than twice the performance of today's existing architectures. This factor of 2x is trivial compared with Cell+'s potential of 10–20x improvement, and does nothing if not widens the existing power efficiency gap.

	Cell+ Speedup over:			Cell+ power efficiency over:		
	X1	AMD64	IA64	X1	AMD64	IA64
GEMM	4.5x	13x	15x	11x	29x	49x
SpMV	-	7.1x	6.3x	-	16x	20x
Stencil	-	40x	17.5x	-	89x	57x
1D FFT	2.4x	8.9x	6.5x	6x	20x	21x
2D FFT	-	14x	20x	-	31x	65x

	Cell Speedup over:			Cell power efficiency over:		
	X1	AMD64	IA64	X1	AMD64	IA64
GEMM	1.3x	3.7x	4.2x	3.3x	8.2x	14x
SpMV	-	6.9x	6.1x	-	15x	20x
Stencil	-	15.5x	6.8x	-	34x	22x
1D FFT	1.05x	3.8x	2.8x	2.6x	8.5x	9.1x
2D FFT	-	6x	8.2x	-	13x	27x

Table 8 - Double precision speedup and increase in power efficiency of (*Top*) Cell+ and (*Bottom*) Cell, relative to the X1, Opteron, and Itanium2 for our evaluated suite of scientific kernels. Results show an impressive improvement in performance and power efficiency.

11. Future Work

A key component missing in this work is cycle-accurate simulation of the Cell architecture. We expect to work on validating the prediction models presented in this paper using a suite of high level Cell architectural simulators that are due to be released by IBM Research late this year. We will report those results in this paper if the software release proceeds as scheduled and NDA restrictions abate. The simulation results will also be checked against runs on Cell-based hardware when it becomes available.

In terms of potential algorithmic improvements, we believe GEMM performs extremely well and there is little room for additional gains. SpMV on the other hand, has many research opportunities from data storage formats, to cache blocking alternatives, to the MFC "get list" command. The FFT has perhaps the most room for improvement. The addition of double buffering, and reduction in memory traffic should help improve the peak performance for 1D FFTs, and alternative strategies to simple transposes are a necessity for more efficient 2D FFT versions. Table 9 presents the potential for further performance speedup for our scientific kernels on the Cell platform, based on our algorithmic analysis.

	Potential further speedup on Cell
GEMM	~0x (for N^3 approaches)
SpMV	~1.5x
Stencil	~0x
1D FFT	2.25x (single), 1.75x(cell+/double)
2D FFT	2x (single), 1.75x(cell+/double)

Table 9 – Potential for further speedup on Cell based on algorithmic analysis.

While peak Cell DP performance is impressive relative to its commodity peers, Cell will not reach its true potential for scientific computing until an SPE implementation that includes at least one fully utilizable pipelined DP floating point unit becomes available, as proposed in our Cell+ implementation. Until then, studies of Cell can provide insights into enhancements that may prove useful for mainstream desktop processors as well as Cell variants that include 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

[22] S.A. Kamil, et al. Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations, *ACM-MSP*, June 2005.

[23] FFTW Speed tests http://www.fftw.org

[24] Chombo homepage http://seesar.lbl.gov/anag/chombo

[25] Cactus homepage http://www.cactuscode.org

[26] Zhiyuan Li, Yonghong Song, Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.* 26(6): 975-1028, 2004

[27] David Wonnacott, Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. *IPDPS*, 171-180, 2000

[28] Guohua Jin, et al., Increasing temporal locality with skewing and recursive blocking, *SC*, 43, 2001

[29] J. A. Kahle, et al., Introduction to the Cell MultiProcessor, *IBM Journal of R&D*, Volume 49, Number 4/5, 2005. pp. 589-604.

[30] G. Blelloch, et. al, Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors, CMU-CS-93-173, 1993.

[31] L. Oliker, et al., A Performance Evaluation of the Cray X1 for Scientific Applications, VECPAR'04, 2004.

APPENDIX

A. GEMM Storage Formats

For GEMM, 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 DMAs 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 A.1 shows a matrix stored in the two formats.

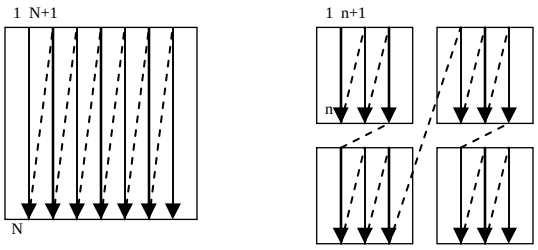


Figure A.1 - *Left*: column major layout. *Right*: BDL. Within each n x n block, values are stored in column major order

B. SGEMM Detailed Results

Figure B.1 shows SGEMM performance for various matrix dimensions, cache block sizes, and storage formats. Small cache blocks lack the computational intensity to keep the processor computationally bound.

