Are Directives Right for My Application?

Thank you for your interest in the 2x in 4 weeks program. This document is intended to help you understand if your code will benefit from the PGI Accelerator Compiler and how to use directives on GPUs.

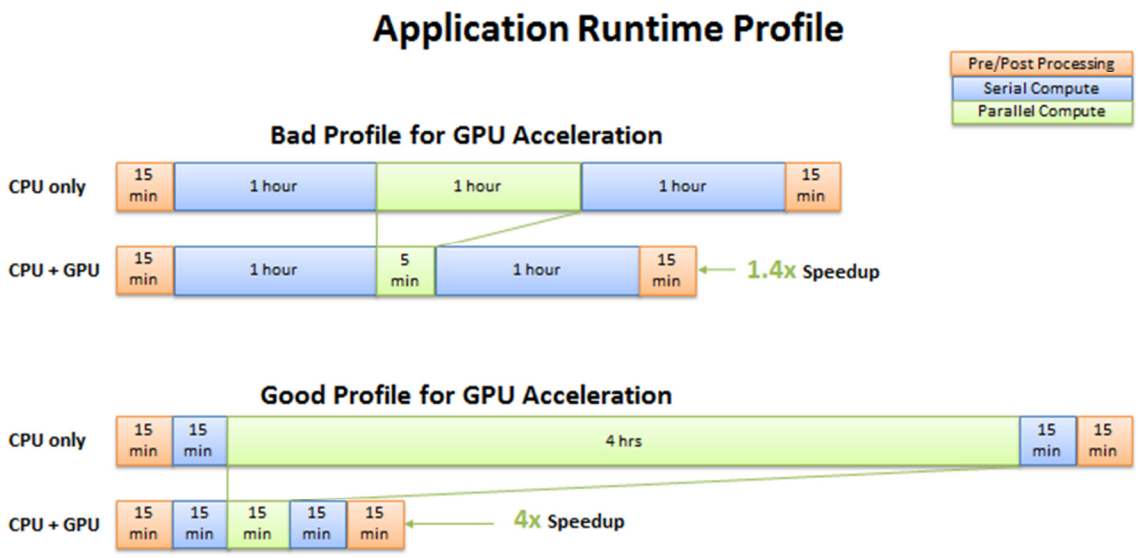
Two Things to Know about NVIDIA GPUs

1. NVIDIA GPUs can compute massive amounts of data in parallel using hundreds of computing cores concurrently.

2. NVIDIA GPUs perform best when the hundreds of cores are kept busy with thousands or even millions of threads of parallel execution.

Applying Amdahl's Law to Your Code

Understanding your application runtime profile is essential to deciding whether directives on GPUs are right for you. **If the application runtime is not dominated by data intensive computation, then GPUs are not right for you.** However, if the application runtime is consumed by a handful of compute intensive segments, then GPUs will offer notable speed-ups.



Despite the GPU significantly accelerating the parallel portion of the first application, it sees only a 40% improvement in run-time performance because non-parallel portions are relatively large. **The second application has a great profile for GPU acceleration.** As the parallel portion dominates the initial run-time, GPUs can make a significant impact on the whole application performance.

Code Structures That Work with Directives

With PGI Directives, parallelism is expressed in parallel loops. A program appropriate for directives on GPUs will have one or more parallel loops with very large aggregate iteration counts. If there is a single parallel loop with thousands of iterations, your code may effectively take advantage of hundreds of cores in the GPU. If loop limits are small, nested parallel loops are necessary to ensure enough parallelism.

> **Hint #1**
>
> Loop nests can be relatively large, running to many hundreds or even thousands of lines with a large number of inner loop nests and complex expressions. However the loop limits must be rectangular.

At least some of the loops to be offloaded **must be fully data parallel**, which means no synchronization or dependences across iterations. Some loops in nests to be offloaded can require limited synchronization, as long as they are vectorizable in the classic sense; for example, reduction loops can be offloaded.

