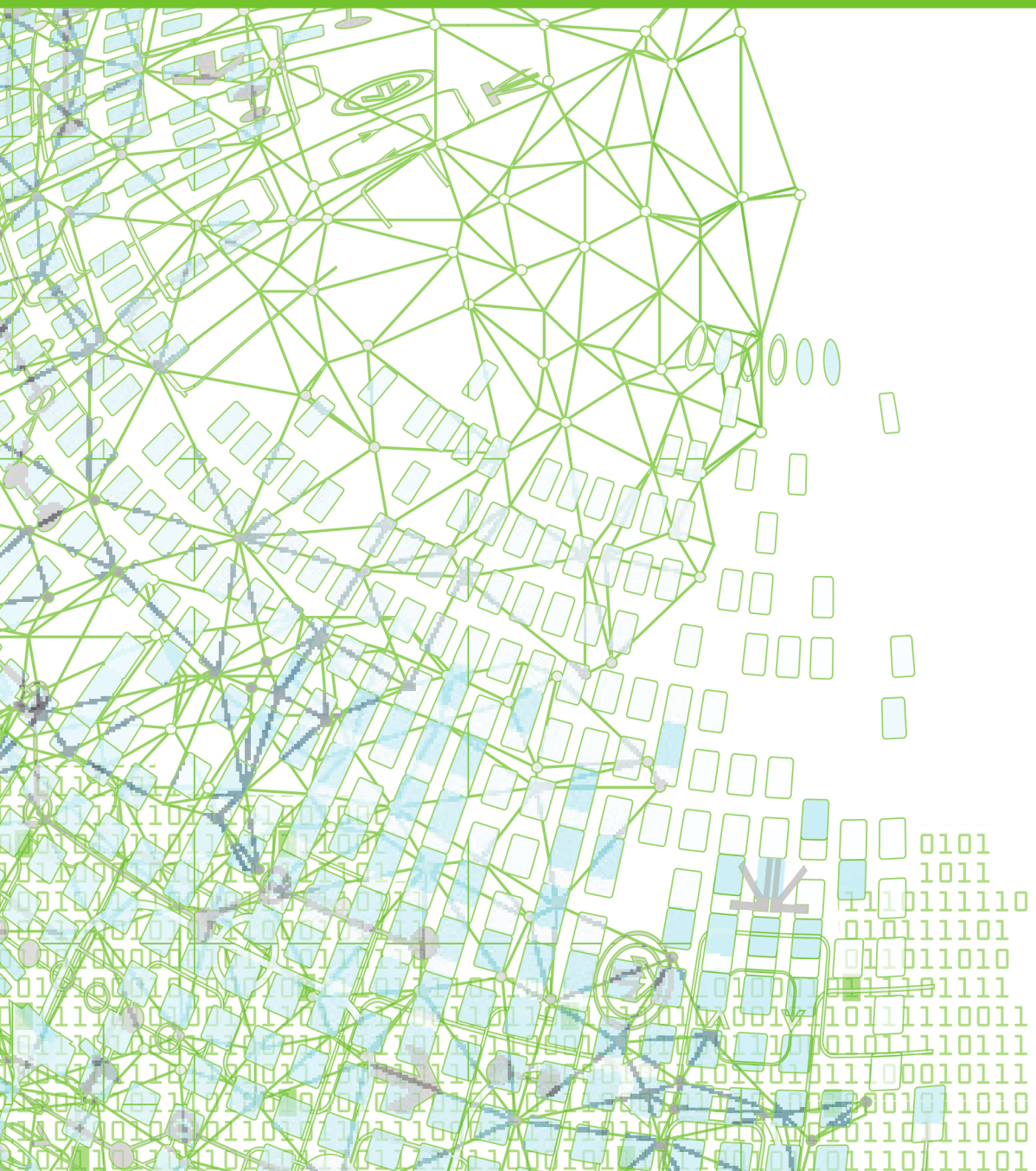
PGI® COMPILERS & TOOLS

PROFILER OPENACC TUTORIAL

Version 2019



NVIDIA

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1. TUTORIAL SETUP

The tutorial assumes you have installed and configured the PGI 2019 compilers with the NVIDIA CUDA toolkit included. The tutorial uses the Cactus BenchADM Benchmark (see http://cactuscode.org/download/). Note that BenchADM uses double-precision arithmetic.

