SAND2007-2869C

Tuning C++ Applications for the Latest Generation x64 Processors with PGI Compilers and Tools

Douglas Doerfler *and* **David Hensinger**
Sandia National Laboratories
Brent Leback *and* **Douglas Miles**
The Portland Group (PGI)

ABSTRACT: *At CUG 2006, a cache oblivious implementation of a two dimensional Lagrangian hydrodynamics model of a single ideal gas material was presented. This paper presents further optimizations to this C++ application to allow packed, consecutive-element storage of vectors, some restructuring of loops containing neighborhood operations, and adding type qualifiers to some C++ pointer declarations to improve performance. In addition to restructuring of the application, analysis of the compiler-generated code resulted in improvements to the latest PGI C++ compiler in the area of loop-carried redundancy elimination, resolution of pointer aliasing conflicts, and vectorization of loops containing min and max reductions. These restructuring and compiler optimization efforts by PGI and Sandia have resulted in application speedups of 1.25 to 1.85 on the latest generation of x64 processors.*

KEYWORDS: Compiler, C, C++, Optimization, Vectorization, Performance Analysis, AMD Opteron™, Intel Core™ 2, Micro-architecture

1. Introduction

Although modern compilers do an admirable job of optimization when provided with nothing other than the "–fast" switch, many times there are still significant performance gains to be obtained with detailed analysis and applied expert knowledge of the application, the compiler, and the underlying processor architecture. The PGI compiler suite has been the mainstream compilation environment at Sandia National Laboratories for several years in its high performance computing initiatives. PGI compilers were used on the 32-bit x86 processor-based ASCI Red supercomputer deployed in 1997, and are used on the 64-bit AMD Opteron processor-based ASC Red Storm platform deployed in 2004. Throughout this period, there has been a continual effort to ensure that optimal performance is extracted and utilized on Sandia's supercomputing platforms. Sandia's Advanced Systems Group is continually evaluating new technologies and technological trends to determine the impact on its application base. One such trend is the growing use of vector or SIMD units to increase FLOP rates in general purpose processors.

While first-generation AMD and Intel 64-bit x86 (x64) processors contain 128-bit wide Streaming SIMD Extensions (SSE) registers, their 64-bit data paths and floating-point units limit the performance benefit of vectorizing double-precision loops. New x64 processors from Intel, known as the "Woodcrest" or Core™ 2 Duo architecture, and a chip from AMD known as "Barcelona" or Quad-Core AMD Opteron™, contain 128-bit-wide data paths and floating-point arithmetic units.

Vectorization is key to extracting maximum performance from floating-point intensive scientific and engineering codes on first-generation x64 processors, and is even more important on the latest generation of x64 processors. In this paper we discuss coding to maximize vectorization, how these techniques were applied to a kernel from Sandia's ALEGRA shock physics code, some compiler improvements driven by this joint project

between Sandia and PGI, and the significant performance gains achieved on the latest generation of x64 processors.

2. SSE Vectorization on First-generation x64 Processors

Vectorization for x64 processors is used to identify and transform loops to take advantage of packed SSE instructions, which process two input operands of 128 bits of data per instruction. Each operand contains two 64-bit double-precision values or four 32-bit single-precision values. The result of the instruction is also a 128-bit packed result, of the same data type as the inputs.

Diagram 1 below gives a logical representation of how packed double-precision operands are stored and computed in SSE registers. This logical representation, and indeed the corresponding x64 assembly code, gives the impression that the two double-precision floating-point additions are performed concurrently. In fact, while these operations are performed using a single packed SSE instruction, they are not necessarily performed concurrently. Rather, in previous generation x64 processors they are simply pipelined in sequence through the 64-bit arithmetic unit.

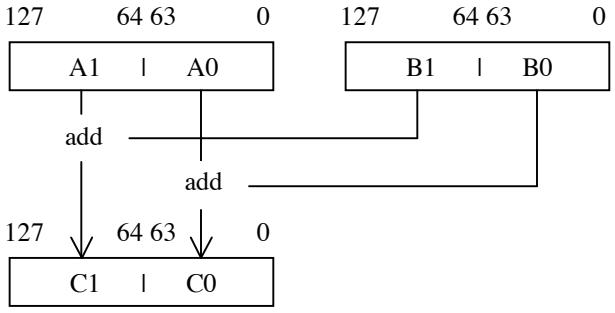

Diagram 1. Packed double-precision add using the `addpd` SSE instruction

Likewise, x64 processors also possess SSE hardware support for a floating-point multiplier. On the first-generation AMD Opteron, the SSE adder is a separate functional unit from the SSE multiplier and both can work concurrently to enable one 64-bit multiply and one 64-bit add per cycle.

