Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L

S. Chatterjee
L. R. Bachega
P. Bergner
K. A. Dockser
J. A. Gunnels
M. Gupta
F. G. Gustavson
C. A. Lapkowski
G. K. Liu
M. Mendell
R. Nair
C. D. Wait
T. J. C. Ward
P. Wu

We describe the design of a dual-issue single-instruction, multiple-data-like (SIMD-like) extension of the IBM PowerPC® 440 floating-point unit (FPU) core and the compiler and algorithmic techniques to exploit it. This extended FPU is targeted at both the IBM massively parallel Blue Gene®/L machine and the more pervasive embedded platforms. We discuss the hardware and software codesign that was essential in order to fully realize the performance benefits of the FPU when constrained by the memory bandwidth limitations and high penalties for misaligned data access imposed by the memory hierarchy on a Blue Gene/L node. Using both hand-optimized and compiled code for key linear algebraic kernels, we validate the architectural design choices, evaluate the success of the compiler, and quantify the effectiveness of the novel algorithm design techniques. Our measurements show that the combination of algorithm, compiler, and hardware delivers a significant fraction of peak floating-point performance for compute-bound-kernels, such as matrix multiplication, and delivers a significant fraction of peak memory bandwidth for memory-bound kernels, such as DAXPY, while remaining largely insensitive to data alignment.

Introduction

Blue Gene*/L [1] is a massively parallel computer system under development at the IBM Thomas J. Watson Research Center and the IBM Engineering and Technology Services Group in Rochester, Minnesota, in collaboration with the Lawrence Livermore National Laboratory. The Blue Gene/L program targets a machine with 65,536 dual-processor nodes and a peak performance of 360 trillion floating-point operations per second (360 Tflops). It is expected to deliver previously unattainable levels of performance for a wide range of scientific applications, such as molecular dynamics, turbulence modeling, and three-dimensional dislocation dynamics.

This paper describes a hardware and software codesign to enhance the floating-point performance of a Blue Gene/L node, based on extensions to the IBM PowerPC* 440 (PPC440) floating-point unit (FPU) core [2], a high-performance dual-issue FPU. We recognized from the beginning that actual performance relies heavily on the optimization of software for the platform. Feedback from the software teams was instrumental in identifying and refining new extensions to the PowerPC instruction set to speed up target applications without adding excessive complexity. On the hardware side, we needed to double the raw performance of our FPU while still being able to connect it to the preexisting auxiliary processor unit (APU) interface of the PPC440 G5 central processing unit (CPU) core [3] under the constraints of working with the dual-issue nature of the CPU and keeping the floating-point pipes fed with data and instructions. On the software side, we needed tight floating-point kernels optimized for the latency and throughput of the FPU and a compiler that could produce code optimized for this unit.

This paper reports four major contributions: First, a minor modification of the PPC440 FPU core design produces a single-instruction, multiple-data-like (SIMD-

0018-8646/05/$5.00 © 2005 IBM

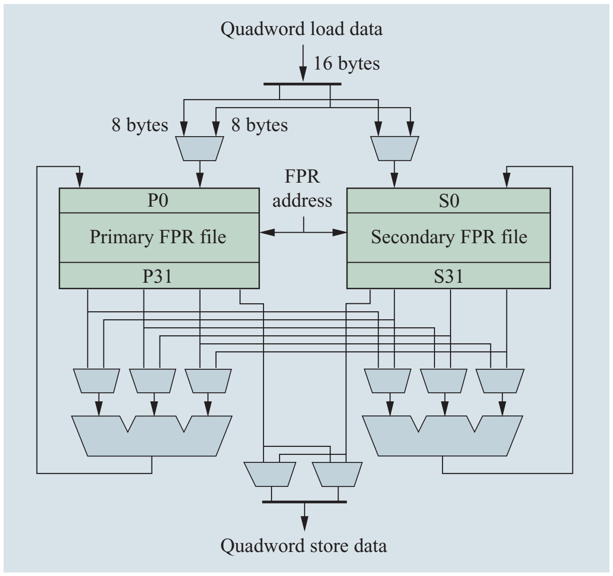

Figure 1

Architecture of the IBM PowerPC 440 FP2—primary and secondary data flow.

like) FPU with novel features at both the instruction-set architecture (ISA) and microarchitecture levels and doubles its performance. Second, the compiler code-generation algorithm incorporates several nontrivial extensions of the Larsen and Amarasinghe superword-level parallelism (SLP) algorithm [4]. Third, the algorithm design techniques explore innovative techniques to double the performance of key kernels while maintaining insensitivity to the alignment of the data. Finally, we document a concrete example of hardware and software codesign, showing how algorithmic requirements drove certain architectural decisions, how architectural choices constrained compiler optimization possibilities, and how algorithm design was creatively altered to work around limitations in both the architecture and the compiler. Some of the descriptions and results herein have been presented in [5].

The remainder of this paper is organized as follows. We describe the innovative features of the FPU and then describe an optimizing compiler targeting this unit. Following that, we present efficient algorithms for key linear algebraic kernels for this unit. We then discuss the evaluation of the level of success for both hand-optimized and compiled code for these kernels, and explore how close we can come to achieving twice the maximum theoretical performance of the PPC440 FPU. We conclude with an evaluation of the various design choices and a discussion of possible future extensions to the FPU architecture.

Architecture

The fused multiply–add (FMA) instruction, along with its variants, $T \leftarrow \pm(B \pm A \cdot C)$ [6, 7], is the workhorse of most modern FPUs. This single instruction delivers the equivalent of two floating-point operations. The PPC440 FPU core, which is capable of performing one FMA instruction per cycle while running at clock speeds in excess of 500 MHz, is considered to have a peak performance of more than a billion floating-point operations per second (i.e., one Gflops).

The embedded PowerPC Architecture*, referred to as Book E [8], allows for user-defined extensions to the ISA. Additionally, the APU interface on the PPC440 G5 core allows coprocessors to support new instructions—referred to as *APU instructions*—without requiring modifications to the CPU core [9]. While APU instructions typically do not become part of the architecture proper, they can still be utilized by assemblers and compilers that target the specific implementation.

To support multiple parallel executions and simultaneous loading (to avoid data starvation), we decided to pursue a SIMD-based approach. This would also have the advantage of reducing the size of the code footprint and the required bandwidth for instruction fetching. While SIMD instruction sets already exist [10, 11], including AltiVec**/VMX [10], which was specifically defined for PowerPC, these units operate primarily on integer and single-precision data. However, our target applications require double-precision data. Additionally, typical SIMD processors contain some sort of vector register file. Each vector register contains multiple elements, and each element, by default, occupies a fixed "slice" of the datapath. While this can be very efficient for simple elementwise calculations, it lacks the flexibility required by our workload.

The architecture of the PPC440 double-precision floating-point (FP2) core is shown in **Figure 1**. The design choice that we adopted goes beyond the advantages of adding another pipeline and of the SIMD approach. Instead of employing a vector register file, we use two copies of the architecturally defined PowerPC floating-point register (FPR) file. The two register files are independently addressable; in addition, they can be accessed jointly in a SIMD-like fashion by the new instructions. One register file is considered *primary*, the other *secondary*. The common register addresses used by both register files have the added advantage of maintaining the same operand hazard and dependency control logic used by the PPC440 FPU. The primary FPR is used in the execution of the preexisting PowerPC floating-point instructions and the new instructions, while the secondary FPR is reserved for use by the new instructions. This allows preexisting PowerPC

Table 1 SIMOMD FMA instructions representing various classes of operations, their semantics, and their bindings to C99 built-in functions.

