# The Potential of the Cell Processor for Scientific Computing

Samuel Williams, John Shalf, Leonid Oliker
Shoaib Kamil, Parry Husbands, Katherine Yelick
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
{swwilliams,jshalf,loliker,sakamil,prjhusbands,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. 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 using the forthcoming STI Cell processor as a building block for future high-end computing systems. Our work contains several novel contributions. First, we introduce a performance model for Cell and apply it to several key scientific computing kernels: dense matrix multiply, sparse matrix vector multiply, stencil computations, and 1D/2D FFTs. The difficulty of programming Cell, which requires assembly level intrinsics for the best performance, makes this model useful as an initial step in algorithm design and evaluation. Next, we validate the accuracy of our model by comparing results against published hardware results, as well as our own implementations on the Cell full system simulator. Additionally, we compare Cell performance to benchmarks run on leading superscalar (AMD Opteron), VLIW (Intel Itanium2), and vector (Cray X1E) architectures. Our work also explores several different mappings of the kernels and demonstrates a simple and effective programming model for Cell's unique architecture. Finally, 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.

## Categories and Subject Descriptors

C.1.2 [Processor Architectures] : Multiple Data Stream Architectures — Single-instruction- stream, multiple-data-stream processors (SIMD)
C.1.3 [Processor Architectures] : Other Architecture Styles — Heterogeneous (hybrid) systems
C.1.4 [Processor Architectures] : Parallel Architectures
C.4 [Performance of Systems] : Design studies, modeling techniques, performance attributes
D.1.3 [Programming Techniques] : Concurrent Programming — Parallel Programming

## General Terms

Performance, Design

## Keywords

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

## 1. INTRODUCTION

Over the last decade the HPC community has moved towards machines composed of commodity microprocessors as a strategy for tracking the tremendous growth in processor performance in that market. As frequency scaling slows, and the power requirements of these mainstream 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 combines the considerable floating point resources required for demanding numerical algorithms with a power-efficient software-controlled memory hierarchy. Despite its radical departure from previous mainstream/commodity processor designs, 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 (~14.6 Gflop/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 X1E) architectures. We believe this study examines the broadest array of scientific algorithms to date on Cell. We developed both analytical models and lightweight simulators to predict kernel performance that we demonstrated to be accurate when compared against published Cell hardware result, as well as our own implementations on the Cell full system simulator. 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 the software-controlled memory. 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 multicore processors.

## 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 [17, 19]. The performance of software-controlled memory is more predictable, thereby making it popular for real-time embedded applications where guaranteed response rates are essential.

Over the last five years, a plethora of alternatives to conventional cache-based architectures have been suggested including scratchpad memories [9,16,30], paged on-chip memories [12, 17], and explicit three-level memory architectures [18, 19]. 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 [20,23,29] and Intel's MXP5800 both achieved high performance at low power by adopting three levels (registers, local memory, external DRAM) of software-managed memory. More recently, the STI Cell processor has adopted a similar approach where data movement between

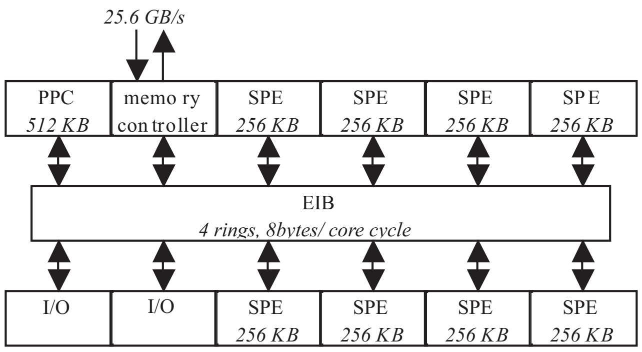

**Figure 1: Overview of the Cell processor**

these three address spaces is explicitly controlled by the application. 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. This more aggressive approach to memory architecture was adopted to meet the demanding cost/performance and real-time responsiveness 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 [8,27] 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), a local memory, and a memory flow controller. An overview of Cell is provided in Figure 1.

Access to external memory is handled via a 25.6GB/s XDR memory controller. The cache coherent 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 (the double precision operations within a SIMD operation must be serialized) 9-cycle pipelined FMA datapath with 4 cycles of overhead for data movement [22]. Cell has a 7 cycle in-order execution pipeline and forwarding network [8]. 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 SPE's DP throughput [14] 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 [10]. 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.

Unlike a typical coprocessor, each SPE has its own local memory from which it fetches code and reads and writes data. All loads and stores issued from the SPE can only access the SPE's local memory. The Cell processor depends on explicit DMA operations to move data from main memory to the local store of the SPE. 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. Dedicated DMA engines allow multiple concurrent DMA loads to run concurrently with the SIMD execution unit, thereby mitigating memory latency overhead via double-buffered DMA loads and stores. The selectable length DMA operations supported by the SPE are much like a traditional unit stride vector load. 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

The Cell architecture poses several challenges to programming: an explicitly controlled memory hierarchy, explicit parallelism between the 8 SPEs and the PowerPC, and a quadword based ISA. 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 memory hierarchy is programmed using explicit DMA intrinsics with the option of user programmed double buffering to overlap data movement with computation on the SPEs. Moving from a hardware managed memory hierarchy to one controlled explicitly by the application significantly complicates the programming model, and pushes it towards a one sided communication model. Unlike MPI, the intrinsics are very low level and map to half a dozen instructions. This allows for very low software overhead and good performance, but requires the user to be capable and either ensure correct usage or provide an interface or abstraction.

