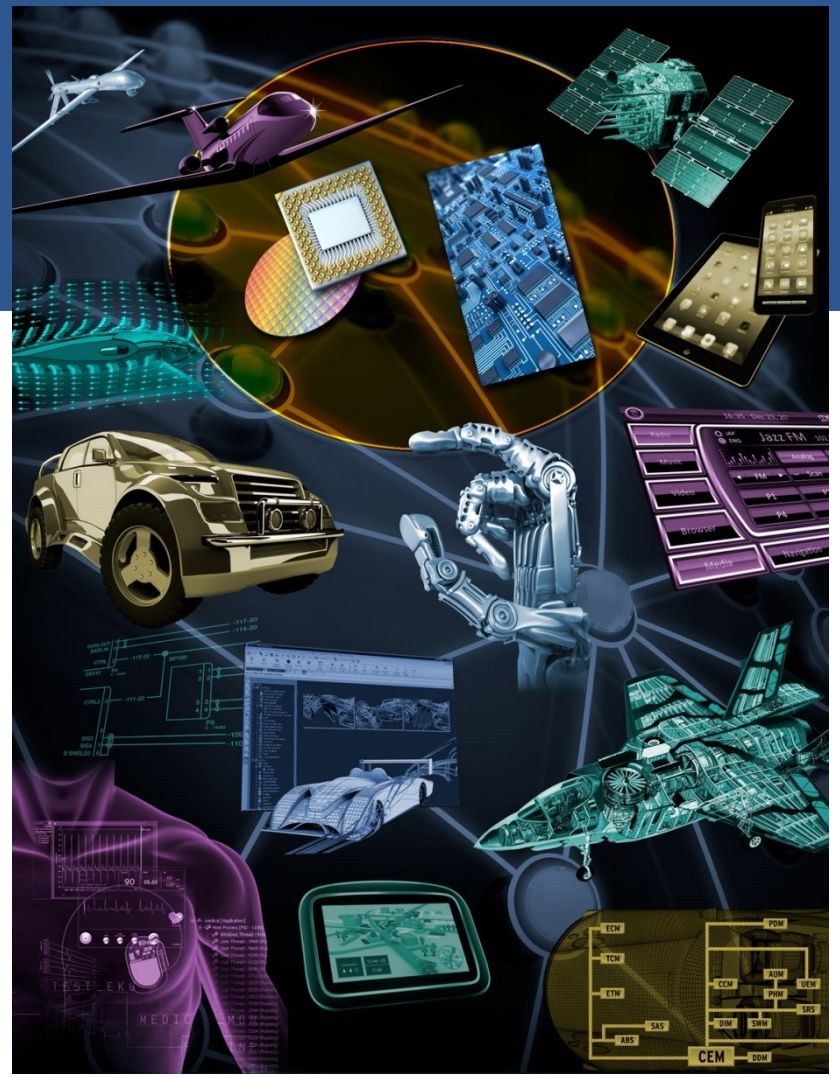


Speeding up Programs with OpenACC in GCC

... using the compute power of GPUs and other accelerators

2019-02-03

Thomas Schwinge, <thomas@codesourcery.com>

Sourcery Services, Embedded Platform Solutions

Mentor, a Siemens Business

<https://fosdem.org/2019/schedule/event/openacc>

HPC, Big Data and Data Science devroom

FOSDEM 2019

Mentor®
A Siemens Business

<https://www.mentor.com/embedded-software/>

Introduction, Agenda

- Proven in production use for decades, GCC (the GNU Compiler Collection) offers C, C++, Fortran, and other compilers for a multitude of target systems.
- Over the last few years, we – formerly known as "CodeSourcery", now a group in "Mentor, a Siemens Business" – added support for the directive-based OpenACC programming model.
- Requiring only few changes to your existing source code, OpenACC allows for easy parallelization and code offloading to GPUs and other accelerators.
- We will present a short introduction of GCC and OpenACC, implementation status, examples and performance results.

GCC (GNU Compiler Collection)

- <https://gcc.gnu.org/>
- <https://en.wikipedia.org/wiki/GNU_Compiler_Collection>

“The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and the standard compiler for most Unix-like operating systems. [...]”

