12 Tips for Maximum Performance with PGI Directives in C

List of Tricks

Trick #1: Eliminate Pointer Arithmetic

Pointer arithmetic is often used to swap pointers or increment through an array. This code is currently **not allowed** within compute regions. For example, the following routine:

```
void memcpy(float * restrict A, float * restrict B, int count) {
float* ptrA = A;
float* ptrB = B;
    while (count--) {
        *ptrA++ = *prtB++;
   }
   return;
}
```

would need to be rewritten to use array indexing:

```
void memcpy(float *restrict A, float * restrict B, int count) {
#pragma acc region
{
    for (int i=0; i<count;++i) {
        A[i] = B[i];
    }
}
return;
}
```

Note the use of the C99 *restrict* keyword. This asserts to the compiler that the arrays do not overlap and hence updates to one will not affect others.

Trick #2: Privatize arrays

Some loops will fail to offload because parallelization is inhibited by arrays that must be privatized for correct parallel execution. In an iterative loop, data which is used only during a particular iteration can be declared private. And in general code regions, data which is used within the region but is not initialized prior to the region, and is re-initialized prior to any use after the region can be declared private.

For example, if the following code is compiled:

```
#pragma acc region
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         for (int ii=0; ii<10;++ii) {
            tmp[ii] = ii;
         }
         sum=0;
         for (int ii=0; ii<10;++ii) {
            sum+=tmp[ii];
         }
         A[i][j] = sum;
      }
   }
}
```

Informational messages similar to the following will be produced:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel private.c
main:
     19, Generating copyout(A[0:N-1][0:M-1])
         Generating copyout(tmp[0:9])
         Generating compute capability 2.0 binary
     21, Parallelization would require privatization of array 'tmp[0:9]'
     22, Parallelization would require privatization of array 'tmp[0:9]'
         Accelerator kernel generated
         21, #pragma acc for seq
         22, #pragma acc for seq
             Non-stride-1 accesses for array 'A'
             CC 2.0 : 18 registers; 0 shared, 64 constant,
                      0 local memory bytes; 16% occupancy
```

A CUDA kernel is generated, but it will be very inefficient because it is sequential. But if you further specify using a `loop` pragma `private` clause that it is safe to privatize array `tmp` in the scope of the do j loop:

```
#pragma acc region
{
   for (int i=0; i<N;++i) {
#pragma acc for private(tmp[0:9])
      for (int j=0; j<M;++j) {
         for (int ii=0; ii<10;++ii) {
            tmp[ii] = ii;
         }
         sum=0;
         for (int ii=0; ii<10;++ii) {
           sum+=tmp[ii];
         }
         A[i][j] = sum;
      }
   }
}
```

It will provide the PGI compiler with the information necessary to successfully compile the nested loop for execution on an NVIDIA GPU:

```
% pgcc -ta=nvidia -Minfo=accel,cc20 private1.c
main:
     19, Generating copyout(A[0:N-1][0:M-1])
         Generating compute capability 2.0 binary
     21, Loop is parallelizable
     23, Loop is parallelizable
         Accelerator kernel generated
         21, #pragma acc for parallel, vector(16) /*blockIdx.y threadIdx.y*/
         23, #pragma acc for parallel, vector(16) /*blockIdx.x threadIdx.x*/
            CC 2.0 : 18 registers; 8 shared, 64 constant,
                     0 local memory bytes; 100% occupancy
```

Trick #3: Make while loops parallelizable

The PGI Accelerator compiler can't automatically convert while loops into a form suitable for running on the GPU. But it is often possible to manually convert a while loop into a countable rectangular do loop. For example, if the following code is compiled:

```
#pragma acc region
{
   while (i<N && found == -1)  {
      if (A[i] >= 102.0f) {
        found = i;
      }
      ++i;
   }
}
```

Informational messages similar to the following will be produced:

```
% pgcc -ta=nvidia -Minfo=accel while.c
     20, Accelerator restriction: loop has multiple exits
         Accelerator region ignored
```

But if the loop is restructured into the following form as a for loop:

```
#pragma acc region
{
   for (i=0;i<N;++i) {
      if (A[i] >= 102.0f) {
        found[i] = i;
      } else {
        found[i] = -1;
      }
   }
}
i=0;
while (i < N && found[i] < 0) {
  ++i;
}
```

It will provide the PGI compiler with the information necessary to successfully compile the nested loop for execution on an NVIDIA GPU:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel while1.c
main:
     21, Generating copyin(A[0:N-1])
         Generating copyout(found[0:N-1])
         Generating compute capability 2.0 binary
     23, Loop is parallelizable
         Accelerator kernel generated
         23, #pragma acc for parallel, vector(256)/*blockIdx.x threadIdx.x*/
             Using register for 'found'
             CC 2.0 : 8 registers; 4 shared, 60 constant,
                      0 local memory bytes; 100% occupancy
```

Trick #4: Rectangles are better than triangles

All loops must be rectangular. For triangular loops, the compiler will either serialize the inner loop or make the inner loop rectangular by adding an implicit if statement to skip the lower part of the triangle. For example, if the following triangular loop is compiled:

```
#pragma acc region copyout(A[0:N-1][0:M-1])
{
   for (int i=0; i<N;++i) {
      for (int j=i; j<M;++j) {
         A[i][j] = i+j;
      }
   }
}
```

Here's the triangular loop!

Informational messages similar to the following will be produced:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel triangle.c
main:
     22, Generating copyout(A[:N-1][:M-1])
         Generating compute capability 2.0 binary
     24, Loop is parallelizable
         Accelerator kernel generated
         24, #pragma acc for parallel, vector(256) /*blockIdx.x threadIdx.x*/
             CC 2.0 : 18 registers; 4 shared, 60 constant,
                      0 local memory bytes; 100% occupancy
     25, Loop is parallelizable
```

While the loops seemed to have been parallelized, the resulting code will **likely fail**. Why? Because the compiler copies out the entire `A` array from device to host and in the process copies garbage values into the lower triangle of the host copy of `A`. However, if a `copy` clause is specified on the accelerator region boundary, correct code will be generated. For example, if the following code is compiled:

```
#pragma acc region copy(A[0:N-1][0:M-1])
{
   for (int i=0; i<N;++i) {
      for (int j=i; j<M;++j) {
         A[i][j] = i+j;
      }
   }
}
```

Informational messages similar to the following will be produced :

```
% pgcc -ta=nvidia,cc20 -Minfo=accel triangle1.c
main:
     22, Generating copy(A[:N-1][:M-1])
         Generating compute capability 2.0 binary
     24, Loop is parallelizable
         Accelerator kernel generated
         24, #pragma acc for parallel, vector(256) /*blockIdx.x threadIdx.x*/
             CC 2.0 : 18 registers; 4 shared, 60 constant,
                       0 local memory bytes; 100% occupancy
     25, Loop is parallelizable
```

Trick #5: Restructure linearized arrays with computed indices

It is not uncommon for legacy codes to use computed indices for computations on multi-dimensional arrays that have been linearized. For example, if the following loop with a computed index into the linearized array `A` is compiled:

```
#pragma acc region copyout(A[0:N*M-1])
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         idx = (i*N)+j;
         A[idx] = B[i][j];
      }
```

```
   }
}
```

Informational messages similar to the following will be produced:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel linearization.c
main:
     23, Generating copyout(A[:M*N-1])
         Generating copyin(B[0:N-1][0:M-1])
         Generating compute capability 2.0 binary
     25, Complex loop carried dependence of '*(A)' prevents parallelization
     26, Complex loop carried dependence of '*(A)' prevents parallelization
         Parallelization would require privatization of array 'A[:M*N-1]'
         Accelerator kernel generated
         25, #pragma acc for seq
         26, #pragma acc for seq
             Non-stride-1 accesses for array 'B'
             CC 2.0 : 15 registers; 0 shared, 72 constant,
                      0 local memory bytes; 16% occupancy
```

The code will run on the GPU but it **will execute sequentially and run very slowly**. You have two options. First, the loop can be restructured to remove linearization:

```
#pragma acc region copyout(A[0:N-1][0:M-1])
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         A[i][j] = B[i][j];
      }
   }
}
```