On first-generation Intel x64 processors (Xeon EM64T), only one SSE floating-point instruction can be issued per cycle. These first-generation Intel x64 processors run at much higher clock rates than first-generation AMD64 processors to provide similar peak SSE vector performance, but are generally less efficient on scalar floating-point code because of the single operation per cycle limitation.

The underlying hardware is free to implement these instructions given the usual constraints of silicon real estate, power, etc. In first generation x64 processors, the throughput of these operations could be modelled as:

```
Cycle i:      [A1|A0]   [B1|B0]
                 \       /
                 A0+B0
                 .....
                 .....

Cycle i+1:    [A1|A0]   [B1|B0]
                 \       /
                 A1+B1
                 A0+A0
                 .....

Cycle i+p-1:     .....
                 A1+B1
                 A0+B0
                    \
                 [..|C0]

Cycle i+p:       .....
                 .....
                 A1+B1
                   /
                 [C1|C0]
```

where the adder pipeline depth *p* varied depending on the x64 processor implementations from AMD and Intel.

Just as double-precision floating-point adds and multiplies are pipelined in sequence on first-generation x64 processors, likewise single-precision packed floating-point add and multiply instructions pipeline data through the SSE arithmetic units. However, as the arithmetic units are 64-bits wide, they are able to perform two 32-bit operations concurrently. This effectively doubles the throughput for single-precision packed SSE operations compared to double-precision packed SSE operations.

One obvious question is, if the underlying hardware only supports 64-bit operations, why is vectorization so important on first-generation x64 processors?

Processor and compiler vendors have been promoting vectorization as equally important on single- or double-precision data, but in fact it has been much more important for single-precision codes. On first-generation x64 processors, highly-vectorized single-precision codes can often achieve speed-ups of nearly 2X over a scalar (non-vectorized) version. With double-precision data it is more common to see at most 20% - 30% speed-ups, and sometimes no speed-up at all. The gains for double-precision codes come not from doing more operations per cycle, but rather from:

1. Doing the same number of arithmetic operations in fewer instructions, increasing instruction efficiency.
2. More efficient utilization of the floating-point arithmetic unit pipelines.
3. Fewer instructions required to move data between SSE registers and memory.
4. More efficient utilization of memory and cache bandwidth by leveraging the alignment of vector data to cache line boundaries.
5. Other vectorization benefits: loop re-ordering, optimal unrolling, software prefetching, etc.

Note that for a loop to vectorize, the compiler must determine that it contains no loop-carried dependence cycles. This is traditionally easiest to do in Fortran programs, which tend to operate on well-defined arrays. It is more difficult in C and C++, where pointers are frequently used in ways that result in real or potential data dependencies that cannot be resolved by the compiler. As we will see below, coding in C and C++ with an eye toward vectorization can pay significant performance dividends. If the compiler determines that a loop can be vectorized, it can often be auto-parallelized. So, coding to maximize vectorization will also maximize the potential for automatic parallel speed-up on multi-core processors, a topic which is not specifically addressed in this paper.

3. SSE Vectorization on the Latest Generation x64 Processors

In the latest generation of x64 processors, the SSE floating-point adder and multiplier have been widened to 128-bits. This basically doubles the potential throughput of these arithmetic units. The new hardware can be modelled performing double-precision packed floating-point adds as follows:

```
Cycle i:      [A1|A0]    [B1|B0]
                \   \      /   /
              [A1+B1|A0+B0]
              .....
              .....

Cycle i+1:
              .....
              [A1+B1|A0+B0]
              .....

Cycle i+p:
              .....
              .....
              [A1+B1|A0+B0]
                   \     /
                  [C1|C0]
```

Double-precision floating-point operations are now performed two-at-a-time, and likewise the throughput of single-precision calculations has been increased to four-at-a-time through the 128-bit SSE floating-point units. Thus, the other stages in the pipeline are available for other calculations. While this is clearly a significant improvement in the theoretical peak performance of the processors, it also increases the need to move data from memory or cache to the SSE registers as efficiently as possible.

