Lawrence Berkeley National Laboratory

Applied Math & Comp Sci

Title

Scientific Computing Kernels on the Cell Processor

Permalink

https://escholarship.org/uc/item/1n87d071

Journal

Lawrence Berkeley National Laboratory

Authors

Williams, Samuel W.
Shalf, John
Oliker, Leonid
et al.

Publication Date

2008-04-18

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Scientific Computing Kernels on the Cell Processor

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,pjrhusbands,kayelick}@lbl.gov

ABSTRACT

The slowing pace of commodity microprocessor performance improvements combined with ever-increasing chip power demands has become of utmost concern to computational scientists. 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 recently-released 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 a 3.2GHz Cell blade. 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.

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 continue 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 recently-released 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 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 double precision applications.

This paper presents several novel results and expands our previous efforts [37]. 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 results, as well as our own implementations on a 3.2GHz Cell blade. 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 would increase the efficiency of double-precision arithmetic calculations 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 exploit Cell's heterogeneity not in computation, but in control and system support. Thus we 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. One such alternative is software-controlled memories, which 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 [20, 22]. 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 [10,19,35], paged on-chip memories [14, 20], and explicit three-level memory architectures [21, 22]. 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 [23, 26, 34] 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 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 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. However, if the data access pattern lacks predictability, 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 newly released 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 [9,30] was designed by a partnership of Sony, Toshiba, and IBM (STI) to be the heart of Sony's recently-released 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.

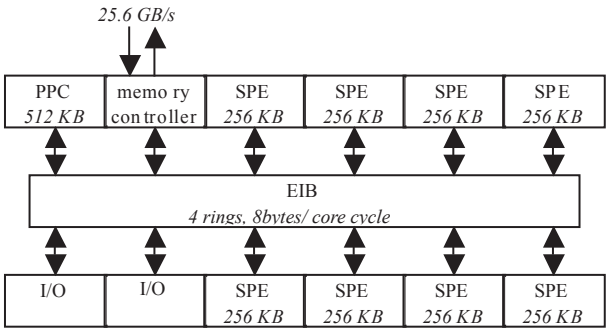


Figure 1: Overview of the Cell processor

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/core cycle to be read or written, and each SPE initiates and waits for its own transfers. Simultaneous transfers on the same ring are possible.

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 [25]. Cell has a 7 cycle in-order execution pipeline and forwarding network [9]. IBM appears to have solved the problem of inserting a 13 (9+4) cycle double precision 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 double precision instruction. The SPE's double precision throughput [16] of one double precision 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 single precision implementations to run near peak, whereas double precision versions would drop to approximately one fourteenth the peak single precision flop rate [12]. Similarly, for bandwidth intensive applications such as sparse matrix vector multiplication (SpMV), we expect single precision versions to be between 1.5x and 4x as fast as double precision, 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, and 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 DMAs to run concurrently with SIMD execution, 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 in order 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 Cell 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 high-level message-passing paradigms such as MPI [32], the DMA intrinsics are very low level and map to half a dozen instructions. This allows for very low software overhead and potentially high performance, but requires the user to either ensure correct usage or provide a specially designed interface or abstraction.

For programming inter-SPE 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, allowing us to treat Cell as a homogeneous data parallel machine, with heterogeneity in control only. For the investigations conducted in this paper, 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. Data pipelining may be suitable for certain classes of algorithms and will be the focus of future investigation.

The focus of this paper is Cell architecture and performance; we do not explore the efficiency of the IBM SPE XLC compiler. Thus, we rely heavily 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.

To gain a perspective of programming complexity, we briefly describe our Cell programming learning curve. Our first Cell implementation, SpMV, required about a month for 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 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 cycle accurate full system simulator (Mambo), we have successfully validated our performance model for SGEMM, SpMV, and Stencil Computations, as will be shown in subsequent sections.

5.1 Cell Performance Modeling

Our modeling approach is broken into two steps commensurate with the two phase double buffered computational model. The kernels are 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 (without register allocation) 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 this 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 (software and hardware) of 1000 cycles.

Our model then computes the total time by adding all the per-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 computation, the memory access pattern depends on the matrix nonzero structure, which varies across iter-

	Cell		X1E	AMD64	IA64
	SPE	Chip	(MSP)		
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)	5	~100	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 chip power and is derived from IBM Cell blade power. Total Gflop/s does not include the PowerPC core.

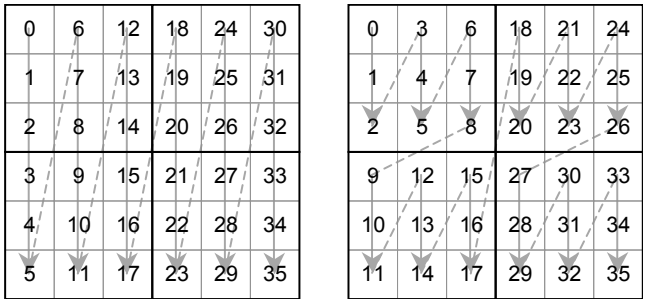


Figure 2: Column major layout versus Block Data Layout. In both cases a 9x9 matrix with 3x3 cache blocks is represented. The numbers are representative of addresses in memory. On the left, the column major layout will necessitate three separate stanzas to be loaded for each cache block - as the columns within a cache block are not contiguous. On the right, the block data layout approach ensures that each cache block is contiguous and thus requires a single long DMA transfer.

ations. 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.

5.2 Hardware Comparison

For simplicity and consistency 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 [33] with perhaps more enabled SPEs. The lower frequency had the simplifying benefit that both the EIB and DRAM controller could deliver two single precision words per cycle. The maximum flop rate of such a machine would be 204.8 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. Both the sparse matrix kernel and the stencil kernels were implemented and run on an IBM 3.2GHz Cell blade using 8 SPEs. The key architectural characteristics of these machines are detailed in Table 1.

5.3 Cell+ Architectural Exploration

The double precision pipeline in Cell is obviously an afterthought as video games have limited need for double precision 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 [14], we believe these design modifications are considerably less complex than a redesigned pipeline and consume very little additional surface area on the chip, but show significant double precision performance improvements for scientific kernels.

