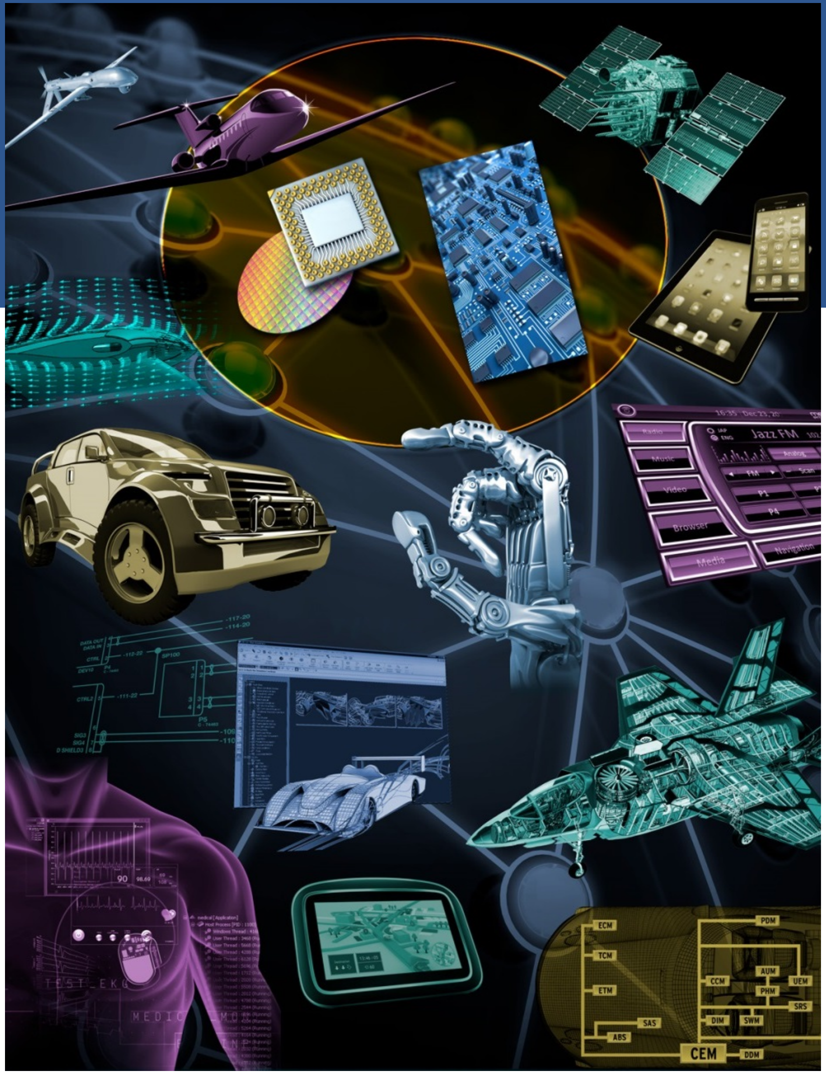
Improving GCC's Performance on OpenACC Applications



March 27, 2018

Randy Allen

Mentor®
A Siemens Business
www.mentor.com/embedded

Overview

- Compiler optimization is guaranteed employment
  - Compilers are complex illustrations of the phrase “NP complete”
  - Tuning an optimizer takes much time and many lines of sample code
- In “compiler” terms, GCC’s support of OpenACC is relatively young
- The omnipresent questions:
  - How well tuned is code generation
  - What are key improvements we can make
- Inquiring minds wanted to know

The Process

- Take a real world application on which GCC performs poorly relative to PGI
- Profile and analyze it to determine the slowdowns
- Effect optimizations in GCC to address the slowdowns
- See how much work is required to get to PGI level performance
- See how general the changes are

The Application: LSDalton

- Large Scaled Coupled-Cluster Calculations of Supramolecular Wires
- Quantum chemistry code targeting
  - Enzyme-catalyzed chemical reactions
  - Carbon nanotubes and graphene
  - Preferred crystal form of organic molecules
- Regular releases, tested with gfortran, ifort, pgf90
- Widely used

Preparation

- LSDalton made extensive use of cuBLAS library, particularly for matrix-vector and matrix-matrix multiplication
  - This defeated the purpose, since the core computation not compiler generated
  - Replaced those calls with netlib source, annotated with OpenACC directives

- LSDalton used a highly-optimized PGI host BLAS library that was incompatible GCC due to OpenMP
  - Given it was host, it provided a small performance boost
  - Replaced with netlib source compiled with respective compilers to normalize the comparisons

- Some minor source changes to work around problems in each compiler

- Baseline execution times on the sample data set:
  - GCC: 216 seconds
  - PGI: 104 seconds

Hardware and Options

- Hardware
  - NVIDIA GeForce GTX 1080 with 8113 MiB RAM
  - Intel® Xeon® CPU ES-2640 v4 @3.10 GHz with 32 GB of RAM
  - CUDA 8.0.44
- PGI Compiler: 17.9-0 64-bit target on x86-64 Linux
  - "-ta=host,tesla:cc60 -lnvidia-fatbinaryloader –lcuda"
- GCC: internal version
  - "-fopenacc –lcuda"