The most efficient way to move data to the SSE registers is through use of the `movapd` and `movaps` instructions. These perform “move aligned packed double” and “move aligned packed single” operations respectively. Each instruction moves 128-bits of data from cache or memory into an SSE register. For the compiler to be able to generate these instructions, it must know the source data is aligned on a 128-bit (16 byte) boundary. If a move aligned instruction is issued on data that is not aligned, the processor will issue an instruction fault.

Generally, if the source data is not aligned the compiler will still vectorize a loop but must use less efficient sequences of instructions to load up the SSE registers. The PGI compilers go to great lengths to maximize the number of aligned move instructions used in vectorized loops. By default, the compilers generate alternate versions of a loop (known as “altcode”) to handle different alignment possibilities, and frequently add "peeling" to perform one or more iterations of a loop in scalar fashion before falling into a vectorized loop where the remaining data in one or more vectors is aligned on a 16-byte boundary.

Also, to maximize the efficiency of SSE vectorization, like data should be stored in sequential vectors to minimize the amount of packing and shuffling of data required by the compiler to vectorize loops. In general, compilers are fairly sophisticated in the amount of "irregular" data packing and vectorization they can support, but these operations are typically inefficient on x64 SSE units. We will see how important this can be in the sections that follow, where we describe the restructuring performed on ALEGRA data structures.

We coded up a simple multiply-add operation, sufficiently unrolled to enable peak processor performance, to illustrate differences in efficiency between scalar operations, unaligned vector operations, and aligned vector operations.

Table 1 compares first-generation x64 processors against the latest generation, in units of "utilization of latest generation peak performance". The first row of Table 1 illustrates performance of scalar (non-packed) SSE floating-point operations in a loop that contains no memory references. The second row illustrates

performance of vector (packed) SSE floating-point operations in a loop that contains no memory references.

All remaining rows use packed vector arithmetic instructions including some number and type of memory accesses. In each successive pair of rows, we've added code to load multiples of 8 bytes (one double operand) per each multiply-add operation (since the processors can do two multiply-adds per cycle at peak, this is actually 16 bytes per cycle). We've used both optimal aligned loads/stores in the first row of each pair, and the less-efficient sequences required when data is potentially unaligned in the second row of each pair. All memory references in this test accessed data known to be resident in the L1 data cache. The range of efficiency for the unaligned cases results from measurements using several different possible unaligned load/store instruction sequences. The most efficient instruction sequence for unaligned vector data accesses typically differs between x64 processor families, and sometimes from generation-to-generation even within a family.

Percentage of Peak versus Latest-Generation x64 Processors				
	AMD		Intel	
Test	First-Gen AMD64	Latest-Gen AMD Opteron	First-Gen EM64T	Latest-Gen Intel Core 2
register-to-register scalar	50%	50%	25%	50%
register-to-register vector	50%	100%	50%	100%
8 bytes aligned	50%	100%	50%	100%
8 bytes unaligned	25-48%	50-90%	28%	38-40%
16 bytes aligned	47%	90%	38%	50%
16 bytes unaligned	22-25%	25-65%	17-20%	25%
24 bytes aligned	33%	65%	27%	33%
24 bytes unaligned	13-22%	17-65%	12-20%	17-25%
32 bytes aligned	24%	50%	20%	25%
32 bytes unaligned	10-14%	13-50%	10-13%	12-17%

Table 1. Percentage of latest-generation x64 SSE peak performance realized by various x64 processors

At 100% efficiency, the latest generation of processors can do four double precision floating point operations per cycle (two multiplies and two adds), meaning a Core 2 Duo processor-based laptop running at 2GHz is theoretically capable of 8 GFLOPS. The achievable peak vector double-precision performance is twice that of first-generation x64 processors, and twice that of scalar code, assuming that the data re-use is sufficiently high.

What should be of interest to performance-oriented programmers is the tail-off in achievable performance as the operations require more data, and when the data is fetched into the SSE registers in a sub-optimal (unaligned) manner. Even when the data resides in L1 cache, the gains of vector over scalar, and latest-generation over previous, may be limited for memory-intensive loops. Thus, it is extremely important to structure code in ways that maximize opportunities for aligned accesses and re-use of data within the SSE unit in order to avoid memory-related performance bottlenecks.