FMA instruction class	*Mnemonic*	*Operation*	*C99 built-ins*
Parallel	`fpmadd fT, fA, fC, fB`	$P_T = P_A \cdot P_C + P_B$ $S_T = S_A \cdot S_C + S_B$	$T = \_fpmadd(B,C,A)$
Cross	`fxmadd fT, fA, fC, fB`	$P_T = P_A \cdot S_C + P_B$ $S_T = S_A \cdot P_C + S_B$	$T = \_fxmadd(B,C,A)$
Replicated	`fxcpmadd fT, fA, fC, fB`	$P_T = P_A \cdot P_C + P_B$ $S_T = P_A \cdot S_C + S_B$	$T = \_fxcpmadd(B,C,a_p)$
Asymmetric	`fxcpnpma fT, fA, fC, fB`	$P_T = -P_A \cdot P_C + P_B$ $S_T = P_A \cdot S_C + S_B$	$T = \_fxcpnpma(B,C,a_p)$
Complex	`fxcxnpma fT, fA, fC, fB`	$P_T = -S_A \cdot S_C + P_B$ $S_T = S_A \cdot P_C + S_B$	$T = \_fxcxnpma(B,C,a_s)$

instructions—which can be intermingled with the new instructions—to operate directly on primary side results from the new instructions, adding flexibility in algorithm design. We describe the use of the new instructions in the section on DAXPY. New move-type instructions allow the transfer of results between the two sides.

Along with the two register files, there are also primary and secondary pairs of datapaths, each consisting of a computational datapath and a load/store datapath. The primary (secondary) datapath pair write their results only to the primary (secondary) register file. Similarly, for each computational datapath, the *B* operand of the FMA is fed from the corresponding register file. However, the real power comes from the operand crossbar, which allows the primary computational datapath to receive its *A* and *C* operands from either register file (see Figure 1). This crossbar mechanism enabled us to create useful operations that accelerate matrix and complex-arithmetic operations. The power of the computational crossbar is enhanced by cross-load and cross-store instructions that add flexibility by allowing the primary and secondary operands to be swapped as they are moved between the register files and memory.

New APU instructions

The newly defined instructions include the typical SIMD parallel operations as well as cross, asymmetric, and complex operations. **Table 1** shows a few examples. The *asymmetric* instructions perform different but related operations in the two datapaths, while the *complex* operations include symmetric and asymmetric instructions specifically targeted to accelerate complex-arithmetic calculations. We call these new types of asymmetric instructions *SIMOMD*, for single-instruction, multiple-operation, multiple-data.

The asymmetric and complex instructions enable efficient arithmetic on complex numbers and the enhancement of fast Fourier transform (FFT) and FFT-like code performance. The cross instructions (and their memory-related counterparts, cross-load and cross-store) help to efficiently implement the transpose operation and have been highly useful in implementing some of our new algorithms for basic linear algebra subroutine (BLAS) codes that involve novel data structures and deal with potentially misaligned data. Finally, the parallel instructions with replicated operands allow important scientific codes that use matrix multiplication to make more efficient use of (always limited) memory bandwidth.

The FP2 core supports parallel load operations, which load two consecutive doublewords from memory into a register pair in the primary and the secondary unit. Similarly, it supports an instruction for parallel store operations. The PPC440 processor local bus supports 128-bit transfers, and these parallel load/store operations represent the fastest way to transfer data between the processor and the memory subsystem. Furthermore, the FP2 core supports a parallel load-and-swap instruction that loads the first doubleword into the secondary unit register and the second doubleword into the primary unit register (and its counterpart for store operation). These instructions help the matrix transpose kernel more efficiently.

IEEE Standard 754 compliance issues

The new APU instructions treat floating-point exceptions as disabled and do not update the PowerPC floating-point status and control register (FPSCR). We believe

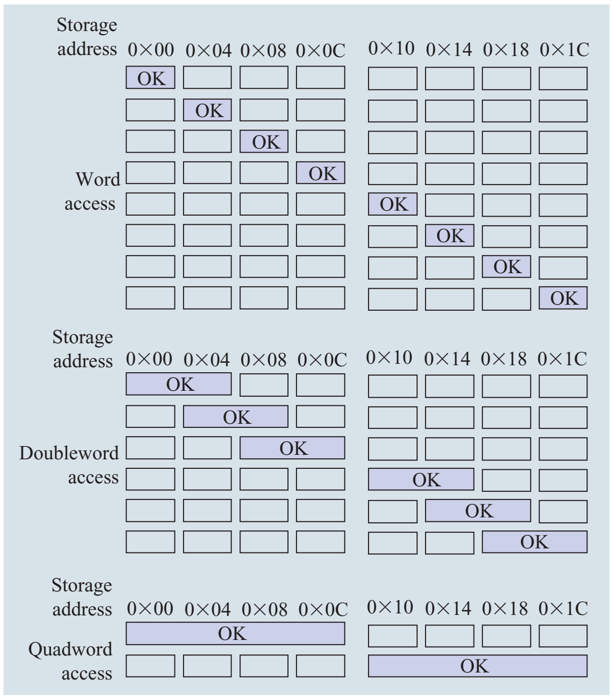

Figure 2

Alignment restrictions.

that the default handling of exceptions, as defined in IEEE Standard 754 [12], is sufficient for the needs of our target applications. Additionally, the complexity of handling enabled exceptions or updating the FPSCR for SIMD-like instructions appeared to outweigh potential benefits. Thus, the PPC440 FP2 core is consistent with IEEE 754, although not strictly compliant. In addition, the new instructions can operate in a non-IEEE mode if an application does not require strict IEEE 754 compliance for denormalized numbers.

The PPC440 FP2 core contains parallel instructions for generating a reciprocal estimate and a reciprocal square-root estimate. These instructions produce an estimate with a relative error of 2^{-13}. While estimate instructions, by their very nature, are not IEEE-compliant, the enhanced accuracy of these estimates allows coders to use a small number of Newton–Raphson-like iterations to obtain properly rounded IEEE-compliant results.

Implementation issues

The dual-issue nature of the CPU allows the initiation in each cycle of a quadword (i.e., two doublewords) load in parallel with two FMAs, yielding a peak performance of four floating-point operations per cycle. Latency and throughput for the new instructions are the same as that for the similar PowerPC floating-point instructions in the PPC440 FPU. In typical cases, most of the instructions have a single-cycle throughput. The associated typical latencies are five cycles for computational (except for divide) and move instructions, two cycles for instructions that update the CPU condition register, three cycles for stores, and four cycles for loads. Also, as in the PPC440 FPU, there is no hardware register renaming.

To avoid excessively depleting the opcode space allocated for APUs, the unit performs only double-precision arithmetic operations. This does not affect performance because the latency of all instructions (except for divide) is precision-independent. Load/store operations convert single-precision operands to double-precision as they enter the unit. Thus, applications that can get by with the reduced precision of single-precision raw data can consume less of the overall memory bandwidth of the system.

To further economize on opcodes, we chose to limit the permutations of operand swapping. For example, the computational crossbar in Figure 1 allows 16 variations on the parallel-multiply instruction by permuting on the source of the *A* and *C* operands (i.e., `A[p|s]*C[p|s]` → `Tp`; `A[p|s]*C[p|s]` → `Ts`). However, analysis of our target algorithms revealed that they generally required only the four basic variations of permuting on the source of the *A* operands (i.e., `A[p|s]*Cp` → `Tp`; `A[p|s]*Cs` → `Ts`). The function of the other permutations could be achieved through careful coding, often without performance penalty. For example, the permutations on the *C* operand could be achieved by using the four defined opcodes and interchanging the roles of the *A* and *C* operands. While it could certainly be argued that allowing a more general sort of "shuffling" ability would ease the burden on the programmer and compiler, it would make the chip more complex and difficult to verify. Thus, the aforementioned interchanges were agreed to as a codesign compromise that allowed enough flexibility without burdening the architecture with too much additional complexity.

