PGI® Compilers and Tools on the ORNL Titan System

16 APR 12
OLCF Spring Training and Users' Meeting

Dave Norton
dave.norton@pgroup.com
530.544.9075
www.pgroup.com
PGI ACC Workstation / Server / CDK
Linux, Windows, MacOS, 32-bit, 64-bit, AMD64, Intel 64, Nvidia
UNIX-heritage Command-level Compilers + Graphical Tools

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Language</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PGFORTRAN™</strong></td>
<td>Fortran 77, Fortran 95, Fortran 2003, CUDA Fortran</td>
<td>pgfortran  pgf90  pgf77</td>
</tr>
<tr>
<td><strong>PGCC®</strong></td>
<td>ANSI C99, K&amp;R C and GNU gcc Extensions</td>
<td>pgcc</td>
</tr>
<tr>
<td><strong>PGC++®</strong></td>
<td>ANSI/ISO C++ and GNU compatible C++</td>
<td>pgCC</td>
</tr>
<tr>
<td><strong>PGDBG®</strong></td>
<td>MPI/OpenMP debugger</td>
<td>pgdbg</td>
</tr>
<tr>
<td><strong>PGPROF®</strong></td>
<td>MPI/OpenMP/ACC profiler</td>
<td>pgprof</td>
</tr>
</tbody>
</table>

Self-contained OpenMP/MPI/Accelerator
Parallel SW Development Solution

The Portland Group
<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1303</td>
<td>FT = c0</td>
</tr>
<tr>
<td>1304</td>
<td>bi &amp; t = this_blockStocal_id</td>
</tr>
<tr>
<td>1305</td>
<td>horizontal diffusion HDiff(T)</td>
</tr>
<tr>
<td>1308</td>
<td></td>
</tr>
<tr>
<td>1310</td>
<td></td>
</tr>
<tr>
<td>1311</td>
<td></td>
</tr>
<tr>
<td>1312</td>
<td></td>
</tr>
<tr>
<td>1313</td>
<td></td>
</tr>
<tr>
<td>1314</td>
<td></td>
</tr>
<tr>
<td>1315</td>
<td></td>
</tr>
<tr>
<td>1316</td>
<td></td>
</tr>
<tr>
<td>1317</td>
<td>if (ltd thrown then)</td>
</tr>
<tr>
<td>1318</td>
<td>if (partial_bottom_cells) then</td>
</tr>
<tr>
<td>1319</td>
<td>00 n,1,nT</td>
</tr>
<tr>
<td>1320</td>
<td>where (k &lt;= RKT(:,;): bid)) &amp;</td>
</tr>
<tr>
<td>1321</td>
<td></td>
</tr>
<tr>
<td>1322</td>
<td></td>
</tr>
</tbody>
</table>

**Line-level information for line 1320**

1. Intensity = 0.40

2. Loop not vectorized: multiple blocks
   - Vectorization Hint: Try splitting the loops or converting conditional blocks into a simpler form

**Information about routines referenced in routine baroclinic_tracer_update_in_file compile/baroclinic.f90**

**Information about how file compile/baroclinic.f90 was compiled**
do j = 1, ny_global
   ULAT_G(i,j) = (-90.0_c8 + j * dlat)/radian
   enddo

! calculate grid spacings and other quantities
! compute here to avoid bad ghost cell values due to dropped land
! blocks
!
else ! not lacion_only

!OMP PARALLEL DO PRIVATE(this_block, i, j, ig, jg, lathalf)
do n=1,nblocks_clinic
   this_block = get_block(blocks_clinic[n], n)
do j=1, ny_block
      jg = this_block%j_gich(j)
      gm1 = jg - 1
      if (jg < 1) gm1 = ny_global
   do i=1, nx_block
      !**** calculate grid lengths
   enddo
   HTW(1, i, n) = dlon/radius/radian ! convert to cm
   HTB(1, i, n) = dlat/radius/radian ! convert to cm
PGI® Compilers & Tools Positioning

- PGI compilers & tools are dedicated to scientific computing, where utilization of latest architecture features and speed on generated code is #1 criteria
  - Not intended to replace infrastructure compilers (GCC/VC++)

- HPC-focused compilers & tools technologies
  - State of the art local, global and inter-procedural optimizations
  - Automatic vectorization and SIMD/SSE code generation
  - Support of OpenMP 3.0 standard
  - Automatic loop parallelization
  - Profile-guided optimization
  - PGI Unified Binary technology to target different ‘flavors’ of same architecture or heterogeneous architectures
  - Graphical tools to debug/profile multithreaded/multiprocess hybrid applications
Compiling codes with PGI

To load the PGI compiler on the ORNL systems

> Module load PrgEng-pgi/4.0.30

Cray supplies wrappers to all of the compilers on the system so that the Fortran compiler is always invoked as “ftn”, the C compiler as “cc”, and C++ as “CC” regardless of the actual compile vendor being used.

> module list

    pgi/12.2.0

> ftn -V foo.f -o foo

    pfgfortran 12.2-0 64-bit target on Linux -tp bulldozer-64

Or you can call the compiler directly with “pfgfortran” but you won’t get the Cray library wrappers for use in the ONRL system.
Using a different version of PGI

On the Cray, to change the version of the PGI compiler, you need to switch modules:

```bash
> module switch pgi/10.2.0 pgi/12.2.0
> ftn -V foo.f -o foo

pgf90 12.2-0 64-bit target on Linux -tp bulldozer
```

On your workstation, if you have multiple versions of PGI installed, you can invoke a different version of the compiler through the compile driver:

```bash
> pgfortran -V9.0-4 hello.f -o hello

pgfortran 9.0-4 64-bit target on Linux -tp bulldozer
```
Changing target processors

The PGI compile driver by default compiles for the processor on which the compilation takes place. The driver allows you to easily cross compile for another target processor:

```bash
> pgfortran -V foo.f -o foo -tp istanbul-64

pgfortran 10.4-0 64-bit target on Linux -tp istanbul-64

The Cray compile driver now also allows this capability.

> pgfortran -V -c foo.f -tp istanbul-64

> ftn -V foo.o -o foo -tp istanbul-64
```
Basic levels of scalar optimization

> ftn foo.f -o foo

Invoking the compiler with no flags for optimization will set the scalar optimization level to 1 if –g is not specified.

> ftn -g foo.f -o foo

Invoking the compiler with no flags for optimization will set the scalar optimization level to 0 if –g is specified.

> ftn -O foo.o -o foo

Invoking the compiler with the -O flag for optimization will set the scalar optimization level to 2 regardless of whether –g is also specified.

Optimization levels O0 through O4 perform increasing aggressive scalar optimizations.

