NAS Benchmarks on the Tera MTA

Jay Boisseau[*], Larry Carter[*†], Kang Su Gatlin[†], Amit Majumdar[*],
and Allan Snavely[*†]

Abstract

The Tera MTA is new, revolutionary commercial computer based on a multithreaded processor architecture. We have compiled and run the five NAS kernel parallel benchmarks on a prototype version of the MTA. This paper briefly describes the MTA architecture, our experience with the compiler, and some performance results.

We compare a single-processor MTA's performance and ease of programming to that of the Cray T90, the most powerful vector supercomputer made by Cray Research. We found both the MTA and the single-processor T90 required no tuning on four of the five benchmarks to get respectable performance. The production MTA should be faster on the CG and IS benchmarks, and the T90 is faster on FT and MG. Except for MG, where the T90's faster clock and higher memory-to-processor bandwidth give it an unbeatable advantage, the differences in performance are relatively small.

We have defined four levels of tuning effort, ranging from "no tuning" to "heroic". The one remaining code, EP, was easily modified to get vectorized or threaded execution. We report some further improvements that were obtained at higher levels of tuning effort on the MTA. In general, for these relatively simple benchmarks, we found tuning codes for the MTA significantly easier than on massively parallel multicomputers or even on high-performance workstations. In fact, most of the tuning consisted of removing unnecessary locality-enhancing "optimizations" that had been introduced into the NAS codes to improve their performance on computers with cache-based memories.

Keywords: Tera multithreaded architecture, performance evaluation, NAS parallel benchmarks, performance tuning, parallelizing compilers.

Contact Author:

Larry Carter
UCSD Mail Code 0114
9500 Gilman Drive
La Jolla, CA 92093-0114

Phone: 619 534-6265
Web: www.cs.ucsd.edu/users/carter
Email: carter@cs.ucsd.edu

1 Introduction

The Tera Computer Company has delivered its first MTA system [TERA], for "Multi-Threaded Architecture", to the San Diego Supercomputer Center at the University of California at San Diego. This paper describes our experience adapting the five NAS kernel benchmarks [NAS] to the Tera MTA. We will report the performance attained, as well as (somewhat subjective) measures of the programming effort required.

[*]San Diego Supercomputer Center
[†]Department of Computer Science and Engineering, University of California at San Diego

The following sections briefly describe the MTA's architecture, the NAS kernel benchmarks, our tuning methodology, and our results and observations.

2 The Tera MTA

The MTA system has from one to 256 processors which share a large memory. The machine has either 1 GB or 2GB of memory per processor, but randomized memory mapping and a high-interconnectivity network provide near-uniform access time from any processor to any memory location. There are no data caches. Instead, multithreading is used to tolerate the latency of memory accesses, which require on the order of 100 or 150 cycles.

Each processor has 128 "streams", where a stream is hardware (including 32 registers and a program counter) that is devoted to running a single thread of control. The processor makes a context switch on each cycle, choosing the next instruction from one of the streams that is ready to execute. Synchronization between threads can be accomplished using full/empty bits in memory, allowing for fine-grained threads. Each 64-bit *instruction* encodes three *operations*, one memory operation and two others which can be arithmetic or control operations. The operations are unusually powerful, including integer and floating-point multiply-and-add, atomic fetch-add to memory, and many bit-manipulation operations and addressing modes.

From the point of view of a thread, when it is assigned to a stream, its instructions are executed one at a time, requiring 21 cycles per instruction. Each instruction includes a "lookahead" number (between 0 and 7) that designates how many additional instructions can be executed before the result of the memory operation is needed. Since memory operations take 5 or 6 times the 21-cycle execution speed, fast thread execution requires that the compiler is able to schedule memory operations ahead of when their results are needed.

From the point of view of a processor, each stream can provide it with at best one operation every 21 cycles (if there is sufficient lookahead), but at worst, fewer than one operation every 100 cycles. Full utilization requires the compiler to schedule enough threads that have sufficient lookahead. If the processor has no stream ready to issue an instruction, the no-op for that issue slot is called a "phantom". Since there are 128 streams, even zero-lookahead threads can fully utilize the processor. However, there is a moderate overhead (on the order of 100's of instructions) associated with creating, scheduling and swapping threads, so there are advantages to using fewer, higher-lookahead threads. These issues raise interesting questions about how effectively the compiler will find the "right" granularity of threads. There are additional questions about how to utilize multiple processors efficiently, though the difference between one-processor and multiple-processor compilation is far smaller on the MTA than for traditional computers.