First download the tutorial source file package, http://www.pgroup.com/lit/samples/benchadm_nvprof.tar.gz from the PGI website and unpack the source in a local directory. Next bring up a shell command window (we use csh), cd to the directory 'PGI_Acc_benchADM', and ensure your environment is initialized for use of the PGI 2019 compilers:

```
% which pgfortran
/opt/pgi/linux86-64/2019/bin/pgfortran
```

If your environment is not yet initialized, execute the following commands to do so:

```
% setenv PGI /opt/pgi
% set path=($PGI/linux86-64/19.1/bin $path)
% setenv LM_LICENSE_FILE=$LM_LICENSE_FILE:/opt/pgi/license.dat
```

You may have to ask your system administrator for the pathname to the FlexNET license file containing PGI 2019 license keys.

Note the information for this tutorial was gathered on a workstation with a Intel Core i7-3930K and NVIDIA Tesla K40c. All examples were compiled with PGI version 16.5, your results may vary.

Chapter 2. PROFILING THE APPLICATION

The first step to effectively working with the PGI Accelerator compilers is to understand which parts of your code can benefit from GPU acceleration. The most efficient way to determine this is to profile your code using the PGI profiler.

First, build BenchADM with the options to enable Common Compiler Feedback Format (CCFF) and profile the code using the profiler.

```
  % make OPT="-fast -Minfo=ccff" build_base
  % pgprof --cpu-profiling-mode flat ./bin/benchADM_base BenchADM_200l.par
======== CPU profiling result (flat):
 99.72%    118.9s  bench_staggeredleapfrog2_
  0.15%     180ms  munmap
  0.07%      80ms  PUGH_ReductionMinVal
  0.06%      70ms  PUGH_ReductionMaxVal

Time(%)      Time  Name
```

Two parameters files have been provided. BenchADM_200l.par sets up a large simulation which will use to measure the application's performance. A much smaller BenchADM_20l.par parameter file has also been provided to do quick validation of the results and the effects of running the application on the GPU.

BenchADM has a fairly simple profile with nearly all of the time spent in a single routine called "bench_staggeredleapfrog2". OpenACC is designed to accelerate loops in your code so let's drill down further and find the particular source lines in "bench_staggeredleapfrog2" that account for most of the time being spent in this routine (notice the addition of the option '--cpu-profiling-scope instruction'):

```
  % pgprof --cpu-profiling-scope instruction --cpu-profiling-mode flat ./bin/
benchADM_base BenchADM_200l.par
======== CPU profiling result (flat):
Time(%)      Time  Name
  1.40%     1.67s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
 0x8880)
  1.22%     1.46s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
 0x87e6)
  1.14%     1.36s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
 0x88c7)
  1.02%     1.22s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
 0x8722)
```

```
 0.94%     1.12s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
0x874a)
 0.88%     1.05s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:341
0x8833)
 0.86%     1.03s  bench_staggeredleapfrog2_ (./src/StaggeredLeapfrog2.F:374
0xc661)
```

The line 341 is part of a loop begining at line 338. Given the high percentage of time spent in this loop, it seems to be a good candidate for GPU acceleration.

Chapter 3.
ADDING OPENACC DIRECTIVES

Now that we have identified a likely candidate loop for acceleration and created a performance baseline, let's add our first OpenACC directives to the code. Open the file "src/StaggeredLeapfrog2.F" in a text editor. Insert "!$ACC KERNELS" on a new line right above line 338, then add a closing directive "!$ACC END KERNELS" on a new line right below line 815 (the **ENDDO** corresponding to the **DO** loop at line 366). Write the modified code out into a file named "src/StaggeredLeapfrog2_acc1.F".

Re-build the code using the command:

```
% make OPT="-fast -ta=tesla -Minfo=accel" build_acc1
```

The "-ta=tesla" target accelerator option instructs the compiler to interpret OpenACC directives and try to offload accelerator regions to an NVIDIA GPU.