The Portland Group
Basic levels of vector optimization

> ftn -fast foo.f -o foo

Invoking the compiler with the –fast (or –fastsse) flag sets common optimizations which include:

- `-O2`
- `-Munroll=c:1`
- `-Mnoframe` (gives the compiler another register)
- `-Mlre`
- `-Mautoinline`
- `-Mvect=sse` <= this is the vectorizer
- `-Mscalarsse`
- `-Mcache_align`
- `-Mflushz`
- `-Mpre`
Vectorization is the key to getting the best performance out of floating point intense codes. Current processors are capable of operating on 128 bits at a time. This means they can do 2 – double precision operations or 4 – single precision operations at the same time – as long as those operations can all be described by a single instruction (i.e. a vector operation).

AVX – used on Bulldozer and Sandybridge - increases this to 256 bit wide units

The vectorizer performs the following operations:

- Loop interchange and loop splitting
- Loop fusion
- Memory-hierarchy (cache tiling) optimizations
- Generation of SSE instructions and prefetch instructions
- Loop peeling to maximize vector alignment
- Alternate code generation
What is AVX?

Before VEX:

movsd (%rax, %r9), %xmm0
movsd (%rax, %r8), %xmm1
movsd %xmm1, %xmm2
addsd %xmm0, %xmm2

After VEX:

vmovsd (%rax, %r9), %xmm0
vmovsd (%rax, %r8), %xmm1
vaddsd %xmm0, %xmm1, %xmm2

---

<table>
<thead>
<tr>
<th>255 ... 128</th>
<th>127 ... 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>xmm0</td>
</tr>
<tr>
<td></td>
<td>ymm0</td>
</tr>
<tr>
<td></td>
<td>xmm1</td>
</tr>
<tr>
<td></td>
<td>ymm1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>xmm14</td>
</tr>
<tr>
<td></td>
<td>ymm14</td>
</tr>
<tr>
<td></td>
<td>xmm15</td>
</tr>
<tr>
<td></td>
<td>ymm15</td>
</tr>
</tbody>
</table>
## Importance of Vectorization

<table>
<thead>
<tr>
<th></th>
<th>255...192</th>
<th>191...128</th>
<th>127...64</th>
<th>63...0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a*x[3]</th>
<th>a*x[2]</th>
<th>a*x[1]</th>
<th>a*x[0]</th>
</tr>
</thead>
</table>

|-------|------------|------------|------------|------------|
Know Your Target Processors

AMD Bulldozer  PGI target processor flag : –tp bulldozer

Specify size of SIMD instructions : -Mvect=simd:[128|256]

Enable/Disable generation of FMA instructions:  -[no]fma

Running FMA4 code on anything but Bulldozer will yield:
   Illegal instruction (core dumped)

Make use of PGI Unified Binary technology to produce optimal code
   paths for multiple x64 architectures within a single executable.
vzeroupper instruction generation

This instruction zeroes out the upper 128 bits of all 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 the transition points.

The PGI compiler currently generates the vzeroupper instruction right before a call is made. This is because we cannot be sure how the callee has been compiled.

When compiling functions that perform AVX instruction sequences, the PGI compiler generates a vzeroupper instruction right before returning, again because we cannot make assumptions about how the caller was compiled.
300 CONTINUE
PERIODIC CONTI

DO 320 J = 1, M
U0L0D(M+1,J) =
V0L0D(M+1,J) =
U(M+1,J) = U(J)
V(M+1,J) = V(J)
P(M+1,J) = P(J)

320 CONTINUE

RETURN
END

bash-4.1$ export OMP_NUM_THREADS=1
bash-4.1$ le
Files
print
print
output
output
output
output
output
output
output
output
output
output

makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile
makefile

The Portland Group
Common impediments to vector optimization

There are several common coding issues that may prevent vectorization. The programmer may have enough knowledge to provide additional information to the compiler to work around these issues.

In C and C++ the compiler may not have enough information about the pointers passed into a subroutine to be able to determine that those pointers don’t overlap. (-Msafeptr option or pragma or `restrict` keyword)

Function calls can be inlined to allow vectorization (-Minline)

Constants may be of the wrong type (-Mfcon)

Loops may be too long or too short. In both cases, additional options to the vectorizer may be successful in generating vector code.
### -Msafeptr Option and Pragma

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>All pointers are safe</td>
</tr>
<tr>
<td>arg</td>
<td>Argument pointers are safe</td>
</tr>
<tr>
<td>local</td>
<td>Local pointers are safe</td>
</tr>
<tr>
<td>static</td>
<td>Static local pointers are safe</td>
</tr>
<tr>
<td>global</td>
<td>Global pointers are safe</td>
</tr>
</tbody>
</table>

**Pragma**

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

Where *scope* is *global*, *routine* or *loop*
Which level of optimization to start?

If you are just starting with a new code, we suggest that you try a short run of the code with optimization level –O2.

If the answers look good, then try the same run with the –fast flag.

If the answers are the same as the first run, use –fast as the basis for further optimizations. If the answers differ, try turning off optimizations one at a time until you find the optimization that is causing the difference. You can then track down in your code where that difference occurs and determine if it can be fixed, or if the optimization needs to be left turned off.
Turning off optimizations

Optimization flags are processed on the command line in the order in which they occur. For example - to turn on all –fast optimizations except loop redundant elimination:

```plaintext
> ftn -fast -Mnolre foo.o -o foo
```

Most optimizations can be turned on with the syntax –Moptimization

Most optimizations can be turned off with the syntax -Mnooptimization
Optimizations and debugging

Optimizations and debugging don’t always go hand in hand, however...

```bash
> ftn -fast -gopt foo.f -o foo
```

-gopt inserts debugging information without disabling optimizations. It is often helpful for tracking down a code bug that only appears in optimized code, or a bug that occurs far enough into a code that running the code with no optimizations takes a painful amount of time.
Generating tracebacks

Linux uses the backtrace system call to create the stacktrace when a fault or error occurs. The only requirement is to link with the -Meh_frame option:

```
> pgfortran -Meh_frame -o x x.f90
```

Then before running the program, the following environment variable is set as follows:

```
> export PGI_TERM=trace
```
Generating tracebacks

Here is a sample traceback from within the PGI runtime.
(An attempt to deallocate an allocatable array more than one time):

```
0: DEALLOCATE: memory at (nil) not allocated
   ./x(__hpf_abort+0x7d) [0x40bb8d]
   ./x(__hpf_dealloc+0xeb) [0x40b57b]
   ./x(MAIN_+0x217) [0x408177]
   ./x(main+0x40) [0x407f40]
/lib64/libc.so.6(__libc_start_main+0xf4) [0x2b877285e154]
   ./x [0x407e69]
```
Here is a sample traceback from a SEGV in user code:

Error: segmentation violation, address not mapped to object
   rax 00000000005f45908, rbx 0000000000000001, rcx 00000000000187f9
   rdx 00000000000187f9, rsp 00007fffffffdaef9a0, rbp 00007fffffffdaef9a0
   rsi 00007fffffffdaef9c4, rdi 00002ab2dd77e020, r8 00000000fffffff
   r9 0000000000000000, r10 0000000000000022, r11 0000000000000246
   r12 0000000000000001, r13 00007fffffffdaefae0, r14 0000000000000000
   r15 0000000000000000

/lib64/libpthread.so.0 [0x2ab2dd1ebc10]
./y(init_+0x1f) [0x4081bf]
./y(MAIN_+0x9b) [0x407ffb]
./y(main+0x40) [0x407f40]
/lib64/libc.so.6(__libc_start_main+0xf4) [0x2ab2dd468154]
./y [0x407e69]
What does this flag do?