4. Optimizing the ALEGRA Kernel

The ALEGRA computational kernel and dataset used in this study represent a two-dimensional Lagrangian hydrodynamics model of a single ideal gas material. The code uses a cache oblivious implementation, by David Hensinger of Sandia, from work developed by Frigo and Strumpen. The cache oblivious implementation was presented in a paper at CUG in 2006, and provides cache locality by recursively walking through multiple time steps over subsets of the spatial data domain. Because of the cache oblivious implementation, memory bandwidth becomes less of an issue and the code is a good candidate to take advantage of the vectorization gains outlined above.

The kernel is written in C++. The original kernel stored data as an "array of structs", where variables representing acceleration, velocity, and force are interleaved in a node structure, and at each time step an element depends on all adjacent elements and nodes at the previous time step. While this may be optimal on direct-mapped or low-set-associativity caches, for x64 SSE vectorization it is suboptimal because it prevents the compiler from generating efficient packed aligned loads to move data into the SSE unit.

So, the first and most significant change to the kernel was to change the storage ordering. For instance, all like accelerations were collected into one vector, all like velocities, etc. This presents data to the compiler in a way that enables packed aligned memory accesses. From a coding standpoint, most of the changes were localized to loop setups where pointers where initialized, and required changes to the scaling and offsetting by the x, y, and struct data dimensions. The total amount of data stored

remained unchanged, as did the cache oblivious techniques in the code.

Once the code restructuring was complete, tests with the PGI C++ compiler ensued. The major goal was to get every loop in the computational kernel to vectorize. As it turned out, this required some compiler work besides just rewriting the application code.

The first section of the kernel contains a loop over elements to accumulate forces to nodes. This contains some special code for boundary conditions, but is mainly a 2x2 neighborhood operation, where a portion of the work is redundant from the previous iteration. PGI compilers implement an optimization called *loop-carried redundancy elimination* (LRE) to improve performance of such loops, but it was not occurring in C++ codes for reasons we won't detail here. Modifications to the compiler enabled loop-carried redundancy elimination for C++, and it is now on by default using the *–fast* compiler option. Occurrences of this optimization, and many others, can be seen using the *–Minfo* compiler option:

```
741, 4 loop-carried redundant expressions
    removed with 8 operations and 12 arrays
783, 4 loop-carried redundant expressions
    removed with 8 operations and 12 arrays
```

As indicated in these compile-time informational messages, the compiler has actually eliminated 8 floating-point operations and 12 memory references from the body of the loops at lines 741 and 783. LRE is an extremely effective optimization when it can be applied. A hint for programmers: to enable LRE to kick in, it is important to code all of the work for a given iteration in the loop, including the redundant parts, and let the compiler find the redundancies itself. Don't try to code it yourself, using temporaries that carry around to the next iteration of the loop. Hand optimizations like this, similar to excessive manual loop unrolling, can obscure the true structure of a loop and result in undue memory pressure due to overuse of local temporary variables.

A second important modification to the PGI C++ compiler was the addition of support for a "restrict" type qualifier. In C99, the `restrict` type qualifier is a part of the language specification. It serves as a "no alias" hint to the compiler on a pointer-by-pointer basis. The C++ language has nothing similar, but several C++ compilers include support for the extension `__restrict`, which carries the same meaning. The code was modified by adding `__restrict` type qualifiers to most of the pointers where they were declared and assigned locations within the x-y grid. This change provides the compiler information it needs to disambiguate potential loop-carried dependencies, removing a barrier to vectorization.

The last loop of the kernel contains a loop over the elements to update material properties before the next time step. To vectorize this loop, coding changes and compiler changes were needed. The original code looked as follows:

```
{
  real sound_speed = sqrt(gxgm1 * edata[ENER]);
  real local_ts = ar/sound_speed;

  if (local_ts/min_ts < 0.9999) min_ts=local_ts;
  edata[AVmPR] = gamma_minus_one * edata[DENS]
                * edata[ENER];
  if(tr_deformation_rate < 0.0) {
     edata[AVmPR] += edata[DENS] * ar
     * tr_deformation_rate *
     * (linear * sound_speed - quadratic * ar
        * tr_deformation_rate);
  }
}
```

and was rewritten to look as follows:

```
{
  real sound_speed = sqrt(gxgm1 * edataENER[i]);
  if (ar/sound_speed < local_ts)
    local_ts = ar/sound_speed;

  edataAVmPR[i] = -gamma_minus_one
                 * edataDENS[i] * edataENER[i];
  real tdrate = (tr_deformation_rate < 0.0) ?
    tr_deformation_rate : 0.0;
  tdrate *= ar;
  edataAVmPR[i] += edataDENS[i] * tdrate
                   *  (linear * sound_speed
                       - quadratic * tdrate);
}
if (local_ts/min_ts < 0.9999) min_ts=local_ts;
```

There are several changes here, and it is useful to go over them one-by-one to understand how they enable more loops to vectorize.