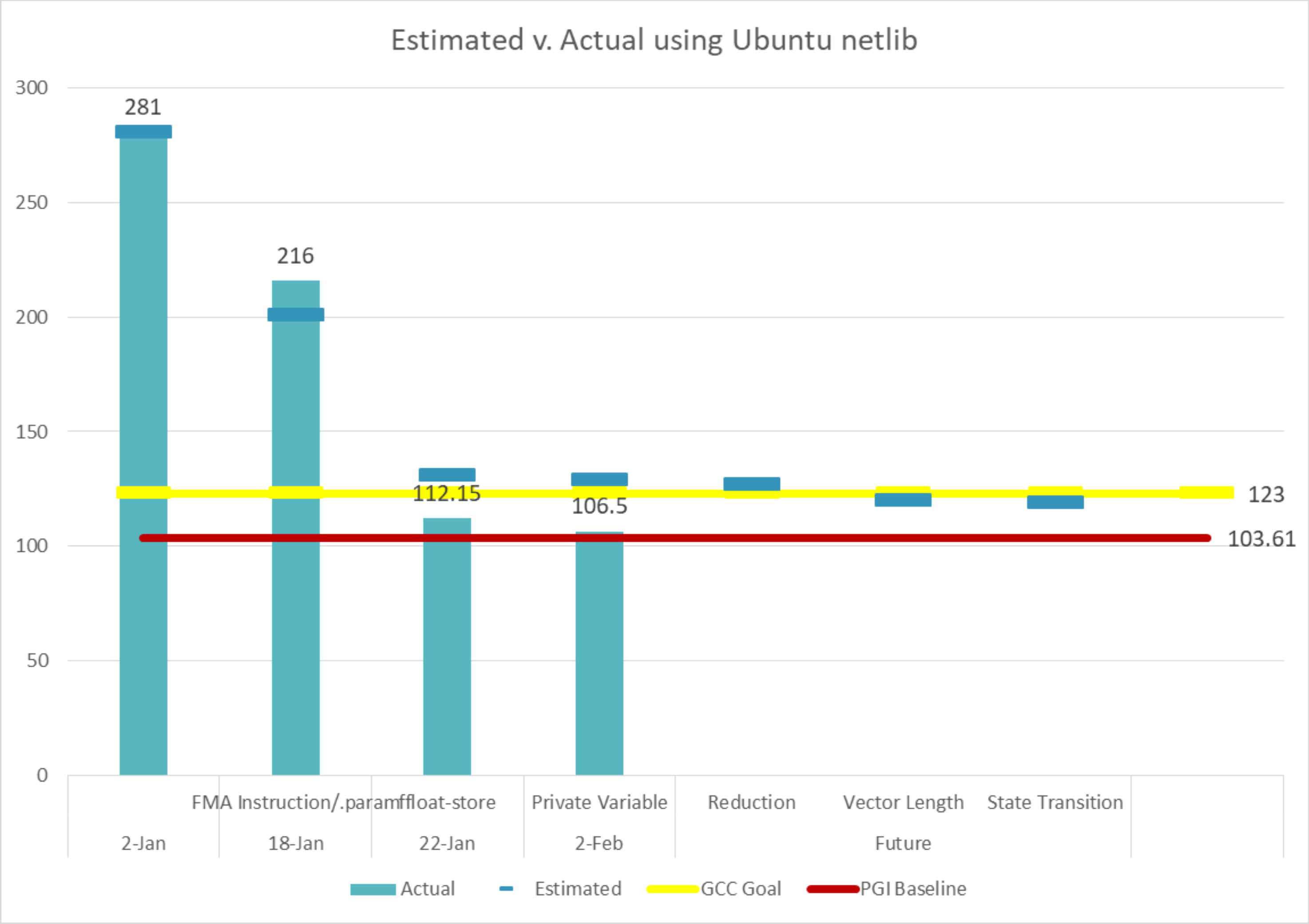
Analysis

- LSDalton enabled the compiler option -ffloat-store
  - Causes every floating value to be stored to memory when computed and loaded on every use
  - Obviously bad for performance
  - Presumably invoked to work around a compiler problem
- Startup code for parallel regions
  - PGI was significantly faster than GCC
  - PGI launched by dispatching parameters as part of the startup
  - GCC launched by dispatching a pointer to global memory location holding parameters
- Treatment of reduction variables
  - Fortran parameters used to accumulate reductions were not well optimized

Improvements and Results

- LSDalton enabled the compiler option -ffloat-store
  - Mentor tracked down and fixed the compiler problem
    - It actually affected only one loop nest in one routine
    - If OpenACC disabled on this one loop nest alone, most of performance gain would have been kept
  - With removal of –ffloat-store, execution time decreased by ~25% (70 seconds)
- Startup code for parallel regions
  - Mentor rewrote initiation code to dispatch parameters directly rather than through global memory
  - The speedup was significant, indicating that lots of threads accessing the same memory, even when no writes are involved, is slow
  - Speed up was ~30 seconds, ~20%
- Treatment of reduction variables
  - Mentor optimized out the reduction variable, giving a small (~6%) speedup

LSDalton performance



Impact on Other Applications

- Overall improvement on Mentor performance regression suite: 3%
- Improvement on Cloverleaf: 10%
- Improvement on subset of SPEC: 6%

Lessons Learned

- The writing of compilers is a noble profession
- Be conservative in using compiler options that disable performance
- Little inefficiencies add up when multiplied by 1000
- For this application, PGI and GCC are now roughly performance equivalent
- For both compilers, the difference between generated code and hand-coded BLAS is non-trivial, but not large either
  - Fastest PGI version (both hand-coded BLAS) 92 seconds – roughly 10% faster than our final with source
  - Means that code can be written in source that will run well across multiple platforms
- Improved version of gcc available from randy_allen@mentor.com.