The load/store datapath pipes allow single-precision or double-precision data to be transferred between memory and the primary or secondary FPR file. Fortunately, the PPC440 G5 core is able to support the loading or storing of quadword operands in a single cycle. This allowed us to define instructions to simultaneously transfer doubleword data to or from both the primary and secondary register files. For efficient transfer, data must be word-aligned and must fit completely within a quadword-aligned quadword, as shown in **Figure 2**. This restriction exists because the cache architecture is such that it can transfer data from either half of the cache line. Each half is a quadword. Misalignments require that a trap handler artificially align the data within a half line and then perform the access again. Thus, the penalty for misaligned data accesses, done in a naive manner, can be of the order of thousands of processor cycles. The sections which follow on alignment handling and DAXPY discuss how we proactively avoid this situation in the compiler and at algorithm design time; we then present empirical evidence that the penalty for such accesses can be greatly reduced.

One compromise in the PPC440 FP2 core design is the lack of "D-form" load/stores. The D-form instructions generate an effective address by adding the contents of a general-purpose register (GPR)—or zero—to a 16-bit immediate value. Unfortunately, the only way to fit this immediate value in the opcode is to use the portion of the instruction normally used to hold the secondary opcode. While the 16-bit immediate values add considerable flexibility, each D-form operation—by virtue of reserving all ten secondary opcode bits—consumes the space of 1,024 opcodes. Since we wanted to leave opcodes for other APUs, we decided to forego these instructions. Instead, we support indexed instructions, in which the effective memory address is determined by the sum of two GPRs or the value in a single GPR. The compiler team felt that the lack of D-form load/stores would affect performance because of the requirement for extra integer registers to hold the displacements. This hypothesis was tested by forcing the compiler to generate X-form load/stores for all floating-point load/stores for a SPEC2000 benchmark suite floating-point (FP) run at `-O3` on a PowerPC POWER4* machine. The results showed an average of 5.5% decrease in SPEC2000 FP performance. This result is worse than a PPC440 FP2 core would experience, since primary load/stores could still use D-form load/stores.

Compilation

Code generation for the PPC440 FP2 core is done within the TOBEY (Toronto Optimizing Back End with Yorktown) back end [13] of the IBM XL family of compilers for Fortran, C, and C++. TOBEY has been enhanced to schedule instructions for the PPC440 FPU core and to generate parallel operations for the PPC440 FP2 core using extensions to the SLP algorithm of Larsen and Amarasinghe [4].

Generation of parallel code is done late in TOBEY, just before scheduling and register allocation. The SLP algorithm works within a basic block. Consecutive load/stores are paired up by matching base registers and displacements, and use–def chains[1] are used to find additional candidate instructions for pairing. Each candidate pair is evaluated to see whether generating the paired instruction is more efficient than the sequential instructions. A paired instruction is considered more efficient if it requires no extra moves to put the operands into the correct registers. For each instruction, the estimated benefit is incremented if the operand is known to be in the correct primary or secondary register (because previously generated instructions have placed it there) or if it is unknown (the register allocator will allocate it properly). Operands known to be in incorrect registers decrease the estimated benefit.

Each instruction may appear in only one parallel instruction. Our implementation generates all viable instruction pairs and then uses a Briggs coloring algorithm [14] to find the sets of paired instructions for which no instruction is in more than one pair. The benefit for each set of pairs is then recalculated, and the set with the largest benefit is selected. This is repeated until no more paired instructions can be found in a block.

At this point, the original SLP algorithm combines paired instructions to form larger sets, but this step can be skipped for the PPC440 FP2 core because it has only two-way SIMD parallelism. Instructions are then scheduled to find a consistent ordering. The implementation of the SLP algorithm is somewhat complicated by the existence of asymmetrical instructions in the PPC440 FP2 core instruction set. These introduce more possible instruction pairs, complicating generation and estimation of the benefits of each instruction. An example of this is the `fxcsmadd` instruction. It can be replaced by an `fxcxma` instruction by swapping the *C* operands. The benefit of the instruction is calculated for both variants, and the better one is used.

In addition, the C and Fortran front ends have been enhanced with built-in functions for generating the PPC440 FP2 core parallel instructions, exploiting the `complex*16` type in Fortran and `double_Complex` type in C99. The built-in functions are unavailable in C++ because that language does not support a built-in complex data type. The section on algorithms contains an example of the use of these facilities.

Alignment handling

In the preceding section on implementation issues, we discuss the significant overhead of misaligned accesses due to the high penalty of alignment traps on this architecture. We have taken a multilevel approach to avoid alignment trap penalties. The first approach is to optimize data layout to maximize occurrences of aligned accesses. Specifically, TOBEY maps data of size 16 bytes or more on a 16-byte-aligned boundary for all stack locals and externally defined data. It also provides a special `malloc` routine that returns 16-byte-aligned memory from the heap.

The second approach is to avoid alignment traps as much as possible. Since alignment trap overhead definitely nullifies any performance gains from

[1]Use–def and def–use chains are a standard data structure in optimizing compilers.

performing parallel load/stores, the compiler generates parallel memory instructions only if the memory to be accessed is known to be 16-byte-aligned. Therefore, utilization of FP2 parallel memory instructions depends heavily on the availability of accurate alignment information. For local variables and external variables defined in the current compilation unit, the alignment information can be directly computed in the back end. In the case of pointers, however, it is very difficult for the back-end compiler to determine the alignment. To address this problem, we implemented three solutions:

- We provided a built-in function for users to assert the alignment of a given address expression at a specific location in the program. The assertion is defined as follows:

  `void _alignx (int alignment, const void * ptr).`

  The compiler can then utilize user-provided alignment information to determine whether to generate parallel load/store instructions. **Figure 3(a)** gives an example of its use.
- We have implemented an interprocedural alignment analysis to track the alignment of pointers. This phase is implemented in the product loop-level optimizer, called the Toronto Portable Optimizer (TPO), to leverage the existing interprocedure and pointer-analysis framework in TPO. TPO then passes the collected alignment information to the back end through the `alignx` assertion.
- When the alignment of some accesses cannot be determined at compiletime, we resort to runtime alignment testing by creating two versions of a loop, with one version guarded under the conditions that all accesses with unknown alignments are 16-byte-aligned. The back end may then be able to generate parallel load/stores for this version. The versioning transformation is implemented in TPO and for loops only. Since there are both performance and code size overheads associated with loop versioning, extra care has been taken to avoid excessive and nonprofitable loop versioning.

Optimization of reductions
The original SLP algorithm is unable to pair instructions that have a true dependence between them, such as an FMA chain for sum reduction. We can break the true dependencies between every set of two instructions if we add temporary registers, facilitating the SLP optimization to pair them up. When the SLP algorithm discovers a true dependence between two isomorphic instructions, TOBEY transforms them as described above if they are part of a reducible chain. This optimization is performed only if strictness is disabled for the procedure. TOBEY currently detects chains of FMA, FP negate multiply subtract (`fnms`), FP multiply subtract (`fms`), and FP negate multiply add (`fnma`), pairing them up by using their parallel equivalents. In the case of `fms` and `fnma` chains, the data in the array must be aligned because they require a cross move between the primary and secondary registers, costing an extra operation. For the reduction to be beneficial, parallel loads on the data are needed.

Register allocation
The challenges in register allocation relate to register pairing in parallel instructions. The TOBEY intermediate representation uses an infinite supply of symbolic registers that are then colored to use a finite set of real registers. The secondary FP registers are treated the same as the primary ones. The instruction descriptions enforce the pairing of one primary with the corresponding secondary operand. For example, a parallel load `LPFL fp500, fp501=Memory (. . . .)` indicates that the first two operands are paired; `fp500` must be a primary register and `fp501` must be secondary. Both symbolic registers are also recorded as an aligned pair.

The register-coloring mechanism, based on the Briggs register-coloring algorithm [14], is modified to allow coloring of register pairs. Additional nodes are added to the interference graph, one for each real hardware register. In the building of the interference graph, each symbolic register used where a primary (secondary) register is required has an interference edge added to all real hardware registers that are not valid primary (secondary) registers. These new interferences restrict the symbolic registers to their corresponding register subset.