In order to explore the limitations of Cell's double precision issue bandwidth, we propose an alternate design with a longer forwarding network to eliminate all but one of the stall cycles — recall the factors that limit double precision throughput as described in Section 3. In this hypothetical implementation, called Cell+, each SPE would still have the single half-pumped double precision datapath, but would be able to dispatch one double precision 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 double precision throughput of Cell (51.2 Gflop/s) by fully utilizing the existing double precision datapath; however, it would maintain the same single precision throughput, frequency, bandwidth, and power as 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 extremely well suited for the Cell architecture.

6.1 Data Layout

We explore two storage formats: column major and block data layout [29] (BDL). In a column major layout, the data within each column is stored contiguously. BDL is a two-stage addressing scheme where the matrix first undergoes a 2D partitioning into blocked rows and blocked columns, the intersection of which will be referred to as a cache block. In the second stage (i.e. within a cache block), the data is stored in a column major format. Figure 2 illustrates the differences in these storage formats.

6.2 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. The BDL approach, on the other hand, will require a single long DMA of length 16KB in single precision.

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 tile size in single precision. 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 is therefore chosen to minimize the average number of pages touched per phase for a column major storage format. The BDL approach, where TLB misses are of little concern, allows us to structure the loop order to minimize memory bandwidth requirements. Whenever possible, we attempt to broadcast the block to all SPEs which may need it.

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.

6.3 GEMM Results

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 achieve near-peak performance — over 200 Gflop/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.

The peak Cell performance of GEMM based on our performance model (referred to as Cell^{pm}) for large matrices is presented in Table 2. For BDL with large cache blocks Cell is capable of reaching over 200 Gflop/s in single precision. In double precision, although the time to load a 64^2 block is twice that of the single precision version, the time required to compute on a 64^2 double precision block is about 14x as long as the single precision counterpart (due to the limitations of the double precision issue logic). This significantly reduces memory performance pressure on achieving peak performance. Nevertheless, when using our proposed Cell+ hardware variant, DGEMM performance jumps to an impressive 51 Gflop/s.

Cell^{pm} achieves 200 Gflop/s for perhaps 100W of power — nearly 2 Gflop/s/Watt. Clearly, for well-suited applications, Cell is extremely power efficient. However, actual power usage may be substantially lower. At 3.2GHz, each SPE may require 5W [9]. Thus with a nearly idle PPC and L2, Cell may actually achieve 4 Gflop/s/Watt.

	Cell$^{pm}_{+}$	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 Cell^{pm}.

6.4 Performance Comparison

Table 2 shows a performance comparison of GEMM between 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, Cell performance is even more impressive, achieving at least 85x the power efficiency of the Itanium2 for SGEMM!

Our Cell$^{pm}_{+}$ exploration architecture is capable, for large tiles, of fully exploiting the double precision pipeline and achieving over 50 Gflop/s. In double precision, the Cell+ architecture would be nearly 10 times faster than the Itanium2 and more than 12 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 3.5 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 has 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

Sparse matrix-vector multiplication is an important computational kernel used in scientific computing, signal and image processor, and many other important applications. In general, the problem is to compute $y = \alpha A \times x + \beta y$, for sparse matrix A, dense vectors x and y, and scalars α and β. We restrict the kernel to $\alpha = 1.0$, and $\beta = 0.0$. Sparse matrix algorithms tend to run much less efficiently than their dense matrix counterparts due to irregular memory access and low computational intensity.

At first glance, SpMV would seem to be a poor application choice for Cell since the SPEs have neither caches nor efficient word-granularity gather/scatter support. However, these considerations are perhaps less important than 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 Storage Formats

Since the sparse matrix is stored in compressed format, a myriad of storage formats have arisen each with specific advantages [31]. The three used in this paper are shown in Figure 3.

The most common format, compressed sparse row (CSR), organizes the matrix into sparse rows as shown in Figure 3 (top). Each sparse row is encoded in two arrays - one to store the nonzero values, and the other to store the corresponding nonzero column indices. The two arrays are stored contiguously with the arrays from the other sparse rows. As such, it is possible to provide a single row start pointer for each sparse row; the row end is just the row start of the next sparse row.

The typical implementation of sparse matrix-vector multiplication using CSR uses two nested loops: one for each of the rows, and one for the nonzeros within the current row. This clearly can be efficient when the average row length is large, but performs poorly, due to loop overhead, when rows are short. A variant on this standard implementation is segmented scan [1] in which the loop is unrolled via a conditional statement. Most compilers do not code the conditional statement efficiently and thus the method is rarely used. The data structure for segmented scan remains the standard CSR; only the code structure is modified. The X1E suite includes this approach as an option for sparse matrix multiplication.

One option to enable SIMDization is to require that all rows be a multiple of the register size. This is illustrated in Figure 3 (middle). Thus for double precision with 128b SIMD registers, all row lengths must be even: they are multiples of four in single precision. One then exploits intra-row parallelism. More succinctly, both even and odd partial sums are created. At the end of a row, they are transposed, and reduced with those on the next row. The single resulting quadword is stored.

The alternate option for enabling SIMDization is to exploit the local structure of the matrix, and register block it. The resultant blocked compressed sparse row (BCSR) format, as shown in Figure 3 (bottom), has the disadvantage of increasing memory bandwidth since zeros are explicitly stored. However it has the advantages of increasing local inner loop parallelism, amortizing loop overhead, and facilitating SIMDization.

Finally, one should note that instead of processing nonzeros one row at a time, it is possible to parallelize across multiple rows. Permuted CSR (CSRP) sorts the rows by length at runtime. Although this will force irregular access to the destination vector as well as the nonzeros, it has the advantage that it is far easier to vectorize the computation. An alternate approach changes the data structure to remove the indirection to the nonzeros and thus stores the i^{th} nonzero of each row contiguously. This storage format, known as jagged diagonal, is popular and efficient on vector machines such as the X1E.

