Targeting AVX-Enabled Processors Using PGI Compilers and Tools

Brent Leback, John Merlin, *and* **Steven Nakamoto,**
The Portland Group (PGI)

ABSTRACT: *AMD and Intel are releasing new microprocessors in 2011 with extended AVX support. In this paper we show examples of compiler code generation and new library and tools capabilities which support these new processors. We also discuss performance issues, comparing the new platforms versus previous generations.*

KEYWORDS: Compiler, Optimization, Micro-architecture

1. Introduction

At CUG 2007 co-authors of this paper presented the characteristics and performance of the first and second generation x64 processors from AMD and Intel [1]. In particular, these second generation micro-architectures, the AMD "Barcelona" and the Intel "Woodcrest", were the first to use 128-bit wide data paths and floating point units. For the first time on the x86 architecture, vectorization of double-precision codes could enable a 2x speedup over non-vectorized, a.k.a. scalar, codes.

Vectorization [2] for x64 processors is used to identify and transform loops to take advantage of packed SSE instructions, which are wide operations on data from multiple iterations of a loop. PGI produced the first x86/SSE automatic vectorizing compiler in 1999. At that time, Intel was providing and promoting a set of SSE intrinsics. Today, there are 9 different header files of intrinsics in the latest Intel compiler distribution, covering each generation of hardware updates going back to the MMX days. In total, they define 781 different routines. Unlike these intrinsics, vectorization is a technology which provides both code and performance portability.

In the CUG 2007 paper it was shown how loop-carried redundancy elimination (LRE), the `restrict` type qualifier for C and C++, ordering data sequentially in memory, and coding reductions so that they can be vectorized by the compiler resulted in an 1.80x speedup on a Sandia benchmark kernel.

The next year, at CUG 2008, we presented a paper that studied the ratio of data bandwidth to peak floating point performance on x64 processors [3]. That year, the first quad-core chips were beginning to show up from AMD and Intel. Most importantly, we showed that the peak transfer rate to and from memory divided by the peak floating point performance had trended dramatically downward over the generations of x64 processors, beginning in 2003. Also in 2008, both Intel and AMD announced future chips, and it was feared that the trend would continue. In May of 2008, we expected these new chips to show up in roughly 12 to 18 months, before the end of 2009. Diagram 1 is a slide from that presentation.

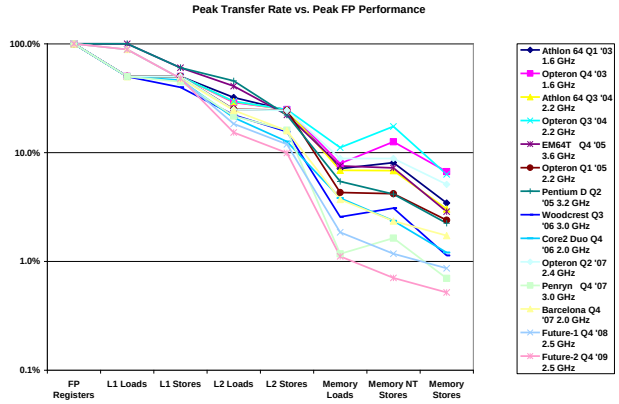


Diagram 1. The decline of peak rate of data transfer compared to peak floating point performance, from a presentation in 2008.

This year, 2011, a new generation of x64 processors containing Advanced Vector Extensions (AVX) [4] are being released by Intel and AMD. Intel has named its micro-architecture "Sandy Bridge" and AMD is using the name "Bulldozer" for their design. While by some accounts these chips may be two years late in arriving, the good news for the High Performance Computing (HPC) community is that the memory hierarchy has had a chance to "catch up" in that time. In our recent tests, all new Intel and AMD memory operation rates divided by the peak floating point performance of the new chips have stayed above the 1% mark. This is an impressive result given the increasing number of cores and wider floating point units in the new chips, and can easily be measured using a benchmark such as Stream [5].