GCC (GNU Compiler Collection)

- Production-quality support:
  - C
  - C++
  - Fortran
- GCC is relevant in HPC

See, for example, “results taken from the XALT database at the National Institute of Computational Sciences (NICS) covering a period from October 2014 through June 2015. This data is from a Cray XC30 supercomputer called Darter” as shown in Figure 2 of "Community Use of XALT in Its First Year in Production," R. Budiardja, M. Fahey, R. McLay, P. M. Don, B. Hadri, and D. James, In Proceedings of the Second International Workshop on HPC User Support Tools, HUST '15, Nov. 2015. doi.acm.org/10.1145/2834996.2835000.

In his WACCPD 2018 keynote, Jack Wells (ORNL) had some more recent data.

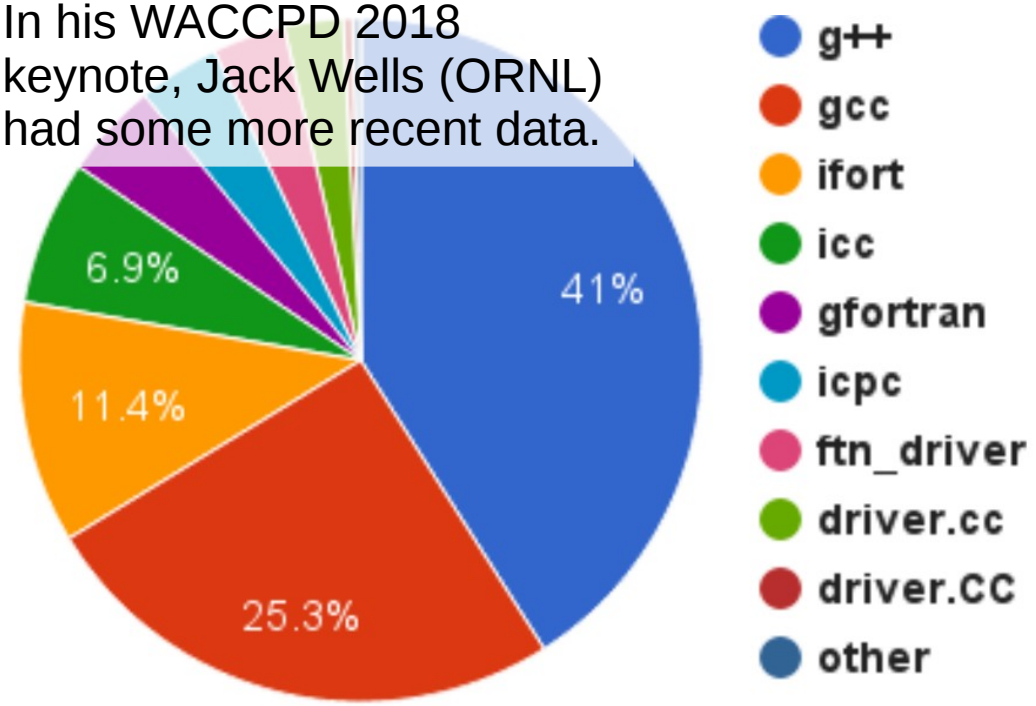


Figure 2: Usage of link program from 105,231 number of links. The GNU compilers (g++, gcc, gfortran) make up about 71% instances in aggregate, followed by the Intel compilers (ifort, icc, icpc) at 22%, and the Cray Compilers (ftn_driver, driver.cc, driver.CC) at 6%.