The prototype MTA computer that we used is a one-processor machine, with a ring topology that connects the processor to two memory modules and the I/O devices. The production machine, which will have a high-throughput network topology, will probably have slightly fewer phantoms than our prototype. Additionally, the clock speed of the prototype was 145 MHz for most of our experiments, whereas the production machine should have a 333 MHz clock. We report the actual times achieved, but also scale the performance to reflect the expected improvement in clock speed. We do not attempt to adjust for the different network topology.

3 The NPB 2-serial benchmarks

Researchers at NASA have developed a set of eight benchmark programs that are representative of computational fluid dynamics applications. These NAS benchmarks have become one of the standard measures of supercomputer performance. Five of the programs, designated the "kernel benchmarks", are relatively small programs that nonetheless provide significant performance challenges to the processors, memory hierarchy, and communications fabric of today's supercomputers. These programs are CG (Conjugate Gradient), EP (Embarassingly Parallel), FT (Fourier Transform), IS (Integer Sort), and MG (Multigrid). NAS publishes performance results of their benchmarks for many supercomputers, which will facilitate a comparison of the MTA to other architectures

There are several versions of the NAS Parallel Benchmarks (NPB). NPB 1 are "paper and pencil" specifications, provided as simple FORTRAN programs that compute the required answers. Motivated computer vendors have devoted considerable programming effort to ensure their machine executes the programs efficiently [AAC94] and performance can reflect the skill of the programmers as well as the capabilities of the machine.

NPB 2 are parallel implementations of the benchmarks written primarily in FORTRAN 77 with MPI. They were designed to run reasonably well on many parallel computers, and "are intended to be run with little or no tuning" [NAS]. They exploit architectural features that are not present on the MTA, such as a memory hierarchy and high-latency interprocessor communication. Since extensive changes would be needed just to eliminate the unnecessary overhead of the NPB 2.0 code, they are not an appropriate starting point for our experiments.

NPB 2-serial are single-processor programs derived by removing all parallelism from NPB 2. They are intended to be tests of parallelization tools, as benchmarks for workstations (run with little or no tuning), and as the starting point for benchmarking shared-memory multiprocessors. Using the 4-level tuning methodology described below, they provide a test of the MTA's compiler as well as the MTA itself, and our performance results can be meaningfully compared to results for other computers.

NAS has defined five different problem sizes (S, W, A, B and C) for each benchmark, which range over several orders of magnitude in the number of operations required. Unless otherwise noted, the results here are for the Class A size problems, intended mainly for small parallel machines.

4 Tuning Methodology

We define four levels of tuning effort. The first two levels are conducted without using feedback from actual timings. The fact that acceptable performance is achieved at these levels indicates that the MTA is exceptionally easy to program for these applications. The remaining levels involve the standard feedback loop of profiling, locating bottlenecks, and reprogramming.

Level 0: no tuning. The only changes, if any, to the code are those that are needed to make the code run correctly. Some of the benchmarks include alternate implementations, for instance, four random number generators are supplied with the EP benchmark. We selected the variant that we think is most appropriate. For instance, since the MTA has 8-byte integer multiply-and-add instruction, the integer variant `randi8` was used.

Level 1: minimal parallelization. The goal here is only to make changes as needed so that the compiler indicates that the significant loops have been parallelized. This will be accomplished by whatever means (either adding compiler directives or rewriting code) that seems the most expedient. For a nested loop, we only require that one of the loop levels is parallel, even if we believe more efficient performance would result if a different loop had been parallelized. The performance that is achieved with Level 1 tuning can be compared to "no tuning" results of the NPB 2 parallel benchmarks, since considerable effort was already made during the design of the latter to enable parallel execution. If anything, the comparison is prejudiced against the MTA.

Level 2: standard performance tuning. At this tuning level, we will perform two types of optimizations. The first is to eliminate extra operations whose purpose appears to be to improve performance on traditional architectures, but are not needed for the MTA. For example, in FT (the Fourier Transform benchmark) there are several copies and transposes whose purpose is to increase the locality of memory references – this is unnecessary for the MTA's flat memory. The second type of optimization will be standard tuning such as restructuring loops to reduce loads and stores, looking for additional parallelism, and adjusting the granularity of threads. We will limit ourselves to changing at most 5% of the lines of code (on the reduced program) to allow plausible comparison with the NAS 2 benchmark results that are limited to a 5% change. Additionally the difference between Level 0 performance and Level 2 performance will be a good indication of the effectiveness of MTA's compiler.