2. The VEX Prefix, 256-bit SIMD Support

Both new processor architectures, the Sandy Bridge from Intel and the Bulldozer from AMD, support the new instruction encoding format called VEX. This instruction prefix allows a set of extensions to the traditional x86 instruction set architecture (ISA) that enable most of the new chip features that are important to compiler implementers, library developers, and applications staff who care about performance at the micro-architecture level.

The VEX prefix enables three and four operand syntax. As we will explain later, even if your code does not take advantage of the wider SIMD floating point units, you can still possibly see performance gains due to the flexibility of these new operations. Traditional x86 instructions contain two operands, where one operand usually plays the role of both a source and destination, thus the operation is destructive to one of the sources. In complicated floating point loops, the compiler may have to insert many register-to-register copy operations, which take slots in the pipelines and increase code size.

For instance, if you need to keep two inputs x and y in xmm0 and xmm1, while adding x+y, up to this point the compiler might generate:

```
movsd   %xmm1, %xmm2
addsd   %xmm0, %xmm2
```

and now with VEX enablement it can generate one instruction:

```
vaddsd %xmm0, %xmm1, %xmm2
```

But probably the most anticipated feature of the new chips for scientists and engineers is the doubling of the SIMD vector width and the width of the floating point pipeline. The new floating point units from both AMD and Intel are now capable of performing four double precision floating-point multiplies and four double precision floating-point adds every cycle. If you use single precision data, you can get twice that. In rough terms, using tuned libraries such as MKL from Intel and ACML from AMD, users could see 20 or more GFlops/core in double precision routines such as dgemm, and 40 or more GFlops/core single precision, depending on the clock frequency.

A new set of registers have been introduced, the ymm registers, which overlay the SSE xmm registers that were introduced with the Pentium III. Instructions operating on these registers are enabled by the VEX prefix. Diagram 2 shows how they are mapped onto the existing hardware.

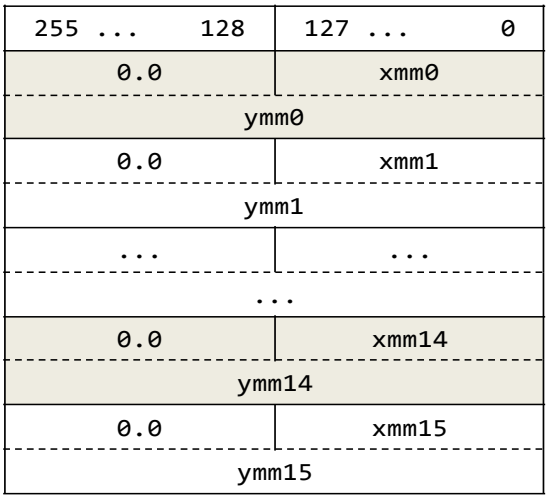
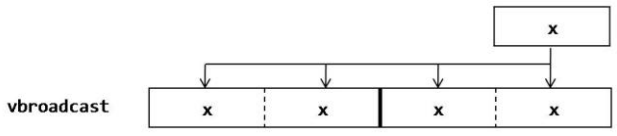


Diagram 2. 16 256-bit wide ymm registers overlay and extend the current 128-bit wide xmm registers.

A complicating factor for compiler writers is that the new instructions available for manipulating ymm registers are somewhat limited. The 256-bit SIMD units are more or less divided into two separate lanes of 128-bits each. There are only four instructions which can move data across lane boundaries, and three of the four only operate on 128-bit chunks:



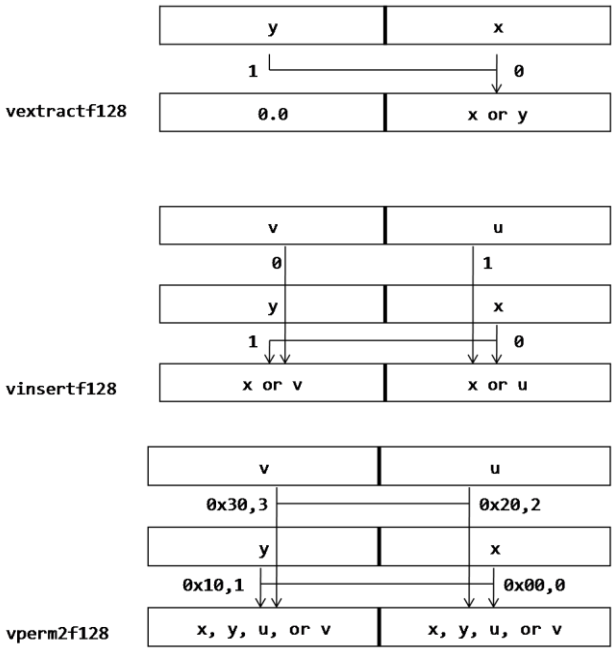


Diagram 3. The four operations which transfer data across AVX lanes. The vbroadcast source must be a memory location, but can be 32, 64, or 128 bits wide.

As an example of how this can become complicated, consider the following simple Fortran subroutine:

```
subroutine sum5(a, c, n)
real*8 a(n+4), c(n)
do i = 1, n
 c(i)=a(i)+a(i+1)+a(i+2)*2.d0+a(i+3)+a(i+4)
end do
end
```

Compiling this for execution on a second generation x64 processor yields this CCFF [6] message:

sum5:
3, 2 loop-carried redundant expressions removed with 2 operations and 4 arrays
Generated 4 alternate versions of the loop
Generated vector sse code for the loop
Generated a prefetch instruction for the loop

Using LRE optimization, the compiler finds redundant expressions in the intermediate sums and carries those around to the next iteration. When using xmm registers, the generated code for this loop requires one shuffle, but then produces two results for every three packed add operations:

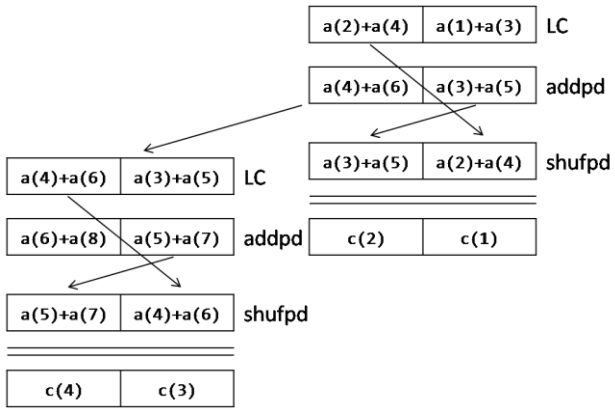


Diagram 4. XMM-based sum5 loop. LC labeled values are loop-carried between iterations.

Now with AVX and the dual-lane nature of the floating point units, the processor can perform twice as many additions per cycle, but the staging and manipulation of the data is more complex. To enable the same operation-reducing strategy requires the new VEX vperm2f128 instruction to move data between lanes.

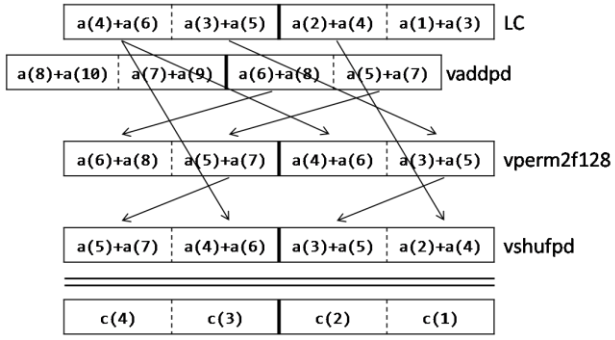


Diagram 5. YMM-based sum5 loop.

