Optimizing Application Performance on Cray Systems with PGI® Compilers and Tools

May 2006

Doug Miles – douglas.miles@st.com
Brent Leback – brent.leback @st.com
Dave Norton – norton@hpfa.com

The Portland Group
www.pgroup.com

The Portland Group

Introduction

- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating examples
- **Function Inlining –** especially important for C and C++
- **Parallelization** – for Cray XD1 and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try
- **Conclusion**

Vectorizable F90 Array Syntax
Data is REAL*4

```
350 !
351 !   Initialize vertex, similarity and coordinate arrays
352 !
353     Do Index = 1, NodeCount
354       IX = MOD (Index - 1, NodesX) + 1
355       IY = ((Index - 1) / NodesX) + 1
356       CoordX (IX, IY) = Position (1) + (IX - 1) * StepX
357       CoordY (IX, IY) = Position (2) + (IY - 1) * StepY
358       JetSim (Index)  = SUM (Graph (:, :, Index) * &
359     &                   GaborTrafo (:, :, CoordX(IX,IY), CoordY(IX,IY)))
360       VertexX (Index) = MOD (Params%Graph%RandomIndex (Index) - 1, NodesX) + 1
361       VertexY (Index) = ((Params%Graph%RandomIndex (Index) - 1) / NodesX) + 1
362     End Do
```

Inner “loop” at line 358 is vectorizable, can used packed SSE instructions

–fastsse to Enable SSE Vectorization
–Minfo to List Optimizations to stderr

```
% pgf95 -fastsse -Mipa=fast -Minfo -S graphRoutines.f90
…
localmove:
   334, Loop unrolled 1 times (completely unrolled)
   343, Loop unrolled 2 times (completely unrolled)
   358, Generated an alternate loop for the inner loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
   …
```

Scalar SSE:

```
.LB6_668:
# lineno: 358
        movss   -12(%rax),%xmm2
        movss   -4(%rax),%xmm3
        subl    $1,%edx
        mulss   -12(%rcx),%xmm2
        addss   %xmm0,%xmm2
        mulss   -4(%rcx),%xmm3
        movss   -8(%rax),%xmm0
        mulss   -8(%rcx),%xmm0
        addss   %xmm0,%xmm2
        movss   (%rax),%xmm0
        addq    $16,%rax
        addss   %xmm3,%xmm2
        mulss   (%rcx),%xmm0
        addq    $16,%rcx
        testl   %edx,%edx
        addss   %xmm0,%xmm2
        movaps  %xmm2,%xmm0
        jg      .LB6_625
```

Vector SSE:

```
.LB6_1245:
# lineno: 358
        movlps  (%rdx,%rcx),%xmm2
        subl    $8,%eax
        movlps  16(%rcx,%rdx),%xmm3
        prefetcht0  64(%rcx,%rsi)
        prefetcht0  64(%rcx,%rdx)
        movhps  8(%rcx,%rdx),%xmm2
        mulps   (%rsi,%rcx),%xmm2
        movhps  24(%rcx,%rdx),%xmm3
        addps   %xmm2,%xmm0
        mulps   16(%rcx,%rsi),%xmm3
        addq    $32,%rcx
        testl   %eax,%eax
        addps   %xmm3,%xmm0
        jg      .LB6_1245:
```

Facerec Scalar: 104.2 sec
Facerec Vector: 84.3 sec

Vectorizable C Code Fragment?

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Minfo functions.c
func4:
    221, Loop unrolled 4 times
    221, Loop not vectorized due to data dependency
    223, Loop not vectorized due to data dependency
```

Pointer Arguments Inhibit Vectorization

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Minfo functions.c
func4:
     221, Generated vector SSE code for inner loop
          Generated 3 prefetch instructions for this loop
     223, Unrolled inner loop 4 times
```

C Constant Inhibits Vectorization

```
217   void func4(float *u1, float *u2, float *u3, …
      …
221   for (i = -NE+1, p1 = u2-ny, p2 = n2+ny; i < nx+NE-1; i++)
222         u3[i] += clz * (p1[i] + p2[i]);
223   for (i = -NI+1, i < nx+NE-1; i++) {
224         float vdt = v[i] * dt;
225         u3[i] = 2.*u2[i]-u1[i]+vdt*vdt*u3[i];
226   }
```

```
% pgcc –fastsse –Msafeptr –Mfcon –Minfo functions.c
func4:
    221, Generated vector SSE code for inner loop
         Generated 3 prefetch instructions for this loop
    223, Generated vector SSE code for inner loop
         Generated 4 prefetch instructions for this loop
```

-Msafeptr Option and Pragma

–M[no]safeptr[=all | arg | auto | dummy | local | static | global]

all	All pointers are safe
arg	Argument pointers are safe
local	local pointers are safe
static	static local pointers are safe
global	global pointers are safe

#pragma [*scope*] [no]safeptr={arg | local | global | static | all},…

Where *scope* is *global*, *routine* or *loop*

Common Barriers to SSE Vectorization

- **Potential Dependencies & C Pointers** – Give compiler more info with –Msafeptr or pragmas
- **Function Calls** – Try inlining with –Minline or –Mipa=inline
- **Large Number of Statements –** Try –Mvect=nosizelimit
- **Too few iterations** – Usually better to unroll the loop
- **Real dependencies** – Must restructure loop, if possible

Barriers to Efficient Execution of Vector SSE Loops

- Not enough work – vectors are too short
- Vectors not aligned to a cache line boundary
- Discussion of –Mcache_align

- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating example
- **Function Inlining –** especially important for C and C++
- **Parallelization** – for Cray XD1 and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try
- **Conclusion**

What can Interprocedural Analysis and Optimization with –Mipa do for You?