Level 3: heroic tuning. Here, we will try (subject to the limitation of our available time) to perform the type of all-out performance programming that was used on the NAS 1 results.[1] Unlike Level 2 tuning, here we will employ detailed knowledge of the MTA architecture. For instance, in CG (the Conjugate Gradient benchmark), several memory load operations might possibly be eliminated for the indirect memory references of sparse matrix multiplication by using "forwarding pointers". (We found this particular idea didn't help, since the processor-memory bandwidth requirements remain the same.)

Prior to starting our experiments, we had received a one-day training course on the Tera MTA's compiler and tools. For tuning levels 0, 1, and 2, we essentially performed the programming without further instruction, although we did receive help *running* the codes on the prototype environment. For some of the level 3 tuning, we also consulted with Tera personnel.

5 Results

We now describe our experience tuning the kernel benchmarks using the Tera compiler. The compiler can perform sophisticated loop transformations and generate multi-threaded code not only for parallelizable loops, but also for some other programming constructs such as reduce and parallel prefix operations. There is a very useful Compiler ANALysis tool, CANAL, that summarizes the compiler's actions. We will show some of the CANAL output in the discussions of the benchmarks that follow.

For comparison, we used a single processor of the 14-processor Cray T916/14 at SDSC running with a 440 MHz clock speed.

5.1 Conjugate Gradient

The NAS Conjugate Gradient (CG) benchmark spends most of its time computing sparse matrix-vector products in the following code:

```
do j=1, lastrow-firstrow+1
   sum = 0.d0
   do k = rowstr(j), rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
   enddo
   w(j) = sum
enddo
```

The CANAL report corresponding to this code segment is:

```
Loop  17 in conj_grad_ at line 573 in loop   16
        Dynamically scheduled
        Single processor implementation
        In parallel phase 4

Loop  18 in conj_grad_ at line 575 in loop   17
        Loop scheduled: 3 memory operations, 2 floating point operations
                 3 instructions, needs 21 streams for full utilization
```

The compiler allocates 21 streams to execute the iterations of the outer loop; "dynamically scheduled" means that the iterations are assigned to streams in a first-come first-serve manner using the fetch-add instruction for inter-stream synchronization. This ensures there is no load imbalance among the streams even if the number of non-zero matrix elements differs from row to row. The inner loop is "loop scheduled", meaning it is unrolled and rescheduled via software pipelining so that the load of `colidx(k)` is issued at least seven instructions before the load of `p(colidx(k))`, which in turn is at least seven instructions before the

[1] Comparisons to NAS 1 figures are not as meaningful as one might hope, since some of the NAS 1 and NAS 2 specifications differ. In particular, IS (the Integer Sort benchmark) has an extra "ranking" step in NAS 1 that was eliminated from NAS 2.

multiply-add instruction. This allows the lookahead number of all instructions in the inner loop to be seven, ensuring maximal instruction-level parallelism. In these and other ways, the compiler does an impressive, near-perfect job of using both instruction-level and thread-level parallelism.

Note however that the inner loop has three memory operations. Since the MTA can issue only one memory operation per cycle, the code requires three instructions per iteration and the maximum performance that can be achieved on CG (without heroic tuning) is two floating point operations per three cycles.

A summary of tuning effort and performance of CG is as follows:

- **Level 0:** No lines of code required changing to make the program run. Performance is 80.9 MFLOP/sec (18.50 seconds) on the class A problem on the 145-MHz MTA prototype. With a 333 MHz clock, the production MTA should achieve at least 186 MFLOP/sec.
- **Level 1:** No changes needed to make the important loop parallelized.
- **Level 2:** We didn't attempt this level of tuning. The percentage of phantoms was 10.5% in the Level 0 code; this might be reduced by tuning, or by the production MTA's improved network. The maximum performance at level 2 tuning, assuming the entire cost of the program is in the matrix-vector product running at two FLOP's per three cycles, would be $333 * 2/3 = 222$ MFLOP/sec.
- **Level 3:** We managed to break the 2/3 FLOP/cycle barrier by packing three values of colidx into an 8-Byte word. It took several attempts (about 12 hours of programmer effort) to find a successful way of generating code that unpacked the packed numbers without introducing extra instructions. Performance was improved to 89.9 MFLOP/sec (16.64 seconds) including 10.3% phantoms. A 333 MHz clock would result in 207 MFLOP/sec.