For programming the parallelism on Cell, we considered three possible programming models: task parallelism with independent tasks scheduled on each SPE; pipelined parallelism where large data blocks are passed from one SPE to the next; and data parallelism, where the processors perform identical computations on distinct data. For simplicity, we do not consider parallelism between the PowerPC and the SPEs, so we can treat this as a homogeneous parallel machine. Data pipelining may be suitable for certain classes of algorithms and will be the focus of future investigation. We adopt the data-parallel programming model, which is a good match to many scientific applications and offers the simplest and most direct method of decomposing the problem. Data-parallel programming is quite similar to loop-level parallelization afforded by OpenMP or the vector-like multistreaming on the Cray X1E and the Hitachi SR-8000.

The focus of this paper is Cell architecture and performance; we do not explore the efficacy of the IBM SPE XLC compiler. Thus, we heavily rely on SIMD intrinsics and do not investigate if appropriate SIMD instructions are generated by the compiler. Although the produced Cell code may appear verbose — due to the use of intrinsics instead of C operators — it delivers readily understandable performance.

Our first Cell implementation, SpMV, required about a month of learning the programming model, the architecture, the compiler, the tools, and deciding on a final algorithmic strategy. The final implementation required about 600 lines of code. The next code development examined two flavors of double precision stencil-based algorithms. These implementations required one week of work and are each about 250 lines, with an additional 200 lines of common code. The programming overhead of these kernels on Cell required significantly more effort than the scalar version's 15 lines, due mainly to loop unrolling and intrinsics use. Although the stencils are a simpler kernel, the SpMV learning experience accelerated the coding process.

Having become experienced Cell programmers, the single precision time skewed stencil — although virtually a complete rewrite from the double precision single step version — required only a single day to code, debug, benchmark, and attain spectacular results of over 65 Gflop/s. This implementation consists of about 450 lines, due once again to unrolling and the heavy use of intrinsics.

## 5. SIMULATION METHODOLOGY

The simplicity of the SPEs and the deterministic behavior of the explicitly controlled memory hierarchy make Cell amenable to performance prediction using a simple analytic model. Using this approach, one can easily explore multiple variations of an algorithm without the effort of programming each variation and running on either a fully cycle-accurate simulator or hardware. With the newly released cycle accurate simulator (Mambo), we have succesfully validated our performance model for SGEMM, SpMV, and Stencil Computations, as will be shown in the subsequent sections.

Our modeling approach is broken into two steps commensurate with the two phase double buffered computational model. The kernels were first segmented into code-snippets that operate only on data present in the local store of the SPE. We sketched the code snippets in SPE assembly and performed static timing analysis. The latency of each operation, issue width limitations, and the operand alignment requirements of the SIMD/quadword SPE execution pipeline determined the number of cycles required. The in-order nature and fixed local store memory latency of the SPEs makes the analysis deterministic and thus more tractable than on cache-based, out-of-order microprocessors.

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 accurately reflects the constraints imposed by resource conflicts in the memory subsystem. For instance, concurrent DMAs issued by multiple SPEs must be serialized, as there is only a single DRAM controller. The model also presumes a conservative fixed DMA initiation latency of 1000 cycles.

The model computes the total time by adding all the per-

| | Cell SPE | Cell Chip | X1E (MSP) | AMD64 | IA64 |
|---|---|---|---|---|---|
| Architecture | SIMD | Multi-core SIMD | Multi chip Vector | Super scalar | VLIW |
| Clock (GHz) | 3.2 | 3.2 | 1.13 | 2.2 | 1.4 |
| DRAM (GB/s) | 25.6 | 25.6 | 34 | 6.4 | 6.4 |
| SP Gflop/s | 25.6 | 204.8 | 36 | 8.8 | 5.6 |
| DP Gflop/s | 1.83 | 14.63 | 18 | 4.4 | 5.6 |
| Local Store | 256KB | 2MB | — | — | — |
| L2 Cache | — | 512KB | 2MB | 1MB | 256KB |
| L3 Cache | — | — | — | — | 3MB |
| Power (W) | 3 | ˜40 | 120 | 89 | 130 |
| Year | — | 2006 | 2005 | 2004 | 2003 |

**Table 1: Architectural overview of STI Cell, Cray X1E MSP, AMD Opteron, and Intel Itanium2. Estimated total Cell power and peak Gflop/s are based on the active SPEs/idle PowerPC programming model.**

iteration (outer loop) times, which are themselves computed by taking the maximum of the snippet and DMA transfer times. In some cases, the per-iteration times are constant across iterations, but in others it varies between iterations and is input-dependent. For example, in a sparse matrix, the memory access pattern depends on the nonzero structure of the matrix, which varies across iterations. Some algorithms may also require separate stages which have different execution times; e.g., the FFT has stages for loading data, loading constants, local computation, transpose, local computation, bit reversal, and storing the results.