GPU architecture

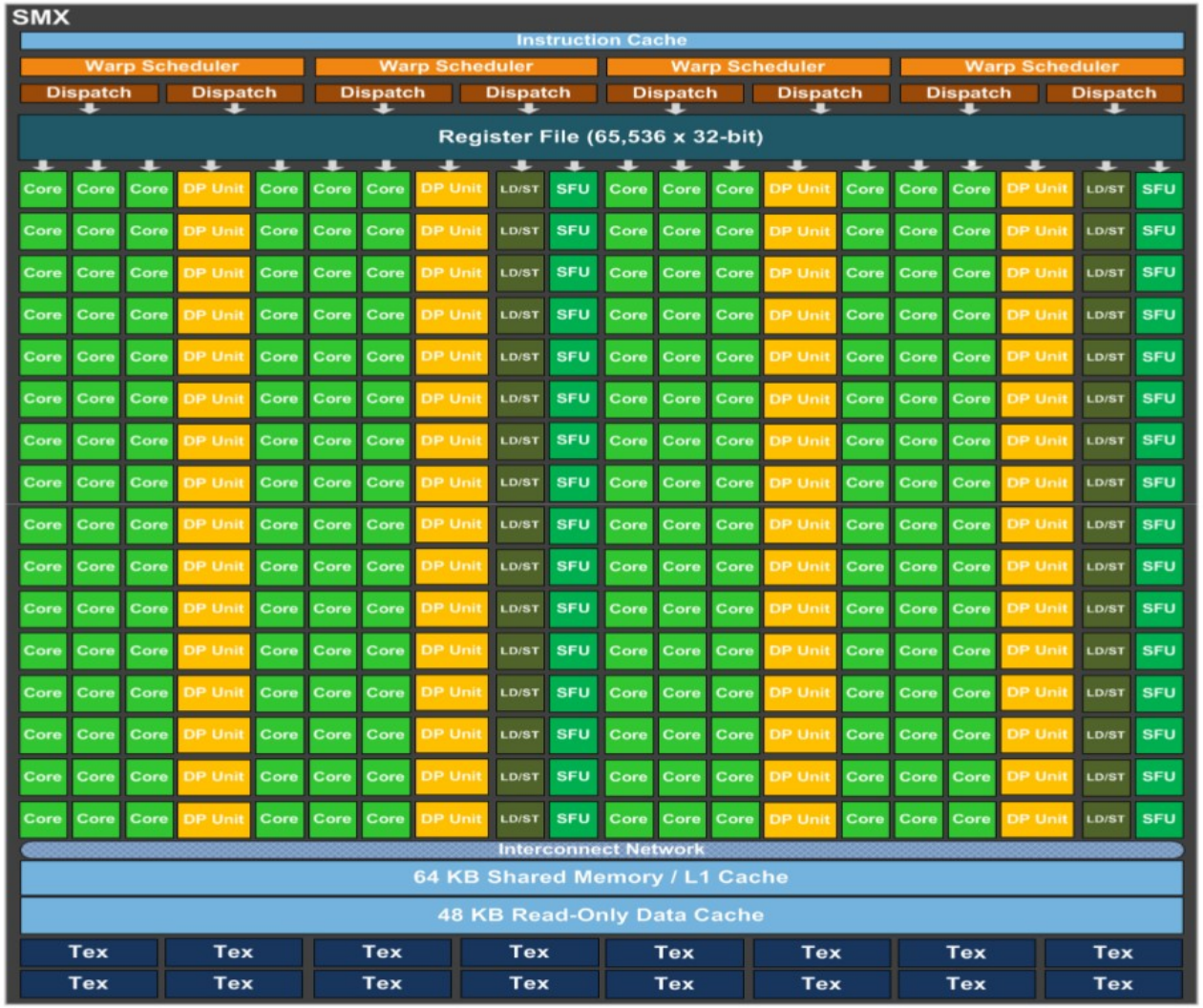
Example: Nvidia GPU Kepler K20 (GK110)

Nvidia GPU Kepler K20 (GK110), which is a few years old, but general concepts are still the same