7.2 Exploiting Locality

For Cell, only CSR will be examined as BCSR is an ongoing research area. As discussed above, because of the

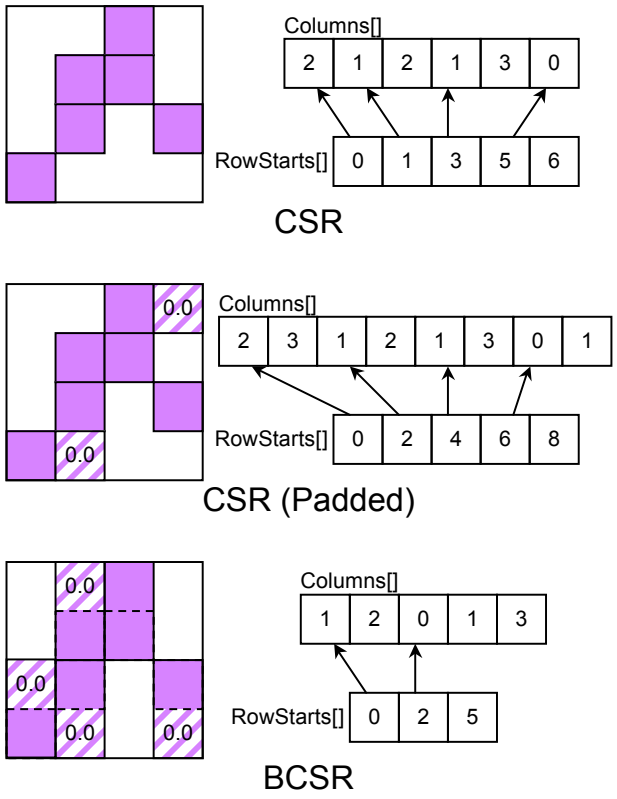


Figure 3: Comparison of nonzero sparse storage formats for a 4x4 matrix. Note: nonzeros are solid, and explicitly zeros are hashed. The value arrays are not shown. Top: compressed sparse row (CSR). Middle: padded compressed sparse row. Each row length may be unique, but must be even. Two explicit zeros are required. Bottom: 2x1 block compressed sparse row (BCSR). The BCSR version requires four explicit zeros to be stored.

quadword nature of the SPEs, all rows within a CSR tile are padded to 128b granularities. This greatly simplifies the programming model at the expense of no more than N explicit zeros.

To perform a stanza gather operation Cell utilizes the MFC "get list" command, where a list of addresses/lengths is created in local store. The MFC gathers these stanzas from the global store and packs them into the local store. It is possible to make every stanza a single float or double; however, the underlying DRAM architecture operates on 1024b (128byte) granularities. Therefore, although the program specifies that only a doubleword needs to be loaded, the underlying hardware will load the full 128 byte line. Thus with a naive implementation, each nonzero would require the transfer of perhaps 140 bytes to perform only two flops. Even if peak memory bandwidth was attainable, this lack of spatial and temporal locality would limit performance to a mere 0.35 Gflops.

In order to exploit spatial locality a matrix cache blocking algorithm was implemented, where the matrix is partitioned

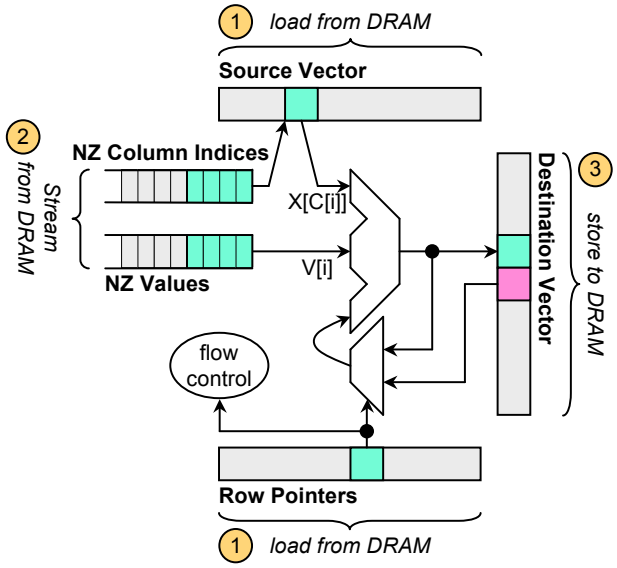


Figure 4: Streaming and caching for SpMV. There are up to three phases for each cache block: 1. load and cache source vector and row pointer, 2. stream nonzeros and update destination vector, 3. if the last cache block in a row, store the destination vector to DRAM.

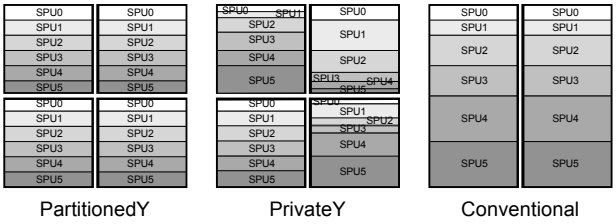


Figure 5: Parallelization strategies for SpMV on a N x N matrix. Note - the source vector cache block size is half the source vector size (N/2). Left: PartitionedY - the destination vector for each cache block is uniformly partitioned, but the computation is not. SPUs are lock step at cache block granularity. It uses an explicit 2D cache blocking strategy. Middle: PrivateY - each cache block is uniquely parallelized and balanced. However, each SPU must maintain a copy of the destination vector, which is reduced at the end of the row. SPUs are lock step at cache block granularity. It also uses an explicit 2D cache blocking strategy. Right: Conventional cache blocked strategy - parallelization is carried across cache block boundaries and the SPUs must only synchronize at the end of the matrix. It uses a 1D blocking, but load balancing is far more difficult as it must be estimated for the entire matrix, not just a cache block.

in 2D such that part of the source vector and part of the destination vector could fit simultaneously in local store. For sufficiently structured matrices there is enough temporal and spatial locality to make this a viable option. In effect, instead of one matrix stored in CSR format, less than a dozen thinner matrices are each stored in CSR format. Note that if each cache block has fewer than 64K columns, then it is possible to store a relative column index as a halfword and save 2 bytes of memory traffic per nonzero.