The Cray T90 executed the untuned CG code at 180.9 MFLOP/sec (8.27 sec). It might benefit more from Level 2 tuning, converting the dot-product style inner loop into daxpy-style code that is more suitable for vector processors.

5.2 Fourier Transform

The NAS Fourier Transform (FT) benchmark performs a three-dimensional FFT using a technique that is well-suited to computers that have a hierarchical memory. It copies 16 one-dimensional transforms at a time into contiguous storage (with a little padding to prevent cache associativity problems), then performs a vector-style transform on them in parallel. This is unfortunate for the MTA. The numerous data copies are unnecessary overhead, and they prevent the compiler from recognizing that many thousands of 1-D FFT's could be executed in parallel.

Nevertheless, without modification the codes runs with some level of multithreading in all important loops.

A summary of tuning effort and performance of CG is as follows:

- **Level 0:** No lines of code required changing to make the program run. Performance was 61.9 MFLOP/sec (6.04 seconds) on the class W problem[2] on the prototype running at 245 MHz. With a 333 MHz clock, the production MTA should achieve at least 84 MFLOP/sec on this small problem.
- **Level 1:** No changes needed to make the important loop parallelized.
- **Level 2:** Removing many the unnecessary copies involved changing 188 lines of code. The Class A problem ran successfully at 66.3 MFLOP/sec (107.61 seconds) using a 145 MHz clock. There were 28.3% phantoms, and we believe further Level 2 tuning is possible. At 333 MHz, the current performance would be 152.3 MFLOP/sec.
- **Level 3:** We tried changing the radix-2 code to a radix-4 algorithm. This caused some register spill to occur, and slightly worse results — 37 instructions for a radix-4 butterfly, which replaces four 9-instruction radix-2 butterflys. We plan to try uses a method that makes better use of the floating-point multiply-and-add operation, as suggested by [LF93, G97].

The Cray T90 performed the untuned FT code at 183.6 MFLOP/sec (38.88 sec).

[2] An unexplained "alignment error" prevented our running the Level 0 class A problem.

5.3 Integer Sort

The NAS Integer Sort (IS) benchmark spends over 85% of its time in the two loops shown below, copying an array (unnecessarily) into a temporary buffer, and counting the number of occurrences of each value in the array.

```
for ( i=0, i<NUM_KEYS; i++)
   key_buff2[i] = key_array[i];

for ( i=0, i<NUM_KEYS; i++)
   key_count[key_buff2[i]]++;
```

The compiler fuses these two loops into a single loop, thereby eliminating the need to load `key_buff2` in the second loop. However, due to the structure of the code, it requires level-2 hand-tuning to eliminate the copy completely. The CANAL report corresponding to the fused loop is:

```
Loop  11 in rank at line 462
        Loop scheduled:  3 memory operations, 0 floating point operations
                3 instructions, needs 35 streams for full utilization
        Block scheduled
```

A remarkable feature of this code is that only three memory references are needed per iteration. They correspond to loading an element of `key_array`, storing it in `key_buff2`, and performing a fetch-add to `key_count` in memory. The fetch-add eliminates the need to load and store `key_count`into a register.

A summary of tuning effort and performance of IS is as follows:

- **Level 0:** No lines of code required changing to make the program run. Performance was 35.0 MOP/sec[3] (2.39 seconds) on the 145 MHz prototype. There were 17.6% phantoms. With a 333 MHz clock, the production MTA would run at 80.5 MOP/sec. assuming no benefit from the improved network.
- **Level 1:** No changes are needed to make the important loop parallelized.
- **Level 2:** Removing the unnecessary copy required changing five lines of code. The result was 41.0 MOP/sec (2.05 seconds), which at 333 MHz would be 94.1 MOP/sec.
- **Level 3:** There isn't much opportunity for improvement, but possibly packing multiple values into the 8-Byte integers could improve the parallel prefix operation a little.

The Cray T90 performed the untuned IS code at 75.3 MOP/sec (1.11 seconds). This is also a remarkable speed – the T90's compiler uses a sophisticated algorithm to vectorize the "unvectorizable" second loop above.

