# The Broader Picture of Using Accelerator Directives in Your Code

Matt Norman

National Center for Computational Sciences

Oak Ridge National Laboratory

Third Workshop on Accelerator Programming Using Directives

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

SC16 Salt Lake City, Utah | hpc matters.

# My Background

- Computational Climate Science
  - Liaise with INCITE projects running on OLCF machines
  - Accelerated Model for Climate and Energy
- Center for Accelerated Application Readiness
  - Funded by OLCF
  - Prepared codes for Titan before it arrived
  - Currently preparing codes for Summit before it arrives
- Fluids algorithm development for modern HPC

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome
  - CUDA syntax / structure is highly foreign to FORTRAN

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome
  - CUDA syntax / structure is highly foreign to FORTRAN
  - Loops cannot be ported in place; separate subroutine needed

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome
  - CUDA syntax / structure is highly foreign to FORTRAN
  - Loops cannot be ported in place; separate subroutine needed
  - Same code cannot run on GPU and CPU

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome
  - CUDA syntax / structure is highly foreign to FORTRAN
  - Loops cannot be ported in place; separate subroutine needed
  - Same code cannot run on GPU and CPU
  - Gradual porting not really possible
    - Detecting bugs significantly more difficult

# Porting work for ACME

- Initial CUDA FORTRAN work was incredibly cumbersome
  - CUDA syntax / structure is highly foreign to FORTRAN
  - Loops cannot be ported in place; separate subroutine needed
  - Same code cannot run on GPU and CPU
  - Gradual porting not really possible
    - Detecting bugs significantly more difficult
  - Optimized code looks nothing like the original code
    - Difficult to merge CPU code changes into CUDA
    - Scientific programmers cannot understand it
    - Unmaintainable and unportable

# Motivators for Using Accelerator Directives

- Large codebase – port & maintain large volume of code

# Motivators for Using Accelerator Directives

- Large codebase – port & maintain large volume of code
- You care about Software Engineering & maintainability

# Motivators for Using Accelerator Directives

- Large codebase – port & maintain large volume of code
- You care about Software Engineering & maintainability
- Support multiple platform targets (CPU, GPU, MIC)

# Motivators for Using Accelerator Directives

- Large codebase – port & maintain large volume of code
- You care about Software Engineering & maintainability
- Support multiple platform targets (CPU, GPU, MIC)
- Codebase is still under development
  - Scientific programmers must understand the code
  - Accelerated and CPU code must look similar

# Motivators for Using Accelerator Directives

- Large codebase – port & maintain large volume of code
- You care about Software Engineering & maintainability
- Support multiple platform targets (CPU, GPU, MIC)
- Codebase is still under development
  - Scientific programmers must understand the code
  - Accelerated and CPU code must look similar
- Easier debugging: run same code on CPU and accelerator

# Other Options

- Templated C++ → “kokkos”
  - However, you have to marry it and stick with it
- “Domain Specific Languages”
  - Are they just glorified if-statements?
  - Ad-hoc to the application in question
  - Still requires significant development effort]

# Other Options

- Templated C++ → “kokkos”
  - However, you have to marry it and stick with it
- “Domain Specific Languages”
  - Are they just glorified if-statements?
  - Ad-hoc to the application in question
  - Still requires significant development effort]

- Directives offer greater flexibility

# Refactoring: Relatively Little Work Per Node

# Refactoring: Relatively Little Work Per Node

- We're used to parallelizing outer loops

```
!$omp parallel do
do ie = 1 , nelements
 do k = 1 , nlevels
  mass(k,ie) = sum(vals(:,:,k,ie))
```

# Refactoring: Relatively Little Work Per Node

- We're used to parallelizing outer loops
- But now we must expose inner loops for vector ops
- However, inner loops often have race conditions
- Overlooking them → bugs

```
!$omp parallel do
do ie = 1 , nelements
 do k = 1 , nlevels
  mass(k,ie) = sum(vals(:,:,k,ie))
```

# Refactoring: Relatively Little Work Per Node

- We're used to parallelizing outer loops
- But now we must expose inner loops for vector ops
- However, inner loops often have race conditions
- Overlooking them → bugs
- Must refactor the code

```
call memset( mass , 0 )
!$acc parallel do collapse(4)
do ie = 1 , nelements
 do k = 1 , nlevels
  do j = 1 , ny
   do i = 1 , nx
    masstmp = mass(k,ie)
    valtmp = vals(i,j,k,ie)
    !$acc atomic update
    masstmp = masstmp + valtmp
```