7.3 Streaming

As the nonzeros are stored contiguously in arrays, it is straightforward to stream them into memory via DMA operations. Here, unlike the source and destination vectors, it is essential to double buffer in order to maximize the SPE's computational throughput. Using buffers of about 8KB aligned to DRAM lines results in high bandwidth utilization and sufficiently amortizes the omnipresent 1000 cycle DMA latency overhead, but results in rather large prologue and epilogue when pipelining. This is shown in Figure 4.

We chose to implement a standard CSR loop structure with the caveat that it has been software pipelined as much as possible. This presents complications as only a few nonzeros are in the buffer, and thus the kernel must be organized around looping through the buffer as opposed to looping through the sparse row. As Cell's SIMD implementation is not scalar friendly, BCSR and blocked segmented scan are areas for further research, although the latter is difficult without a predicated store instruction.

7.4 Parallelization

When parallelizing the work among the SPEs, we explored three variations (shown in Figure 5): parallelization of a single blocked row, parallelization of a single cache block, and parallelization of the entire matrix via standard row blocking. If synchronization is performed on a per cache block basis, then it is possible to use a broadcast mechanism to leverage the vast on-chip bandwidth it disseminate the source vector to all SPEs. Within this subclass, one approach, referred to as PartitionedY, partitions the destination vector evenly among the SPEs. With this approach, there is no guarantee that the SPEs' computations will remain balanced. Thus, the execution time of the entire tile can be limited by the most heavily loaded SPE. An alternate method, referred to as PrivateY, divides work among SPEs within a cache 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. Thus for uniform cache blocks, the PartitionedY approach is generally advantageous; however, for matrices exhibiting irregular (uneven) nonzero patterns, we expect higher performance using PrivateY. The third method, is the more standard approach, with synchronization only at the end of the matrix. No broadcast is possible, but load balancing is more conventional. Only the third approach was actually implemented.

Note that there is a potential performance benefit by writing a kernel specifically optimized for symmetric matrices. However, this algorithm is substantially more complicated than the unsymmetric variant for the parallel case, and will be the subject of future work.

In order to effectively evaluate SpMV performance, we examine ten real matrices used in numerical calculations from

#	spyplot	Name	N	NNZ	Description
15		Vavasis	40K	1.6M	2D PDE Problem
17		FEM	22K	1M	Fluid Mechanics Problem
18		Memory	17K	125K	Motorola Memory Circuit
36		CFD	75K	325K	Navier-Stokes, viscous fllow
6		FEM Crystal	14K	490K	FEM Stiffness matrix
9		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 Programming	31K	1M	AA^T

Figure 6: Suite of matrices used to evaluate SpMV performance. Matrix numbers as defined in the SPARSITY suite are shown in the first column. The second column is a spyplot representing the density of nonzeros - clearly some matrices are very irregular, and some average very few nonzeros per row.

the BeBop SPARSITY suite [13,36] (four nonsymmetric and six symmetric). Figure 6 presents an overview of the evaluated matrices.

7.5 SpMV Results

Before we examine results, remember that Cell is doubly disadvantaged on this kernel: not only is double precision performance far less than single precision, but no autotuning has been implemented, and the current implementation is rather immature. Table 3 details SpMV performance for the various machines, precision and matrices. Note that a double precision nonsymmetric kernel was written and run on a 3.2 GHz Cell system to obtain performance numbers for both the symmetric and nonsymmetric matrices. The performance models show best performance after varying cache block size and parallelization strategy. Given Cell's inherent architectural limitations in the context of SpMV, it is surprising that it achieves nearly 4 Gflop/s (on average) in single precision and nearly 3 Gflop/s (on average) in double precision, indicating that the algorithm achieves a high fraction of memory bandwidth.

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 will be explored in future work.

As evidenced by the data, matrices that average few nonzeros per row (e.g. `memory`, `CFD`, `portfolio`, and `NASA`) perform poorly as the loop overhead, including mispredicted branches, can be very expensive. On the contrary, matrices with denser cache blocks perform well on Cell. We expect the segmented scan and BCSR approaches (currently under development) to improve performance in these instances.

As can clearly be seen when comparing Cell and Cell^{pm} performance in Table 3, the actual implementation which attempts to statically balance the matrix at runtime and does not rely on broadcasting, compares quite favorably with our performance model, which is perfectly balanced and exploits a broadcast operation. Once again DMA latency plays a relatively small role in this algorithm. In fact, reducing the DMA latency by a factor of ten causes only a 5% increase in performance. This is actually an encouraging result which indicates that 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 single precision performance. Solely increasing the clock frequency by a factor of 2 (to 6.4GHz) provides only a 6% increase in performance on the SPARSITY nonsymmetric matrix suite. Similarly, cutting the frequency in half (to 1.6GHz) results in only a 30% decrease in performance. Simply put, for the common case, more time is used in transferring the data than performing the computation.

Results from our performance estimator show that single precision SPMV is only 50% faster than double precision, even though the peak single precision performance is fourteen times that of double precision. However the fact that double precision nonzero memory traffic is about 50% larger indicates that much of the kernel time is determined by memory bandwidth rather than computation. This provides some hope that a symmetric kernel, although difficult to implement, will show significant benefits in single precision by cutting the memory traffic in half.

As seen in Table 3, the double precision Cell$^{pm}_{+}$ performance is only slightly faster than Cell^{pm} on the SpMV kernel, indicating that most cycles are not the double precision stall cycles, but rather other instructions as well as memory bandwidth cycles.