5.4 Embarassingly Parallel

The NAS "Embarassingly Parallel" (EP) benchmark had a loop in the random number generator that was the only important loop that was not handled well by either the MTA or the T90 compiler. We summarize:

- **Level 0:** The untuned code ran slowly — .12 MOP/sec[4] on the small Class W problem. We didn't try running the Class A problem.
- **Level 1:** It required changing five lines of code to create a multi-threaded version of the program. The NPB 2-serial code version of EP computes a vector of random numbers with each call to the generator. There needs to be separate storage for this vector for each thread. Effecting this change involves adding two directives and modifying three declarations. Performance was 2.43 MOP/sec (220.88 sec) at 145 MHz, corresponding to 5.59 MOP/sec at 333 MHz.

[3]Each "OP" corresponds to ranking one key.

[4]In this case, an "OP" is generating a random number and performing some calculations on it.

- **Level 2:** Five more lines of code changed eliminated a hot spot where the results are accumulated. Performance improved to 3.50 MOP/sec (153.47 sec), corresponding to 8.04 MOP/sec at 333 MHz.
- **Level 3:** Calls to Fortran intrinsics were inlined, improving performance to 4.32 MOP/sec (124.31 sec), corresponding to 9.92 MOP/sec at 333 MHz. Arguably, this could be called level-2 tuning, but we classify it as Level 3 since we received help from Tera personnel.

The untuned code ran at 6.91 MOP/sec (77.73 seconds) on the T90. Since the code was not completely vectorized, we performed level-1 tuning for the T90 as well. A moderately simple change to the random number generator improved the T90's performance to 7.73 MOP/sec (69.48 seconds).

5.5 Multigrid

The NAS Multigrid (MG) benchmark spends most of its time computing 27-point stencil computations, using temporary arrays (`u1` and `u2` in the code below) to take advantage of 4-point common subexpressions that are used multiple times. These computations are performed in four subroutines; the inner loops for a typical one is shown below:

```
  do i = 1, n
     u1(i) = u(i,j  ,k  ) + u(i,j+1,k  ) + u(i,j  ,k-1) + u(i,j  ,k+1)
     u2(i) = u(i,j-1,k-1) + u(i,j+1,k-1) + u(i,j-1,k+1) + u(i,j+1,k+1)
  enddo
  do i = 2, n-1
     r(i,j,k) = v(i,j,k) - a(0)*u(i,j,k)
>              - a(2)*(u2(i)+u1(i-1)+u1(i+1)) - a(3)*(u2(i-1)+u2(i+1))
  enddo
```

The first loop has 6 floating point operations but needs 10 memory operations, thereby requiring 10 instructions per iteration. The second loop has 5 memory operations and 9 floating point instructions. Since the MTA has a multiply-and-add operation, it requires only 6 instructions per iteration. Without fusing the two loops into a single loop, the maximum possible performance is 15 floating point operations per 16 cycles.

A summary of tuning effort and performance of MG is as follows:

- **Level 0:** No lines of code required changing to make the program run correctly. Performance was 91.1 MFLOP/sec (42.74 sec) at 145 MHz, corresponding to 209 MFLOP/sec at 333 MHz.
- **Level 1:** No changes are needed to make all the important loops parallelized.
- **Level 2:** We performed (by hand) the intricate loop transformations needed to fuse the two loops described above with 3-way unrolling to avoid unnecessary register copies. This unfortunately resulted in some register spilling, and the CANAL output showed that 49 instructions would be needed for the 45 floating point operations (as opposed to 3*16=48 instructions in the original code).
- **Level 3:** It might be possible to further improve the FLOP to memory operation ratio, but it appears to require heroic effort.

The Cray T90 performed the untuned MG code is at 601 MFLOP/sec (6.48 seconds), significantly faster than the MTA's limit. Clearly, the T90 benefits from being able to load and store multiple values per cycle.

These results are summarized in Figure 5.5.