There are too many compiler flags to remember all of their options. You can get help in several places:

> man pgfortran

> pgfortran -fast -help -- gives help on -fast

Full PDF manuals are online in (e.g)

/opt/pgi/12.2.0/linux86-64/2012/doc

Manuals are also available at:

http://www.pgroup.com/resources/docs.htm
Optimization is as much a user exercise as it is a compiler exercise. To see what the compiler thinks of your code, compile using the –Minfo flag.

```bash
> pgfortran -fast -Minfo foo.f -o foo
```

Use the information generated by –Minfo to help identify coding issues and locate places where code can be improved so the compiler can do an optimal job on it.

```bash
> pgfortran -Minfo -help
```
Use –Minfo to see which loops vectorize

\[
\begin{align*}
\text{localmove:} & \\
334, & \text{Loop unrolled 1 times (completely unrolled)} \\
343, & \text{Loop unrolled 2 times (completely unrolled)} \\
358, & \text{Generating vector sse code for inner loop} \\
364, & \text{Generating vector sse code for inner loop} \\
& \quad \text{Generating vector sse code for inner loop} \\
392, & \text{Generating vector sse code for inner loop} \\
423, & \text{Generating vector sse code for inner loop}
\end{align*}
\]

Use –Mneginfo to see why things don’t vectorize
Additional compiler optimizations

The –fast flag is the 90/90 solution for code optimization. That is, it achieves about 90% of the possible performance for about 90% of the codes.

That means there are some additional areas that can be explored.

Interprocedural analysis can be helpful for C codes and Fortran codes without interface blocks. (Interface blocks are to the language specification what IPA is to the compiler)

> ftn -fast -Minfo -Mipa=fast foo.f -o foo

***If compiling and linking are done in separate steps, you must be sure to pass the IPA flag to the linker too.

IPA involves an additional pass of the compiler.

The Portland Group
The suggested usage for IPA is to apply –Mipa=fast globally

The –Mipa flag has a large number of options that may be helpful in certain circumstances. These options are generally best applied to a specific subroutine to address a specific issue.

A couple of the more interesting flags include:

-Mipa=libopt  This allows recompiling and optimization of routines from libraries using IPA information. If you make extensive use of libraries in your code, try compiling those libraries with –Mipa=fast so that you have the option of using IPA when you link your application to that library

-Mipa=safeall  This declares that all unknown procedures are safe.
Several memory management options are available and may be beneficial depending on how your code accesses memory. *Smartalloc* tends to do a better job managing memory then standard Unix malloc.

Smartalloc can make use of “big pages”. Using big pages helps to minimize the number to TLB misses. This option tends to be helpful for codes that do a big initial allocate and then manage their own memory.

```plaintext
> ftn -fast -Minfo -Mipa=fast -Msmartalloc= HUGE foo.f -o foo
```

***-Msmartalloc must be used to compile main, and also to link the program***

Inlining can have a significant impact on application performance. It’s most dramatic effects tend to be on C++ codes which have many many small functions.

Inlining can be done at several different points in the compilation.

-Minline/autoinline - during the regular compilation phase

-Mipa=inline - during the recompile for IPA

Inline libraries - created during the “make” process
The auto inliner is for C/C++ only. This enables inlining functions with the inline attribute. The suboptions control how the auto inliner operates.

-M[no]autoinline
   Enable inlining of functions with the inline attribute.
   -Mautoinline is implied with the -fast switch. The options are:

   levels:n Inline up to n levels of function calls; the default is to inline up to 10 levels.

   maxsize:n Only inline functions with a size of n or less. The size roughly corresponds to the number of statements in the function, though the correspondence is not direct. The default is to inline functions with a size of 100 or less.

   totalsize:n
      Stop inlining when this function reaches a size of n. The default is to stop inlining when a size of 8000 has been reached.
Creating and Using Inline Libraries

Use of -Minline/-Mextract to create an inline library. This works for all languages (C/C++/FORTRAN). To create an inline library with -Mextract do the following:

```
pgfortran -Mextract=lib:libfloat.il -c add.f90
pgfortran -Mextract=lib:libfloat.il -c sub.f90
pgfortran -Mextract=lib:libfloat.il -c mul.f90
pgfortran -Mextract=lib:libfloat.il -c div.f90
```

This creates an inline library name libfloat.il which can be used during compilation as follows:

```
pgf90 -fast -Minline=libfloat.il -c -Minfo -Mneginfo driver.f90
```
The -Minfo messages for this compile are:

test:
  14, Generated an alternate loop for the loop
      Generated vector sse code for the loop
  21, Generated an alternate loop for the loop
      Generated vector sse code for the loop
  22, add inlined, size=2, file add.f90 (2)
  33, Generated an alternate loop for the loop
      Generated vector sse code for the loop
  34, sub inlined, size=2, file sub.f90 (2)
  45, Generated an alternate loop for the loop
      Generated vector sse code for the loop
  46, mul inlined, size=2, file mul.f90 (2)
  57, Generated an alternate loop for the loop
      Generated vector sse code for the loop
  58, div inlined, size=2, file div.f90 (2)

As a result of inlining the functions add, sub, mul, and div the compiler was then able to vectorize the loops that contained those calls.
Use of -Mipa=inline to inline functions/subroutines. This works for all languages (C/C++/FORTRAN). Create the library using the -Mipa=inline option as follows:

```
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c add.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c sub.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c mul.f90
pgfortran -Mipa=fast,inline -Minfo -Mneginfo -c div.f90
```

```
ar cr libfloat.a add.o sub.o mul.o div.o
```

This creates a library named libfloat.a which can be used during compilation as follows (need to use the libinline suboption):

```
pgf90 -fast -Mipa=fast,inline,libinline -c -Minfo -Mneginfo driver.f90
pgf90 -fast -Mipa=fast,inline,libinline -o d driver.o libfloat.a
```
The -MINFO messages for this compile are:

test:
  14, Generated an alternate loop for the loop
      Generated vector SSE code for the loop
  21, Loop not vectorized/parallelized: contains call
  33, Loop not vectorized/parallelized: contains call
  45, Loop not vectorized/parallelized: contains call
  57, Loop not vectorized/parallelized: contains call

IPA: Recompiling driver.o: stale object file

test:
  0, Pointer c is only set via allocate statements
     Pointer b is only set via allocate statements
     Pointer a is only set via allocate statements
     Function add does not write to any of its arguments
     Function add does not reallocate any of its arguments
     Function add does not reassociate any of its pointer arguments
     Function add does not reallocate any global variables
     Function add does not reassociate any global pointers
     Function add does not read any global (common/module) variables
     Function sub does not write any global (common/module) variables
     Function sub does not write to any of its arguments
     Function sub does not reallocate any of its arguments
     Function sub does not reassociate any of its pointer arguments
     Function sub does not reallocate any global variables
     Function sub does not reassociate any global pointers
     Function sub does not read any global (common/module) variables
There are a number of compiler options that offer the possibility of significant performance improvement at the expense of accuracy. If you are having numerical issues, you might tighten some restrictions.

