Domain-Specific Acceleration and Pragmatic Auto-Parallelization of Legacy Scientific Code in FORTRAN 77 using Source-to-Source Compilation

Wim Vanderbauwhede, Gavin Davidson
School of Computing Science, University of Glasgow, UK

Overview

- A lot of scientic code is “legacy” code written in FORTRAN 77
- Massively parallel accelerators such as GPGPUs, manycores and FPGAs are powerful and affordable tools for scientists to speed up simulations of complex systems.
- Porting code to such devices requires a detailed understanding of heterogeneous programming tools and effective strategies for parallelization.
- We present our work on source to source compilation to
  - transform sequential FORTRAN 77 legacy code
  - into OpenCL-accelerated programs
  - with auto-parallelized kernels
  - without need for directives or extra information from the user.

Accelerator Programming

- **OpenCL**: open standard for heterogeneous computing
- Main advantage of OpenCL over proprietary solutions such as e.g. CUDA (to which it is very similar): it is supported by a wide range of devices, including multicore CPUs, manycores, GPGPUs and FPGAs.
- Programmer writes
  - one or more kernels that are run directly by the accelerator and
  - a host program that is run on the system's main CPU.
- The host program handles memory transfers to the device and initializing computations
- The kernels perform the parallel computations.

OpenCL

- From the programmer perspective, OpenCL is
  - very flexible
  - quite low level
  - requires a lot of complex code to be written
- Considerable barrier for adoption
- No official Fortran support for OpenCL:
  - Host API is C/C++
  - Kernel language is based on a subset of C99.

Our Work

- A compiler to
  - transform FORTRAN 77 code into accelerator-ready Fortran 95 `https://github.com/wimvanderbauwhede/RefactorF4ACC`
  - transform the Fortran program into a Fortran-OpenCL program with automatic parallelisation `https://github.com/wimvanderbauwhede/AutoParallel-Fortran`
  - compile Fortran 95 kernels into the OpenCL C language `https://github.com/wimvanderbauwhede/RefactorF4ACC`
- A Fortran API for OpenCL `https://github.com/wimvanderbauwhede/OpenCLIntegration`

Source-to-source compilation

- A conventional compiler consumes source code and produces binaries.
- A source-to-source compiler produces a transformed version of the original source.
  - e.g. refactoring, parallelization or translation to a different language.
- The advantage of this approach is that the resulting code can be modified by the programmer if desired and compiled with a compiler of choice.
- We combine source-to-source compilation with whole-program analysis:
  - most compilers only consider a single source file as the context.
  - our compiler analyses the complete source of a program and establishes the relationships between all entities

Maintainable and Extensible

In a first step the compiler transforms the FORTRAN 77 program into modern, maintainable, extensible and accelerator-ready Fortran 95 code

- No IMPLICIT typing: our compiler creates explicit type declarations for all variables.
- Fully explicit INTENT: our compiler infers the INTENT for all subroutine and function arguments.
- Modules with export lists: our compiler converts all non-program code units into modules USEd with an explicit export (ONLY) declaration.

Accelerator-ready

- Most current accelerators have a separate memory space from the host memory.
- It is therefore crucial to separate the memory spaces of the kernel and the host programs.
- Our compiler transforms global variables (e.g COMMON block variables) into subroutine arguments across the complete call tree of the program.

OpenCL-accelerated Code

1. *Allow any subroutine in the code to be offloaded* to an accelerator using whole-source refactoring.
2. *Minimization of the data transfer* between the host and the accelerator by eliminating redundant transfers.
   Includes determining which transfers need to be made only once in the run of the program.
3. *Pragmatic auto-parallelization* of the code to be offloaded to the accelerator by identification of parallelizable maps and folds (reductions).
   Includes partial parallelization of loops and fusion of loops with non-identical bounds.

Code Transformation Validation

- We have validated the code transformation performance of the compiler on the *NIST FORTRAN78* test suite (72,4K loc) as well as on following real-world codes:
  - *Large Eddy Simulator for Urban Flows* (LES), a high-resolution turbulent flow model (1.4K loc)
  - Shallow water component of the *Gmodel ocean model* (1.5K loc)
  - *Flexpart-WRF*, a particle dispersion simulator (13.8K loc)
  - *Linear Baroclinic Model*, an atmospheric climate model (39.3K loc)
- All these codes are successfully refactored:
  - no errors on code generation
  - no errors on compilation
  - transformed code performance and results are identical to original code

EPSRC
Engineering and Physical Sciences Research Council

Automatic Parallelization Validation

- Test case: Large Eddy Simulator for Urban Flows (LES)
- Developed by the Disaster Prevention Research Institute of Kyoto University and the Japan Atomic Energy Agency:
  - Generates turbulent flows by using mesoscale meteorological simulations.
  - Explicitly represents the urban surface geometry.
  - Used to conduct building-resolving large-eddy simulations of boundary-layer flows over urban areas under realistic meteorological conditions.
  - Essentially solves the Poisson equation for the pressure, using Successive Over-Relaxation
  - Written in Fortran-77, single-threaded, about a thousand lines of code.