Kepler GK110 Full chip block diagram

NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

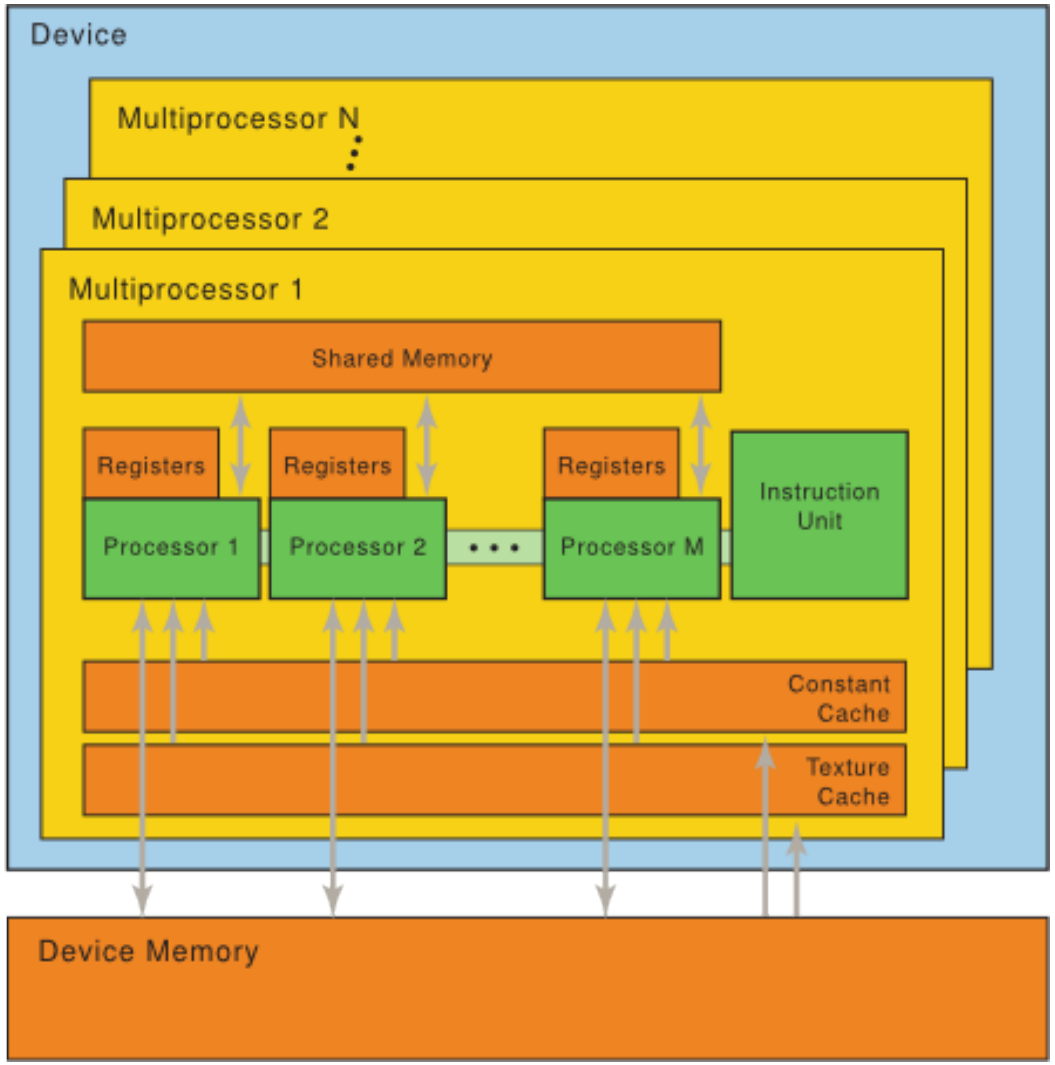
GPU architecture
Example: Nvidia GPU Kepler K20 (GK110)



SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

GPU architecture
abstractly



<http://www.martinpeniak.com/images/cuda_fig2.png>

OpenACC

- Base on your existing source code, few modifications required
- Set of simple directives
  - Similar to OpenMP
- Easy for the user to mark up:
  - Memory regions for data copy
  - Parallel/vector loops
  - Reduction operations
  - Etc.
- Provide hints to the compiler to better use available parallelism
- Abstract enough to apply to a wide range of parallel architectures

<https://www.openacc.org/>

Example: matrix multiplication

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    float t = 0;
    for (k = 0; k < n; k++)
      t += a[i][k] * b[k][j];
    c[i][j] = t;
}}
```

Example: matrix multiplication

OpenACC parallel construct

```
#pragma acc parallel \ // spawn parallel execution
  copyin(a[0:n][0:n], b[0:n][0:n]) copyout(c[0:n][0:n]) // data copy
{
#pragma acc loop gang // multiprocessors
  for (i = 0; i < n; i++) {
#pragma acc loop worker // groups of PTX “warps”
    for (j = 0; j < n; j++) {
      float t = 0;
#pragma acc loop vector \ // “threads” in 32-size PTX “warps”
  reduction(+: t) // “t” needs a “+” reduction operation
      for (k = 0; k < n; k++)
        t += a[i][k] * b[k][j];
      c[i][j] = t;
}}}
```

Example: matrix multiplication
OpenACC kernels construct