Allowing the compiler to successfully generate a parallel GPU code:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel linearization1.c
main:
     24, Generating copyout(A[:N-1][:M-1])
         Generating copyin(B[0:N-1][0:M-1])
         Generating compute capability 2.0 binary
     26, Loop is parallelizable
     27, Loop is parallelizable
         Accelerator kernel generated
         26, #pragma acc for parallel, vector(16) /*blockIdx.y threadIdx.y*/
         27, #pragma acc for parallel, vector(16) /*blockIdx.x threadIdx.x*/
             CC 2.0 : 12 registers; 8 shared, 64 constant,
                      0 local memory bytes; 100% occupancy
```

Or second, independent clauses can be specified on the do loops to provide the compiler with the information it needs to safely parallelize the loops:

```
#pragma acc region copyout(A[0:N*M-1])
{
#pragma acc for independent
   for (int i=0; i<N;++i) {
#pragma acc for independent
```

```
      for (int j=0; j<M;++j) {
         idx = (i*N)+j;
         A[idx] = B[i][j];
      }
   }
}
```

Trick #6: Privatize live-out scalars

It is common for loops to initialize scalar work variables, and for those variables to be referenced or re-used after the loop. Such a variable is called a “live out” scalar, because correct execution may depend on its having the last value it was assigned in a serial execution of the loop(s). For example, if the following loop with a live out variable `idx` is compiled:

```
#pragma acc region
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         idx = i+j;
         A[i][j] = idx;
      }
   }
}
printf("%d %d %d\n",idx, A[1][1], A[2][1]);
```

Informational messages similar to the following will be produced:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel live.c
main:
     20, Generating copyout(A[0:N-1][0:M-1])
         Generating compute capability 2.0 binary
     22, Loop is parallelizable
     23, Inner sequential loop scheduled on accelerator
         Accelerator kernel generated
         22, #pragma acc for parallel, vector(32) /*blockIdx.x threadIdx.x*/
         23, #pragma acc for seq
             Non-stride-1 accesses for array 'A'
             CC 2.0 : 17 registers; 4 shared, 60 constant,
                        0 local memory bytes; 16% occupancy
     24, Accelerator restriction: induction variable live-out from loop: idx
     25, Accelerator restriction: induction variable live-out from loop: idx
```

While some code will run on the GPU, the inner loop is executed sequentially. Looking at the code, the use of `idx` in the print statement is only for debugging purposes. In this case, you know the computations will still be valid even if `idx` is privatized so the code can be modified as follows:

```
#pragma acc region
{
#pragma acc for private(idx)
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         idx = i+j;
         A[i][j] = idx;
      }
   }
```

```
}
printf("%d %d %d\n",idx, A[1][1], A[2][1])
```

A much more efficient fully parallel kernel will be generated:

```
% pgcc -ta=nvidia,cc20 -Minfo=accel live1.c
main:
     20, Generating copyout(A[0:N-1][0:M-1])
         Generating compute capability 2.0 binary
     23, Loop is parallelizable
     24, Loop is parallelizable
         Accelerator kernel generated
         23, #pragma acc for parallel, vector(16) /*blockIdx.y threadIdx.y*/
         24, #pragma acc for parallel, vector(16) /*blockIdx.x threadIdx.x*/
             CC 2.0 : 10 registers; 8 shared, 60 constant,
                      0 local memory bytes; 100% occupancy
```

Note that the value printed out for `idx` in the print statement will be different than in a sequential execution of the program.

Trick #7: Inline function calls in directives regions

One of the most common barriers to maximum GPU performance is the presence of function calls in the region. To run efficiently on the GPU, the compiler must be able to inline function calls.

There are two ways to invoke automatic function inlining with the PGI Accelerator compilers:

First, if the function(s) to be inlined are in the same file as the section of code containing the accelerator region, you can use the `–Minline` compiler command-line option to enable automatic procedure inlining. This will enable automatic inlining of functions throughout the file, not only within the accelerator region.

If you would like to restrict inlining to specific functions, say func1 and func2, use the option `–Minline=func1,func2`. To learn more about controlling inlining with `–Minline`, just type `pgcc -help -Minline` in a shell window.