With the latest generation processors, we can produce four results for every three packed add operations, at the cost of one cross-lane transfer and one shuffle within lanes. Contrast this with the scalar, straight-forward method which requires five scalar adds (or four adds and one multiply) for every element, and you can begin to see the gains that are possible with an optimizing compiler. Of course, with an uncountable number of loop construct possibilities, heuristics and strategies need to be developed into our compilers for optimal code generation, and that work is well underway.

Another complicating factor for AVX code generation is the difficulty in generating aligned loads and stores. For ymm registers, the most optimal alignment is

32 bytes. Using current toolchains, 32 byte alignment can be forced for static data, but is not produced by most malloc() implementations, nor required of the stack by the ABIs. Thus it is not easy to force 32 byte alignment for local variables. We showed in our 2007 CUG paper that unaligned loads and stores, even when the data resides in L1 cache, run substantially slower than aligned loads, and that is still true with today's processors. While it would be an easy fallback position for the compiler to generate a loop schedule for the sum5 subroutine similar to what is shown in Diagram 6, we've found on both Intel and AMD processors that this runs slower than the original, non-AVX code. In fact, we've verified that in some cases breaking up a single 256-bit unaligned load or store (a movupd operation) into two 128-bit unaligned operations can actually run faster.

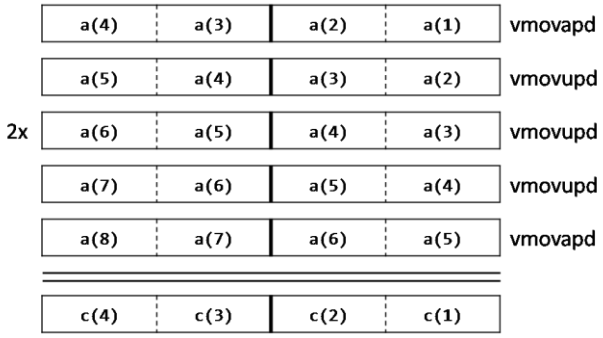


Diagram 6. Straight-forward implementation performs poorly due to unaligned memory accesses.

Finally we should mention the new AVX masked move instruction. While we've made some use of this capability in our new hand-coded AVX-enabled library routines, because the vmaskmov load operation loads a zero if the mask is zero, and there is no corresponding way to mask the exception generation, another level of compiler analysis (or recklessness) would be needed for the compiler to make use of this generally for residual loops. Performance issues currently make it unappealing for commonly executed code paths.

3. AMD-Specific Code Generation

The AMD Bulldozer chip not only implements the AVX instruction set, but extends it by adding FMA4 instructions as well. FMA, which stands for fused multiply-add, has not, up to this point, been a part of the x86 instruction set, but has long been an instrumental part of traditional embedded processors. In an FMA operation, the result of the multiplier directly feeds the adder, without being rounded or moved to a register. This results in a shorter total pipeline for common multiply-followed-by-add operations found in signal and image processing, matrix multiplication, dot products, and other BLAS routines, and in FFTs. The FMA4 instructions use four operands total, three for sources, and one destination.

On the Bulldozer chips, the only way to get peak floating-point performance is to use FMA4 instructions. It is easiest to model the performance by thinking of a single pipelined floating-point unit with three inputs, and on each cycle it can initiate some variant of an add, a multiply, or a multiply-add. The results may come out of the unit at different points (cycles) in the pipe, but the input ports basically control the rate of execution.

Because of this design, legacy objects and executables written or compiled using a style of vector packed adds and packed multiply operations will never run at peak performance on a Bulldozer chip. It is important to recompile using the proper target processor flags, and use Bulldozer-tuned math libraries, for best performance.

As an added wrinkle, the floating point unit on a Bulldozer chip is shared between two integer cores. The two integer cores can perform address generation support and sequencer control for the floating-point operations independently, of course.