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 [28]. 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 204.8 Gflop/s, with a computational intensity of 32 FLOPs/word. For comparison, we ran these kernels on actual hardware of several leading processor designs: the vector Cray X1E MSP, superscalar AMD Opteron 248 and VLIW Intel Itanium2. The key architectural characteristics are detailed in Table 1.

### 5.1 Cell+ Architectural Exploration

The Double Precision (DP) pipeline in Cell is obviously an afterthought as video games have limited need for DP arithmetic. Certainly a redesigned pipeline would rectify the performance limitations, but would do so at a cost of additional design complexity and power consumption. We offer a more modest alternative that can reuse most of the existing circuitry. Based on our experience designing the VIRAM vector processor-in-memory chip [12], we believe these “Cell+” design modifications are considerably less complex than a redesigned pipeline, consume very little additional surface area on the chip, but show significant DP performance improvements for scientific kernels.

In order to explore the limitations of Cell's DP issue bandwidth, we propose an alternate design with a longer forwarding network to eliminate the all but one of the stall cycles — recall the factors that limit DP throughput as described in Section 3. 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 not stall issuing other instructions and 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.

## 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 an extremely well suited for the Cell architecture. We explored two storage formats: column major and block data layout [26] (BDL). BDL is a two-stage addressing scheme (block row/column, element sub row/column).

### 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 element tiles designed to fit into the memory available on the Cell chip, which are in turn split into eight n x n element tiles that can fit into the 8 SPE local stores. For the column layout, the matrix will be accessed via a number of short DMAs equal to the dimension of the tile — 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 tiles, when double buffered, require $6n^2$ words of local store (one from each matrix) — thus making $96^2$ the maximum square tiles in SP. Additionally, in column layout, there is added pressure on the maximum tile size for large matrices, as each column within a tile will be on a different page resulting in TLB misses. The minimum size of a tile is determined by the FLOPs to word ratio of the processor. In the middle, there is a tile-size ”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 to minimize memory bandwidth requirements.

A possible alternate approach is to adapt Cannon's algorithm [3] 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 an algorithm 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

The Cell performance of GEMM based on our performance model (referred to as $\text{Cell}^{pm}$) for large matrices is presented in Table 2. SGEMM simulation data show that $32^2$ blocks do not achieve sufficient computational intensity to fully utilize the processor. The choice of loop order

|  | $\text{Cell}_{+}^{pm}$ | $\text{Cell}^{pm}$ | X1E | AMD64 | IA64 |
|---|---|---|---|---|---|
| DP (Gflop/s) | 51.1 | 14.6 | 16.9 | 4.0 | 5.4 |
| SP (Gflop/s) | — | 204.7 | 29.5 | 7.8 | 3.0 |

**Table 2: GEMM performance (in Gflop/s) for large square matrices on Cell, X1E, Opteron, and Itanium2. Only the best performing numbers are shown. Cell data based on our performance model is referred to as $\text{Cell}^{pm}$.**

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 [8]. Thus with a nearly idle PPC and L2, $\text{Cell}^{pm}$achieves over 200 Gflop/s for approximately 40W of power — nearly 5 Gflop/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 DGEMM. 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 $\text{Cell}^{pm}$and the set of modern processors evaluated in our study. Note the impressive performance characteristics of the Cell processors, achieving 69x, 26x, and 7x speed up for SGEMM compared with the Itanium2, Opteron, and X1E respectively. For DGEMM, the default Cell processor is 2.7x and 3.7x faster than the Itanium2 and Opteron. In terms of power, the Cell performance is even more impressive, achieving over 200x the efficiency of the Itanium2 for SGEMM!

Our $\text{Cell}_{+}^{pm}$ exploration architecture is capable, for large tiles, of fully exploiting the DP pipeline and achieving over 50 Gflop/s. In DP, the Cell+ architecture would be nearly 10 times faster than the Itanium2 and nearly 30 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 X1E, Cell+ would be 3 times as fast and 9 times more power efficient.

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.

## 6.5 Model Validation

IBM recently released their in-house performance evaluation of their prototype hardware [4]. On SGEMM, they achieve about 201 Gflop/s, which is within 2% of our predicated performance.

## 7. SPARSE MATRIX VECTOR MULTIPLY

At first glance, SpMV would seem to be a poor application choice for the Cell since the SPEs have neither caches nor word-granularity gather/scatter support. Furthermore, SpMV has a relatively low O(1) computational intensity. However, these considerations are perhaps less important than the Cell's low functional unit and local store latency (<2ns), the task parallelism afforded by the SPEs, the eight independent load store units, and the ability to stream nonzeros via DMAs.

## 7.1 Algorithmic Considerations

Two storage formats are presented in this paper: Compressed Sparse Row (CSR) and Blocked Compressed Sparse Row (BCSR). 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 tile 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..

To perform a stanza gather operation the Cell utilizes the MFC "get list" command, where a list of addresses/lengths is created in local store. The MFC then gathers these stanzas from the global store and packs them into the local store. It is possible to make every stanza a single quadword, however, without an accurate performance model of the MFC "get list" command, one must resort to tiling to provide a reasonable estimate for performance. For simplicity all benchmarks were run using square tiles. The data structure required to store the entire matrix is a 2D array of tiles, where each block stores its nonzeros and row pointers as if it were an entire matrix. We chose not to buffer the source and destination vector tiles as this would 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 [1]. As there are no conditional stores in the SPE 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 completing 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 consuming DRAM bandwidth or corrupting the actual arrays.