# Refactoring: Low-Level Calls w/ Too Little Work

- Good SE practices → reusable low-level routines

```
do ie = 1 , nelements
 do k = 1 , nlevels
  grad = gradient(dat(:,k,ie))
 enddo
enddo

function gradient(dat) result(r)
 r = matmul( grad_mat , dat )
end function gradient
```

# Refactoring: Low-Level Calls w/ Too Little Work

- Good SE practices → reusable low-level routines
- “gradient” = matrix-vector multiply over 4 values
- Not enough for vector units on MIC or GPU

```
do ie = 1 , nelements
 do k = 1 , nlevels
  grad = gradient(dat(:,k,ie))
 enddo
enddo

function gradient(dat) result(r)
 r = matmul( grad_mat , dat )
end function gradient
```

# Refactoring: Low-Level Calls w/ Too Little Work

- Good SE practices → reusable low-level routines
- "gradient" = matrix-vector multiply over 4 values
- Not enough for vector units on MIC or GPU
- Manually fission & push some looping down callstack

```
do ie = 1 , nelements
 do kc = 1 , kchunk
  grad = gradient(dat(:,:,ie),kc)
 enddo
enddo

function gradient(dat,kc) result(r)
 do kk = 1 , kchunk
  do i = 1 , n
   k = (kc-1)*kchunk + kk
   tmp = 0
   do m = 1 , n
    tmp = tmp + grad_mat(i,m)*dat(m)
   enddo
   r(i,k) = tmp
  enddo
end function gradient
```

# Refactoring: Low-Level Calls w/ Too Little Work

- Good SE practices → reusable low-level routines
- "gradient" = matrix-vector multiply over 4 values
- Not enough for vector units on MIC or GPU
- Manually fission & push some looping down callstack
- Stop using "matmul"

```
do ie = 1 , nelements
 do kc = 1 , kchunk
  grad = gradient(dat(:,:,ie),kc)
 enddo
enddo

function gradient(dat,kc) result(r)
 do kk = 1 , kchunk
  do i = 1 , n
   k = (kc-1)*kchunk + kk
   tmp = 0
   do m = 1 , n
    tmp = tmp + grad_mat(i,m)*dat(m)
   enddo
   r(i,k) = tmp
  enddo
end function gradient
```

# Other Refactoring

- Array of structures: Outer index cannot be threaded easily
  - Reusable routines require flattened arrays

# Other Refactoring

- Array of structures: Outer index cannot be threaded easily
  - Reusable routines require flattened arrays
- Loop Collapsing
  - Poorly-sized inner loop dimension (vectorization)
  - Too many nested loops (cannot nest “omp do” or “acc loop”)
  - If-statements in middle of loop nest pushed into inner loop
    - Fine on GPU (already vectorized); Terrible on CPU (cannot vectorize)

# Other Refactoring

- Array of structures: Outer index cannot be threaded easily
  - Reusable routines require flattened arrays
- Loop Collapsing
  - Poorly-sized inner loop dimension (vectorization)
  - Too many nested loops (cannot nest “omp do” or “acc loop”)
  - If-statements in middle of loop nest pushed into inner loop
    - Fine on GPU (already vectorized); Terrible on CPU (cannot vectorize)
- Indirect addressing on fastest-varying dimension
  - Doesn’t saturate wide memory bus;  Doesn’t vectorize efficiently
  - Best to pad indirect addressing with contiguous dimension

# Other Refactoring

- Modern Fortran, C++ often not supported
  - Functions defined inside functions
  - Many layers of function interfaces
  - Deeply nested classes, structs, derived type data structures

# Other Refactoring

- Modern Fortran, C++ often not supported
  - Functions defined inside functions
  - Many layers of function interfaces
  - Deeply nested classes, structs, derived type data structures
- Very old Fortran often not supported
  - “Data”, goto (improving), equivalence, mysterious subroutines