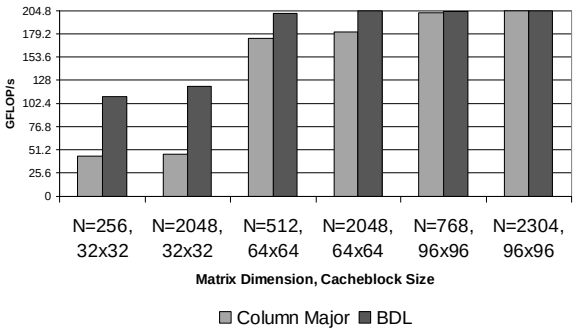
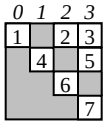


Figure B.1 - SGEMM on Cell. Even with the minimum overhead of BDL, the lack of computational intensity prevents 32^2 cache blocks from attaining 60% of peak. The inefficiency of column major layout prevents it from reaching peak performance without very large cache blocks.

C. SpMV Storage Formats

For SpMV, 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.

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 C.1 provides an example matrix and the corresponding data structures used in CSR and BCSR.



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 C.1 - 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.

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.

D. Detailed Single Precision SpMV Results

Cell SpMV performance is detailed in figures D.1 & D.2. For each matrix a number of storage and partitioning strategies were employed. BCSR does well on real world matrices, dense matrices achieve higher performance, and unbalanced matrices perform poorly in the PartitionedY strategy.

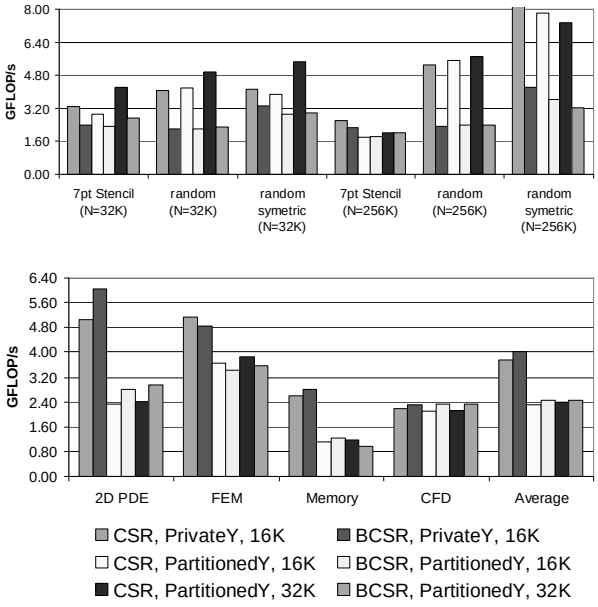


Figure D.1 - Top: SP SpMV using synthetic matrices – clear benefits from density and uniformity. Bottom: using SPARSITY unsymmetric matrices – PrivateY shows superior performance due to unbalance.

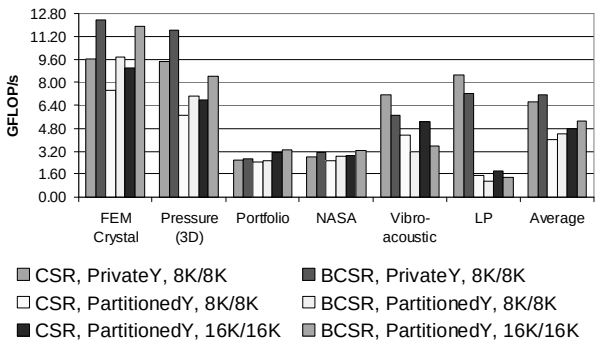


Figure D.2 - SP SpMV using SPARSITY symmetric matrices – Significant performance boost from minimization of nonzero traffic. Each of the cache blocks is half as big. Imbalance in PartitionedY strategy can generate serious performance degradation.

E. Detailed Stencil Results

The performance estimates for the *heattut* and *WaveToy* stencil kernels on the Cell is detailed in Figure E.1. Note that as the number of time steps increases, a corresponding decrease in the grid size is required due to the limited memory footprint of the local store. Observe that in SP, the heat equation is effectively computationally bound when time skewing with two time steps. More specifically, the permute unit becomes fully utilized as discussed in Section 6.1. In DP, however, the heat equation is truly computationally bound for only a single time step.

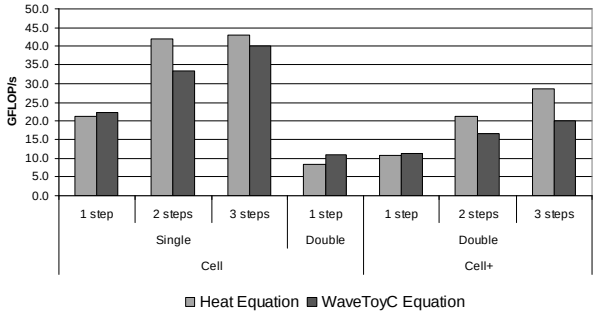


Figure E.1 - Performance in GFlop/s for the two stencils examined. For each, up to 3 time steps (time skewing) were taken. On Cell, DP is computationally bound with only a single time step. The Cell+ analysis showed that the heat equation is memory bound with a one time step, but time skewing will improve performance.