Following the construction of the interference graph, nodes (representing registers) are removed from the graph one by one, starting with the lowest-degree nodes. As long as there is a node with a degree smaller than the number of physical registers, it is guaranteed to color and can be reduced. If there is none, a heuristic is used to select a node to spill. The removed or spilled node is put on top of the reduction stack. To help in assigning colors for pairs, reduced registers that are paired are set aside until their partner register is reduced before they are put on the reduction stack. Once both members of the pair are reduced, the two registers are pushed onto the reduction stack together.

When the interference graph is completely reduced, registers are popped off the reduction stack and assigned colors. In the modified algorithm, the hardware registers are assigned colors first in order. This assigns each color a primary or secondary attribute and matches each color with its aligned partner. As the symbolic registers are popped off the reduction stack, they are assigned colors according to the Briggs algorithm. A symbolic register

```
static void _daxpy1(double a, double *x, double *y, int n) {
__alignx(16, x); __alignx(16, y); /* computation elided */
}

static void _daxpy2(double a, double *x, double *y, int n) {
__alignx(16, x+1); __alignx(16, y); /* computation elided */
}

void daxpy1(double a, double *x, double *y, int n) {
  int x_is_aligned, y_is_aligned, i;
  for (i = 0; i < n%8; i++) y[i] = a*x[i]+y[i];
  x += n%8; x_is_aligned = ((x & 0x0000000f) == 0)?1:0;
  y += n%8; y_is_aligned = ((y & 0x0000000f) == 0)?1:0;
  n -= n%8;
  switch (2*x_is_aligned + y_is_aligned) {
  case 3: _daxpy1(a, x, y, n); return;
  case 1: _daxpy2(a, x, y, n); return;
  case 2:
    y[0] = a*x[0] + y[0];
    _daxpy2(a, x+1, y+1, n-8); /* x+1 unaligned, y+1 aligned */
    for (i = 0; i < 7; i++) (y+n-7)[i] = a*(x+n-7)[i]+(y+n-7)[i];
    return;
  case 0:
    y[0] = a*x[0] + y[0];
    _daxpy1(a, x+1, y+1, n-8); /* x+1 aligned, y+1 aligned */
    for (i = 0; i < 7; i++) (y+n-7)[i] = a*(x+n-7)[i]+(y+n-7)[i];
    return;

  }
}
```

(a)

```
function Cout = DATB( m1, n2, k1, AR, BC, CC )
% m1, n2, and k1 are integer multiples of m0, n0, and k0.
% CReg, AReg, and BReg represent 16, 4, and 4, registers
% used to hold submatrices of C, A, and B, respectively.
m0 = 4; n0 = 4; k0 = 1;
for j=1:n0:n2
  for i=1:m0:m1
    CReg = CC( i:i+m0-1, j:j+n0-1 );   % load submatrix of C
    for p=1:k0:k1
      AReg = AR( p:p+k0-1, i:i+m0-1 ); % load panel of A
      BReg = BC( p:p+k0-1, j:j+n0-1 ); % load panel of B
      CReg = AReg' * BReg + CReg;
    end
    CC( i:i+m0-1, j:j+n0-1 ) = CReg;   % store submatrix of C
  end
end
Cout = CC;
```

(b)

Figure 3

(a) Driver routine for DAXPY. The routines _daxpy1 and _daxpy2 assume that the length of their input vectors is a multiple of 8 to simplify unrolling and software pipelining, and terminate early if $n \leq 0$. (b) MATLAB** prototype matrix-panel kernel for level L1, $C = A^T \cdot B + C$.

that is a member of a pair is popped off the reduction stack with its partner. The two registers are then assigned colors together, thus ensuring a valid color pair.

Another challenge is combining two spill load/stores into a parallel load/store. Intermediate spill instructions with compatible paired registers are combined first; only afterward is space dedicated in the spill area. An execution count estimate is used to prioritize combining of these spill locations.

The remaining challenge is to rematerialize parallel loads as individuals. Memory locations determined as constant are added to the rematerialization table. A parallel load, such as in the previous example, also generates two individual entries (`fp500` and `fp501`) using scalar load instructions. With these additional entries, these two registers can be rematerialized individually if needed.

Future work

The PPC440 FP2 core provides parallel load instructions that can load successive floating-point values into the primary and secondary registers either as <primary,secondary> or <secondary,primary> pairs. This decision must be made in the first phase of the SLP algorithm and will change the estimated savings benefits for future pairs of instructions. For example, one choice may prevent parallel code from being generated. An algorithm to estimate the impact of each operation (load) choice must be employed.

The current SLP algorithm generates code independently for each basic block. The SLP algorithm must be enhanced to support extended basic blocks. This may allow more parallel instructions to be generated.

Another important method for generating FP2 instructions is to exploit loop-level SIMD code-generation techniques (*SIMDization*). Compared with the SLP algorithm, loop-level SIMDization keeps information in a more compact form and is more deterministic than the greedy-based packing approach used by SLP. Another advantage of loop-level SIMDization is its ability to handle misaligned accesses. We have recently proposed a general framework to SIMDize loops with arbitrary compiletime and runtime misalignment [15]. This framework has been implemented in TPO for VMX extensions for PPC970 processors. We are in the process of retargeting this framework for the PPC440 FP2 core.

Algorithms

This section discusses several important ideas in the design of high-performance BLAS that exploit the unique characteristics of the PPC440 FP2 core while steering clear of potential pitfalls that carry a large performance penalty. Here, we restrict our discussion to two examples: matrix multiplication, a key level 3 BLAS [16] kernel whose data reuse allows near-peak floating-point performance to be reached, and DAXPY, a representative level 1 BLAS [17] kernel in which performance is limited by the memory bandwidth, and data alignment becomes important.

Matrix multiplication: $C \leftarrow \alpha C + \beta A \cdot B$

Traditionally, high-performance matrix-multiplication algorithms have involved the use of a kernel routine that takes advantage of the low-latency, high-bandwidth L1 cache. Several approaches to blocking for this routine at higher levels of memory have been published [18–21]. For example, if the A matrix is of size $M \times K$, a blocking of this matrix yields submatrices (blocks) that are of size $MB \times KB$. For the L1 level of memory, our model indicates that one should load most of the L1 cache with either the A or the B matrix operand. The other operands, C and B or A, are respectively streamed into and out of (through) the remainder of the L1 cache, while the large A or B operand remains consistently cache-resident. Our kernel places the B operand, of size $KB \times NB$, in the L1 cache, filling most of the cache. However, we can stream $m_1 = M/MB$ blocks of size $MB \times NB$ and $MB \times KB$, of C and A, respectively, through the L1 cache. We observe two practical benefits from our form of kernel construction and usage: First, a rectangular blocking where $NB < KB$ leads to a higher computational rate because of the inherent asymmetry that results from the fact that we have to load *and* store C. Second, the streaming feature allows a factor of M/MB fewer invocations of the kernel routine.

In theory: The PPC440 FPU core L1 kernel

We sketch how to optimally implement a matrix-multiply kernel for $C = C + A \cdot B$ under the assumptions mentioned above. Only a register block, a panel, and a resident matrix will occupy the L1 cache during a given time interval. See **Figure 3(b)** and [6, 19, 22] for further details.

The amount of memory at the L0 level corresponds to the number of registers. Let an $m_0 \times n_0$ block occupy "most" of the registers and be used to hold an $m_0 \times n_0$ submatrix of C, while the remainder of the registers are used to stream in row/column elements of A and B. This requires $m_0 \cdot n_0 + m_0 + n_0$ FPRs. Since registers can be loaded more efficiently from contiguous memory, traditionally this has meant "preparing A," stored by column (the result is A^T), by storing it by row and computing $C = C + (A^T)^T \cdot B$. To take advantage of the prefetching abilities of the L2 cache and not exceed its seven-stream limit, we prepare both the A and B matrices. This results in utilizing no more than three streams per core.