Second, if the function(s) to be inlined are in a separate file from the code containing the accelerator region, you need to use the inter-procedural optimizer with automatic inlining enabled by specifying `–Mipa=inline` on the compiler command-line. `–Mipa` is both a compile-time and link-time option, so you need to specify it on the command-line when linking your program as well for inlining to occur. As with `–Minline`, you can learn more about controlling inter-procedural optimizations and inlining by using `pgcc -help -Mipa`.

The following types of C and C++ functions cannot be inlined:

- Functions containing switch statements
- Functions which reference a static variable whose definition is nested within the function
- Function which accept a variable number of arguments

Certain C/C++ functions can only be inlined into the file that contains their definition:

- Static functions
- Functions which call a static function
- Functions which reference a static variable

If you encounter these or any other restrictions that prevent automatic inlining of functions called in accelerator regions, the only alternative is to inline them manually.

Trick #8: Watch for runtime device errors

Once you have successfully offloaded code in an accelerator region for execution on the GPU, you can still encounter errors at runtime due to common porting or coding errors that are not exposed by execution on the host CPU.

If you encounter the following error message when executing a program:

```
Call to cuMemcpyDtoH returned error 700: Launch failed
```

This typically occurs when the device kernel returns an execution error due to an out-of-bounds or other memory access violation. For example the following code will generate such an error:

```
#pragma acc region copyin(B[0:N-1][0:M-1])
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         A[i][j] = B[i][j+1];
      }
   }
}
```

The only way to isolate such errors currently is through inspection of the code in the accelerator region, or by compiling and executing on the host using the -Mbounds command-line option which will instrument the executable to print an error message for out-of-bounds array accesses.

If you encounter the following error message when executing a program:

```
Call to cuMemcpy2D returned error 1: Invalid value
```

This typically occurs if there is an error copying data to/from the device. For example, the following code will generate such an error:

```
#pragma acc region copyin(B[0:N-1][0:M+1])
{
   for (int i=0; i<N;++i) {
      for (int j=0; j<M;++j) {
         A[i][j] = B[i][j];
      }
   }
}
```

The only way to isolate such errors currently is through inspection of the code in the accelerator region or inspection of the `-Minfo` informational messages at compile time.

Trick #9: Accelerating C++

The PGI Accelerator programming model is currently supported in Fortran 2003 and C99, but is not directly supported in C++. However, it is possible to offload portions of C++ applications by refactoring code regions and loop nests into extern 'C' program units and compiling them with the PGI Accelerator C compiler. While this requires additional work, the resulting code will still be 100% portable to other compilers and platforms.

Code that is heavily reliant on C++ will be more difficult to port using PGI Accelerator C than code that is already C99 or mostly so.

To build C++ applications with PGI Accelerator C program units, compile each C++ file (including the main program) with the PGI C++ compiler, each C file with the PGI C compiler, and link the executable with the PGI C compiler driver. For example, a file main.cpp containing this C++ main program:

```
#include <iostream>
extern "C" int matit();
int
main()
{
     int i;
     i=matit();
     cout << "return from matit ==" << i <<endl;
}
```

And a file csub.c containing the C function matit and potentially several other C functions it calls can be compiled using the following steps :

```
% pgcpp -fast -c main.cpp <ret>
% pgcc -fast -ta=nvidia -c csub.c <ret>
% pgcc -pgcpplibs -ta=nvidia:time main.o csub.o <ret>
```

The option `-pgcpplibs` to the `pgcc` compiler driver will append all required C++ libraries to the link line and enable linking of executables where the main program and potentially other program units are C++.

Trick #10: Be Aware of Data Movement

Once you have successfully offloaded a CUDA kernel using PGI Accelerator pragmas, you should understand and try to optimize the data movement between host memory and GPU device memory.