7.6 Performance Comparison

Table 3 compares Cell's performance for SpMV with results from the Itanium2 and Opteron using SPARSITY, a highly tuned sparse matrix numerical library, on nonsymmetric (top) and symmetric matrix suites. X1E results were gathered using a high-performance X1-specific SpMV imple-

	SPARSITY nonsymmetric matrix suite								
	Double Precision (Gflop/s)						Single Precision (Gflop/s)		
Matrix	Cell	Cell$^{pm}_{+}$	Cell^{pm}	X1E	AMD64	IA64	Cell^{pm}	AMD64	IA64
Vavasis	3.35	3.20	3.12	0.84	0.44	0.46	4.93	0.70	0.49
FEM	3.92	3.49	3.43	1.55	0.42	0.49	5.00	0.59	0.62
Mem	1.77	1.68	1.45	0.57	0.30	0.27	2.43	0.45	0.31
CFD	1.61	1.72	1.53	1.61	0.28	0.21	1.99	0.38	0.23
Average	**2.66**	**2.52**	**2.38**	**1.14**	**0.36**	**0.36**	**3.59**	**0.53**	**0.41**

	SPARSITY symmetric matrix suite								
	Double Precision (Gflop/s)						Single Precision (Gflop/s)		
Matrix	Cell	Cell$^{pm}_{+}$	Cell^{pm}	X1E	AMD64	IA64	Cell^{pm}	AMD64	IA64
FEM	4.23	3.66	3.62	—	0.93	1.14	5.39	1.46	1.37
3D Tube	3.26	3.63	3.59	—	0.86	1.16	5.30	1.36	1.31
Portfolio	1.46	1.78	1.55	—	0.37	0.24	2.35	0.42	0.32
NASA	1.61	1.86	1.60	—	0.42	0.32	2.56	0.46	0.40
Vibro	3.42	3.19	3.01	—	0.57	0.56	4.87	0.56	0.64
LP	3.48	3.48	3.43	—	0.47	0.63	5.20	0.55	0.92
Average	**2.91**	**2.93**	**2.80**	**—**	**0.60**	**0.67**	**4.28**	**0.80**	**0.83**

Table 3: SpMV performance in single and double precision on the SPARSITY (top) nonsymmetric and (bottom) symmetric matrix suites. Note: on Cell, the symmetric matrices were only run on a nonsymmetric kernel

mentation [7].

Considering that the Itanium2 and Opteron each have a 6.4GB/s bus compared to 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 7x faster in double precision for the nonsymmetric matrices. 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, avoiding loads of superfluous zeros. For example, in matrix #17, Cell uses more than 50% of its bandwidth loading just the double precision nonzero values, while the Itanium2 utilizes only 33% of its bandwidth. The rest of Itanium2's bandwidth is used for zeros and metadata. In this sample of symmetric matrices, OSKI was able to exploit BCSR to achieve good performance on the superscalar machines, and Cell is only four times faster. Of course this optimization could be applied to Cell to achieve even better performance. It should be noted that while runs 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 10 times more power efficient than the Itanium2 and more than 6 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, Cell performance on average is more than twice that of the best X1E kernel. The fact that the X1E consumes about 30% more power than Cell guarantees that Cell, in double precision, is three times as power efficient as the X1E.

7.7 Model Validation

As seen in Table 3, we evaluated our implementation of the double precision SpMV kernel on actual Cell hardware. The implementation makes blocking and partitioning decisions at run time, based on the lessons learned while exploring optimization strategies for the performance model. Although SpMV performance is ostensibly difficult to predict, our results clearly show hardware performance very close to the Cell^{pm} estimator. This builds confidence in our belief that Cell performance is far easier to predict than traditional superscalar architectures.

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 7.

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 two previous timesteps. Additionally, WaveToy

$$X_{next}[i,j,k,t+1] = X[i-1,j,k,t]+X[i+1,j,k,t]+ X[i,j-1,k,t]+X[i,j+1,k,t]+ X[i,j,k-1,t]+X[i,j,k+1,t]+ \alpha X[i,j,k,t]$$

$$X_{next}[i,j,k,t+1] = \frac{dt^2}{dx^2}(X[i-1,j,k,t]+X[i+1,j,k,t])+ \frac{dt^2}{dy^2}(X[i,j-1,k,t]+X[i,j+1,k,t])+ \frac{dt^2}{dz^2}(X[i,j,k-1,t]+X[i,j,k+1,t])+ \alpha X[i,j,k,t]-X[i,j,k,t-1]$$

Figure 7: 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.

has a higher computational intensity, allowing it to more readily exploit the Cell FMA pipeline.

8.1 Algorithmic Considerations

The basic algorithmic approach to update the 3D cubic data array is to sweep across the domain, updating one point at a time. The simplest implementation one might implement on Cell would be similar to how it would be implemented on a a cacheless vector machine. Basically read 7 streams of data (one for each each point in the stencil) and write one stream of data. This clearly doesn't exploit the inherit temporal locality in the method, and would result in 48 bytes of traffic for every 7 flops - a very poor computational intensity. A cache based machine can partially avoid this for problems where 6 pencils fit in cache, and more completely when 3 planes fit in cache.

8.2 Exploiting Temporal Locality

The algorithm used on Cell is behaves virtually identically 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 implementation sweeps through planes updating them one at a time. Since a stencil requires both the next and previous plane, to exploit a caching effect 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).

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 single precision 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 obviate this problem for the stencil kernel by choosing a problem size that is a multiple of the register size. That is, in double precision, the problem size must be a multiple of two, and in single precision, a multiple of four. This ensures that no more than two permutes are required for a registers worth of stencils.

8.3 Spatial Blocking and Parallelization

As each Cell SPE is a SIMD processor, it is necessary to unroll in the unit stride direction, and beneficial to unroll in the Y dimension (within the current plane). This two dimensional unrolling produces register blocks that share loads and permutes and thus help reduce total instructions issued. These register blocks are then extruded into ribbons, and thus planes are processed in ribbons.

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 about 256^2 elements plus ghost regions. Thus each SPE updates the core 256x32 points from a 258x34 slab (as slabs also contain ghost regions). The full blocking and parallelization procedure is shown in Figure 8

8.4 Blocking the Time Dimension

To improve performance of stencil computations on cache-based architectures, previous research has shown multiple time steps can be combined to increase performance [15, 18, 24, 38]. This temporal blocking 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 little increase in memory traffic, thus increasing computational intensity and improving overall performance. Figure 9 details a flow diagram for the heat equation, showing both the simple and temporally blocked implementations. Note: each block represents a plane at the specified coordinates in space and time. It should be emphasized that not all applications, even if they are stencil computations, can exploit this optimization.