The pattern of matrix multiplication requires that we access rows of A and columns of B. Since maximizing the number of independent dot products in the inner kernel dictates the use of an outer-product matrix-multiplication algorithm at the register level, the format of these two submatrices is prescribed. We reformat (part of) the A matrix in a "column-internal, row-external" format. This means that we store some number (m_0, above) of the rows of A consecutively (row-external), and within that row-block, we store the elements in column-major format (column-internal). When m_0 is 1, this is the same as row-major format. For analogous reasons, we store (part of) the B matrix in row-internal, column-external format. In most cases, the elements of C are not reformatted, but may be rearranged in the register file. This approach leads to an access pattern utilizing two L2 cache streams per core and, in some exceptional cases, a third stream is consumed by C.

The PPC440 unit has 32 FPRs, and it is capable of performing one load/one store and one FMA, in parallel, in one cycle for data residing in the L1 cache. As shown in Figure 3(b), the inner kernel is easy to understand at a high level. In practice, loads, calculations, and stores are staggered (i.e., preloads are performed) in such a manner as to allow the floating-point pipeline to "flow" efficiently. Thus, it follows that $m_0 = n_0 = 4$ is a good choice because this utilizes 24 FPRs.

Figure 3(b) gives a simplified prototype MATLAB[2] implementation of our L1 kernel algorithm: 16 FPRs (`T00-T33`) are needed to hold a 4×4 block of C, `CC(i: i+3, j: j+3)`. Its inner loop consists of four loads of A (`AR(p, i: i+3)`) into four FPRs (`A0A3`), four loads of B (`BC(p, j: j+3)`) into four FPRs (`B0-B3`), and 16 independent dot-product operations (FMAs); for example,

```
T12 = T12 + A1 * B2
```

where `T12` holds `CC(i+1, j+2)`, `A1` holds `AR(p, i+1)` `(A(i+1, p))`, and `B2` contains `BC(p, j+2)`.

Note that the use of $k_0 = 1$ is a simplification. To minimize the impact of latency, k_0 is kept greater than 1 so that the time between the load and use of the registers holding the A and B submatrices can be increased.

In practice: The PPC440 FP2 core L1 kernel

Given the nature of the architecture, we want to use the PPC440 FP2 core as a SIMD vector machine of vector length 2. Conceptually, this can be realized by performing $2 \times 2 \times 1$ submatrix computations on 2×2 submatrices. Let us choose the A matrix to hold vectors of length 2. The B matrix will also hold vectors of length 2; however, the components of B will be used as scale factors. A simple example will clarify what we have in mind.

We compute the first column of

```
C(i: i+1, j:j+1) =
                A(i:i+1, l:l+1) * B(l:l+1, j:j+1)
```

as

$$\begin{bmatrix} a(i, l) \\ a(i+1, l) \end{bmatrix} b(l, j) + \begin{bmatrix} a(i, l+1) \\ a(i+1, l+1) \end{bmatrix} b(l+1, j)$$

and calculate the second column in a similar fashion, using the two scalars, `b(l: l+1, j+1)`.

To respect the cache-line size, we have designed our inner kernel as two k-interleaved $4 \times 2 \times 8 (m_0 \times k_0 \times n_0)$ matrix multiplications. Even though this architecture exhibits extremely good latency characteristics, we load the non-cache-resident matrix, A, in 4×4 "chunks." This yields a latency tolerance greater than 40 cycles for elements of A and greater than 20 cycles for elements of B, the cache-resident matrix. Since the cold load time from the L2 cache to the registers is no more than 18 cycles, this should allow the inner loop of our algorithm to proceed at the peak rate of the machine. The extra latency tolerance is built into the algorithm so that it covers most of the L3 latency for the nonresident A matrix. The 20-cycle latency coverage for the cache-resident matrix is advisable, because the L2 prefetch stream dictates that the first load of a column block of B will result in an L2 miss, and we wish to tolerate that as well.

In practice: The PPC440 FP2 core L3 kernel

Unfortunately, as can be seen in the performance graphs in the results section, while this algorithm results in good performance, there is room for improvement. While the design is generally sound, there is one fairly obvious problem. One invocation of the $4 \times 2 \times 8$ update requires 32 cycles for computation as well as the loading of 12 dual registers with A and B. When the B operands are in L1, this presents no problem. However, the bandwidth from L3 is 5.3 bytes per cycle, and this implies a 36-cycle requirement for operand loading, resulting in an update that proceeds at 89% of peak. Because the L1 cache replacement policy is not *least recently used* (LRU), this unfortunate eviction of B occurs often enough to affect performance.

Fortunately, the large register set and low latency of this architecture allows us to code around this problem by blocking for and targeting the L3 cache. Changing the inner kernel update to a $6 \times 2 \times 6$ format yields an inner kernel that requires 36 cycles for computations and 36 cycles for operand loading from L3. The efficacy of this approach is shown by data presented in the results section.

[2]Notice that the redundant data copying often required by call-by-value MATLAB can easily be avoided when the kernel is implemented in C, Fortran, or assembly code.

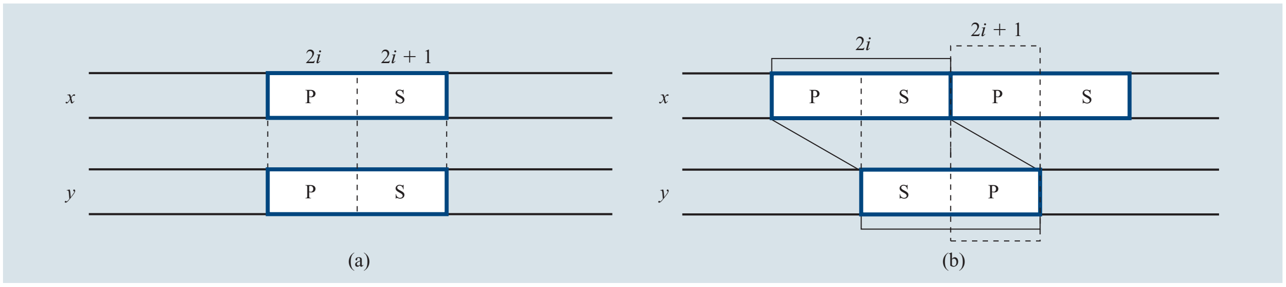

Figure 4

Alignment possibilities in DAXPY: (a) Vectors x and y are both quadword-aligned. (b) Vector x is misaligned, while y is quadword-aligned.

DAXPY: $y \leftarrow ax + y$

The BLAS1 routine DAXPY adds to vector y a multiple of vector x. In contrast to the matrix-multiplication routine, there is no reuse of the vector arguments, and the computation of a single element of the result involves two loads, an FMA and a store. Performance is therefore limited by the speed at which data can be transferred between the memory hierarchy and the register file, and it is therefore important to use quadword load/stores. Given the substantial penalty for unaligned quadword accesses, the overriding issue in gaining efficiency for DAXPY is that of being able to maintain quadword access to and from memory regardless of the alignment of the x and y vectors (assuming that these are stored contiguously in memory).

Depending on the alignments of vectors x and y, there are four possibilities. When both vectors are quadword-aligned, it is trivial to perform quadword load/stores and SIMD FMAs. If both vectors are misaligned, the first element is "peeled," thereby aligning the remaining subvectors. This situation is illustrated in **Figure 4(a)** and represented by the routine `_daxpy1` in Figure 3(a).

The more interesting case arises when one vector is quadword-aligned while the other is not. This situation can arise, for instance, with successive rows of a C array with an odd number of columns, or successive columns of a Fortran array with an odd-sized leading dimension. This case is illustrated in **Figure 4(b)** and called `_daxpy2` in Figure 3(a). Assume, without loss of generality, that vector y is aligned and vector x is not; then vector elements y_{2i} and y_{2i+1} are computed as follows:

1. Assume that registers P0 and S0 contain x_{2i-1} and x_{2i} .
2. Load y_{2i} into register S1 and y_{2i+1} into register P1 using a *cross-load.*
3. Perform an `fxcpmadd` operation: P3/S3 $= \alpha \cdot$ P0/S0 $+$ P1/S1. Register S3 contains $\alpha x_{2i} + y_{2i}$, while register P3 contains junk.
4. Load x_{2i+1} and x_{2i+2} into registers P0 and S0 using a parallel load. (This operation also sets up the required precondition for $i + 1$.)
5. Perform a scalar FMA: P3 $= \alpha \cdot$ P0 $+$ P1. Register P3 now contains $ax_{2i+1} + y_{2i+1}$.
6. Store y_{2i} and y_{2i+1} to memory from register pair P3/S3 using a *cross-store*.