As the nonzeros are stored contiguously in arrays, it is

| # | Name | N | NNZ | Comments |
|---|---|---|---|---|
| 15 | Vavasis | 40K | 1.6M | 2D PDE Problem |
| 17 | FEM | 22K | 1M | Fluid Mechanics Problem |
| 18 | Memory | 17K | 125K | MotorolaMemory Circuit |
| 36 | CFD | 75K | 325K | Navier-Stokes, viscous flow |
| 06 | FEM Crystal | 14K | 490K | FEM stiffness matrix |
| 09 | 3D Tube | 45K | 1.6M | 3D pressure Tube |
| 25 | Portfolio | 74K | 335K | Financial Portfolio |
| 27 | NASA | 36K | 180K | PWT NASA Matrix |
| 28 | Vibroacoustic | 12K | 177K | Flexible box structure |
| 40 | Linear Prog. | 31K | 1M | $AA^T$ |
| — | 7pt_64 | 256K | 1.8M | $64^3$ 7pt stencil |

**Table 3: Suite of matrices used to evaluate SpMV performance. Matrix numbers as defined in the SPARSITY suite are shown in the first column.**

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. 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 — loading nonzeros and indices — there is the omnipresent 1000 cycle DMA latency overhead in addition to the startup and finalize penalties (as in traditional 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, was 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. However there is no longer any guarantee that the SPEs' computations will remain balanced, causing the execution time of the entire tile 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 tiles 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 a synthetic stencil matrix, as well as ten real matrices used in numerical calculations from the BeBop SPARSITY suite [11, 31] (four nonsymmetric and six symmetric). Table 3 presents an overview of the evaluated matrices.

## 7.3 Single Precision SpMV Results

Single and double precision tuned SpMV results for the SPARSITY matrices are show in Tables 4 and 5. Surprisingly, given Cell's inherent SpMV limitations, the SPARSITY nonsymmetric matrices average over 4 Gflop/s, while the symmetric matrices average nearly 8 Gflop/s. Unfortunately, many of these matrices are so small that they utilize only a fraction of the default tile size. 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.

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 nonsymmetric 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. Solely increasing the clock frequency by a factor of 2 (to 6.4GHz) provides only a 1% increase in performance on the SPARSITY nonsymmetric 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 tile, where twice as many blocked rows are present. For example, when using $16K^2$ SP tiles on a $128K^2$ matrix, the 512KB source vector must be loaded 8 times. However, in DP, the tiles 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.

## 7.5 Performance Comparison

Table 4 compares Cell's estimated performance (the best partitioning and blocking combination) for SpMV with re-

| | **SPARSITY nonsymmetric matrix suite** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Double Precision (Gflop/s) | | | | | | Single Precision (Gflop/s) | | |
| Matrix | Cell$^{FSS}$ | Cell$^{pm}_{+}$ | Cell$^{pm}$ | X1E | AMD64 | IA64 | Cell$^{pm}$ | AMD64 | IA64 |
| Vavasis | 3.79 | 3.17 | 3.06 | 0.84 | 0.44 | 0.46 | 6.06 | 0.70 | 0.49 |
| FEM | 4.28 | 3.44 | 3.39 | 1.55 | 0.42 | 0.49 | 5.14 | 0.59 | 0.62 |
| Mem | 2.21 | 1.69 | 1.46 | 0.57 | 0.30 | 0.27 | 2.79 | 0.45 | 0.31 |
| CFD | 1.87 | 1.52 | 1.44 | 1.61 | 0.28 | 0.21 | 2.33 | 0.38 | 0.23 |
| **Average** | **3.04** | **2.46** | **2.34** | **1.14** | **0.36** | **0.36** | **4.08** | **0.53** | **0.41** |

| | **SPARSITY symmetric matrix suite** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Double Precision (Gflop/s) | | | | | | Single Precision (Gflop/s) | | |
| Matrix | Cell$^{FSS}$ | Cell$^{pm}_{+}$ | Cell$^{pm}$ | X1E | AMD64 | IA64 | Cell$^{pm}$ | AMD64 | IA64 |
| FEM | — | 6.79 | 6.32 | 3.12 | 0.93 | 1.14 | 12.37 | 1.46 | 1.37 |
| 3D Tube | — | 6.48 | 6.06 | 2.62 | 0.86 | 1.16 | 11.66 | 1.36 | 1.31 |
| Portfolio | — | 1.83 | 1.60 | 2.99 | 0.37 | 0.24 | 3.26 | 0.42 | 0.32 |
| NASA | — | 1.92 | 1.66 | 3.30 | 0.42 | 0.32 | 3.17 | 0.46 | 0.40 |
| Vibro | — | 3.90 | 3.47 | 2.54 | 0.57 | 0.56 | 7.08 | 0.56 | 0.64 |
| LP | — | 5.17 | 4.87 | 1.27 | 0.47 | 0.63 | 8.54 | 0.55 | 0.92 |
| **Average** | — | **4.35** | **4.00** | **2.64** | **0.60** | **0.67** | **7.68** | **0.80** | **0.83** |

| | **Synthetic Matrices** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Double Precision (Gflop/s) | | | | | | Single Precision (Gflop/s) | | |
| Matrix | Cell$^{FSS}$ | Cell$^{pm}_{+}$ | Cell$^{pm}$ | X1E | AMD64 | IA64 | Cell$^{pm}$ | AMD64 | IA64 |
| 7pt_64 Stencil | 2.20 | 1.44 | 1.29 | — | 0.30 | 0.29 | 2.61 | 0.51 | 0.32 |