8.5 Stencil Kernel Results

The performance for the heattut and WaveToy stencil kernels is shown in Table 4. 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 double precision, the heat equation is truly computationally bound for only a single time step, achieving over 7.1 Gflop/s. Analysis also shows that in the Cell+ approach, the heat equation is memory bound when using a single time step, attaining nearly 10 Gflop/s; when temporally blocked, performance of Cell+ double precision jumps to over 27 Gflop/s. In single precision, it is possible with temporal blocking to achieve an impressive 65 Gflop/s!

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

8.6 Performance Comparison

Table 4 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 [17].

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

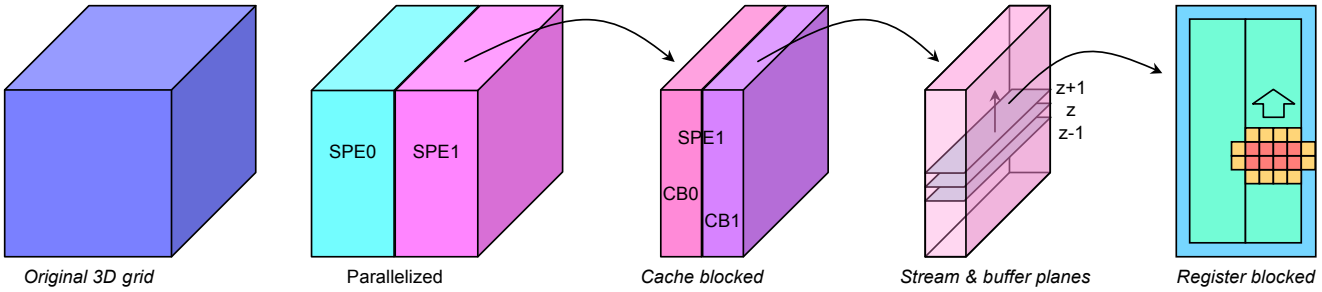


Figure 8: The original 3D grid is first parallelized among SPEs. Each SPE then partitions its block into several cache blocks (CB), so that multiple planes fit simultaneously in the local store. The planes are streamed into the double buffer within the local store. Once there, the cache blocked planes are further blocked into ribbons. Finally, each ribbon is register blocked, with a 4x2 core, surrounded in 3D by a ghost zone.

	Double Precision (Gflop/s)						
Stencil	Cell^{pm}_{+} (4 step)	Cell^{pm}_{+}	Cell	Cell^{pm}	X1E	AMD64	IA64
Heat	27.6	9.93	7.16	6.98	3.91	0.57	1.19
WaveToy	32.1	10.33	9.48	9.44	4.99	0.68	2.05

	Single Precision (Gflop/s)					
Stencil	Cell (4 step)	Cell^{pm} (4 step)	Cell^{pm}	X1E	AMD64	IA64
Heat	65.0	63.5	21.1	3.26	1.07	1.97
WaveToy	—	63.6	22.0	5.13	1.53	3.11

Table 4: 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 temporally blocked version where n time steps are computed.

Opteron respectively. Recall that Cell has only four times the memory bandwidth the scalar machines, and 75% the bandwidth of the X1E, indicating that Cell's potential to perform this class of computations in a much more efficient manner — 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 of single precision, Cell achieves a 1.8x, 6x, and 12x 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 nearly triple the performance in either single precision or in double precision on the Cell+ variant.

8.7 Model Validation

As with SpMV, we implemented an actual double precision kernel on a Cell blade with results shown in Table 4. The 3% increase in actual performance is from 2D register blocking which is not present in the performance model, and can be seen on both single double precision kernels. Overall, these results clearly show that Cell delivers highly predictable performance on the stencil kernel.

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 [27] and cache-based [8] 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. Our model corresponds well with IBM's [6] and Mercury's [11] implementations.

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 first evaluate 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. Note that 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.

One may now consider outer loop parallelism (multiple

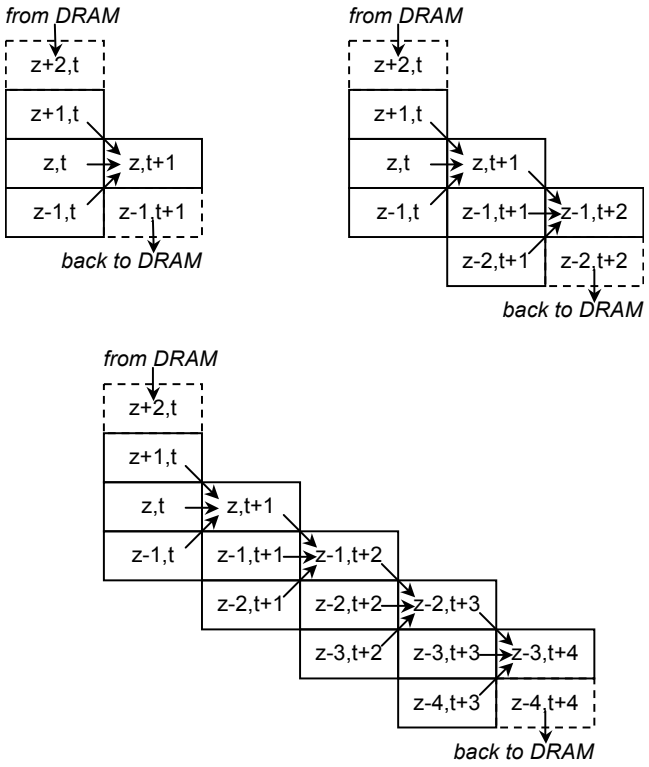


Figure 9: Flow Diagram for Heat equation flow diagram. Each rectangle represents a cache blocked plane. Top left: Without temporal blocking, only two queues are required within each SPE. Top right: Temporally blocked version for two time steps. Bottom: Temporally blocked for four time steps. For each phase, all queues from left to right are updated, and then rolled.

independent FFTs) that allows reuse of the roots of unity, and overlap of communication and computation. 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 then 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 FFT Results

Table 5 presents performance results for the Cell^{pm} 1D and 2D FFT; a Cell hardware implementation will be undertaken in future work. 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 single 4K point single precision 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 loading the roots of unity in order to reach this upper bound.