Thus, we have exploited the inherent 3:1 imbalance in memory accesses to FMAs in this operation to perform redundant floating-point operations while operating at peak cache bandwidth. With four-way loop unrolling and software pipelining, these operations can be scheduled to initiate a quadword load or store at each cycle. Note the critical use of the cross-loads and cross-stores on the y vector and the use of redundant computation to compensate for the relative misalignment between x and y. More elaborate versions of this technique are used when computing the dot product of two vectors and when computing a matrix–vector product.

Figure 3(a) shows the driver routine for a DAXPY routine that works for vectors of arbitrary size and alignment. It illustrates the use of the `_alignx` directive discussed previously in the alignment-handling section.

Experimental results

We now describe the performance of various computational kernels on Blue Gene/L hardware. This system runs at 700 MHz and consists of 2,048 compute nodes. We limit ourselves to describing the results obtained on a single node of the BG/L system, since our focus in this work has been on exploiting the dual-issue FPU. Clearly, the performance results we have obtained on these computational kernels have been important for achieving high performance on the parallel applications that use these kernels.

To "measure" the programming effort involved, we coded multiple versions of the test codes, as appropriate.

Their nomenclature and descriptions are as follows. With the exception of the DGEMM routines, all codes are in C and are compiled using the modified XL C compiler described above in the compilation section, with optimization level -O3. The DGEMM routines were written in assembly language and compiled using the GCC (GNU compiler collection) compiler. For the XL C routines, the digit appended to the name of the routine indicates the level of optimization employed, as follows:

0 The source code of this version contains no architecture-specific optimizations.

1 This version augments version 0 with alignment information (_alignx(16, x)) and pragmas asserting the absence of pointer aliasing (#pragma disjoint (*x, *y)). These enable the compiler to generate more PPC440 FP2 core instructions—in particular, more parallel load/stores. Figure 3(a) shows an example. Additionally, this version modifies Version 1 by using the C99 built-in functions shown in Table 1 to select PPC440 FP2 core instructions, making the compiler responsible for instruction scheduling and register allocation.

2 This version, in addition to containing the information present in Version 1, incorporates loop unrolling and software pipelining to help the compiler identify PPC440 FP2 core operations and to reduce pipeline stalls.

In **Figures 5(a)–5(g)**, we present performance results for the DAXPY, DGEMV, and Streams benchmark suite, and matrix–matrix product (DGEMM).

DAXPY

The DAXPY results presented in Figures 5(a) and 5(b) demonstrate that the relative performance of small DAXPY routines can be virtually insensitive to data alignment. In one case, both vectors are misaligned; in the other, both are aligned. The same trend holds when the two vectors are relatively misaligned.

Figure 5(b) shows that this trend continues when the vector sizes become large. The same figure also illustrates that there are several kinds of data (mis)alignment that come into play. The most important kind of data alignment, the 16-byte alignment that allows vector load/stores to occur, has already been discussed. However, L1 and L3 cache-line alignment are also relevant to performance. This figure illustrates the effect of different relative alignment on L3 cache lines. Because of the manner in which memory banks are accessed, different relative alignments lead to different performance characteristics. While we have done further work on this issue and have been able to mitigate some of these effects, we do not present such results here.

DGEMV

Like the DAXPY routine, DGEMV is a BLAS routine whose performance is bandwidth-limited. Similar techniques are used to render this routine relatively insensitive to data alignment, and our experiments have shown that, using a single core, this routine operates at nearly the bandwidth limit of the architecture.

Figure 5(c) demonstrates both the advantages of hand-tuned code over code containing no architecture-specific optimizations and, less obviously, the performance differences ("bumps") encountered when crossing into a new level of the machine memory hierarchy. This cache-boundary effect is perhaps more apparent in Figure 5(b). Since DAXPY has no useful operand reuse and DGEMV has only minimal exploitable data reuse for large problem sizes, this appears to be unavoidable. Below, in matrix–matrix multiplication, we see that, thanks to the bandwidth and other characteristics of this architecture, this effect can be virtually eliminated for computationally intensive kernels, such as DGEMM.

Finally, it should be pointed out that these results were obtained with an older version of the compiler. The compiler gives relatively good performance when given data alignment and pointer alias information in many situations, and efforts are being made to use techniques such as interprocedural analysis and loop versioning to make what we refer to here as a "0type" code perform far more like a "1type" code. Nonetheless, when instruction scheduling and cache hierarchy blocking are extremely important, the performance of architecture-independent codes does not match that of hand-tuned code, such as the assembly-coded DGEMM kernel discussed in this section.

Stream-oriented codes and SIMDization

The advantages of SIMDization and careful instruction scheduling are illustrated in Figures 5(d) and 5(e). These figures also illustrate how efficient compiler-generated code can be when information regarding pointer aliasing and alignment is exploited.

Figure 5(e) makes this most apparent, as it expresses the speedup advantage of the various codes in the Streams benchmark suite. Notice that all of the benchmarks experience a gain from approximately 50% to 100% (2×) in achieved bandwidth, greatly narrowing the gap between achieved and optimal performance for these codes.

Matrix–matrix multiplication

Figures 5(f) and 5(g) respectively illustrate the performance characteristics of L1 and L3 cache-blocked

