# Compilers

Tools for Scientists and Engineers

*September 2007*

Jason Beech-Brandt – jason@cray.com
Kevin Roy – kroy@cray.com

www.cray.com

# Outline of Today's Topics

- **Introduction to PGI Compilers and Tools**
- **Documentation.  Getting Help**
- **Basic Compiler Options**
- **Optimization Strategies**
- **Questions and Answers**

## PGI Compilers and Tools, features

- **Optimization** – State-of-the-art vector, parallel, IPA, Feedback, …
- **Cross-platform** – AMD & Intel, 32/64-bit, Linux & Windows
- **PGI Unified Binary for AMD and Intel processors**
- **Tools** – Integrated OpenMP/MPI debug & profile, IDE integration
- **Parallel** – MPI, OpenMP 2.5, auto-parallel for Multi-core
- **Comprehensive OS Support** – Red Hat 7.3 – 9.0, RHEL 3.0/4.0, Fedora Core 2/3/4/5, SuSE 7.1 – 10.1, SLES 8/9/10, Windows XP, Windows x64

# PGI Tools Enable Developers to:

- **View x64 as a unified CPU architecture**
- **Extract peak performance from x64 CPUs**
- **Ride innovation waves from both Intel and AMD**
- **Use a single source base and toolset across Linux and Windows**
- **Develop, debug, tune parallel applications for Multi-core, Multi-core SMP, Clustered Multi-core SMP**

# PGI Documentation and Support

- **PGI provided documentation**
- **PGI User Forums, at www.pgroup.com**
- **PGI FAQs, Tips & Techniques pages**
- **Email support, via trs@pgroup.com**
- **Web support, a form-based system similar to email support**
- **Fax support**

# PGI Docs & Support, cont.

- **Legacy phone support, direct access, etc.**
- **PGI download web page**
- **PGI prepared/personalized training**
- **PGI ISV program**
- **PGI Premier Service program**

# PGI Basic Compiler Options

- **Basic Usage**
- **Language Dialects**
- **Target Architectures**
- **Debugging aids**
- **Optimization switches**

# PGI Basic Compiler Usage

- **A compiler driver interprets options and invokes pre-processors, compilers, assembler, linker, etc.**
- **Options precedence: if options conflict, last option on command line takes precedence**
- **Use -Minfo to see a listing of optimizations and transformations performed by the compiler**
- **Use -help to list all options or see details on how to use a given option, e.g. pgf90 -Mvect -help**
- **Use man pages for more details on options, e.g. "man pgf90"**
- **Use –v to see under the hood**

# Flags to support language dialects

- **Fortran**
  - **pgf77, pgf90, pgf95, pghpf tools**
  - **Suffixes .f, .F, .for, .fpp, .f90, .F90, .f95, .F95, .hpf, .HPF**
  - **-Mextend, -Mfixed, -Mfreeform**
  - **Type size –i2, -i4, -i8, -r4, -r8, etc.**
  - **-Mcray, -Mbyteswapio, -Mupcase, -Mnomain, -Mrecursive, etc.**
- **C/C++**
  - **pgcc, pgCC, aka pgcpp**
  - **Suffixes .c, .C, .cc, .cpp, .i**
  - **-B, -c89, -c9x, -Xa, -Xc, -Xs, -Xt**
  - **-Msignextend, -Mfcon, -Msingle, -Muchar, -Mgccbugs**

## Specifying the target architecture

- **Not an issue on XT.**
- **Defaults to the type of processor/OS you are running on**
- **Use the “tp” switch.**
  - **-tp k8-64 or –tp p7-64 or –tp core2-64 for 64-bit code.**
  - **-tp amd64e for AMD opteron rev E or later**
  - **-tp x64 for unified binary**
  - **-tp k8-32, k7, p7, piv, piii, p6, p5, px for 32 bit code**

## Flags for debugging aids

- **-g generates symbolic debug information used by a debugger**
- **-gopt generates debug information in the presence of optimization**
- **-Mbounds adds array bounds checking**
- **-v gives verbose output, useful for debugging system or build problems**
- **-Mlist will generate a listing**
- **-Minfo provides feedback on optimizations made by the compiler**
- **-S or –Mkeepasm to see the exact assembly generated**

## Basic optimization switches

- **Traditional optimization controlled through -O[<n>], n is 0 to 4.**
- **-fast switch combines common set into one simple switch, is equal to -O2 -Munroll=c:1 -Mnoframe -Mlre**
  - **For -Munroll, c specifies completely unroll loops with this loop count or less**
  - **-Munroll=n:<m> says unroll other loops m times**
- **-Mnoframe does not set up a stack frame**
- **-Mlre is loop-carried redundancy elimination**

# Basic optimization switches, cont.

- **fastsse switch is commonly used, extends –fast to SSE hardware, and vectorization**
- **-fastsse is equal to -O2 -Munroll=c:1 -Mnoframe -Mlre (-fast) plus -Mvect=sse, -Mscalarsse -Mcache_align, -Mflushz**
- **-Mcache_align aligns top level arrays and objects on cache-line boundaries**
- **-Mflushz flushes SSE denormal numbers to zero**

# Node level tuning

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

# 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

```
% ftn -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
   …
```

## 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, pragmas, or restrict type qualifer
- **Function Calls** – Try inlining with –Minline or –Mipa=inline
- **Type conversions** – manually convert constants or use flags
- **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**
- **Non unity strides**
- **Code bloat if altcode is generated**

- **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

# 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

# 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!***

## Other C++ recommendations

- **Encapsulation, Data Hiding  -** small functions, inline!
- **Exception Handling –** use –no_exceptions until 7.0
- **Overloaded operators, overloaded functions -** okay
- **Pointer Chasing -** -Msafeptr, restrict qualifer, 32 bits?
- **Templates, Generic Programming –** now okay
- **Inheritance, polymorphism, virtual functions –** runtime lookup or check, no inlining, potential performance penalties

- **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 XT CNL and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try

# SMP Parallelization

- **–mp=nonuma to enable OpenMP 2.5 parallel programming model**
  - **See PGI User's Guide or OpenMP 2.5 standard**
  - **OpenMP programs compiled w/out –mp=nonuma "just work"**
  - **Supported on Cray XT**

- **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 XT CNL and multi-core processors
- **Miscellaneous Optimizations** – hit or miss, but worth a try

## Miscellaneous Optimizations (1)

- **–Mfprelaxed – single-precision sqrt, rsqrt, div performed using reduced-precision reciprocal approximation**
- **–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**
- **–Mvect=noaltcode – disable multiple versions of loops**

# Pathscale programming environment

- Pathscale module available
  - **module load pathscale**
- Use standard compiler driver: ftn
  - **ftn -O3 -OPT:Ofast …**

# Pathscale compilers

- **Pathscale compiler flags for a first start**
  - **Preprocessor Options:**
    - -cpp runs cpp on source files
    - -ftpp runs the fortran source preprocessor
  - **Optimisation Options:**
    - -LNO: specify transformations performed on loop nests by the Loop Nest Optimizer
    - -OPT: controls miscellaneous optimizations
    - -ipa Inter Procedural Analysis
    - -Ofast Equivalent to
    - -O3 -ipa -OPT:Ofast -fno-math-errno -ffast-math
  - Default: -O2 -mcpu=opteron -m64 -msse -msse2 -mno-sse3 -mno-3dnow
  - Start: -O3 –OPT:Ofast
- More info: man eko, man pathf95