You can see exactly what data movement is occurring for each generated CUDA kernel by looking at the informational messages emitted by the PGI Accelerator compiler:

```
% pgcc -ta=nvidia test.c -Minfo=accel
testgpu1:
     49, Generating copyin(a[:19999])
         Generating copyin(ix[0:97][0:197])
         Generating copy(b[1:98][1:198])
         ...
```

Arrays a and ix being copied from host memory to GPU device memory before CUDA kernel launch

Elements of arrays b copied both to the GPU and back to host memory after CUDA kernel execution

You can see how much execution time is spent moving data between host memory and device memory by linking your executable with the `time` sub-option added to `-ta=nvidia` command-line option:

```
% pgcc -ta=nvidia,time test.c
% a.out

Accelerator Kernel Timing data
test.c
  testgpu1
    49: region entered 1000 times
        time(us): total=22568519 init=107681 region=22460838
                  kernels=20116 data=21971428
        w/o init: total=22460838 max=50259 min=22290 avg=22460
        52: kernel launched 1000 times
            grid: [13x7]  block: [16x16]
            time(us): total=20116 max=28 min=18 a
```

20,116 microseconds spent executing kernels

21,971,428 microseconds spent on moving data between host memory and GPU device

Once you have examined and timed the data movement required at accelerator region boundaries, there are several techniques you can use to minimize and optimize data movement.

Trick #11: Use Contiguous Memory for Multi-dimensional Arrays

In this example, arrays `b` and `ix` are declared as pointer arrays. Because data is copied to and from the host and device in contiguous segments, each row of these arrays will be copied separately. We can speed up the data transfers by dynamically allocating the two pointer arrays as a single contiguous block of memory and passing them to our function as a multi-dimensional C99 variable length array (VLA). With this approach, both arrays can be copied in a single transfer.

```
float *restrict b0;
int *restrict ix;

void
testgpu1( int N, int M, float b[N][M], float *restrict a, int ix[N][M],
          const int niter )
{
    int i,j;
    for (int it=1; it <= niter ; ++it) {
#pragma acc region copyin(a[0:(N*M)-1])
   {
        for( i = 1; i < N-1; ++i ){
            for( j = 1; j < M-1; ++j ){
              b[i][j] += 0.5f*(a[ix[i-1][j-1]] + a[i*M+j]);
            }
        }
   }
   }
}
…

testgpu1( N, M, b0, a0, ix, 1000 );
```

Running the program again after linking once more with the `-ta=nvidia,time` command-line option shows these results;

```
%a.out

Accelerator Kernel Timing data
test1a.c
  testgpu1
    47: region entered 1000 times
        time(us): total=1090302 init=102621 region=987681
                  kernels=20471 data=497893
        w/o init: total=987681 max=1144 min=980 avg=987
        50: kernel launched 1000 times
            grid: [13x7]  block: [16x16]
            time(us): total=20471 max=32 min=19
```

Kernels execution time is remains unchanged

Time to move data between host memory and GPU device memory falls by over 97%

Trick #12: Use Data Regions to Avoid Inefficiencies

In this test example, the PGI Accelerator compute region is contained within a loop. The data are being unnecessarily copied between the host and device every iteration of the loop. Instead, the data need only be copied to the device once before the outer loop and back to the host after the loop. To accomplish this, we will add a data region.

```
void
testgpu1( int N, int M, float b[N][M], float *restrict a, int ix[N][M],
          const int niter )
{
    int i,j;
#pragma acc data region copyin(a[0:(N*M)-1]), copy(b[0:N-1][0:M-1]),
                        copyin(ix[0:N-1][0:M-1])
   {
    for (int it=1; it <= niter ; ++it) {
#pragma acc region
   {
        for( i = 1; i < N-1; ++i ){
            for( j = 1; j < M-1; ++j ){
              b[i][j] += 0.5f*(a[ix[i-1][j-1]] + a[i*M+j]);
            }
        }
   }}}
}
```

Running the program again after linking once more with the `-ta=nvidia,time` command-line option shows these results:

```
% a.out
```

```
Accelerator Kernel Timing data
  testgpu1
    49: region entered 1000 times
        time(us): total=139372 init=106 region=139266
                  kernels=17941 data=0
```

No data movement in the compute kernel

```
        w/o init: total=139266 max=213 min=136 avg=139
        52: kernel launched 1000 times
            grid: [13x7]  block: [16x16]
            time(us): total=17941 max=28 min=17 avg=17
  testgpu1
    46: region entered 1 time
```

Result with only one data transfer each way.

```
        time(us): total=244478 init=103975 region=140503
                  data=514
        w/o init: total=140503 max=140503 min=140503 avg=140503
```