- **-Kieee** – floating point strictly conforms to IEEE 754 standard. (off by default)

- **-Ktrap** – turns on the behavior of the processor when exceptions occur

- **-Mdaz** – mode to treat IEEE denormalized input numbers as zero

- **-Mflushz** – set SSE to flush-to-zero mode (on with –fast)

- **-Mfprelaxed** - perform certain floating point operations using relaxed precision when it improves the speed. (This is the default mode on most other vendor’s compilers)
Using more then one core

There are three general techniques for using more then one core for a computation. Of course, on large XT6 machines, all codes implement parallelism through MPI.

While most codes are MPI everywhere, some codes benefit by using the shared memory on the node through either automagic parallelizing by the compiler or/and OpenMP. OpenMP compilation is invoked with the –mp flag, automagic parallelization with the –Mconcur flag.

Environment variables which can effect OpenMP performance include:

OMP_SCHEDULE – can be static, dynamic, guided or auto

OMP_NUM_THREADS – specifies the number of threads to use

OMP_STACKSIZE – override the default stack size for new threads.
We developed Gnu compatible pgc++ (PGI’s current C++ compiler is call “pgcpp” or “pgCC”) to become link compatible with the growing number of Gnu compiled libraries available, including the Gnu STL and Boost.

The link compatibility changes include:

• mangled names
• run time type information
• virtual function tables
• subobject creation
• exception handling
• complex types
• gnu header file support
There are currently some limitations to our link compatibility with GCC code:

* Our long double size is not compatible. Users who call functions with long double parameters should compile all of that code with the same compiler.

* pgc++ supports pthreads except for code compiled -mp (OpenMP).

* Some GNU builtins introduced in 4.4.0 are not yet supported.

* We use our math routines. We recommend users link with pgc++.

* (C99) tgmath.h is not supported.

* pgc++ objects are not compatible with pgcpp objects/libraries.

pgc++ supports gcc versions 4.1 through 4.5. The compiler installation process determines which g++ version is installed on your machine, and configures your localrc file accordingly. **If you upgrade your gcc compiler, you will have to reinstall the PGI compilers.**

We are currently at EDG release 4.1, and plan to have full C++11 support by mid 2013.
PGI OpenCL Framework for Multi-core ARM
PGI OpenCL compiler features

- All OpenCL 1.1 embedded profile language features implemented
  - new vector/scalar data types
  - vector literals & components
  - Neon/SIMD code generation for operations on vector data types
  - function / addr space qualifiers
  - built-ins

- ATI and NVIDIA OpenCL SDK examples
  - 21 NVIDIA SDK examples and 26 ATI SDK examples compile and run correctly with PGCL on Android/ARM

- 24 of 24 OpenCL 1.1 Conformance Test Suites passing on Android/ARM; working on a formal compliance submission

- Supports both static compilation of OpenCL kernels, and native dynamic compilation of kernels on Android/ARM devices

- DWARF generation for OpenCL kernels and basic debugging using gdb/gdbserver
PGI OpenCL compiler driver

pgcl compiler driver

- cmd-line interface to compile both OpenCL host code and statically compiled OpenCL kernels
- Minimizes changes to makefiles
- Pre-configured to use GCC as OpenCL host compiler & PGI OpenCL compiler for Multi-core ARM as a compute device
- From the command line you can specify host compiler to use, pass compile/link options, no need to specify location for OpenCL lib/include
- From command line you can specify options to be passed to the OpenCL language compiler

```
% pgcl --opencl-flags -O2 -o kernel.so -- kernel.cl
% pgcl -hostcomp=g++ -O1 -c hostcode.cpp
% pgcl -O2 -c hostcode.c
```
OpenCL compiler development approach

PGI OpenCL Front End & Optimizing Core

LLVM IR assembly file .ll

- Cortex-A9
- Accelerators
- x86

- Pre-production
- For validation purposes only
- Potential development

The Portland Group
PGI Optimizing Compilers infrastructure

- **C/C99**
- **C++**
- **OpenCL**
- **F95/03**

**Global Optimization**
- InterProcedural Optz
- Auto-Parallel
- OpenMP Parallel Function Inlining

**Dependence Analysis**
- SIMD Vectorization
- Loop Tiling/Unrolling
- Loop Interchg/

**OpenCL Intrinsics**
- DSP Intrinsics
- SSE/AVX Intrinsics
- Heterogeneous Targets
- PGI Unified Binary

**Profile Feedback**
- SW Prefetching
- Alignment Optz
- CCFF

**Local Regs**
- Global Regs
- SIMD Vector
- Peephole
- 32-bit only

**x86**
- Local Regs
- Global Regs
- SIMD Vector
- Peephole
- 32-bit only

**x64**
- Local Regs
- Global Regs
- SIMD Vector
- Peephole
- 32/64-bit

**U**
- Device
- Mgmt
- Kernel Gen
- Data Ld/St
- Local Alloc
- AutoPar
- AutoSIMD

**ST100**
- Local Regs
- VLIW
- SW Pipe
- AutoSIMD
- Predication
- 16- & 32-bit

**ARM**
- Local Regs
- Global Regs
- Scheduling
- Peephole
- AutoSIMD
- Code Sel

**Others**
- Any other LLVM target

**x64+GP**
Cray provides some excellent tools for profiling using hardware counters.

PGI also provides some mechanisms for profiling of code. The simplest method is to use pgcollect. No special build process is needed, although compiling with –Minfo=ccff may provide useful feedback. This imbeds the –Minfo messages into the executable which can then be viewed with the performance profile.

