Performance Analysis of the PSyKAI Approach for a NEMO-based Benchmark

Mike Ashworth, Rupert Ford and Andrew Porter

Scientific Computing Department
and
STFC Hartree Centre
STFC Daresbury Laboratory
United Kingdom
Contents

● Motivation (the GungHo Project)
● The GOcean Project
● Separation of Concerns (PSyKAI)
● Application of PSyKAI to shallow-water models
● Performance Impact of PSyKAI
  ● Serial
  ● OpenMP
  ● GPU
● Summary
GungHo – What and Why?

• GungHo is a UK Met Office, NERC and STFC project aiming to research, design and develop a new dynamical core suitable for operational, global and regional, weather and climate simulation

• Computer architectures are in a state of flux with a variety of competing technologies
  – GungHo is developing code for a computer that does not yet exist
  – Many cores, accelerators (PCIe or socket), FPGAs…

• How can we produce maintainable, scientifically-correct code that will perform well on a range of future architectures?
• NERC-funded Technology Proof of Concept fund
• 03/2014 – 02/2015
• Collaboration between National Oceanography Centre, Liverpool (NOC) and STFC
• Investigate the feasibility of applying technology from the GungHo project to ocean modelling
• Extend the developing GungHo infrastructure to support finite difference on regular, lat-long grids
Separation of Concerns in GungHo

Diagram:
- Single model
  - PSy
  - Algorithms (Algs)
  - Kernel
- Driver
  - Read partition
  - Initialisation
  - Finalise
- Infrastructure:
  - E.g. halo_exchange
The Parallel System, Kernel, Algorithm (PSyKAl) Approach…

• Oceanographer writes the algorithm (top) and kernel (bottom) layers, following certain rules
  – no need to worry about relative indexing of various fields
  – no need to worry about parallelism (algorithm layer deals with logically global field quantities)

• A code-generation system (PSyclone) generates the PSy middle layer
  – glues the algorithm and kernels together
  – incorporates all code related to parallelism
Two shallow-water codes…

- We have applied the PSyKAI approach to two codes:
  - ‘Shallow’ originally written by Swarztrauber, NCAR
  - ‘NEMOLite2D’ 2D, free-surface part of NEMO extracted by NOC

- Both use Finite Difference on Arakawa C grid
- But there are important differences:
  - Boundary conditions (bi-periodic vs. forced/closed)
  - Relative indexing of variables on the grid

- Understanding and expressing these differences is essential for correct code generation
Two benchmarks...

• Have re-structured both Shallow and NEMOLite2D following PSyKAI separation of concerns
• Manually optimise Shallow while obeying PSyKAI rules
  – Serial benchmark
• Manually optimise NEMOLite2D and implement OpenMP parallelisation
  – Parallel benchmark
• These benchmarks have also been supplied to vendors (IBM, NVIDIA) for them to optimise for their hardware while obeying PSyKAI rules
Body of time-stepping loop consists of kernel calls:

```c
    call invoke(
        continuity(ssha_t, sshn_t, sshn_u, sshn_v, 
                    hu, hv, un, vn, rdt), 
                  &
        momentum_u(ua, un, vn, hu, hv, ht, 
                   ssha_u, sshn_t, sshn_u, sshn_v), 
                  &
        momentum_v(va, un, vn, hu, hv, ht, 
                   ssha_v, sshn_t, sshn_u, sshn_v), 
                  &
        bc_ssh(istp, ssha_t), 
                  &
        bc_solid_u(ua), 
                  &
        bc_solid_v(va), 
                  &
        bc_flather_u(ua, hu, sshn_u), 
        bc_flather_v(va, hv, sshn_v), 
                  &
        copy(un, ua), 
                  &
        copy(vn, va), 
                  &
        copy(sshn_t, ssha_t), 
                  &
        next_sshu(sshn_u, sshn_t), 
        next_sshv(sshn_v, sshn_t) 
    )
```
A kernel looks like:

```fortran
subroutine continuity_code(ji, jj, &
  ssha, sshn, sshn_u, sshn_v, &
  hu, hv, un, vn, rdt, e12t)

  implicit none
  integer, intent(in) :: ji, jj
  real(wp), intent(in) :: rdt
  real(wp), dimension(::,:), intent(in) :: e12t
  real(wp), dimension(::,:), intent(out) :: ssha
  real(wp), dimension(::,:), intent(in) :: sshn, sshn_u, sshn_v, &
                             hu, hv, un, vn

! Locals
  real(wp) :: rtmp1, rtmp2, rtmp3, rtmp4

  rtmp1 = (sshn_u(ji, jj) + hu(ji, jj)) * un(ji, jj)
  rtmp2 = (sshn_u(ji-1, jj) + hu(ji-1, jj)) * un(ji-1, jj)
  rtmp3 = (sshn_v(ji, jj) + hv(ji, jj)) * vn(ji, jj)
  rtmp4 = (sshn_v(ji, jj-1) + hv(ji, jj-1)) * vn(ji, jj-1)

  ssha(ji, jj) = sshn(ji, jj) + (rtmp2 - rtmp1 + rtmp4 - rtmp3) * &
                  rdt / e12t(ji, jj)

end subroutine continuity_code
```
A kernel operates on a single grid-point

```
subroutine continuity_code(ij, ji, jj,
                          ssha, sshn, sshn_u, sshn_v, &
                          hu, hv, un, vn, rdt, e12t)
    implicit none
    integer, intent(in) :: ji, jj
    real(wp), intent(in) :: rdt
    real(wp), dimension(:,:), intent(in) :: e12t
    real(wp), dimension(:,:), intent(out) :: ssha
    real(wp), dimension(:,:), intent(in) :: sshn, sshn_u, sshn_v, &
                                       hu, hv, un, vn

    ! Locals
    real(wp) :: rtmp1, rtmp2, rtmp3, rtmp4

    rtmp1 = (sshn_u(ji+1,jj ) + hu(ji+1,jj ) ) * un(ji+1,jj )
    rtmp2 = (sshn_u(ji  ,jj ) + hu(ji  ,jj ) ) * un(ji  ,jj )
    rtmp3 = (sshn_v(ji+1,jj ) + hv(ji+1,jj ) ) * vn(ji+1,jj )
    rtmp4 = (sshn_v(ji  ,jj+1) + hv(ji  ,jj+1)) * vn(ji  ,jj+1)

    ssha(ji,jj) = sshn(ji,jj) + (rtmp2 - rtmp1 + rtmp4 - rtmp3) * &
                   rdt / e12t(ji,jj)

end subroutine continuity_code
```
Kernel meta-data