6 Conclusions

The MTA's compiler effective at finding enough threads in the NAS kernel benchmarks to keep a single processor busy, and indeed the parallelization strategies it used could generate thousands of threads if needed. There were only two programs that presented any difficulties: EP needed a few directives, and FT was limited

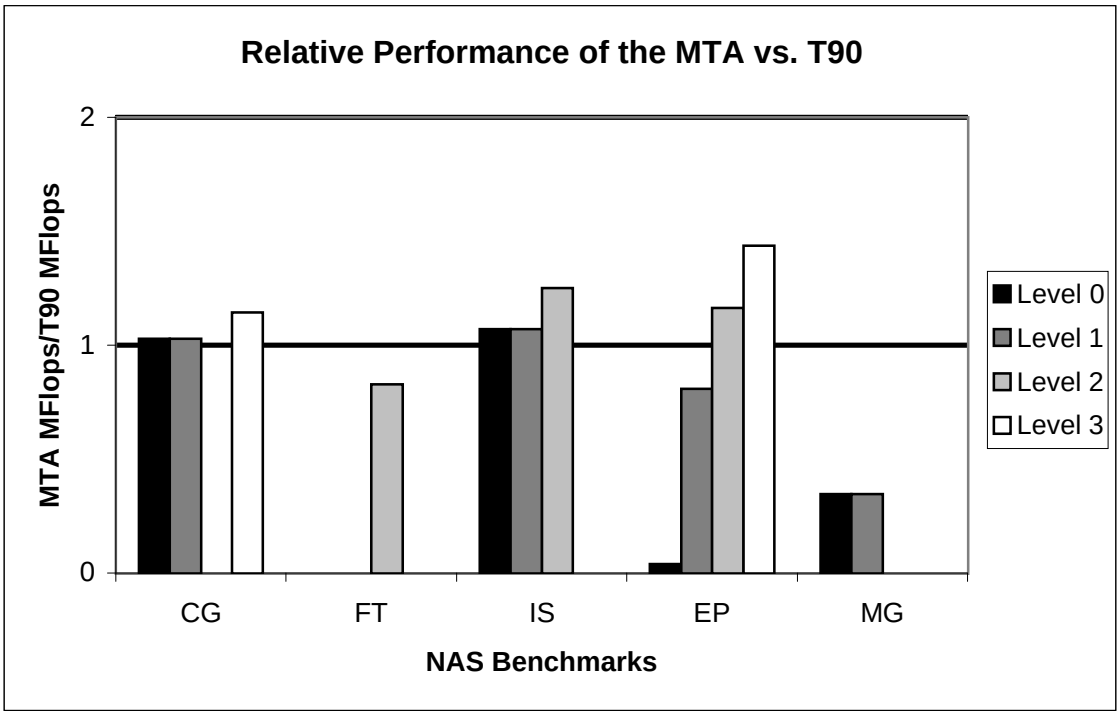

Figure 1: Performance of the Tera MTA on the five kernal NAS benchmarks, shown relative to the performance of the (untuned) codes on a Cray T916/14.

to 16-way parallelism due to a coding style that was intended to improve cache performance on workstations. A significantly simpler version of FT allows $O(n^2)$ threads for an $O(n^3 lgn)$ computation.

A second observation is that the compiler makes very efficient use of the VLIW instructions. In all of the benchmarks except possibly EP (which spends most of its time in library functions that we didn't study), the major bottleneck is the memory-register bandwidth. The MTA can only perform one memory operation per cycle, and the compiler always generated code that issues memory operations at nearly the maximum rate.

Our performance figures show the overhead introduced by multithreading is small. For CG and IS, we calculate that more than two thirds of the cycles were spent doing necessary memory operations. We expect this ratio to improve further with the memory system of the production machine.

Our comparisons with a single processor of the Cray T90 reveal that the T90 benefits by having a faster clock and a higher memory-register bandwidth. Nevertheless, the performance of the production MTA should be reasonably comparable. Both have compilers that are effective in keeping a single processor utilized on these scientific benchmarks. For the MTA, scaling to multiple processors may be equally easy.

References

[AAC94] Agarwal, R.C., B. Alpern, L. Carter, F.G. Gustavson, D. Klepacki, R. Lawrence and M. Zubair, "High Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2," *IBM Systems Journal,* Vol 34, pp. 263-272 (1995).

[LF93] Linzer, E. and E. Feig, "Modified FFT's for Fused Multiply-Add Architectures", *Mathematics of Computation,* Vol 60, pp. 347-361 (January 1993).

[G97] Goedecker, S., "Fast Radix 2,3,4, and 5 Kernels for Fast Fourier Transformations on Computers with overlapping multiply-add instructions", *SIAM Journal on Scientific Computing*, Vol 16, pp 1605-1611, (November 1997).

[NAS] See http://science.nas.nasa.gov/Software/NPB.

[TERA] See http://www.tera.com.