```
#pragma acc kernels \ // spawn parallel execution
  copyin(a[0:n][0:n], b[0:n][0:n]) copyout(c[0:n][0:n]) // data copy
{
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      float t = 0;
      for (k = 0; k < n; k++)
        t += a[i][k] * b[k][j];
      c[i][j] = t;
}}}
```

(But with GCC, the OpenACC kernels construct does not yet deliver the expected performance.)

OpenACC support in GCC upstream

- Most of OpenACC 2.0a, 2.5; 2.6 in development branch
- OpenACC 2.7 not yet
- Code offloading to:
  - Nvidia GPUs (nvptx): upstream
  - AMD GPUs (GCN): integrating upstream (GCC 10, 2020?)
  - Any others could be done, too, including multi-threaded CPU
- Host system
  - Any with suitable drivers to talk to the accelerator
  - We’re testing on x86_64 and ppc64le GNU/Linux
- In you’re interested to help or fund development, talk to us :-)

Real-world application example: lsdalton

- Program suite for “calculations of molecular properties”
- Mixed C/some C++/a lot of Fortran source code
- OpenACC directives in legacy Fortran code
  - “The history of Dalton starts in fall of 1983 [...]”
- Project: use GCC/OpenACC with Nvidia GPU for acceleration, then tune performance
  - Compare to PGI compiler

lsdalton: SLOCCount

```
SLOC    Directory       SLOC-by-Language (Sorted)
519483  src             f90=349006,fortran=64312,ansic=63756,sh=41296,cpp=1113
343934  external        ansic=149330,f90=89252,cpp=83139,python=11996,perl=9705,sh=475,pascal=37
Totals grouped by language (dominant language first):
F90:         438258 (50.76%)    ansic:       213086 (24.68%)    cpp:          84252 (9.76%)   Fortran:      64312 (7.45%)
sh:           41771 (4.84%)     python:       11996 (1.39%)     Perl:          9705 (1.12%)   pascal:          37 (0.00%)

Total Physical Source Lines of Code (SLOC)                = 863,417 [for comparison: GCC w/o testsuites: ~3,500,000]
Development Effort Estimate, Person-Years (Person-Months) = 242.14 (2,905.65) (Basic COCOMO model, Person-Months = 2.4 *
(KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 4.31 (51.76) (Basic COCOMO model, Months = 2.5 * (person-
months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 56.14
Total Estimated Cost to Develop                           = $ 32,709,451 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to redistribute it under certain conditions as specified by
the GNU GPL license; see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

lsdalton: build

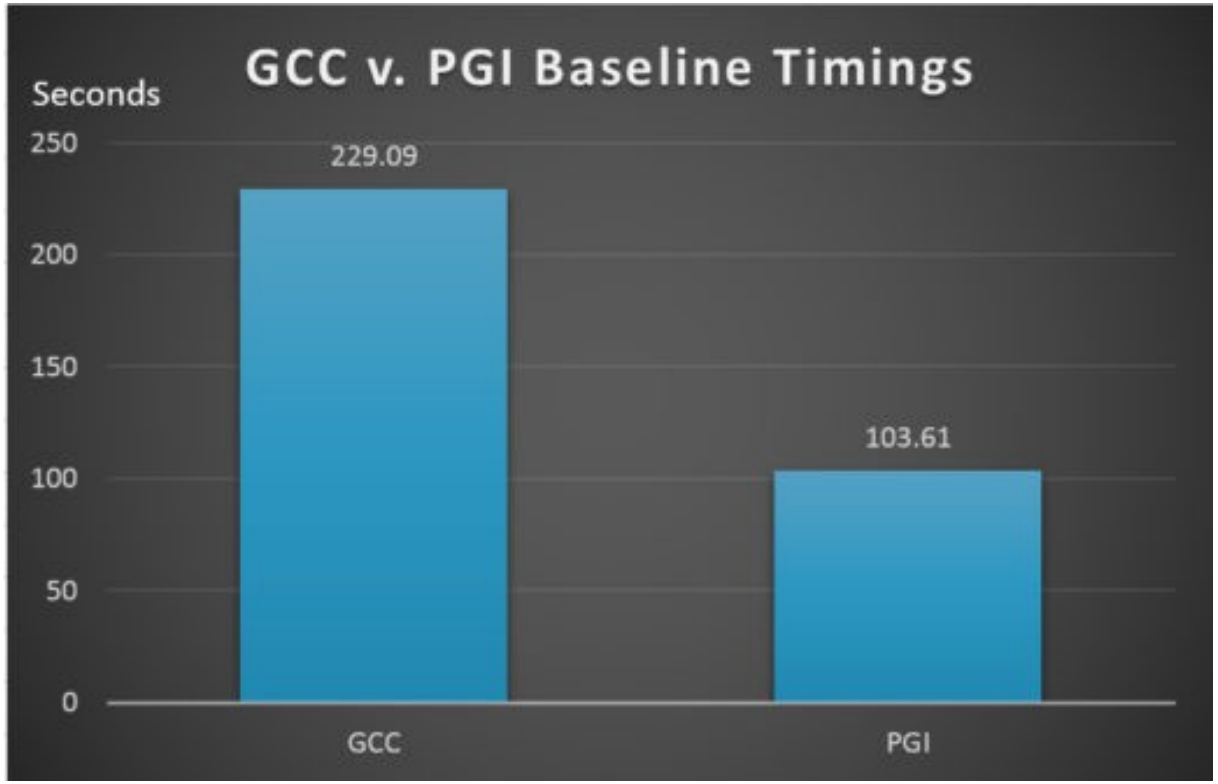
- This includes seven external Git submodules
- Non-trivial build system
  - Works for the PGI compiler
  - Also for “stock” GCC (no acceleration)
  - But not for GCC with OpenACC/nvptx offloading
    - Figure out how to use the desired GCC for all components, with -fopenacc enabled etc.