- Kernels make use of several grid-related quantities, e.g. area of cell around a T point, T-point mask etc.

```fortran
subroutine next_sshu_code(ji,jj, sshn_u, sshn, &
    tmask,e12t,e12u)
    implicit none
    integer, intent(in) :: ji, jj
    integer, dimension(::,:), intent(in) :: tmask
    real(wp), dimension(::,:), intent(in) :: e12t, e12u
    real(wp), dimension(::,:), intent(inout) :: sshn_u
    real(wp), dimension(::,:), intent(in) :: sshn
```

- The algorithm layer should not/cannot supply these:

```fortran
call invoke( next_sshu(sshn_u, sshn), &
    next_sshv(sshn_v, sshn) )
```
• Extend meta-data to specify what quantities a kernel requires from the infrastructure:

```python
    type, extends(kernel_type) :: next_sshu
    type(arg), dimension(5) :: meta_args = &
       (/* arg(READWRITE, CU, POINTWISE), &
            arg(READ, CU, POINTWISE), &
            arg(READ, GRID_MASK_T), &
            arg(READ, GRID_AREA_T), &
            arg(READ, GRID_AREA_U) &
       */)
```

• PSyclone then supplies these quantities from the generated middle layer
What about performance?
(256 x 256 case)

Shallow optimisation stages (serial)
Gnu cannot optimise across separate source files
Loop fusion not always beneficial
Kernel source in-lining important for all except Cray
Summary for Shallow (serial)
Second benchmark: NEMOLite2D (serial)
Performance with OpenMP
NEMOLite2D, OpenMP optimisation stages

(256 x 256 case, Intel v.14)
NEMOLite2D, OpenMP

Solid lines – manual OpenMP
Dashed lines – PSyclone-generated
NEMOLite2D, OpenMP

Solid lines – manual OpenMP
Dashed lines – PSyclone-generated

Mainly due to lack of kernel inlining in generated code
GPU results courtesy of Jeremy Appleyard, NVIDIA
Next steps...

• Compilers are complex!
  – Recovering performance is not straightforward
  – More transformations required (e.g. in-lining of kernel code for Gnu and Intel)

• Currently not exploring the optimisation space
  – Only attempting to recover original code structure

• Three dimensions
  – Current test cases are two-dimensional
  – Full models are a mixture of 2D and 3D…
  – NOC working on introducing some 3D aspects to NEMOLite
Summary I

- Separation of Concerns: Introduces flexibility needed to achieve performance on different architectures
  - potentially enables e.g. OpenMP or OpenACC to be used, depending on target hardware
  - No need to modify source code containing the Natural Science (Algorithm and Kernel layers)

- Optimal code structure is both system- and compiler-dependent
Summary II

• Framework now supports two distinct shallow-water models
  – PSyKAI proven as applicable to unstructured (GungHo) AND structured (GOcean) codes

• Code generation
  – Support for loop fusion and OpenMP transformations (parallel, parallel do and do)

• Work continuing on PSyclone in the GungHo project
  – More OpenMP support (loop colouring)
  – Distributed memory support (MPI)
  – Support for OpenACC
Thank you!