The -Minfo=accel option instructs the compiler to emit CCFF messages to stdout during compilation. The messages indicate whether or not an accelerator region is generated, which arrays are copied to/from the host to the GPU accelerator, and report the schedule used (parallel and/or vector) to map the kernel onto the GPU accelerator processors.

Scrolling back in your command window, you should see output from the PGI compiler that looks something like the following:

```
pgfortran  -fast -ta=tesla -Minfo=accel   -c -o objdir/
StaggeredLeapfrog2_acc1.o ./src/StaggeredLeapfrog2_acc1.F
bench_staggeredleapfrog2:
    366, Generating copyout(adm_kzz_stag(2:nx-1,2:ny-1,2:nz-1)...)
         Generating copyin(lalp(:nx,:ny,:nz))
         Generating copyout(adm_kxx_stag(2:nx-1,2:ny-1,2:nz-1))
         Generating copyin(lgzz(:nx,:ny,:nz),lgyz(:nx,:ny,:nz)...)
    367, Loop is parallelizable
    371, Loop is parallelizable
    375, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        367, !$acc loop gang ! blockidx%y
        371, !$acc loop gang, vector(4) ! blockidx%z threadidx%y
        375, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

Voila! You've succeeded in mapping a the loop start at line 367 onto a GPU accelerator. If you don't see such output, you may have made an error editing the source. You will find versions pre-edited with the directives and known to work in the file "src/STEPS/

StaggeredLeapfrog2_acc1.F". Feel free to compare to that version, or to simply copy it to "src/StaggeredLeapfrog2_acc1.F" and try again. There are similar pre-edited versions in "src/STEPS" for each step of this tutorial.

Since this is our first attempt to accelarate this code onto the GPU, lets just profile the sanity run using the following command:

```
  % pgprof --cpu-profiling off ./bin/benchADM_acc1 BenchADM_20l.par
```

The timing data produced by the PGI profiler now shows information about the operations executed on the GPU. Both the kernel time (50ms) and data transfer times (444s down / 419s up) is included.

```
==23973== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.69%  445.31ms    199500  2.2320us  1.9890us  273.48us  [CUDA memcpy HtoD]
 45.82%  419.04ms    195700  2.1410us  1.8880us  186.04us  [CUDA memcpy DtoH]
  5.04%  46.073ms       100  460.73us  457.03us  464.10us
 bench_staggeredleapfrog2_375_gpu
  0.45%  4.0812ms       100  40.811us  39.456us  42.112us
 bench_staggeredleapfrog2_342_gpu
```

Before we continue, let us fix the high number of data transfers that time it takes to transfer this data between the CPU and GPU. If these numbers scale and we continue to see ~20 times as much time spent moving data between the host and device we are not going to see any performance gain by accelerating this loop on the GPU. The next chapter will begin by adding directives to move data explictly.

The GPU accelerator card needs to be initialized before use. This is done by default the first time an accelerator region is entered. Alternatively, you can add a call to the function "acc_init()" to initialize the device. We have added the following code to BenchADM's initialization routine in the file "src/Cactus/InitialiseCactus_acc.c":

```
#ifdef _OPENACC
#  include "accel.h"
#endif
....
#ifdef _OPENACC
acc_init(acc_device_nvidia);
#endif
```

The _OPENACC macro name is defined when OpenACC directives are enabled. This is an example of a portable method for factoring GPU initialization out of accelerator regions, to ensure the timings for the regions do not include initialization overhead.

Chapter 4.
IMPROVING DATA TRANSFER SPEED

In order to minimize data transfers between the host memory and GPU accelerator device memory, the PGI Accelerator compilers compute the minimal amount of data that must be copied. The following -Minfo message shows the compiler has determined that only a section of the array "adm_kzz_stag_p_p" must be copied to the GPU.