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.

When double precision 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. Thus, the single precision simultaneous Cell^{pm} FFT is 10 times faster than the double precision 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 single precision case. 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^{pm} for all problem sizes.

9.3 Performance Comparison

The peak Cell^{pm} FFT performance is compared to a number of other processors in the Table 5. These results are conservative given the naive 1D FFT implementation modeled on Cell whereas the other systems in the comparison used highly tuned FFTW [8] or vendor-tuned FFT implementations [28]. Nonetheless, in double precision, Cell^{pm} is at least 3x faster than the Itanium2 for a mid sized 1D FFT, and Cell$_+^{pm}$ could be as much as 150x faster for a large 2D FFT! Cell$_+$ more than doubles the double precision FFT performance of Cell for all problem sizes. Cell^{pm} 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 so long as the points fit within the local store, FFT performance on Cell improves as the number of points increases. 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 pre-

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

	N	Double Precision (Gflop/s)				
		Cell^{pm}_{+}	Cell^{pm}	X1E*	AMD64	IA64
1D	4K	12.6	6.06	2.92	1.88	3.51
	16K	14.2	6.39	6.13	1.34	1.88
	64K	—	—	7.56	0.90	1.57
2D	$1K^2$	15.9	6.67	6.99	1.19	0.52
	$2K^2$	16.5	6.75	7.10	0.19	0.11

	N	Single Precision (Gflop/s)				
		Cell^{pm}_{+}	Cell^{pm}	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 5: Performance of 1D and 2D FFT in double precision (top) and single precision (bottom). For large FFTs, Cell is more than 10 times faster in single precision 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.

cipitously once the associativity of the cache is exhausted and cache lines are 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 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 superscalar processors with set-associative caches since local store provides all of the benefits of 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 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 of Cell's performance on scientific kernels to date and directly compares performance to tuned kernels running on leading superscalar (Opteron), VLIW (Itanium2), and vector (X1E) architectures.

In the process of porting the scientific kernels to the Cell architecture, we introduce a number of novel algorithmic techniques to fully utilize unique features of the Cell architecture. First, we explore a partitioning and multi-buffering approach to facilitate parallelization and exploit temporal locality for the Stencil calculations on block-structured grids. This was expanded into a more general approach to aggregate multiple time-steps that complements prior work on time-skewing approaches. Finally, we introduced an efficient matrix partitioning strategy for SpMV that selects the appropriate cache blocking at run time, resulting in improved temporal and spatial locality as well as load balanced SPE computations.

Additionally, 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 that makes makes Cell programming time-consuming. The model proves to be quite accurate, because the programmer has explicit control over parallelism and data movement through the memory subsystem.

Our benchmark suite was selected to reflect a broad range of application characteristics, from the compute-intensive dense matrix multiply and FFT computations, to the memory-intensive sparse matrix vector multiply and stencil computations. Figure 10 summarizes the results, showing Cell's speedup over each of the evaluated platforms for both double and single precision. The graph shows the best implementation for each machine and is based on our most accurate data, which is our analytic model for FFTs and DGEMM (which closely matches other published results) and hardware runs on SPMV and Stencil.

Results show that only the Cray X1E supercomputer outperforms Cell processor in any setting, and only then for double precision DGEMM and 2D FFTs. The benefits of the Cell architectural model is clearly demonstrated for single-precision compute intensive problems where cell is 119-270x faster than conventional microprocessors for the $2K^2$ FFTs and 26-68x faster for SGEMM. The single-precision results for compute-intensive problems such as DGEMM highlight the performance benefits of a heterogeneous multicore approach, which gracefully improves the on-chip peak floating point computational capability relative to a conventional homogeneous multicore design.

Memory intensive problems such as SpMV and Stencil saw less dramatic performance benefits both in single and double precision. However the relative speedups of the memory intensive problems — which exceeded 7x for memory-bound problems such as SpMV — were far greater than the the 4x memory bandwidth advantage of Cell's XDR memory over the DDR memory in the AMD64 and IA64 systems (see Table 1). The Cell processor is nearly at parity with the X1E for double precision performance on the memory-intensive algorithms, which indicates that it is competitive with the vector approach to improving bandwidth utilization by hiding memory latency. These benefits were also apparent for the double-precision problems where, in spite of the handicapped double precision performance, its high memory bandwidth and explicit DMA capabilities gives the Cell a tremendous advantage on memory-intensive problems. This provides strong evidence that Cell's software controlled memory hierarchy is better able to exploit available memory bandwidth than conventional cache architectures, demonstrating that Cell's performance advantages are not derived exclusively from its high peak performance infrastructure.

To further explore the trade-offs in the Cell design, we proposed Cell+, a modest micro-architectural variant to Cell that improves double precision throughput. Its performance reflects that of the recently announced Cell BE2, which is anticipated to include fully pipelined double precision floating point capability. Table 6 compares the advantage of Cell and Cell+ in terms of absolute performance and power efficiency (and includes the raw data used in Figure 10). Again, these results represent actual hardware experiments