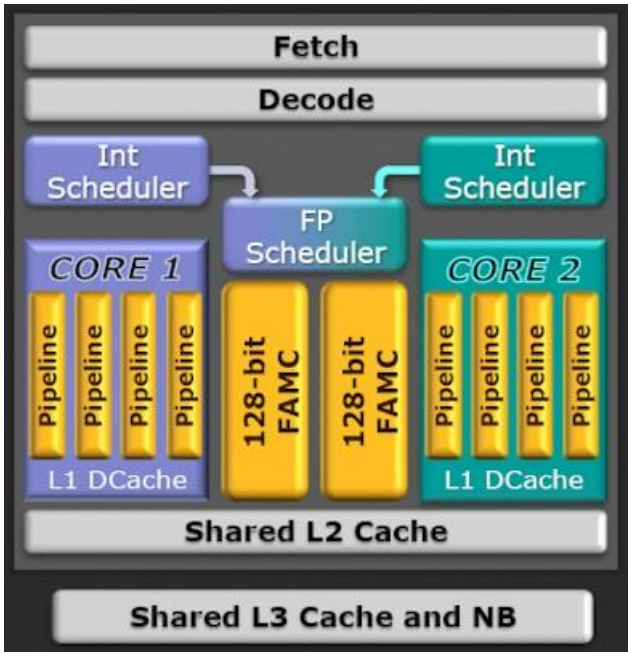


Diagram 7. Block diagram of a Bulldozer module containing two integer cores and a shared floating-point unit.

What is interesting about the Bulldozer chip is that you can compile and tune your floating-point codes to run in 256-bit wide "AVX Mode", where each thread assumes control of, and schedules ymm-based operations

over, the entire floating point unit. Or, you can compile for a "Shared Mode", where each thread uses either traditional SSE code generation, or newer VEX-based operations on xmm registers, and utilizes just one lane of the shared floating point resource. There are VEX-based FMA4 opcodes which work on xmm registers and these can be used to achieve peak performance on multi-threaded codes.

Diagram 8 shows two versions of the generated code for a daxpy routine and the FMA4 operations that are used. The PGI compiler can produce either mode based on compiler command-line options. At this point, it is still too early to determine which mode should be default.

```
.LB1_427:
  vmovapd  (%rax,%r9), %xmm1
  vfmaddpd %xmm1,(%rax,%r10),%xmm0,%xmm3
  vmovapd  %xmm3, (%rax,%r9)
  vmovapd  16(%rax,%r9), %xmm2
  vfmaddpd %xmm2,16(%rax,%r10),%xmm0,%xmm3
  vmovapd  %xmm3, 16(%rax,%r9)
  . . .
  addq     $64, %rax
  subl     $8, %edi
  testl    %edi, %edi
  jg       .LB1_427
-------------------------------
.LB1_427:
  vmovapd  (%rax,%r9), %ymm2
  vfmaddpd %ymm2,(%rax,%r10),%ymm0,%ymm3
  vmovapd  %ymm3, (%rax,%r9)
  vmovapd  32(%rax,%r9), %ymm4
  vfmaddpd %ymm4,32(%rax,%r10),%ymm0,%ymm3
  vmovapd  %ymm3, 32(%rax,%r9)
  . . .
  addq     $128, %rax
  subl     $16, %edi
  testl    %edi, %edi
  jg       .LB1_427
```

Diagram 8. Partial assembly output for two versions of daxpy(), the top using 128-bit xmm registers and the bottom using 256-bit ymm registers. Each is unrolled to perform four vfmaddpd operations per loop iteration.

Given the discussions in the previous section concerning the difficulties manipulating data between lanes, and guaranteeing aligned loads and stores of 256-bit data, it might turn out that for Bulldozer targets, 128-bit VEX enabled code is the best approach. Also given the graph in the introduction, for codes where the data is not all in registers or the lowest level caches, the floating point unit might be idle for many cycles. Tighter code might run more efficiently than wider-vector code, especially when it is multi-threaded. We will continue our experiments in this area.

Users should be aware that results obtained using FMA operations may differ in the lowest bits from results obtained on other X64 processors. The intermediate result fed from the multiplier to the adder is not rounded to 64 bits.