First, the change from `edata[ENER]` to `edateENER[i]` is a result of the restructuring of the storage order, so that like-data is vectored into sequential locations in memory rather than being split across multiple C structs.

Second, the computation of `local_ts` is different. The purpose of this bit of code is so the function can return the minimum computed time step, but only if that minimum (`local_ts`) is somewhat less than the timestep passed in to the function (`min_ts`). For this kernel, somewhat less is defined as

```
(local_ts/min_ts < 0.9999)
```

This creates a loop-carried dependence. If the condition is true, `min_ts` is updated, and it MUST be available for the computation in the next iteration.

As it turns out, there is also a slight subtlety in this computation and it is not exactly what the author intended. If `min_ts` does get updated to a value of `local_ts`, there is a chance that another later value of `local_ts` could be less than `min_ts`, but not enough less to update the `min_ts` value. So, as originally

written, the function may return a new `min_ts`, but not necessarily the minimum `local_ts` as computed over all elements.

The rewritten code always locates the `min_ts` value. In addition, the rewritten code removes the loop-carried dependence. Why? Because as it is now written, it finds the minimum of an array of values (`ar/sound_speed`), which is just a reduction, similar to a sum reduction. An improvement was made to the PGI compilers to enable vectorization of these min and max reductions, leveraging the hardware support for min and max operations in the x64 SSE unit. SSE min and max instructions have similar characteristics to adds and multiplies – with scalar & packed variants, support for the various data types, appropriate pipeline lengths, etc.

Once the vectorized reduction recognition was in place in the compiler, it enabled a rewrite of the original conditional at the end of this section of code. We noted that a temporary, `tdrate`, could be set to the min of `tr_deformation_rate` and 0.0. Assuming `tdrate` is zero, the final update just becomes a wasted operation (aka a "nop"). It may seem like a very expensive nop, but if it facilitates vectorization it is worthwhile.

There were a few other minor issues which could easily have been avoided with a re-work of the source code, but instead we took the opportunity to enhance the PGI C++ compiler to automatically eliminate these barriers to vectorization. The new optimization features presented above are all available in the commercial release of the PGI 7.0 PGC++ compiler. Work is continuing on PGC++ performance, and another set of improvements will be available in PGI 7.1.

4. Results

On first-generation x64 processors containing 64-bit wide floating-point units, the speedup attained is roughly 1.25x over the previous version of ALEGRA kernels. On the latest generation x64 processors containing 128-bit wide floating point units we have tested on thus far, the speedup attained is roughly 1.85x due to these code transformations and compiler enhancements which maximize the effectiveness of double-precision vectorization.

While performance improvements will obviously vary from application to application, these results highlight the need to structure code to maximize vectorization now and into the future.

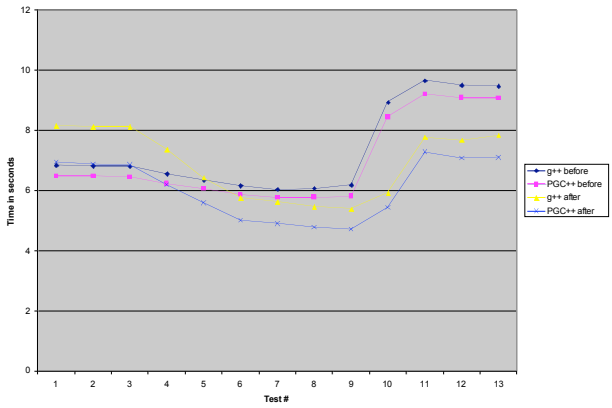

Diagram 2. First-generation x64 ALEGRA kernel performance before/after restructuring for double-precision vectorization

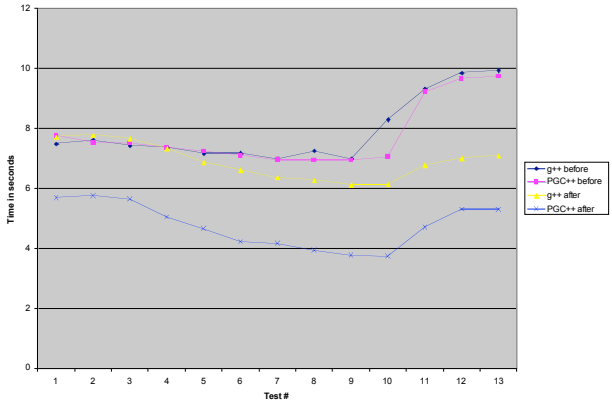

Diagram 3. Latest-generation x64 ALEGRA kernel performance before/after restructuring for double-precision vectorization