**Table 4: SpMV performance in single and double precision on the SPARSITY (top) nonsymmetric and (bottom) symmetric matrix suites. Note: Cell$^{FSS}$ represents the actual implementation and runs on the cycle accurate full system simulator**

sults from the Itanium2 and Opteron using SPARSITY, a highly tuned sparse matrix numerical library, on nonsymmetric (top) and symmetric matrix suites. X1E results where gathered using a high-performance X1-specific SpMV implementation [6].

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$^{pm}$ 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 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 X1E MSP is far more difficult. For unsymmetric matrices, the Cell$^{pm}$ performance on average is twice that of the X1E. For symmetric matrices, Cell$^{pm}$ performs somewhere between half and triple the performance of the X1E, but on average is 50% faster. The fact that the X1E consumes about three times the power of Cell guarantees Cell, in double precision, is at least as power efficient as the X1E

## 7.6 Model Validation

Some might claim that matrix-matrix multiplication performance can be easily predictable. Most, however, would agree that SpMV is very difficult to predict. As seen in Table 4, we tested our implementation of the DP SpMV kernel on the cycle accurate IBM full system simulator, referred to as Cell$^{FSS}$. The actual implementation makes dynamic blocking and partitioning decisions at run time, based on the lessons learned while exploring optimization strategies for the performance model; however, the current version but does not include the BCSR approach, and only pads rows to the nearest even number.

The cycle accurate simulations with a superior implementation proved to be about 30% faster than the initial performance estimate, and averages impressive results of more than 3 Gflop/s for nonsymmetric matrices. The 30% discrepancy disappears when static partitioning and blocking strategies used. We can clearly see how the actual implementation's run time search for structure boosted performance of the heat equation from about 1.3 Gflop/s to 2.2 Gflop/s — achieving a 7x speedup over the Itanium2. Cell$^{FSS}$, for double precision nonsymmetric matrices, is more than 8 times faster than the Itanium2, and 27 times more power efficient. These results confirm our performance model's predictive

$$\begin{aligned}X_{next}[i,j,k,t+1] &= X[i-1,j,k,t]+X[i+1,j,k,t]+\\ &\quad X[i,j-1,k,t]+X[i,j+1,k,t]+\\ &\quad X[i,j,k-1,t]+X[i,j,k+1,t]+\\ &\quad \alpha X[i,j,k,t]\end{aligned}$$

$$\begin{aligned}X[i,j,k,t+1] &= \frac{dt^2}{dx^2}(X[i-1,j,k,t]+X[i+1,j,k,t])+\\ &\quad \frac{dt^2}{dy^2}(X[i,j-1,k,t]+X[i,j+1,k,t])+\\ &\quad \frac{dt^2}{dz^2}(X[i,j,k-1,t]+X[i,j,k+1,t])+\\ &\quad \alpha X[i,j,k,t]-X[i,j,k,t-1]\end{aligned}$$

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

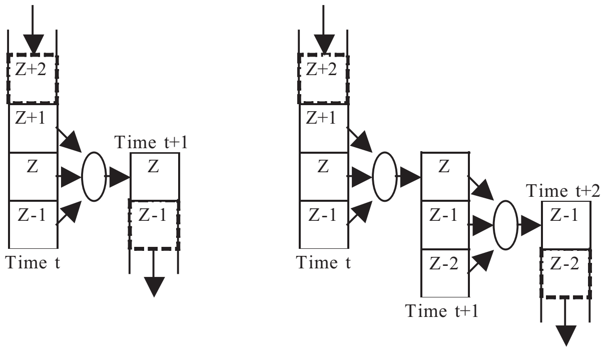

**Figure 3: Flow Diagram for Heat equation flow diagram. Left: Queues implemented within each SPE perform only one time step. Right: Time skewing version requires an additional circular queue to hold intermediate results.**

abilities on complex kernels, and clearly demonstrate Cell's performance superiority when compared with leading microarchitectural approaches.

## 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 [5] and Cactus [2] 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 previous time step, while WaveToy requires values from the two previous timesteps.. Additionally, WaveToy has a higher computational intensity, and can more readily exploit the FMA pipeline.

### 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).

To improve performance of stencil computations on cache-based architectures, previous research has shown multiple time steps can be combined to increase performance [13, 21, 32]. 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 SPE 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.