Bulldozer code generation is enabled in the PGI compiler suite by using the –tp=bulldozer command line option.

4. Intel-Specific Code Generation

Of course if you run either of the above loops on a Sandy Bridge machine, or any Intel-based processor for that matter, you get this result:

```
bash-4.1$ ./a.out
Illegal instruction (core dumped)
```

There seems to be no question that on a Sandy Bridge processor it is best to run simple, cache-based loops using 256-bit wide AVX enabled code. The optimal code that we've generated for the same daxpy operation is shown in Diagram 9.

```
.LB1_485:
  vmulpd  (%rdi,%r10), %ymm0, %ymm1
  vaddpd  (%rdi,%r9), %ymm1, %ymm2
  vmovapd %ymm2, (%rdi,%r9)
  vmulpd  32(%rdi,%r10), %ymm0, %ymm1
  vaddpd  32(%rdi,%r9), %ymm1, %ymm2
  vmovapd %ymm2, 32(%rdi,%r9)
  . . .
  addq    $128, %rdi
  subl    $16, %eax
  testl   %eax, %eax
  jg      .LB1_485
```

Diagram 9. Partial assembly output for a version of daxpy() targeting Sandy Bridge.

This version runs fairly well on an AMD Bulldozer system, but it is not optimal. On both chips, loops like this are limited by the L1 bandwidth. For instance, the Sandy Bridge has two 128-bit ports for loading data from L1 cache, and so it is capable of loading four double precision floating-point values every cycle. This daxpy operation using ymm registers could consume eight inputs each cycle: four for the multiplier and four for the adder. And on Sandy Bridge, as is typical of many x64 processors, the bandwidth storing to L1 cache is half of that for loads.

One final issue worth mentioning concerning Sandy Bridge code generation is the need for the vzeroupper instruction. This instruction zeroes out the upper 128 bits of all of the ymm registers and marks them as clean. If you mix 256-bit AVX instructions with legacy SSE instructions that use xmm registers, you will incur performance penalties of roughly one hundred cycles at each transition between legacy code and AVX code. This is because the processor must save the state of the upper 128-bits of the ymm registers at the first legacy instruction so that they can be restored upon issuing another AVX instruction. To avoid this problem, use the vzeroupper before the transition between these two code sequences can occur. This instruction seems to have minimal overhead.

In the PGI compiler, if we are generating AVX instruction sequences, we will generate the vzeroupper instruction right before a call is made. This is because we cannot be sure how the callee has been compiled. Also, when writing library routines or compiling functions that perform AVX instruction sequences, we generate a vzeroupper instruction right before returning, again because we cannot make assumptions about how the caller was compiled. This is always safe because the ABI never specifies that parameters are passed to, or returned in, the high half of the ymm registers.

Sandy Bridge code generation is enabled in the PGI compiler suite by using the –tp=sandybridge command line option.

5. Libraries and Tools

PGI is well on the way to finishing their port of the runtime libraries for these two new architectures. We have added VEX encoding and FMA4 instructions where appropriate to produce new scalar transcendental and other functions for single and double precision. We are also working on new 128-bit and 256-bit wide vector versions, which can produce up to 4 double precision and 8 single precision results per invocation. The port to the new architectures has not been necessarily straight-forward, as there are decisions and tradeoffs made in the balance of integer, table lookup address generation, and floating point operations that must be tuned for optimal performance.

PGI's GUI-based development tools, PGDBG and PGPROF, have been updated to support AVX. Both tools utilize a disassembler which has been updated to properly handle AVX and FMA4 code sequences. PGDBG has been recently enhanced to show ymm register contents in a variety of formats: 32 bit floats, 64 bit floats, and hexadecimal. The GUI-based tools are both capable of debugging and profiling OpenMP and MPI parallel programs.