lsdalton: a few code changes required

- Getting it to run with GCC with OpenACC/nvptx offloading
  - Apples-to-apples comparison
    - Replace OpenACC kernels with OpenACC parallel constructs
    - Do not link against optimized Nvidia cuBLAS/PGI BLAS library (… for PGI compiler only)
      - Replaced by Netlib Fortran BLAS functions, annotated with OpenACC directives
  - Replace a few “questionable” uses of OpenACC directives with ones supported by GCC

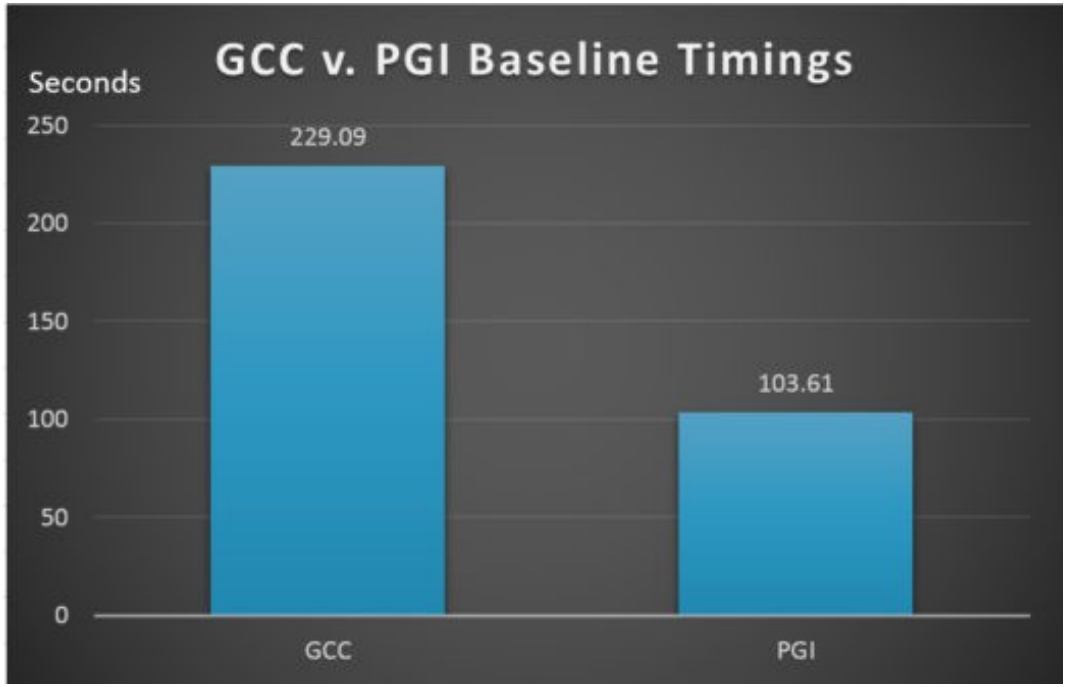
lsdalton: performance optimization

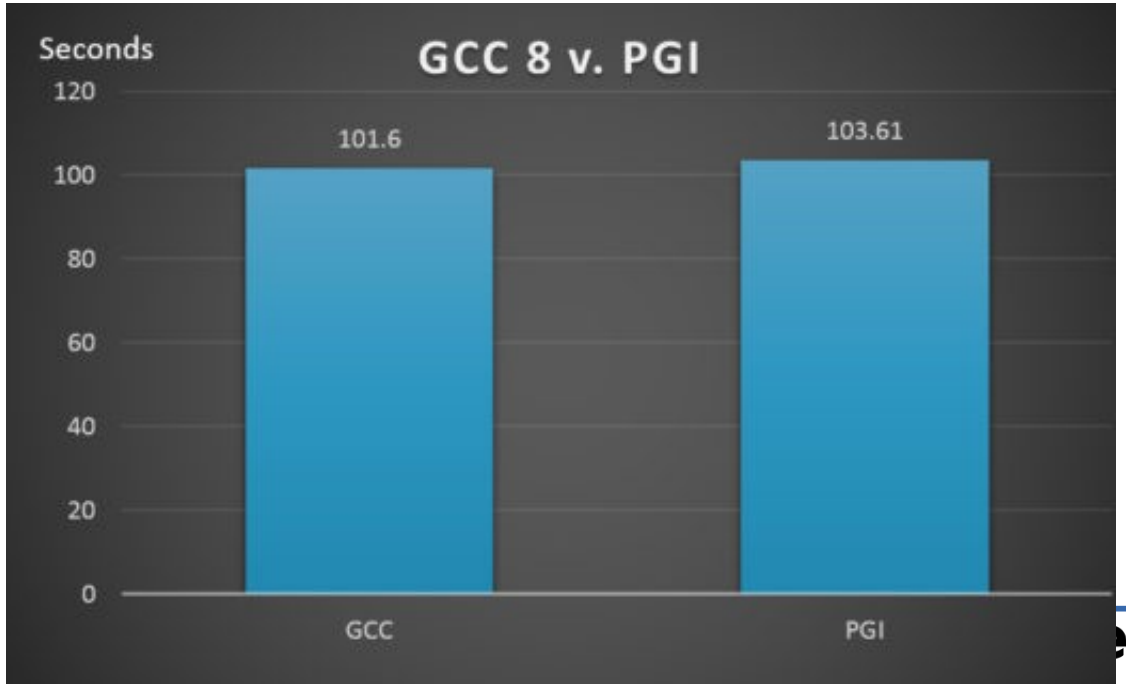
- Several cycles of:
  - Profile
  - Analyze
  - Tune
    - Teach GCC how to do “the right things”
      - Did not alter lsdalton sources anymore
    - Infrequently report issues to Nvidia
      - PTX to hardware JIT compilation



lsdalton: performance optimization

- Eventually achieved great performance testing results for the specific lsdalton configuration tested in this project
  - https://blogs.mentor.com/embedded/blog/2018/06/06/evaluating-the-performance-of-openacc-in-gcc/
  - Optimizing GCC's code generation also benefited other benchmarks





Real-world application example: n-body

- Set of n individual bodies, each with initial position and velocity
- Distance-dependent force between each pair