LES Code Structure – Functional

- LES structure is sequential in each time step:

velnw: Update velocity for current time step
bondv1: Calculate boundary conditions (initial wind profile, inflow, outflow)
velfg: Calculate the body force
feedbf: Calculation of building effects (Goldstein damping model)
les: Calculation of viscosity terms (Smagorinsky model)
adam: Adams-Bashforth time integration
press: Solving of Poisson equation using SOR

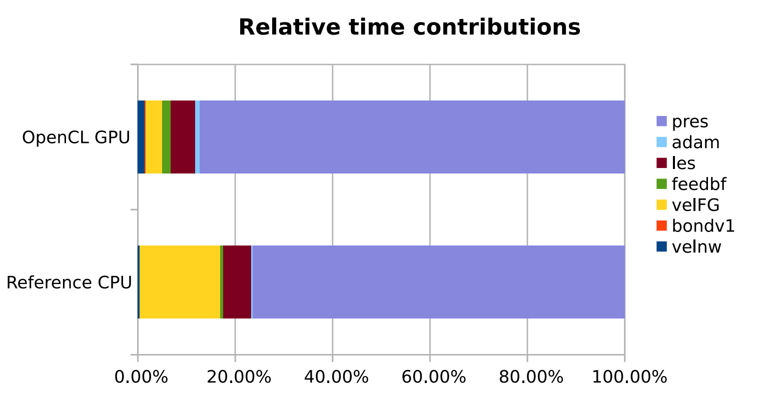
```
      program main
c-----------------------------------------------------------------------
      include 'common.sn' ! Parameter & variable setting, declaration

      call set
      call grid

      call timdata
      call init
      call ifdata

c--main loop
      do 1000 n = n0,nmax
        time = float(n-1)*dt
c-------calculate turbulent flow--------c
        call velnw
        call bondv1
        call velFG

        if(ifbf.eq.1) then
        call feedbf            !calculate building effect
        end if

        call les
        call adam
        call press
c-------data output --------------------c
        call timseris
        call aveflow
        if(ianime.eq.1) then
        call anime
        endif
c
        if(n.eq.nmax) then
        stop
        end if
 1000 continue
c
      end program
```

OpenCL LES Evaluation

- Platform
  - Host platform: Intel Xeon CPU E5-2620 0 @ 2.00GHz, 6-core CPU with hyperthreading (12 threads), AVX, 32GB RAM, cache 15MB
  - GPU platform: NVIDIA GeForce GTX TITAN, 980 MHz, 15 compute units, 16GB RAM, OpenCL 1.1 CUDA 6.5.14
- Original LES code on CPU (reference)
  - Compiler: gfortran 4.8.2, flags for auto-vectorization and auto-parallelization:
    - `-Ofast -floop-parallelize-all -ftree-parallelize-loops=12 -fopenmp -pthread`
    - Auto-parallelization provides only 4% speed-up because the most time-consuming loops are not parallelised
- Our compiler:
  - Auto-parallelizes all 34 loop nests in the code base
  - Produces a complete OpenCL-enabled code base.
  - Runs on GPU and CPU
  - 20x speed-up compared to the original code on CPU

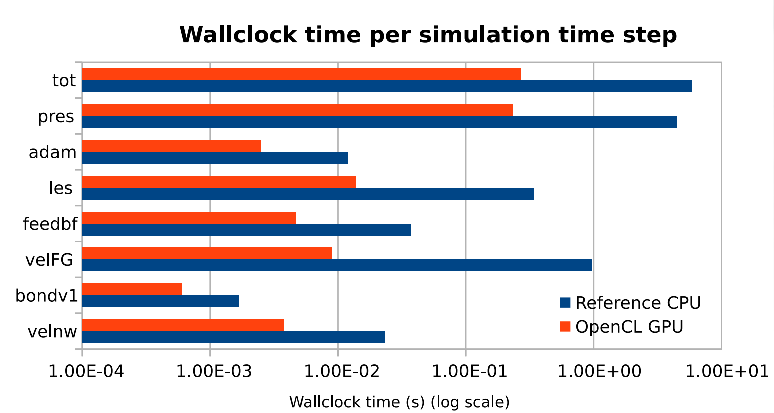
Results

Breakdown per subroutine
(domain size 300x300x90, SOR 50 iterations)



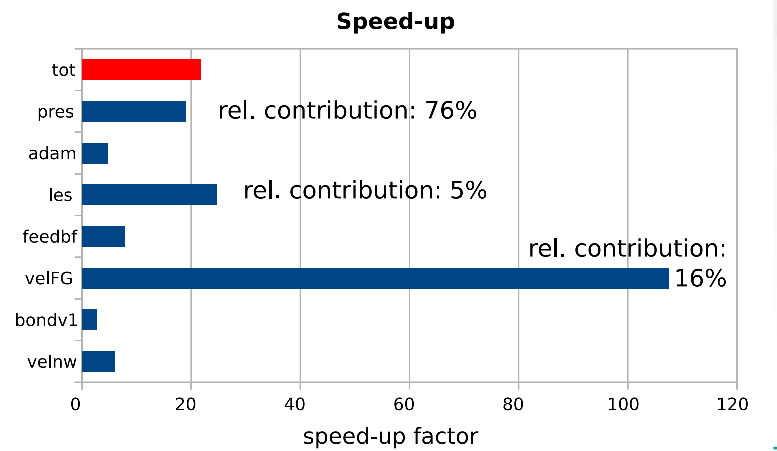
Results

Wallclock time comparison



Results

Speed-up



Engineering and Physical Sciences Research Council

Conclusion

- We have developed a proof of concept compiler for
  - OpenCL acceleration and pragmatic auto-parallelization
  - of domain-specific legacy FORTRAN 77 scientific code
  - using whole-program analysis and source-to-source compilation.
- Future work
  - improving the compiler to extract more parallelism from the original code and improve the performance
  - FPGA back-end
  - CUDA and OpenMP back-ends

The authors acknowledge the support of the EPSRC (EP/L00058X/1).

Thank You!

Engineering and Physical Sciences Research Council