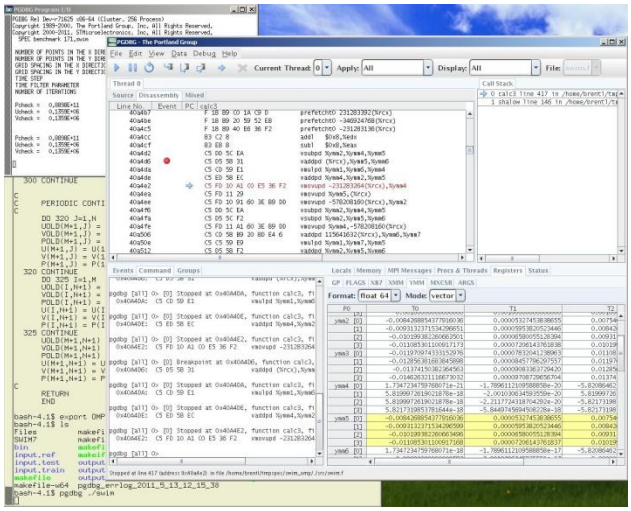


Diagram 10. PGDBG, the PGI debugger showing debugging support for AVX code and viewing ymm register contents.

6. Concluding Remarks and Future Work

We've shown that the new AVX-enabled processors require substantial changes in the compiler generated code to achieve optimal performance. The major points of difference are well understood, but we expect tuning to continue for the foreseeable future, and continue to evolve as new revisions of these architectures are released. There is not only compiler work to do; we rely on changes to the entire software stack, from the OS which has to save and restore the new registers, down to new runtime libraries for math, i/o, and memcpy-type operations, some of which PGI provides ourselves. We've shown an example where an incompatible mix of code can lead to disastrous performance penalties. This leads to increased burdens (or opportunities) for support and training.

Despite these changes to the underlying hardware, vectorizing compilers allow code and performance portability throughout the processor generations. There is no need for programmers to perform tedious manual unrolling of their loops, or for them to insert SSE intrinsic calls, to take advantage of the latest hardware features.

The PGI Unified Binary technology [7] can assist ISVs and other users who need to produce optimal code paths for all possible x64 architectures in a single executable. Though the need for this capability has

waned in the last few years, with the release of the Bulldozer processor and its use of the FMA4 instruction, care must be taken in choosing a set of target processors to avoid instruction faults.

About the Authors

Brent Leback is the Engineering Manager at PGI. He has worked in various positions over the last 27 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*.

John Merlin joined The Portland Group as a compiler engineer in 1999. From 1986 to 1999 he was a research fellow at the University of Southampton, UK, and then at VCPC, Vienna, Austria, with research interests including computational physics, automatic parallelisation, parallel programming languages and models, and compiler technology. His e-mail address is *john.merlin @pgroup.com*.

Steven Nakamoto is the Compiler Architect at PGI. Prior to joining PGI in 1989, he worked in various software and compiler engineering positions at Floating Point Systems, BiiN, Mentor Graphics, and Honeywell Information Systems. He can be reached by e-mail at *steven.nakamoto@pgroup.com*.

References

[1] Doerfler, Hensinger, Miles, and Leback, *Tuning C++ Applications for the Latest Generation x64 Processors with PGI Compilers and Tools*, CUG 2007 Proceedings

[2] Michael Wolfe, *High Performance Compilers for Parallel Computers,* Addison-Wesley, 1996

[3] Leback, Doerfler, Heroux, *Performance Analysis and Optimization of the Trilinos Epetra Package on the Quad-Core AMD Opteron Processor*, CUG 2008 Proceedings

[4] Intel Advanced Vector Extensions Programming Reference, http://software.intel.com/en-us/avx

[5] STREAM: Sustainable Memory Bandwidth in High Performance Computers, www.cs.virginia.edu/stream

[6] CCFF - Common Compiler Feedback Format, *www.pgroup.com/resources/ccff.htm*

[7] PGI Unified Binary, www.pgroup.com/resources/ unified_binary.htm