Run your code as:

> pgcollect a.out

Then view the results with the GUI tool - pgprof

> pgprof -exe a.out
To get a general profile for an MPI code, you may wish to just profile one of the MPI processes. Running the code is where things change. Instead of launching the executable via mpiexec, launch a script instead:

```bash
> mpiexec -np 2 ./doit
```

The `doit` script for code compiled and linked with MPICH2 might look like the following:

```csh
#!/bin/csh

if ($PMI_RANK == 0) then
    pgcollect ./test
else
    ./test
endif
```

After the run is complete, there will be only one `pgprof.out` file which can be viewed using:

```bash
> pgprof -exe ./test pgprof.out
```
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 parallel programming model

OpenMP programs compiled w/out `–mp` “just work”

- `Mconcur` and `–mp` can be used together!
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
Extending Host-side x64 Compilers to Enable Incremental use of GPGPUs
NVIDIA TESLA C1060
- Lots of available performance ~1 TFlops peak SP
- Programming is a challenge
- Getting high performance is lots of work

NVIDIA CUDA programming model and C for CUDA simplify GPGPU programming
- Much easier than OpenGL/DirectX, still challenging
- PGI CUDA Fortran simplifies it even further

PGI Accelerator compilers do for GPU programming what OpenMP did for Posix Threads
Emerging Cluster Node Architecture
Commodity Multicore x86 + Commodity Manycore GPUs

CPU Cores
GPU/Accelerator Cores
NVIDIA Streaming Multiprocessor features

- **One Control Unit per SM**
  - SM operates in SIMT fashion by « Warps » of 32 threads
  - Up to 32 Warps in flight
  - Computation structured in 1D, 2D or 3D blocks of threads
  - Blocks are organized in a 1D or 2D Grid

- **16384 32-bits registers per SM**
  - No cost to tolerate heavily multi-threaded computations
  - Hide long access time to device memory

- **SM can execute up to 8 blocks**
  - Block execution can’t migrate from one SM to another

- **16 KB of shared memory**
  - Must be managed as a software cache by the programmer
Today’s architectures

<table>
<thead>
<tr>
<th>Chip</th>
<th>Cores</th>
<th>Vector length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandybridge</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>Interlagos</td>
<td>8/16</td>
<td>2/4</td>
</tr>
<tr>
<td>MIC</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>Fermi</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

(2 cycle instruction latency makes effective vector length 32)
CUDA Fortran

Implicit (through the language syntax) rather then explicit (through an API)

As with CUDA-C, requires both host code and GPU code

(Almost) all Fortran 2003 language features are Accessible through CUDA Fortran

Understanding of CUDA-C kernel launch mechanism and underlying hardware architecture helpful*

*"Supercomputer is computing which requires the programmer to have an intimate understanding of the underlying hardware architecture”
- Paul Anderson
subroutine vadd( A, B, C )
    use cudafor
    use kmod
    real, dimension(:) :: A, B
    real, pinned, dimension(:) :: C
    real, device, allocatable:: Ad(:,), Bd(:,), Cd(:,)
    integer :: N
    N = size( A, 1 )
    allocate( Ad(N), Bd(N), Cd(N) )
    Ad = A(1:N)
    Bd = B(1:N)
    call vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
    C = Cd
    deallocate( Ad, Bd, Cd )
end subroutine
CUDA Fortran VADD Device Code

module kmod
    use cudafor
contains
    attributes(global) subroutine vaddkernel(A,B,C,N)
    real, device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadIdx%x
    if( i <= N ) C(i) = A(i) + B(i)
end subroutine
end module
Building a CUDA Fortran Program

CUDA Fortran is supported by the PGI Fortran compilers when the filename uses a CUDA Fortran extension. The .cuf extension specifies that the file is a free-format CUDA Fortran program;

The .CUF extension may also be used, in which case the program is processed by the preprocessor before being compiled.

To compile a fixed-format program, add the command line option –Mfixed.

CUDA Fortran extensions can be enabled in any Fortran source file by adding the –Mcuda command line option.

Most F2003 features should work in CUDA Fortran.

There is a (CUDA-like) API to access features

– Streams supported through API rather then language
Accelerator Directives for flat performance profile codes

Source code directives provide a second method for programming GPU’s.

The goal is to move data to the GPU and compute on the GPU until off-node communications is required.

Directives allow for flexibility “under the hood”

Allow for single program source – similar to OMP
module kmod
  contains
  subroutine vaddkernel(A,B,C)
    real :: A(:,,:), B(:,,:), C(:,:)
    !$acc region
    C(:, :) = A(:, :) + B(:, :)
    <lots of other code to do neat stuff>
    <special code to do even neater stuff>
    !$acc end region
  end subroutine
end module

 !$acc region clauses can surround many individual loops and compute kernels. There is no implicit GPU/CPU data movement within a region
Compiling the subroutine:

PGI$ pgfortran -Minfo=accel -ta=nvidia -c vadd.F90

vaddkernel:
  5, Generating copyout(c(1:z_b_14,1:z_b_17))
     Generating copyin(a(1:z_b_14,1:z_b_17))
     Generating copyin(b(1:z_b_14,1:z_b_17))
     Generating compute capability 1.0 binary
     Generating compute capability 1.3 binary
     Generating compute capability 2.0 binary
  6, Loop is parallelizable
     Accelerator kernel generated
     6, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
       !$acc do parallel, vector(16) ! blockidx%y threadidx%y
     CC 1.0 : 7 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy
     CC 1.3 : 8 registers; 64 shared, 8 constant, 0 local memory bytes; 100% occupancy
     CC 2.0 : 15 registers; 8 shared, 72 constant, 0 local memory bytes; 100% occupancy
Tuning the compute kernel

Accelerator VADD Device Code

module kmod
contains
subroutine vaddkernel(A,B,C) ! We know array size
  real :: A(:,,:), B(:,,:), C(:,:)! dimension(2560,96)
  integer :: i,j
!$acc region
!$acc do parallel
  do j = 1,size(A,1)
!$acc do vector(96)
    do i = 1,size(A,2)
      C(j,i) = A(j,i) + B (j,i)
    enddo
  enddo
enddo
!$acc end region
end subroutine
end module
Keeping the data on the GPU

Accelerator VADD Device Code

module kmod
  contains
  subroutine vaddkernel(A,B,C)
    real :: A(:,:), B(:,:), C(:,:)
    !$acc reflected (A,B,C)
    !$acc region
      C(:,:) = A(:,:) + B (:,:)
    !$acc end region
  end subroutine
end module

The !$reflected clause must be visible to the caller so it knows to pass pointers to arrays on the GPU rather then copyin actual array data.
Compiling the subroutine:

PGI$  pgfortran -Minfo=accel -ta=nvidia -c vadd.F90
vaddkernel:
  5, Generating reflected(c(:, :))
    Generating reflected(b(:, :))
    Generating reflected(a(:, :))
  6, Generating compute capability 1.0 binary
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
  7, Loop is parallelizable
    Accelerator kernel generated
    7, !$acc do parallel, vector(16) ! blockIdx%ix threadIdx%ix
    !$acc do parallel, vector(16) ! blockIdx%iy threadIdx%iy
    CC 1.0 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 66% occupancy
    CC 1.3 : 11 registers; 80 shared, 8 constant, 0 local memory bytes; 100% occupancy
    CC 2.0 : 17 registers; 8 shared, 88 constant, 0 local memory bytes; 100% occupancy
subroutine vadd(M,N,C)
    use kmod ! Visibility of !$acc reflected
    real, dimension(:,:) :: A, B, C
    integer :: N
!$acc mirror(A,B)  !device resident clause in 1.3
    allocate(A(M,N),B(M,N))
! C has been mirrored and allocated previously
    A = 1.0
    B = 2.0
!$acc update device(A,B,C)
    call vaddkernel (A,B,C)
call kernel2 (A,B,C)
call kernel3 (A,B,C)
call kernel4 (A,B,C)
!$acc update host(C)
    deallocate( A, B)
end subroutine
Using GPU device-resident data across subroutines

```
subroutine timestep(Input, Result, M, N)
  use kmod    ! Make reflected var’s visible
  real, dimension(M, N) :: Input, Result
  integer :: M, N
  real, allocatable :: B, C, D
  dimension(:, :) :: B, C, D
  !$acc mirror(B, C, D)
  allocate(B(M, N), C(M, N), D(M, N))
  B = 2.0
  !$acc update device(Input, B)
  call vaddkernel(Input, B)
  ... call kernel2(C, D)
  ... call kernel3(D, Result)
  !$acc update host(Result)
  deallocate(B, C, D)
end subroutine

module kmod
  Contains
    !
    subroutine vaddkernel(A, B, C)
      real :: A(:, :), B(:, :), C(:, :)
      !$acc reflected (A, B, C)
      !$acc region
      C(:, :) = A(:, :) + B(:, :)
      !$acc end region
    end subroutine
    !
    subroutine kernel2(C, D)
      real :: C(:, :), D(:, :)
      !$acc reflected (C, D)
      !$acc region
      < compute-intensive loops >
      !$acc end region
    end subroutine
    ...
end module
```
% pgfortran -help -ta

-ta=nvidia:{analysis|nofma|[no]flushz|keepbin|keepptx|keepgpu|maxregcount:<n>|c10|cc11|cc12|cc13|cc20|fastmath|mul24|time|cuda2.3|cuda3.0|cuda3.1|cuda3.2|cuda4.0|[no]wait}|host

Choose target accelerator

nvidia Select NVIDIA accelerator target
analysis Analysis only, no code generation
nofma Don't generate fused mul-add instructions
[no]flushz Enable flush-to-zero mode on the GPU
keepbin Keep kernel .bin files
keepptx Keep kernel .ptx files
keepgpu Keep kernel source files
maxregcount:<n> Set maximum number of registers to use on the GPU
cc10 Compile for compute capability 1.0
...
cc20 Compile for compute capability 2.0
fastmath Use fast math library
mul24 Use 24-bit multiplication for subscripting
time Collect simple timing information
cuda2.3 Use CUDA 2.3 Toolkit compatibility
...
cuda4.0 Use CUDA 4.0 Toolkit compatibility
[no]wait Wait for each kernel to finish; overrides nowait clause
host Compile for the host, i.e. no accelerator target
Compute region directive clauses for tuning data allocation and movement

<table>
<thead>
<tr>
<th>Clause</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (condition)</td>
<td>Execute on GPU conditionally</td>
</tr>
<tr>
<td>copy (list)</td>
<td>Copy in and out of GPU memory</td>
</tr>
<tr>
<td>copyin (list)</td>
<td>Only copy in to GPU memory</td>
</tr>
<tr>
<td>copyout (list)</td>
<td>Only copy out of GPU memory</td>
</tr>
<tr>
<td>local (list)</td>
<td>Allocate locally on GPU</td>
</tr>
<tr>
<td>deviceptr (list)</td>
<td>C pointers in list are device pointers</td>
</tr>
<tr>
<td>update device (list)</td>
<td>Update device copies of the arrays</td>
</tr>
<tr>
<td>update host (list)</td>
<td>Update host copies of the arrays</td>
</tr>
</tbody>
</table>
Loop directive clauses for tuning GPU kernel schedules

<table>
<thead>
<tr>
<th>Clause</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel [(width)]</td>
<td>Parallelize the loop across the multi-processors</td>
</tr>
<tr>
<td>vector [(width)]</td>
<td>SIMD vectorize the loop within a multi-processor</td>
</tr>
<tr>
<td>seq [(width)]</td>
<td>Execute the loop sequentially on each thread processor</td>
</tr>
<tr>
<td>independent</td>
<td>Iterations of this loop are data independent of each other</td>
</tr>
<tr>
<td>unroll (factor)</td>
<td>Unroll the loop factor times</td>
</tr>
<tr>
<td>cache (list)</td>
<td>Try to place these variables in shared memory</td>
</tr>
<tr>
<td>private (list)</td>
<td>Allocate a copy of each variable in list for each loop iteration</td>
</tr>
</tbody>
</table>
Timing / Profiling

How long does my program take to run?
- `time ./myprogram`

How long do my kernels take to run?
- `pgfortran -ta=nvidia,time`

Environment variables:
- `export ACC_NOTIFY=1`
- `export NVDEBUG=1`
- `# cuda profiler settings`
- `#export CUDA_PROFILE=1`
- `#export CUDA_PROFILE_CONFIG=cudaprof.cfg`
- `#export CUDA_PROFILE_CSV=1`
- `#export CUDA_PROFILE_LOG=cudaprof.log`
Compiler-to-Programmer Feedback
Incremental porting/tuning for GPUs

Restructuring for accelerators will be more difficult than vectorization
Obstacles to GPU code generation

Loop nests to be offloaded to the GPU must be rectangular

At least some of the loops to be offloaded must be fully data parallel with no synchronization or dependences across iterations

Computed array indices should be avoided

All function calls must be inlined within loops to be offloaded

In Fortran, the pointer attribute is not supported; pointer arrays may be specified, but pointer association is not preserved in GPU device memory

In C

- Loops that operate on structs can be offloaded, but those that operate on nested structs cannot

- Pointers used to access arrays in loops to be offloaded must be declared with C99 restrict (or compiled with -Msafeptr, but it is file scope)

- Pointer arithmetic is not allowed within loops to be offloaded
The programming model

The directive based models apply to user-directed accelerator programming, where the user specifies the regions of a host program to be targeted for offloading to an accelerator device. The remained of the program will be executed on the host.

It’s important for the programmer to understand where the data is located that is being computed on. Depending on how data is allocated on the accelerator, there may be two copies of the same array – one located on the accelerator and one on the host.
The programming model

The accelerator is a slave processor, the host is the master processor. The host sends work to the accelerator and waits* for the accelerator to complete the work.

Sending work to the accelerator is independent from sending data to the accelerator.

The programmer must have a mode for managing the program data as well as the program computation.

*various methods of waiting are defined by the model
The programming model

The device execute parallel regions which typically contain work sharing loops* or kernel regions**.

The host:
- Allocates memory on the device
- Transfers data to the device
- Sends the code to the accelerator, passes device arguments to the parallel region, queues the device code, and waits for completion
- Transfers the data back to the host
- Deallocates the device memory

*from the Cray GPU programming mode
**from the PGI Accelerator model
The host+ACC memory model

Accelerator memory may be completely separate from the host memory, as is the case with most current GPUs.