5. Conclusions

Although many application developers rely on just applying the "–fast" switch to optimise their code, significant performance gains may still be possible and it is important to apply classic performance analysis and tuning techniques to realize some of the untapped performance modern processor architectures can provide. However, this requires expert knowledge from not only the application developer, but also the compiler team, which in many instances has an expert knowledge of the underlying architecture and how to best utilize it for a given application's particular needs. In this case, it was possible to achieve a near 2x performance improvement.

Results derived from this work will benefit users of ALEGRA on current and future DOE and DoD computing resources. In addition, the compiler techniques developed have been integrated into the generally available PGI compiler suite to benefit all C/C++ codes

with similar source code structure. The coding practices that we outlined will also help other programmers in applying the same techniques and tools for their particular application. Finally, the cooperation between Sandia's application developers and the Portland Group product development and support teams is shown to be a model for future joint endeavours.

6. Future Work

As the trend to multiple cores per processor continues, it is anticipated that a similar trend will be observed in the number of floating-point math units per processor. This will most likely come about with an increase in the size of each core's vector unit to 256 bits, 512 bits or maybe even larger. Perhaps architectures will utilize multiple vector units per core? Or maybe even chain vector units from separate cores into a single virtual vector unit? And then there is the general purpose GPU trend with the potential for streaming multiple vector units together. Whatever the outcome, it will become increasingly important to revisit techniques abandoned with the advent of the cache based microprocessor in order to extract the full potential of the next generation of processors for science and engineering applications.

We will continue to examine the application programming and compiler techniques required to support larger vector operations that are most likely to appear in future sequential and parallel architectures deployed by the commercial processor vendors.

About the Authors

Douglas Doerfler is a Principal Member of Technical Staff at Sandia National Laboratories. He started his career at Sandia 22 years ago in the field of analog instrumentation, then migrated to embedded computing, then to embedded high performance computing, and that lead to an interest in supercomputing. His current job responsibilities include leading Sandia's HPC Advanced Systems Group and performing research in computational architectures and performance modeling. He can be reached by e-mail at *dwdoerf@sandia.gov*.

David Hensinger is a Principal Member of Technical Staff at Sandia National Laboratories. He is part of the Computational Shock and Multi-Physics department and an ALEGRA developer. His work at Sandia has included thermal analysis, robotics automation, solid modelling, computational geometry, and parallel mesh generation. He can be reached by e-mail at *dmhensi@sandia.gov*.

Brent Leback is the Applications Engineering manager for PGI. He has worked in various positions over the last 20 years in HPC customer support, math library development, applications engineering and consulting at QTC, Axian, PGI and STMicroelectronics. He can be reached by e-mail at *brent.leback@pgroup.com*.

Douglas Miles is responsible for all business and technical operations of The Portland Group (PGI). He has worked in various positions over the last 20 years in HPC applications engineering, math library development and technical marketing at Floating Point Systems, Cray Research Superservers, PGI and STMicroelectronics. He can be reached by e-mail at *douglas.miles@pgroup.com*.

References

1. Hensinger, Luchini, Frigo and Strumpen, *Performance Characteristics of Cache Oblivious Implementation Strategies for Hyperbolic Equations on Opteron Based Super Computers*, CUG 2006 Proceedings.

2. Frigo, Leiserson, Prokop and Ramachandran, *Cache Oblivious Algorithms*, Proc. 40th Annual Symp. Foundations of Computer Science (FOCS'99).

3. Frigo and Strumpen, *Cache Oblivious Stencil Computations*, Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05).

4. D. Miles, B. Leback and D. Norton, *Optimizing Application Performance on Cray Systems with PGI Compilers and Tools*, CUG 2006 Proceedings.

5. Michael Wolfe, *High Performance Compilers for Parallel Computers,* Addison-Wesley, 1996.

6. Bik, Girkar, Grey & Tian, *Automatic Intra-register Vectorization for the Intel Architecture*, IJPP Vol. 30, No. 2, April 2002.