> **Hint #2**
>
> Fully data parallel = no synchronization or data dependencies across iterations

Code Structures That Do Not Work with Directives

Code regions and loops that are not good candidates for GPU acceleration using PGI directives include:

- Non-loop-based, object-oriented or branch-intensive codes
- Loops that are not parallelizable or vectorizable
- Loops with indirect memory accesses
- Loops with conditionals that result in significant divergence of the code paths
- Loops that perform unstructured accumulations (simple structured accumulations like dot products are OK)
- Loops with calls to functions in pre-compiled libraries whose source code is not visible to the PGI Accelerator compilers
- Loops with I/O statements
- Loops with branches into or out of the loop, or return statements

- Loops with Fortran STOP statements

Strongly Recommended Coding Guidelines

If an application has a few isolated compute-intensive portions in the source code, it is easier to accelerate on the GPU. On the other hand, if an application has a flat performance profile (small percentages of time spent in each of a large number of separate functions), large portions of the source code will have to be visited and potentially adapted for offloading with directives.

> **Hint #3**
>
> For applications with a flat performance profile, it may help to use the device-resident data features of the PGI Accelerator programming model to leave large data structures in device memory across kernels and even across host subroutine boundaries.

Here are some general guidelines concerning coding style that should be followed in both C and Fortran:

- Computed array indices should be avoided
- Use simple expressions in array subscripts
- All function calls in code regions to be offloaded must be either automatically inlinable by the PGI Accelerator compilers or manually inlined by the programmer
- Conditionals are OK but be aware of divergence
- Loops to be offloaded must have rectangular loop limits and have invariant trip counts
- In C, loops can only operate on data of any integer type, float, double or struct (nested structs cannot offload to GPU)
- In C, pointers used to access arrays in loops must be declared with C99 restrict

Hint #4

It is also common in C to use literal constants (e.g. 3.14159) in single-precision calculations without explicitly typing the constant as float (e.g. 3.14159f). In such cases, the C standard requires that the constant be treated as double precision and necessarily all calculations involving it are promoted to double precision. So be careful!

A common idiom used in C programs is to save a value in a scalar for later reuse. For instance, instead of writing:

```
for( i = 0; i < n-1; ++i ){
  b[i] = a[i] + a[i+1];
}
```

You might try to save one array fetch in the loop by using a scalar:

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

Such micro-optimization is particularly common in C programs, but the second formulation has a loop-carried dependence for the scalar variable `x`. The value of `x` is assigned in one iteration and then used in the next iteration, so the iterations are no longer independent. The original loop was completely parallel, so the problem is not the algorithm, it's just the coding style used in the program.

Hint #5

In porting C codes, you may run into cases (like the one above) where coding style needs to be changed to use directives effectively. For more examples, see http://www.pgroup.com/lit/articles/insider/v1n2a1.htm.

In Fortran, only nested loops can be offloaded—no array syntax, no reshape(), etc. In general, it is best to avoid exotic F90/F2003 features. Loops to be offloaded can only operate on data of

type integer, real, double precision, complex, double complex, derived types, allocatables, and CUDA device data. The pointer attribute is not supported; pointer arrays may be specified, but pointer association is not preserved in GPU device memory. Most scalar arithmetic and transcendental intrinsic functions, including reductions (e.g. max(), sum()) are supported.

Now What?

Have you determined that computationally-intensive portions of your application are suitable for GPU acceleration using directives?

If so, then we encourage you to register and participate in the 2x in 4 Weeks program! We also have a few tools to help you hit the ground running.

- For an introduction to directives video, see <link>.
- For Top Tricks for Optimal Performance, see <link>.
- For a complete specification of the PGI Accelerator directives syntax, see http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf.

If you found that directives are not for you, then you may try exposing more parallelism in your code to make it better. If this doesn't work, then it may be that your code is not appropriate for GPU computing.