The host may not be able to read or write device memory directly because it is not mapped into the host’s virtual memory space.

All data movement must be done by the host via runtime libraries.

Two primary programmer concerns:

- Memory bandwidth between host and device memory determines the amount of computation that must be available to make running on the accelerator profitable.
- The limited device memory may prohibit offloading regions of code that operate on very large arrays.
The accelerator memory model

Some accelerators implement a weak memory model. They do not support coherence between operations executed by different computation units on the accelerator.

Even on the same computation unit, memory coherence is only guaranteed when the memory operations are separated by an explicit barrier.

Compilers can warn about some of these situations, but may not be able to determine all. Programmer beware!

Some accelerators have memory caches – software managed, hardware managed, or hardware managed and constrained, such as read only caches. The accelerative directive model allows the compiler, with hints from the programmer, to manage these caches rather than requiring the programmer to explicitly manage them.
Directive format

In C and C++, OpenACC directives are specified with the #pragma mechanism

```
#pragma acc directive-name [clause[,clause]…] new-line
```

In Fortran, OpenACC directives in free form source files:

```
!$acc directive-name [clause[,clause]…]
```

- In Fortran, OpenACC directives in fixed form source files:

```
!$acc directive-name [clause[,clause]…]
c$acc directive-name [clause[,clause]…]
*$acc directive-name [clause[,clause]…]
```
Internal control variables

An OpenACC implementation acts as if there are internal control variables that control the behavior of the program. These can be queried and set by the program or programmer.

<table>
<thead>
<tr>
<th>ICV</th>
<th>Ways to modify</th>
<th>Ways to retrieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc-device-type-var</td>
<td>ACC_DEVICE_TYPE</td>
<td>acc_get_device_type</td>
</tr>
<tr>
<td></td>
<td>acc_set_device_type</td>
<td></td>
</tr>
<tr>
<td>acc-device-num-var</td>
<td>ACC_DEVICE_NUM</td>
<td>acc_get_device_num</td>
</tr>
<tr>
<td></td>
<td>acc_set_device_num</td>
<td></td>
</tr>
</tbody>
</table>
Primary OpenACC constructs

 !$acc parallel

When the program encounters an accelerator parallel construct, gangs of workers are created and execute the accelerator parallel region. Once the gangs are created, the number of gangs and the number of workers in each gang remain constant for the duration of the region.

 !$acc kernels

This construct defines a region of the program that is to be compiled into a sequence of kernels for execution on the accelerator. The compiler will break the code into a sequence of accelerator kernels. Typically, each loop nest will be a distinct kernel.

The kernels construct is closely related to the current PGI Accelerator directive model.
OpenACC compilers

The OpenACC standard is being implemented by PGI, Cray, and CAPS.

PGI’s current plan is to release an OpenACC compiler once the complete 1.0 standard is fully implemented in the compiler.

Until then, users are encouraged to continue with the PGI Accelerator model as this is very closely related to the kernels model, and a transition from PGI to OpenACC should be relatively straightforward.
Implicit Programming of Accelerators

The PGI Accelerator directive based approach to programming.

Maximize the work that the compiler is able to do

Concentrate programmer efforts on performance of kernels rather then management and placement of data

The OpenACC “parallel” method put more responsibility on the programmer for data placement
PGI Accelerator -> OpenACC

region  -> kernels
region for -> kernels loop
region do -> kernels loop
data region -> data

local() -> create()
parallel() -> gang()
PGI Accelerator -> OpenACC

PGI Accelerator uses *mirror* and *reflect* for data placement on the GPU. Visibility of these directives by caller at compile time (compiler time check)

OpenACC uses *present* for data placement. There is no visibility of *present* at compile time, so it requires a runtime check. (IPA may pick this up in the future to eliminate runtime check?)
PGI Accelerator -> OpenACC

C subarrays go from \([\text{lowerbound:upperbound}]\)
to \([\text{lowerbound:length}]\)
Subarrays in data clauses must be contiguous
Add *reduction* clauses in loops and regions
Remove *cache* clauses
Replace *reflect* with *present*
Remove *mirror*
Getting started with a real code

Is the code already well structured to take advantage of an accelerator? Is there sufficient parallelism in the code?

What is the performance profile of the code?

What parts of the code can run on the accelerator?
Profile the code
Are the results correct?

How do you determine the correctness of the original code?

Having a graphical way of evaluating code results is highly encouraged

How correct is correct? GPU results will be different from CPU results. You need to have a method for determining how big a delta is acceptable to you

Debug by comparing intermediate output of original code to the results on the GPU
VH-1 code output
# Specify search path for subroutines that perform 1D ppmlr hydrodynamics
VPATH = ../PPMLR
#
# System-dependent parameters
#
# F90 FORTRAN 90 Compiler
# LDR Program to load the objects, typically the same as F90
# LDFLAGS Flags to pass to the compiler during linking
# LIBS A list of libraries to link, normally only netCDF
#
F90 = pgf90
FFLAGS = -c -fast -ta=nvidia,nofma -Minfo=accel
LDR = pgf90
LD.flags = -ta=nvidia
LIBS =
Move most costly routine to GPU

Parabola.F90 contains no I/O, MPI, or subroutine calls

subroutine parabola( nmin, nmax, para, a, deltaa, a6, al, flat )
! GLOBALS
use sweepsize
IMPLICIT NONE
! LOCALS
integer :: n, nmin, nmax
real :: onemfl
real, dimension(maxsweep) :: a, deltaa, a6, al, flat, da, ar, diffa, scrch1, scrch2, scrch3
real, dimension(maxsweep,5) :: para
!$acc region
...
!$acc end region
eeturn
end !subroutine parabola
Compiling the subroutine:

```plaintext
PGI$ make parabola.o
pgf90  -c -fast -ta=nvidia -Minfo=accel ../PPMLR/parabola.f90
parabola:

  23, Generating copyout(deltaa(nmin:nmax))
      Generating copyout(a6(nmin:nmax))
      Generating copyout(scrch1(nmin:nmax))
      Generating copyout(scrch2(nmin:nmax))
      Generating copyout(scrch3(nmin:nmax))
      Generating copyout(diffa(nmin-2:nmax+1))
      Generating copyout(da(nmin-1:nmax+1))
      Generating copyout(al(nmin:nmax+1))
      Generating copyout(ar(nmin-1:nmax))
      Generating copyin(para(nmin-1:nmax+1,1:5))
      Generating copyin(a(nmin-2:nmax+2))
      Generating copyin(flat(nmin:nmax))
      Generating compute capability 1.0 binary
      Generating compute capability 1.3 binary
      Generating compute capability 2.0 binary

  25, Loop is parallelizable
  Accelerator kernel generated
  25, !$acc do parallel, vector(256) ! blockidx%x threadidx%x
      Cached references to size [257] block of 'a'
      CC 1.0 : 5 registers; 1092 shared, 4 constant, 0 local memory bytes; 100% occupancy
      CC 1.3 : 5 registers; 1092 shared, 4 constant, 0 local memory bytes; 100% occupancy
      CC 2.0 : 10 registers; 1036 shared, 68 constant, 0 local memory bytes; 100% occupancy
```
Run the code, check results
Tuning the compute kernel