- Interprocedural constant propagation
- Pointer disambiguation
- Alignment detection, Alignment propagation
- Global variable mod/ref detection
- F90 shape propagation
- Function inlining
- IPA optimization of libraries, including inlining

Effect of IPA on the WUPWISE Benchmark

PGF95 Compiler Options	Execution Time in Seconds
–fastsse	156.49
–fastsse –Mipa=fast	121.65
–fastsse –Mipa=fast,inline	91.72

- –Mipa=fast => constant propagation => compiler sees complex matrices are all 4x3 => completely unrolls loops
- –Mipa=fast,inline => small matrix multiplies are all inlined

Using Interprocedural Analysis

- Must be used at both compile time and link time
- Non-disruptive to development process – edit/build/run
- Speed-ups of 5% - 10% are common
- –Mipa=safe:<*name*> - safe to optimize functions which call or are called from unknown function/library *name*
- –Mipa=libopt – perform IPA optimizations on libraries
- –Mipa=libinline – perform IPA inlining from libraries

- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating examples
- **Function Inlining –** especially important for C and C++
- **SMP Parallelization** – for Cray XD1 and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try
- **Conclusion**

Explicit Function Inlining

–Minline[=[lib:]<inlib> | [name:]<func> | except:<func> | size:<n> | levels:<n>]

[lib:]<inlib>	Inline extracted functions from *inlib*
[name:]<func>	Inline function func
except:<func>	Do not inline function func
size:<n>	Inline only functions smaller than n statements (approximate)
levels:<n>	Inline n levels of functions

For C++ Codes, PGI Recommends IPA-based inlining or –Minline=levels:10!

- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating examples
- **Function Inlining –** especially important for C and C++
- **SMP Parallelization** – for Cray XD1 and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try
- **Conclusion**

SMP Parallelization

- –Mconcur for auto-parallelization on multi-core
  - Compiler strives for parallel outer loops, vector SSE inner loops
  - –Mconcur=innermost forces a vector/parallel innermost loop
  - –Mconcur=cncall enables parallelization of loops with calls
- –mp to enable OpenMP 2.5 parallel programming model
  - See PGI User's Guide or OpenMP 2.5 standard
  - OpenMP programs compiled w/out –mp "just work"
  - Not supported on Cray XT3 – would require some custom work
- –Mconcur and –mp can be used together!