### 8.2 Stencil Kernel Results

The performance estimation for the heattut and WaveToy stencil kernels is shown in Table 5. 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$^{pm}$ is effectively computationally bound with two steps of time skewing, resulting in over 41 Gflop/s. More specifically, the permute unit becomes fully utilized as discussed

| | Double Precision (Gflop/s) | | | | | | |
|---|---|---|---|---|---|---|---|
| Stencil | Cell$^{FSS}$ | Cell$^{pm}_{+}$ (2 step) | Cell$^{pm}_{+}$ | Cell$^{pm}$ | X1E | AMD64 | IA64 |
| Heat | 7.25 | 21.1 | 10.6 | 8.2 | 3.91 | 0.57 | 1.19 |
| WaveToy | 9.68 | 16.7 | 11.1 | 10.8 | 4.99 | 0.68 | 2.05 |

| | Single Precision (Gflop/s) | | | | | |
|---|---|---|---|---|---|---|
| Stencil | Cell$^{FSS}$ (4 step) | Cell$^{pm}$ (2 step) | Cell$^{pm}$ | X1E | AMD64 | IA64 |
| Heat | 65.8 | 41.9 | 21.2 | 3.26 | 1.07 | 1.97 |
| WaveToy | — | 33.4 | 22.3 | 5.13 | 1.53 | 3.11 |

**Table 5: Performance for the Heat equation and WaveToy stencils. X1E and Itanium2 experiments use 256$^3$ grids. The Opteron uses a 128$^3$. Cell uses the largest grid that would fit within the local stores. The (n steps) versions denote a time skewed version where n time steps are computed.**

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.

## 8.3 Performance Comparison

Table 5 presents a performance comparison of the stencil computations across our evaluated set of leading processors. Note that stencil performance has been optimized for the cache-based platforms as described in [15].

In single precision, for this memory bound computation, even without time skewing, Cell$^{pm}$ achieves 6.5x, 11x, and 20x speedup compared with the X1E, the Itanium2 and the Opteron respectively. Recall that the Cell has only four times the memory bandwidth the scalar machines, and 75% the bandwidth of the X1E indicating that Cells 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. In double precision, with 1/14$^{th}$ the floating point throughput, Cell$^{pm}$ achieves a 2x, 7x, and 14x speedup compared to the X1E, the Itanium2, and the Opteron for the heat equation — a truly impressive result. Additionally, unlike the Opteron and Itanium2, simple time skewing has the potential to at least double the 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 are equivalent in both cases. Results show that for two time step calculations, the single precision structured grid approach achieves a 23x 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), the ability to time skew (increased computational intensity), and that stencils on a structured grid dont require multiplications by 1.0 like a sparse matrix would. For double precision, the stencil algorithm advantage is diminished to approximately 12x, due mainly to the lack of time skewing.

## 8.4 Model Validation

As with SpMV, we implemented an actual double precision kernel on the full system simulator, with Cell$^{FSS}$ results shown in Table 5. At first, we were surprised that measured performance fell short of our prediction by 13%. However, upon closer examination it was discovered that the actual Cell implementation prohibits dual issuing of DP instructions with loads or permutes, even though it allows SP with loads or permutes to be dual issued. Thus for kernels with streaming behavior, it is realistic to assume that one double precision SIMD instruction can be executed every 8 cycles — instead of every 7 as we had predicted previously. This discrepancy results in a 14% architectural performance reduction, which corresponding very well to the 13% difference observed in Table 5 between the predicted (*Cellpm*) and simulated (*Cell*$^{FSS}$) DP data.

Nonetheless, the actual DP Cell$^{FSS}$ implementation of our evaluated stencil kernel is about 13x faster, and nearly 30x more power efficient than the Opteron. We also developed a SP version of the heat equation that allowed four time-skewed stencil steps. (Our original performance estimation assumed one or two time steps.) Results show spectacular SP Cell$^{FSS}$ performance of nearly 66 Gflop/s — more than 60x faster and 136x power efficient compared with the Opteron, even though Cell has only four times the bandwidth and 20 times the single precision throughput.

## 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 [24] and cache-based [7] 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 76 Gflop/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 6 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 $(25.6GB/s/8Bytes) * 5NlogN/3N$ cycles or approximately $5.3logN$ Gflop/s. This means performance is limited to 64 Gflop/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 8K simultaneous 4K point FFTs achieve 76 Gflop/s (after optimizing away the loading of roots of unity), a $4K^2$ 2D FFT only reaches 46 Gflop/s — an impressive figure nonetheless. Without the bit reversal approach, the performance would have further dropped to about 40 Gflop/s. The smaller FFT's shown in the table show even poorer performance.

## 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. $\text{Cell}_{+}^{pm}$ 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.

| | N | Double Precision (Gflop/s) | | | | |
|---|---|---|---|---|---|---|
| | | $\text{Cell}_{+}^{pm}$ | $\text{Cell}^{pm}$ | $\text{X1E}^{*}$ | AMD64 | IA64 |
| 1D | 4K | 12.6 | 5.6 | 2.92 | 1.88 | 3.51 |
| | 16K | 14.2 | 6.1 | 6.13 | 1.34 | 1.88 |
| | 64K | — | — | 7.56 | 0.90 | 1.57 |
| 2D | $1K^2$ | 15.9 | 6.6 | 6.99 | 1.19 | 0.52 |
| | $2K^2$ | 16.5 | 6.7 | 7.10 | 0.19 | 0.11 |

| | N | Single Precision (Gflop/s) | | | | |
|---|---|---|---|---|---|---|
| | | $\text{Cell}_{+}^{pm}$ | $\text{Cell}^{pm}$ | $\text{X1E}^{*}$ | AMD64 | IA64 |
| 1D | 4K | — | 29.9 | 3.11 | 4.24 | 1.68 |
| | 16K | — | 37.4 | 7.48 | 2.24 | 1.75 |
| | 64K | — | 41.8 | 11.2 | 1.81 | 1.48 |
| 2D | $1K^2$ | — | 35.9 | 7.59 | 2.30 | 0.69 |
| | $2K^2$ | — | 40.5 | 8.27 | 0.34 | 0.15 |