```
   Generating copyin(...,adm_kzz_stag_p_p(2:nx-2+1,2:ny-2+1,2:nz-2+1),...)
```

However, this array section is not contiguous; because the data transfer is done via DMA (direct memory access) operation, the transfer will send each contiguous subarray separately. Often, it's more efficient to copy an entire array. Even though it involves transferring more data, it can be done in one large DMA operation.

Copy "src/StaggeredLeapfrog2_acc1.F" to "src/StaggerredLeapfrog2_acc2.F" and bring up the latter in a text editor. Modify the accelerator region directive by inserting data copy clauses that will cause whole arrays to be copied, rather than array sections.

Hint: look at the -Minfo messages output by the compiler to determine which arrays are copied in/out, and insert directives that look as follows to explicitly specify the whole-array copies:

```
!$ACC KERNELS
!$ACC& COPYOUT (
!$ACC& ADM_kxx_stag(1:nx,1:ny,1:nz),
!$ACC& ADM_kxy_stag(1:nx,1:ny,1:nz),
...
```

Alternatively, just copy the file "src/STEPS/StaggeredLeapfrog2_acc2.F" to the "src" directory.

Build and profile the revised code using the following commands:

```
  % make OPT="-fast -ta=tesla -Minfo=accel" build_acc2
  % pgprof ./bin/benchADM_acc2 BenchADM_200l.par

...
==24179== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 60.82%  23.6223s       100  236.22ms  235.14ms  236.50ms
 bench_staggeredleapfrog2_406_gpu
 29.77%  11.5609s      9400  1.2299ms  2.1120us  1.6457ms  [CUDA memcpy HtoD]
  9.41%  3.65443s      2400  1.5227ms  1.2963ms  1.6557ms  [CUDA memcpy DtoH]
...
```

The revised code with directives helps the compiler optimize data movement and reduces the host-to-GPU data transfer time significantly. Moving the data between the CPU and GPU can now be completed in about 12s.

Chapter 5.
WHAT MORE CAN BE DONE?

In the final example file "src/StaggeredLeapfrog2_acc4.F", we can improve benchADM by expanding the accelerator region to include the entire function. Although the first three loops have a low intensity and would not be good candidates for acceleration by themselves, most of the data transfer cost has already been paid so it's beneficial to offload them to the GPU.

Try copying "src/StaggeredLeapfrog2_acc3.F" to "src/StaggeredLeapfrog2_acc4.F" and bring up the latter in your editor. Move the !$ACC KERNELS directive to encompass all four loops. Which arrays must be added to the COPYIN or COPYOUT clauses? Which arrays can be allocated locally using the LOCAL clause? Feel free to take a look at "src/STEPS/StaggeredLeapfrog2_acc4.F" for hints.

Note that the LOCAL clause on several of the arrays used only within the accelerator region eliminates the need to copy them at all, but some arrays must be added as inputs or outputs to the first three loops in the routine. With these changes, the data transfer and aggregate kernel execution times increase slightly, but overall wall-clock time drops.

```
  % make OPT="-fast -ta=tesla -Minfo=accel" build_acc4
  % pgprof ./bin/benchADM_acc4 BenchADM_200.par
  ...
==24586== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.11%  26.7256s       100  267.26ms  265.94ms  268.89ms
 bench_staggeredleapfrog2_410_gpu
 35.09%  19.4933s     15900  1.2260ms  2.1440us  1.6592ms  [CUDA memcpy HtoD]
 13.16%  7.31232s      4800  1.5234ms  1.2969ms  1.6274ms  [CUDA memcpy DtoH]
  3.60%  1.99779s       100  19.978ms  19.917ms  20.043ms
 bench_staggeredleapfrog2_374_gpu
  0.02%  11.469ms       100  114.69us  113.54us  120.67us
 bench_staggeredleapfrog2_334_gpu
  0.02%  11.439ms       100  114.39us  113.63us  115.20us
 bench_staggeredleapfrog2_353_gpu
```

The following table summarizes the performance at each step of this tutorial.

Table 1 Performance Results

Step	Total Execution (s)	Kernel(s) Execution (s)	Data Movement (s)
CPU Only (benchADM_base)	109	108	0
Explict Data Tranfers (benchADM_acc2)	85	23	16