MGRID Benchmark Main Loop

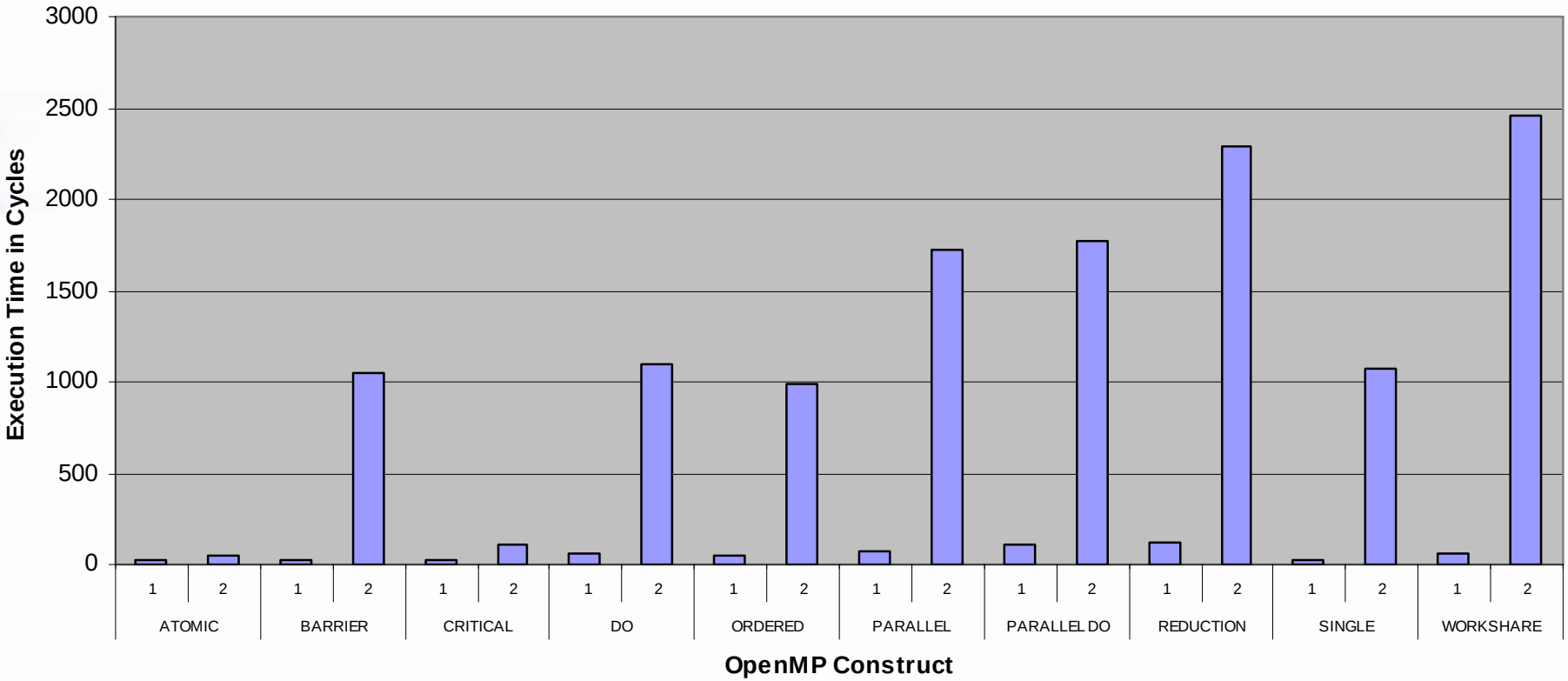
```
      DO 10 I3=2, N-1
      DO 10 I2=2,N-1
      DO 10 I1=2,N-1
10          R(I1,I2,I3) = V(I1,I2,I3)
     &                  -A(0)*(U(I1,I2,I3))
     &                  -A(1)*(U(I1-1,I2,I3)+U(I1+1,I2,I3)
     &                        +U(I1,I2-1,I3)+U(I1,I2+1,I3)
     &                        +U(I1,I2,I3-1)+U(I1,I2,I3+1))
     &                  -A(2)*(U(I1-1,I2-1,I3)+U(I1+1,I2-1,I3)
     &                        +U(I1-1,I2+1,I3)+U(I1+1,I2+1,I3)
     &                        +U(I1,I2-1,I3-1)+U(I1,I2+1,I3-1)
     &                        +U(I1,I2-1,I3+1)+U(I1,I2+1,I3+1)
     &                        +U(I1-1,I2,I3-1)+U(I1-1,I2,I3+1)
     &                        +U(I1+1,I2,I3-1)+U(I1+1,I2,I3+1) )
     &                  -A(3)*(U(I1-1,I2-1,I3-1)+U(I1+1,I2-1,I3-1)
     &                        +U(I1-1,I2+1,I3-1)+U(I1+1,I2+1,I3-1)
     &                        +U(I1-1,I2-1,I3+1)+U(I1+1,I2-1,I3+1)
     &                        +U(I1-1,I2+1,I3+1)+U(I1+1,I2+1,I3+1))
```

Auto-parallel MGRID Overall Speed-up is 40% on Dual-core AMD Opteron

```
% pgf95 –fastsse –Mipa=fast,inline –Minfo –Mconcur mgrid.f
resid:
. . .
    189, Parallel code for non-innermost loop activated
             if loop count >= 33; block distribution
    291, 4 loop-carried redundant expressions removed
             with 12 operations and 16 arrays
         Generated vector SSE code for inner loop
         Generated 8 prefetch instructions for this loop
         Generated vector SSE code for inner loop
         Generated 8 prefetch instructions for this loop
```

EPCC 2.0 OpenMP Microbencmarks

Dual-core 2.2Ghz AMD Opteron, SuSE 9.3
PGI 6.1 Options: -fastsse -Mipa=fast,inline -mp



- **Vectorization** – packed SSE instructions maximize performance
- **Interprocedural Analysis (IPA)** – use it! motivating examples
- **Function Inlining –** especially important for C and C++
- **SMP Parallelization** – for Cray XD1 and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try
- **Conclusion**

Miscellaneous Optimizations (1)

- –Mfprelaxed – single-precision sqrt, rsqrt, div performed using reduced-precision reciprocal approximation
- –lacml and –lacml_mp – link in the AMD Core Math Library
- –Mprefetch=d:<p>,n:<q> – control prefetching distance, max number of prefetch instructions per loop
- –tp k8-32 – can result in big performance win on some C/C++ codes that don't require > 2GB addressing; pointer and long data become 32-bits

Miscellaneous Optimizations (2)

- –O3 – more aggressive hoisting and scalar replacement; not part of –fastsse, always time your code to make sure it's faster
- For C++ codes: ––no_exceptions –Minline=levels:10
- –M[no]movnt – disable / force non-temporal moves
- –V[version] to switch between PGI releases at file level

Conclusion

- Use –fastsse => drive for maximize SSE vectorization, learn common barriers to vectorization and avoid them
- Use –Mipa=fast or –Mipa=fast,inline => IPA and inlining often add 5% - 10% or more to performance
- Function Inlining – especially critical for C++; use –Minline=levels:10 if you are not using IPA inlining
- Parallelization – for Cray XD1 and multi-core processors
- Miscellaneous Optimizations – hit or miss, but worth a try; sometimes make a huge difference in single-node performance