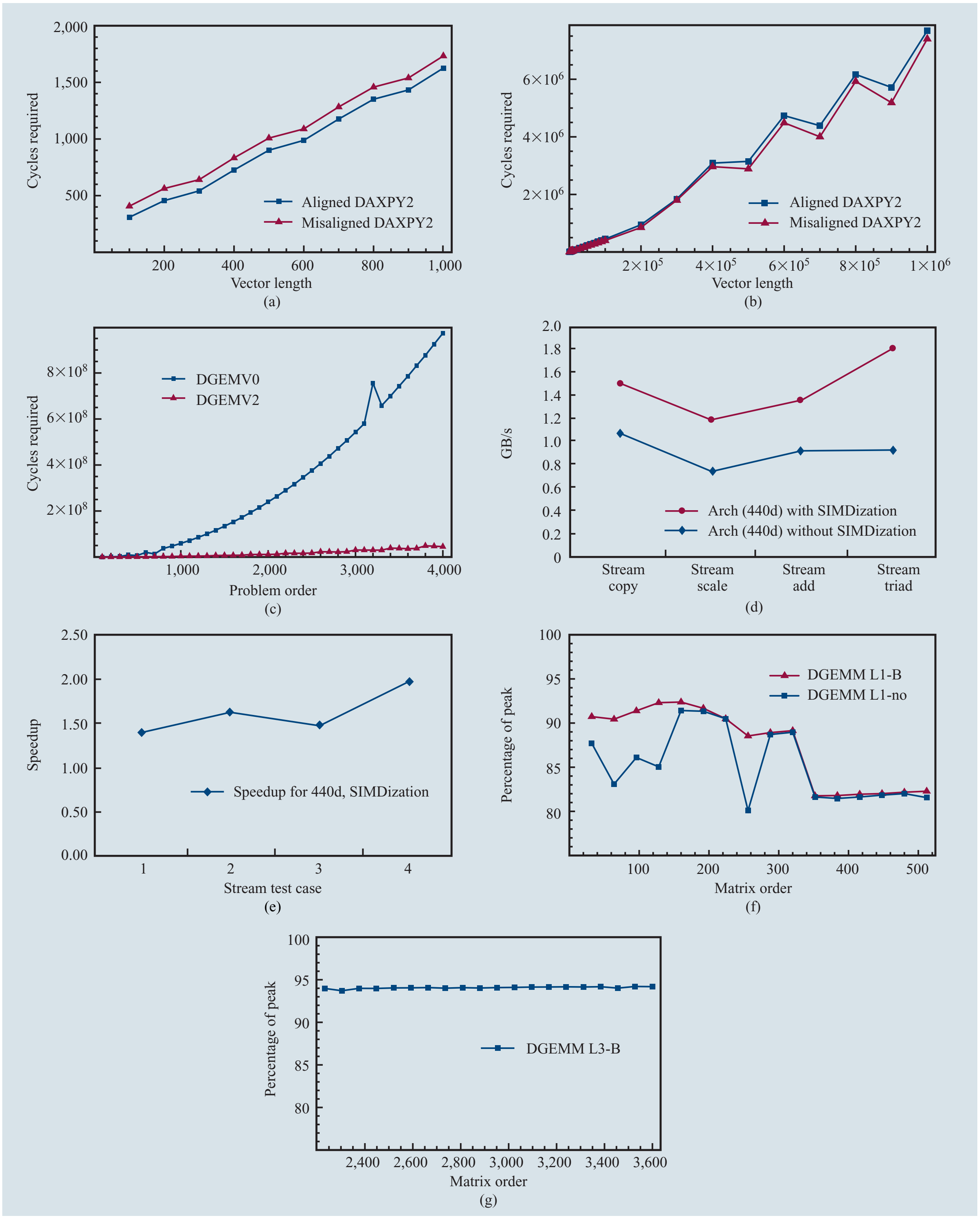

Figure 5

Running time for various versions of test codes on the PPC440 FP2 core.

algorithms. Both matrix-multiplication routines were coded in assembly to handle instruction scheduling and more carefully manage the register file.

Figure 5(f) demonstrates that failing to block for the targeted level of cache can be extremely detrimental to performance. The L1-B trend shows the performance of a properly blocked kernel, while the L1-no line demonstrates what can happen when blocking is not carefully managed. As the matrix size increases past that which can fit into the L1 cache, performance tapers off. It is largely because of the relatively robust and sophisticated memory system that performance does not fall off significantly faster than it does here.

In many cases, especially in computationally intensive codes, it is possible to avoid any perceptible bumps in performance as memory levels are crossed. Preliminary results involving an L3-based matrix-multiplication kernel [Figure 5(g)] reflect this fact. Because of the healthy bandwidth from the L3 cache and the large register file, this architecture is capable of attaining almost the theoretical peak computational rate of the machine for such codes.

Taken together, Figures 5(f) and 5(g) indicate that the architecture is capable of sustaining very close to peak performance when utilizing suitably blocked BLAS3-like algorithms. Note that these results reflect kernel speeds and do not take into account the overhead of data transformations, although these are almost negligible for the (larger) L3-cache-based results. Finally, the DGEMM kernels scale well enough to make presenting single-core results redundant. Thus, for this kernel, only dual-core results are shown.

Conclusions and future work

At a very high level, high-performance engineering and scientific software can be viewed as an interaction among algorithms, compilers, and hardware. This paper is an illustration that such an interaction can produce an FPU and tool chain for engineering and scientific computations that can deliver a substantial fraction of its advertised peak performance.

The PPC440 FP2 core hardware extends the PPC ISA in novel ways to support fast floating-point kernels, while being able to reuse much of the logic and layout of the PPC440 FPU. The new design can run PPC floating-point code as well as code employing our new instructions, and effectively double the peak arithmetic and memory access performance of the core. The compiler for the unit extends both the SLP algorithm for parallelism detection and the Briggs register allocator to handle register pairs. The design of high-performance algorithms for this unit involves innovative techniques to balance the memory bandwidth and floating-point requirements of the operations without unnecessarily constraining data alignment. Initial results show that we are able to sustain a large fraction of the peak performance for key floating-point routines.

There are several possible directions for future work. On the hardware front, while the paired approach worked very well, we could effectively double the performance again had we added a second pair and a means to keep the data fed. On the compiler front, we discussed several capabilities not yet in the compiler whose inclusion would enhance its capabilities. On the algorithm design front, there are many more operations for which we plan to design efficient algorithms.

Acknowledgments

The Blue Gene/L project has been supported and partially funded by the Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Lawrence Livermore National Laboratory Subcontract No. B517552.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Motorola, Inc. or The Mathworks, Inc. in the United States, other countries, or both.

References

1. G. Almási, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castaños, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, A. Deutsch, M. B. Dombrowa, W. Donath, M. Eleftheriou, B. Fitch, J. Gagliano, A. Gara, R. Germain, M. E. Giampapa, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, A. P. Lanzetta, D. Lieber, M. Lu, M. Mendell, L. Mok, J. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, R. Rand, R. Regan, R. Sahoo, A. Sanomiya, E. Schenfeld, S. Singh, P. Song, B. D. Steinmacher-Burow, K. Strauss, R. Swetz, T. Takken, P. Vranas, and T. J. C. Ward, "Cellular Supercomputing with System-on-a-Chip," *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'02)*, 2002, pp. 196–197.
2. K. Dockser, " 'Honey, I Shrunk the Supercomputer'—The PowerPC 440 FPU Brings Supercomputing to IBM's Blue Logic Library," *IBM MicroNews* **7**, No. 4, 29–31 (November 2001).
3. IBM Corporation, *PowerPC 440 Embedded Processor Core Users Manual,* Document No. SA14-2523-01, June 2001.
4. S. Larsen and S. P. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 145–156.
5. L. Bachega, S. Chatterjee, K. A. Dockser, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. P. Mendell, C. D. Wait, and T. J. C. Ward, "A High-Performance SIMD Floating Point Unit for BlueGene/L: Architecture, Compilation, and Algorithm Design," *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, 2004, pp. 85–96.
6. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-

Performance Numerical Algorithms," *IBM J. Res. & Dev.* **38**, No. 5, 563–576 (1994).
7. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. & Dev.* **34**, No. 1, 59–70 (1990).
8. IBM Corporation, *Book E: Enhanced PowerPC Architecture*, March 2000; see *http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC*.
9. T. R. Halfhill, "IBM PowerPC Hits 1,000 MIPS," *Microprocessor Report* **13**, No. 14 (October 25, 1999).
10. K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales, "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro* **20**, No. 2, 85–89 (2000).
11. C.-L. Yang, B. Sano, and A. R. Lebeck, "Exploiting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions," *IEEE Trans. Computers* **49**, No. 9, 934–946 (September 2000).
12. *IEEE Standard 754-1985,* "IEEE Standard for Binary Floating-Point Arithmetic," ©2004 IEEE; see *http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html*.
13. K. O'Brien, B. Hay, J. Minish, H. Schaffer, B. Schloss, A. Shepherd, and M. Zaleski, "Advanced Compiler Technology for the RISC System/6000 Architecture," *IBM RISC System/6000 Technology*, IBM Corporation, Armonk, NY, 1990, pp. 154–161.
14. P. Briggs, "Register Allocation via Graph Coloring," Ph.D. thesis, Rice University, Houston, TX, 1992.
15. A. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for Short SIMD Architectures with Alignment Constraints," *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004, pp. 82–93.
16. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **16**, No. 1, 1–17 (March 1990).
17. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Software* **5**, No. 3, 308–323 (September 1979).
18. K. Goto and R. van de Geijn, "On Reducing TLB Misses in Matrix Multiplication," *Technical Report TR-2002-55* (FLAME Working Note No. 9), University of Texas at Austin, Department of Computer Sciences, 2002.
19. J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal Linear Algebra Methods Environment," *ACM Trans. Math. Software* **27**, No. 4, 422–455 (December 2001).
20. R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)," *Proceedings of the IEEE/ACM Supercomputing Conference*, 1998, p. 38; Best Paper Award for Systems; see *http://www.cs.utk.edu/~rwhaley/ATL/INDEX.HTM*.
21. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-Driven Optimization," *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003, pp. 63–76; see *http://iss.cs.cornell.edu/Publications/Papers/PLDI2003.pdf*.
22. F. G. Gustavson, "New Generalized Matrix Data Structures Lead to a Variety of High-Performance Algorithms," *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, 2000, pp. 211–234.