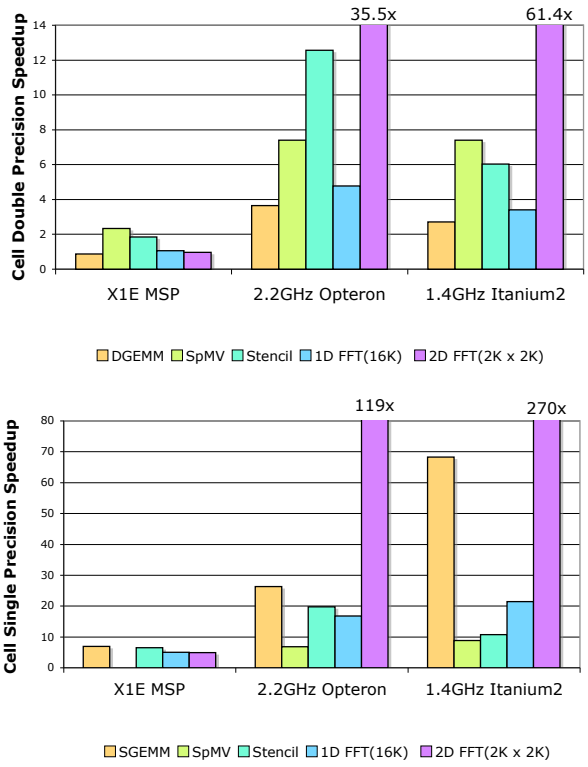


Figure 10: Cell's Speedup. Top: Cell's double precision speedup over the X1E, Opteron, and Itanium2. Even in double precision Cell is easily a match for an X1E MSP and more than a match for the Opteron and Itanium2. Bottom: Cell's single precision performance outshines the others.

DP Cell+	Speedup vs.			Power Efficiency vs.		
	X1E	AMD64	IA64	X1E	AMD64	IA64
DGEMM	3.02	12.8	9.46	3.63	11.4	12.3
SpMV	2.33	7.39	7.39	2.80	6.58	9.61
Stencil	2.54	17.4	8.34	3.05	15.5	10.9
16K FFT	2.32	10.6	7.55	2.78	9.43	9.82
$2K^2$ FFT	2.32	86.8	150	2.79	77.3	195

DP Cell	Speedup vs.			Power Efficiency vs.		
	X1E	AMD64	IA64	X1E	AMD64	IA64
DGEMM	0.86	3.65	2.70	1.04	3.25	3.51
SpMV	2.33	7.39	7.39	2.80	6.58	9.61
Stencil	1.83	12.6	6.02	2.20	11.2	7.82
16K FFT	1.04	4.77	3.40	1.25	4.24	4.42
$2K^2$ FFT	0.95	35.5	61.4	1.14	31.6	79.8

SP Cell	Speedup vs.			Power Efficiency vs.		
	X1E	AMD64	IA64	X1E	AMD64	IA64
SGEMM	6.94	26.2	68.2	8.33	23.4	88.7
SpMV	—	6.77	8.76	—	6.03	11.4
Stencil	6.47	19.7	10.7	7.77	17.6	13.9
16K FFT	5.00	16.7	21.4	6.00	14.9	27.8
$2K^2$ FFT	4.90	119	270	5.88	106	351

Table 6: Speedup and increase in power efficiency of double precision Cell+ (Top), Cell (Middle), and single precision Cell (Bottom) relative to the X1E, Opteron, and Itanium2 for our evaluated suite of scientific kernels. Results show an impressive improvement in performance and power efficiency. Note that stencil results do not include temporal blocking since a complementary kernel version was not available for the X1E, Opteron or Itanium2.

when available, and are otherwise based on our performance model. The Cell+ results confirm the potential of the Cell concept for computationally intensive problems: Across the benchmarks, Cell+ is 7x to 150x faster than the Opteron and Itanium processors and 2x faster than the Cray X1E. However, memory intensive codes such as SpMV derive virtually no benefit from the improved double precision performance and moderately memory-intensive problems such as Stencil and some of the FFTs derive a modest 2x performance improvement for a nearly 5x improvement in double-precision performance. Power efficiency results are similar, although the advantage grows for Cell+ relative to the more power-hungry Itanium processor.

Overall our analysis shows that Cell's three level software-controlled memory architecture, which completely decouples main memory accesses 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, large transfers (either a single long DMA or many smaller DMAs) can maintain a much higher percentage of memory bandwidth than individual loads in much the same way a vector load or 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 and is competitive with the vector approach to latency hiding. 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.

It is important to consider our performance results in the context of current trends in scientific applications and in hardware. On the application side, as problem sizes scale, computational scientists are moving away from computationally-intensive algorithms based on $O(n^3)$ computations like matrix multiply, and are increasingly relying on more scalable algorithms built from $O(n)$ computational kernels — such as variations on SPMV and Stencil. Thus, even without the Cell+ boost, the memory advantages of Cell offer real performance advantages, albeit at a significantly increased software development cost. The potential is significant enough to warrant more investment in software tools and techniques to reduce the software development overhead associated with Cell's novel architecture.

On the hardware side, there are increasingly prevalent trends towards homogeneous multicore commodity processors. The first generation of this technology has instantiated at most two cores per chip, and thus will deliver less than twice the performance of today's existing architectures. Our work demonstrates that the heterogeneous multicore

approach of the Cell architecture enables chip designers to instantiate eight simpler cores using the same generation process technology while deriving considerable performance advantages over the conventional multicore approach. The potential factor of 2x achieved by a homogeneous multicore is trivial compared with Cell+'s potential of a 10-20x improvement using the same technology generation.

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] A. Chow, G. Fossumm, D. Brokenshire A programming example: Large FFT on the Cell Broadband Engine 2005.

[7] 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.

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

[9] 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.

[10] 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.

[11] J. Greene, and R. Cooper. A Parallel 64K complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine Processor. In *Tech. Conf. Proc. of the Global Signal Processing Expo (GSPx)*, 2005.

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

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

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

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

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

[17] 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.

[18] S. Kamil, K. Datta, S. Williams, et al. Implicit and explicit optimizations for stencil computations. In *ACM Workshop on Memory System Performance and Correctness*, October 2006.

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

[20] 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.

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

[22] 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.

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

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

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] Y. Saad. *Iterative Methods for Sprarse Linear Systems*, PWS, Boston, MA, 1996.

[32] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI: The Complete Reference (Vol. 1). *The MIT Press*, 1998. ISBN 0262692155.

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

[34] 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.

[35] 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.

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

[37] S. Williams, J. Shalf, L. Oliker, et al. The Potential of the Cell Processor for Scientific Computing. In *Computing Frontiers*, 2006.

[38] 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.