- Problem is to calculate the trajectory of each body

Stock Ubuntu 18.04: GCC 8 with OpenACC/nvptx offloading support

```
$ sudo aptitude install gcc-8-offload-nvptx
The following NEW packages will be installed:
  gcc-8-offload-nvptx  libgomp-plugin-nvptx1{a}  nvptx-tools{a}
0 packages upgraded, 3 newly installed, 0 to remove and 239 not upgraded.
Need to get 8057 kB of archives. After unpacking 54.4 MB will be used.
Do you want to continue? [Y/n/?] y
Get: 1 [...] nvptx-tools amd64 0.20180301-1 [27.8 kB]
Get: 2 [...] libgomp-plugin-nvptx1 amd64 8.2.0-1ubuntu2~18.04 [13.4 kB]
Get: 3 [...] gcc-8-offload-nvptx amd64 8.2.0-1ubuntu2~18.04 [8016 kB]
Fetched 8057 kB in 16s (501 kB/s)
[...]
Setting up nvptx-tools (0.20180301-1) ...
[...]
Setting up libgomp-plugin-nvptx1:amd64 (8.2.0-1ubuntu2~18.04) ...
Setting up gcc-8-offload-nvptx (8.2.0-1ubuntu2~18.04) ...
[...]
```

Beware: these distribution packages are many months behind our current (public!) development branch: in terms of bug fixes, features, performance optimizations.

(Packaging work not done by us, but by Matthias "doko" Klose who is maintainer of the Debian and Ubuntu GCC packages.)

Real-world application example: n-body

- Live demo
  - … on this 2013 laptop, with a powerful CPU, but mediocre GPU

Thank you!