Use loop directives

module kmod
  contains
  subroutine vaddkernel(A,B,C)  ! We know array size
    real :: A(:,,:), B(:,,:), C(:, :)! dimension(2560,96)
    integer :: i,j
    !$acc region
    !$acc do parallel
      do j = 1, size(A,1)
        !$acc do vector(96)
          do i = 1, size(A,2)
            C(j,i) = A(j,i) + B(j,i)
          enddo
        enddo
      enddo
    !$acc end region
  end subroutine
end module
Loop directive clauses for tuning GPU kernel schedules

<table>
<thead>
<tr>
<th>Clause</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel [(width)]</td>
<td>Parallelize the loop across the multi-processors</td>
</tr>
<tr>
<td>vector [(width)]</td>
<td>SIMD vectorize the loop within a multi-processor</td>
</tr>
<tr>
<td>seq [(width)]</td>
<td>Execute the loop sequentially on each thread processor</td>
</tr>
<tr>
<td>independent</td>
<td>Iterations of this loop are data independent of each other</td>
</tr>
<tr>
<td>unroll (factor)</td>
<td>Unroll the loop \textit{factor} times</td>
</tr>
<tr>
<td>cache (list)</td>
<td>Try to place these variables in shared memory</td>
</tr>
<tr>
<td>private (list)</td>
<td>Allocate a copy of each variable in \textit{list} for each loop iteration</td>
</tr>
</tbody>
</table>
Run the code, check results
Can any of the data movement be optimized?

Look at the –Minfo=accel messages for clues to see if any of the compiler generated data movement is non-optimal.

Insert directives to override compiler generated data movement.

Best solution: keep all data on the GPU and only move off for I/O and message passing
Using GPU device-resident data across subroutines

subroutine timestep(Input, Result, M, N)
    use kmod
    ! Make reflected var's visible
    real, dimension(M,N) :: Input, Result
    integer :: M, N
    real, allocatable :: B, C, D
dimension(::) :: B, C, D
!$acc mirror(B,C,D)
allocate(B(M,N), C(M,N), D(M,N))
    B = 2.0
!$acc update device(Input,B)
call vaddkernel (Input, B)
    ...
call kernel2 (C, D)
    ...
call kernel3 (D, Result)
!$acc update host(Result)
deallocate(B, C, D)
end subroutine

module kmod
    Contains
    subroutine vaddkernel(A, B, C)
        real :: A(:,::), B(:,::), C(:,::)
!$acc reflected (A,B,C)
!$acc region
        C(:,::) = A(:,::) + B(:,::)
!$acc end region
    end subroutine
    ...
end module
Compute region directive clauses for tuning data allocation and movement

<table>
<thead>
<tr>
<th>Clause</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflected((list))</td>
<td>Arguments already on GPU</td>
</tr>
<tr>
<td>copy ((list))</td>
<td>Copy in and out of GPU memory</td>
</tr>
<tr>
<td>copyin ((list))</td>
<td>Only copy in to GPU memory</td>
</tr>
<tr>
<td>copyout ((list))</td>
<td>Only copy out of GPU memory</td>
</tr>
<tr>
<td>local ((list))</td>
<td>Allocate locally on GPU</td>
</tr>
<tr>
<td>mirror ((list))</td>
<td>Allocation state on GPU = CPU</td>
</tr>
<tr>
<td>update device ((list))</td>
<td>Update device copies of the arrays</td>
</tr>
<tr>
<td>update host ((list))</td>
<td>Update host copies of the arrays</td>
</tr>
</tbody>
</table>
Running the rest of the code

Now that all of the arrays live on the GPU, go through the code and put !$acc region’s in.

Subroutine calls cannot be in regions

Moving data back to host:
  – I/O cannot be in regions
  – MPI message passing cannot be in regions

Code inside of region must be parallel
  – Non-parallel code in !$acc scalar regions
Timing / Profiling

How long does my program take to run?
- `time ./myprogram`

How long do my kernels take to run?
- `pgfortran –ta=nvidia,time`

Environment variables:
- `export ACC_NOTIFY=1`
- `export NVDEBUG=1`
- `export CUDA_PROFILE=1`
- `export CUDA_PROFILE_CONFIG= ~/cudaprof.cfg`
- `export CUDA_PROFILE_CSV=1`
- `export CUDA_PROFILE_LOG= ~/cudaprof.log`
How did we make Vectors Work?
Compiler-to-Programmer Feedback – a classic “Virtuous Cycle”

We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators.
Compiler-to-Programmer Feedback
Incremental porting/tuning for GPUs

Directives, Options, RESTRUCTURING

HPC Code → PGI Compiler → CCFF → HPC User

Performance

x64 + Acc → Trace → PGPROF

The Portland Group
Common Compiler Feedback Format

http://www.pgroup.com/ccff
PGPROF with CCFF Messages

Line-level information for line 256:

1. Intensity = 0.0
2. Generating copy(q(its,site,kts,kte,jts,jte))
3. Generating copyin(p(its,site,kts,kte,jts,jte))
4. Generating copyin(w(its,site,jts,jte))
5. Generating copyin(den(its,site,jts,jte))
6. Generating copyin(dalz(its,site,jts,jte))

Parallelism  Histogram  Compiler Feedback  System Information

Browsing: ./wrf
General Compiler Feedback

How the function was compiled
Interprocedural optimizations
Profile-feedback runtime data
  – Block execution counts
  – Loop counts, range of counts
Compiler optimizations, missed opportunities
  – Vectorization, parallelization
  – Altcode, re-ordering of loops, inlining
  – X64+GPU code generation, GPU kernel mapping, data movement
Compute intensity – important for GPUs & Multi-core
program Main
use accel_lib
...
 !$omp parallel private(ilo, iho, k, flux) num_threads(2)
  do iter = 1, 100
   call acc_set_device_num(omp_get_thread_num(), ACC_DEVICE_NVIDIA)
   if ( first ) then
     dkm = km/omp_get_thread_num()
     ilo = dkm *omp_get_thread_num() + 1
     if (omp_get_thread_num() + 1 == omp_get_num_threads()) then
       iho = km
     else
       iho = dkm*(omp_get_thread_num() + 1)
     endif
   endif
   !$acc region
   do k=ilo,iho
     ...
   end do
   !$acc end region
  !$omp end parallel
end do
end program Main
Reference Materials

- **PGI Accelerator programming model**

- **CUDA Fortran**

- **CUDA-x86**
  - http://www.pgroup.com/resources/cuda-x86.htm

- **Understanding the CUDA Threading Model**
  - http://www.pgroup.com/lit/articles/insider/v2n1a5.htm