Received May 28, 2004; accepted for publication August 18, 2004; Internet publication April 12, 2005

Siddhartha Chatterjee *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (sc@us.ibm.com)*. Dr. Chatterjee is a Research Staff Member at the IBM Thomas J. Watson Research Center. He is the manager of the Firmware Architecture and Quality Department at IBM Research. He received a B.Tech. degree (with honors) in electronics and electrical communications engineering from the Indian Institute of Technology, Kharagpur, in 1985, and a Ph.D. degree in computer science from Carnegie Mellon University in 1991. He was a visiting scientist at the Research Institute for Advanced Computer Science (RIACS) in Mountain View, California, from 1991 through 1994, and was an assistant and associate professor of computer science at the University of North Carolina at Chapel Hill from 1994 through 2001. His research interests include the design and implementation of programming languages and systems for high-performance computations, high-performance parallel architectures, and parallel algorithms and applications. Dr. Chatterjee has published in the areas of compilers for parallel languages, computer architecture, and parallel algorithms.

Leonardo R. Bachega *School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, Indiana 47907 (lbachega@purdue.edu)*. Mr. Bachega worked at the IBM Thomas J. Watson Research Center as a co-op student from October 2002 to March 2004. As a member of the Blue Gene software team, he worked on the implementation and test of an efficient subset of the basic linear algebra subroutine (BLAS) for the Blue Gene/L machine. He received a B.S. degree in physics and an M.S. degree in computer engineering, both from the University of São Paulo, Brazil, in 2000 and 2004, respectively. Mr. Bachega is currently a doctoral student at Purdue University. His research interests include compiler optimization for parallel architectures, automatic generation of efficient numerical code, and computer architecture.

Peter Bergner *IBM Systems and Technology Group, 3605 Highway 52 N., Rochester, Minnesota 55901 (bergner@us.ibm.com)*. Dr. Bergner is a member of the Blue Gene/L software team working on compiler and runtime library development. He received a Ph.D. degree in electrical engineering from the University of Minnesota in 1997. His thesis work involved developing techniques for minimizing spill code in graph coloring register allocators. Dr. Bergner has worked in a variety of areas since joining IBM, including AS/400* compiler optimizer development and Linux kernel and compiler development for PowerPC hardware platforms.

Kenneth A. Dockser *Qualcomm CDMA Technologies, 2000 CentreGreen Way, Cary, North Carolina 27513 (kdockser@qualcomm.com)*. Mr. Dockser joined IBM in 1997 as part of the PowerPC Embedded Processor Solutions Team. He headed up the development of the PowerPC 440 floating-point unit and developed the architecture for the Blue Gene SIMD floating-point unit. He also led the efforts to convert the PowerPC 401* and PowerPC 440 processors from technology-specific implementations to portable designs. Mr. Dockser has written several papers; he holds 16 U.S. patents.

John A. Gunnels *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gunnels@us.ibm.com)*. Dr. Gunnels received his Ph.D. degree in computer sciences from the University of Texas at Austin. He joined the Mathematical Sciences Department of the

IBM Thomas J. Watson Research Center in 2001. His research interests include high-performance mathematical routines, parallel algorithms, library construction, compiler construction, and graphics processors. Dr. Gunnels has coauthored several journal papers and conference papers on these topics.

Manish Gupta *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (mgupta@us.ibm.com).* Dr. Gupta is a Research Staff Member and Senior Manager of the Emerging System Software Department at the IBM Thomas J. Watson Research Center. His group has developed system software for the Blue Gene/L machine and conducts research on software issues for high-performance server systems. In 1992 he received a Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign, and he has worked in the IBM Research Division since then. Dr. Gupta has coauthored several papers in the areas of high-performance compilers, parallel computing, and high-performance Java** Virtual Machines.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gustav@us.ibm.com).* Dr. Gustavson leads the Algorithms and Architectures project in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. degree in physics and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for the IBM Power Family* of processors for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Invention Achievement Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.

Christopher A. Lapkowski *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada (lapkow@ca.ibm.com).* Mr. Lapkowski joined the IBM Canada Development Laboratory in Toronto in 1997 after completing an M.A. degree in computer science at McGill University, with a thesis in optimizing compilers. His areas of interest at IBM Canada include working on developing the back end of IBM compilers for the pSeries* and zSeries* computers.

Gary K. Liu *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario L6G 1C7, Canada (garyliu@ca.ibm.com).* Mr. Liu worked as a TOBEY developer on the optimizer for Blue Gene/L. He has a compiler background and received a B.E. degree from McMaster University and an M.E. degree from the University of Toronto. His areas of interest have included the JIT. Mr. Liu is currently focusing on J2ME space.

Mark Mendell *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (mendell@ca.ibm.com).* Mr. Mendell graduated from Cornell University in 1980 with a B.S. degree in computer engineering. He received his M.S. degree in computer science from the University of Toronto in 1983. At the University of Toronto, he helped develop the Concurrent Euclid, Turing, and Turing Plus compilers and worked on the Tunis operating system project. In 1991 he joined IBM, working initially on the AIX* C++ compiler from V1.0 to V5.0. He has been the team leader for the TOBEY Optimizer Group since 2000. Mr. Mendell implemented the automatic compiler support of the dual floating-point unit (FPU) for the Blue Gene/L project.

Rohini Nair *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (nrohini@in.ibm.com).* Mrs. Nair received a B.Tech. degree in computer science and technology from Mahatma Gandhi University, India, in 2000. She joined the IBM India Software Laboratory in 2000. Her work interests to date have included research in the areas of software agents, optimizing code, and compiler development. Mrs. Nair is currently working at the IBM Thomas J. Watson Research Center in the areas of SIMDization for BG/L and porting of ESSL for BG/L.

Charles D. Wait *IBM Engineering and Technology Services, 3605 Highway 52 N., Rochester, Minnesota 55901 (cdwait@us.ibm.com).* Mr. Wait was the PPC440 FP2 team leader. He began his career with IBM in Kingston, New York, in 1986 after graduating from Iowa State University with a B.S. degree in computer engineering. He worked on multiple hardware implementations of the System/390* vector processor in a variety of development positions beginning with logic entry to floating-point team leader. He transferred to IBM Rochester in 1993 to work on system integration simulation of the central electronics complex for the IBM AS/400. He later worked in floating-point hardware development and as processor timing leader of the star processors used in the IBM iSeries* and pSeries systems. Mr. Wait has received numerous formal and informal awards, including an IBM Outstanding Technical Achievement Award for his work as the star processor timing leader.

T. J. Christopher Ward *IBM United Kingdom Limited, Hursley House, Hursley Park, Winchester, Hants SO21 2JN, England (tjcw@uk.ibm.com).* Mr. Ward graduated from Cambridge University in 1982 with a first-class honors degree in electrical engineering. He has worked for IBM in various hardware and software development roles, always finding ways of improving performance of products and processes. From 2001 to 2004, he was a member of the IBM Computational Biology Center at Yorktown Heights, arranging for the Blue Gene/L hardware and compilers and the Blue Matter protein folding application to work effectively together and achieve the performance entitlement. Mr. Ward currently works for IBM Hursley as part of the IBM Center for Business Optimization, enabling customers of IBM to take advantage of the opportunities afforded by the rapidly decreasing cost of supercomputing services.

Peng Wu *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pengwu@us.ibm.com).* Dr. Wu received her M.S. degree in computer science in 1999 and her Ph.D. degree from the University of Illinois at Urbana–Champaign in 2001. She joined the IBM Thomas J. Watson Research Center as a Research Staff Member in 2001. Dr. Wu is currently working on compiler optimization and automatic SIMDization for multimedia extensions such as VMX for PowerPC and double-hummer FPU for BG/L.