**Table 6: Performance of 1D and 2D FFT in DP (top) and SP (bottom). For large FFTs, Cell is more than 10 times faster in SP than either the Opteron or Itanium2. The Gflop/s number is calculated based on a naive radix-2 FFT algorithm. For 2D FFTs the naive algorithm computes $2N$ N-point FFTs.**

## 9.4 Performance Comparison

The peak Cell FFT performance is compared to a number of other processors in the Table 6. These results are conservative given the naive 1D FFT implementation we used on Cell whereas the other systems in the comparison used highly tuned FFTW [7] or vendor-tuned FFT implementations [25]. Nonetheless, in DP, $\text{Cell}^{pm}$ is at least 12x faster than the Itanium2 for a 1D FFT, and $\text{Cell}_{+}^{pm}$ could be as much as 30x faster for a large 2D FFT. $\text{Cell}_{+}$ more than doubles the DP FFT performance of Cell for all problem sizes. Cell performance is nearly at parity with the X1E in double precision; however, we believe considerable headroom remains for more sophisticated Cell FFT implementations. In single precision, Cell is unparalleled.

Note that FFT performance on Cell 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 drops precipitously once the associativity of the cache is exhausted and cache lines start getting evicted due to aliasing. Elimination of cache evictions requires extensive algorithmic changes for the power-of-two problem sizes required by the FFT algorithm, but such evictions will not occur on Cells software-managed local store. Furthermore, we believe that even for problems that are larger than local store, 1D FFTs will con-

*X1E FFT numbers provided by Cray's Bracy Elton and Adrian Tate.

tinue to scale much better on Cell than typical cache-based superscalar processors with set-associative caches since local store provides all of the 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 AND FUTURE WORK

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 broadest 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 (X1E) architectures. We developed an analytic framework to predict Cell performance on dense and sparse matrix operations, stencil computations, and 1D and 2D FFTs. Using this approach allowed us to explore numerous algorithmic approaches without the effort of implementing each variation. We believe this analytical model is especially important given the relatively immature software environment makes Cell time-consuming to program currently; the model proves to be quite accurate, because the programmer has explicit control over parallelism and features of the memory system.

Furthermore, we propose Cell+, a modest architectural variant to the Cell architecture designed to improve DP behavior. 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. In addition, we show that Cell+ significantly outperforms Cell for most of our evaluated DP kernels, while requiring minimal microarchitectural modifications to the existing design.

Analysis shows that Cell's three level software-controlled 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 load time from local store is constant. 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 overlapped more effectively than conventional cache-based approaches. Increasing the size of the local store or reducing the DMA startup overhead on future Cell implementations may further enhance the scheduling efficiency by enabling more effective overlap of communication and computation.

There are also disadvantages to the Cell architecture for kernels such as SpMV. With its lack of unaligned load support, Cell must issue additional instructions simply to permute data, yet still manages to outperform conventional scalar processor architectures. Even memory bandwidth may be wasted since SpMV is constrained to use tiling to remove the indirectly indexed 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 that is heavily influenced by the performance of the permute pipeline. Here, the lack of support for an unaligned load instruction is a more significant performance bottleneck than either the SP execution rate or the memory bandwidth.

| **Cell+** | Speedup vs. | | | Power Efficiency vs. | | |
|---|---|---|---|---|---|---|
| | X1E | AMD64 | IA64 | X1E | AMD64 | IA64 |
| GEMM | 3x | 12.7x | 9.5x | 9x | 28.3x | 30.9x |
| SpMV | >2.7x | >8.4x | >8.4x | >8.0x | >18.7x | >27.3x |
| Stencil | 5.4x | 37.0x | 17.7x | 16.2x | 82.4x | 57.5x |
| 1D FFT | 2.3x | 10.6x | 7.6x | 6.9x | 23.6x | 24.7x |
| 2D FFT | 2.3x | 13.4x | 30.6x | 6.9x | 29.8x | 99.5x |

| **Cell** | Speedup vs. | | | Power Efficiency vs. | | |
|---|---|---|---|---|---|---|
| | X1E | AMD64 | IA64 | X1E | AMD64 | IA64 |
| GEMM | 0.8x | 3.7x | 2.7x | 2.4x | 8.2x | 8.78x |
| SpMV | 2.7x | 8.4x | 8.4x | 8.0x | 18.7x | 27.3x |
| Stencil | 1.9x | 12.7x | 6.1x | 5.7x | 28.3x | 19.8x |
| 1D FFT | 1.0x | 4.6x | 3.2x | 3.0x | 10.2x | 10.4x |
| 2D FFT | 0.9x | 5.5x | 12.7x | 2.7x | 12.2x | 41.3x |

**Table 7: Double precision speedup and increase in power efficiency of (Top) Cell+ and (Bottom) Cell, relative to the X1E, Opteron, and Itanium2 for our evaluated suite of scientific kernels. Results show an impressive improvement in performance and power efficiency.**

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 7 compares the advantage of Cell and Cell+ based on the better of performance model or actual implementation (where available) in terms of DP performance and power efficiency for our suite of evaluated kernels and architectural platforms. Observe that Cell+ has the potential to greatly increase the already impressive performance characteristics of Cell.

By using the insight gained in the development of our estimation model, we developed an optimized SpMV version that outperformed our initial predictions by 25% – 70%. If a full system simulator could model the modest improvements of our Cell+ variant, we feel confident that we could demonstrate comparable improvements to DP performance as well. We also note that DP stencil performance fell short of our model by 13% due to previously unknown microarchitectural limitations. However, time skewing showed a huge benefit in SP and we believe a similar benefit would be present in DP on Cell+ variant.

It is important to consider these performance differences in the context of increasingly 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.

While peak Cell DP performance is impressive relative to its commodity peers, a fully utilizable pipelined DP floating point unit would boost Cell (i.e. Cell+) performance and efficiency significantly.

## Acknowledgments

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The authors gratefully thank Bracy Elton and Adrian Tate for their assistance in obtaining X1E FFT performance data, and Eduardo D'Azevedo for providing us with an optimized X1E SpMV implementation.

## 11. REFERENCES

[1] G. Blelloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, CMU, 1993.

[2] Cactus homepage. `http://www.cactuscode.org`.

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

[4] Cell broadband engine architecture and its first implementation. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`.

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

[6] E. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science (ICCS)*, pages 99–106, 2005.

[7] FFTW speed tests. `http://www.fftw.org`.

[8] B. Flachs, S. Asano, S. Dhong, et al. A streaming processor unit for a cell processor. *ISSCC Dig. Tech. Papers*, pages 134–135, February 2005.

[9] P. Francesco, P. Marchal, D. Atienzaothers, et al. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Design Automation Conference*, June 2004.

[10] Ibm cell specifications. `http://www.research.ibm.com/cell/home.html`.

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

[12] The Berkeley Intelligent RAM (IRAM) Project. `http://iram.cs.berkeley.edu`.

[13] G. Jin, J. Mellor-Crummey, and R. Fowlerothers. Increasing temporal locality with skewing and recursive blocking. In *Proc. SC2001*, 2001.

[14] J. Kahle, M. Day, H. Hofstee, et al. Introduction to the cell multiprocessor. *IBM Journal of R&D*, 49(4), 2005.

[15] S. Kamil, P. Husbands, L. Oliker, et al. Impact of modern memory subsystems on cache optimizations for stencil computations. In *ACM Workshop on Memory System Performance*, June 2005.

[16] M. Kandemir, J. Ramanujam, M. Irwin, et al. Dynamic management of scratch-pad memory space. In *Proceedings of the Design Automation Conference*, June 2001.

[17] P. Keltcher, S. Richardson, S. Siu, et al. An equal area comparison of embedded dram and sram memory architectures for a chip multiprocessor. Technical report, HP Laboratories, April 2000.

[18] B. Khailany, W. Dally, S. Rixner, et al. Imagine: Media processing with streams. *IEEE Micro*, 21(2), March-April 2001.

[19] M. Kondo, H. Okawara, H. Nakamura, et al. Scima: A novel processor architecture for high performance comp uting. In *4th International Conference on High Performance Computing in the Asia Pacific Region*, volume 1, May 2000.

[20] A. Kunimatsu, N. Ide, T. Sato, et al. Vector unit architecture for emotion synthesis. *IEEE Micro*, 20(2), March 2000.

[21] Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Language Systems*, 26(6), 2004.

[22] S. Mueller, C. Jacobi, C. Hwa-Joon, et al. The vector floating-point unit in a synergistic processor element of a cell processor. In *17th IEEE Annual Symposium on Computer Arithmetic (ISCA)*, June 2005.

[23] M. Oka and M. Suzuoki. Designing and programming the emotion engine. *IEEE Micro*, 19(6), November 1999.

[24] L. Oliker, R. Biswas, J. Borrill, et al. A performance evaluation of the Cray X1 for scientific applications. In *Proc. 6th International Meeting on High Performance Computing for Computational Science*, 2004.

[25] Ornl cray x1 evaluation. `http://www.csm.ornl.gov/~dunigan/cray`.

[26] N. Park, B. Hong, and V. Prasanna. Analysis of memory hierarchy performance of block data layout. In *International Conference on Parallel Processing (ICPP)*, August 2002.

[27] D. Pham, S. Asano, M. Bollier, et al. The design and implementation of a first-generation cell processor. *ISSCC Dig. Tech. Papers*, pages 184–185, February 2005.

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

[29] M. Suzuoki et al. A microprocessor with a 128-bit cpu, ten floating point macs, four floating-point dividers, and an mpeg-2 decoder. *IEEE Solid State Circuits*, 34(1), November 1999.

[30] S. Tomar, S. Kim, N. Vijaykrishnan, et al. Use of local memory for efficient java execution. In *Proceedings of the International Conference on Computer Design*, September 2001.

[31] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California at Berkeley, 2003.

[32] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.