# Other Refactoring

- Modern Fortran, C++ often not supported
  - Functions defined inside functions
  - Many layers of function interfaces
  - Deeply nested classes, structs, derived type data structures
- Very old Fortran often not supported
  - “Data”, goto (improving), equivalence, mysterious subroutines
- “Just because you can do a thing doesn’t mean you should”

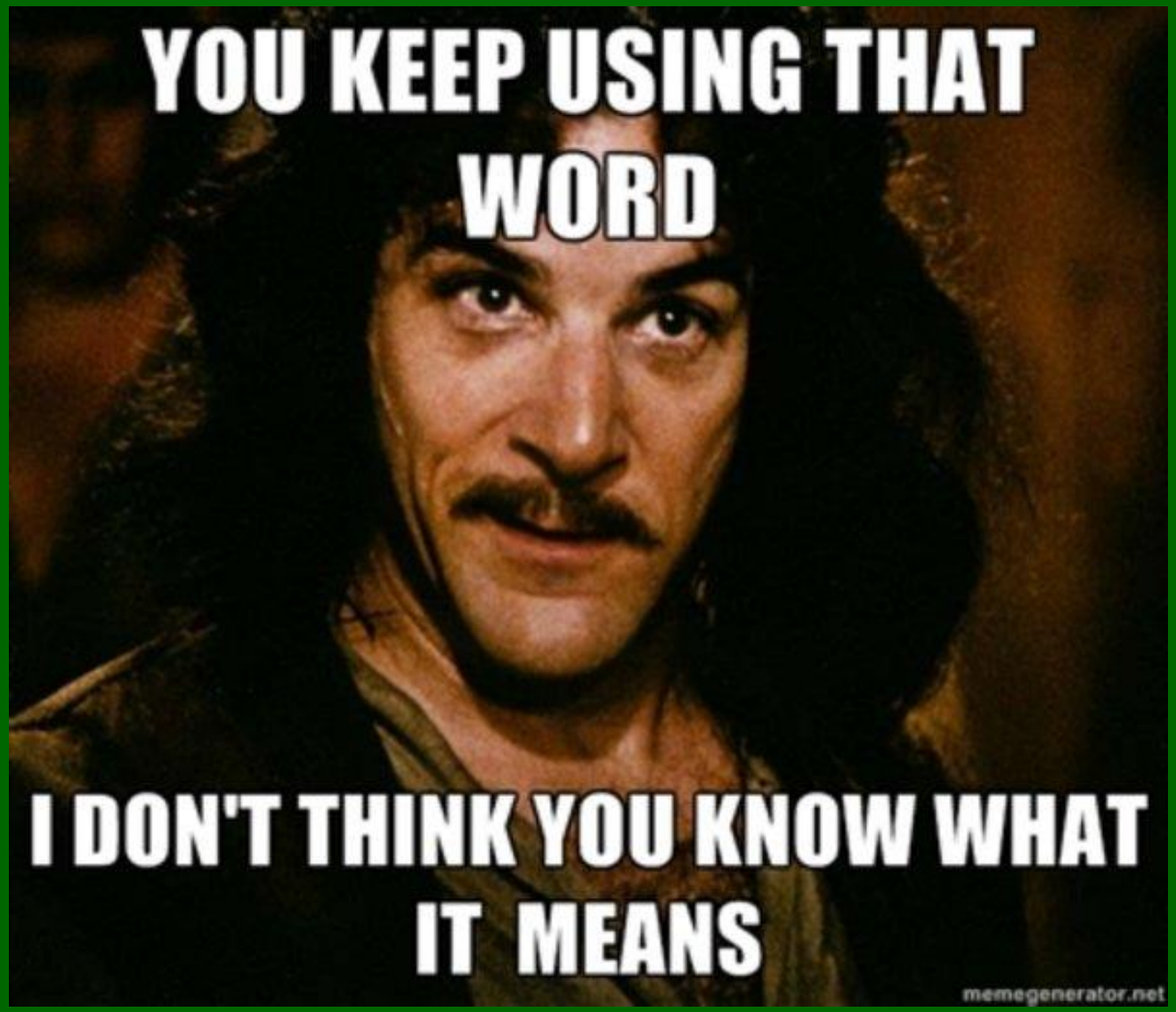
# Other Refactoring

- Modern Fortran, C++ often not supported
  - Functions defined inside functions
  - Many layers of function interfaces
  - Deeply nested classes, structs, derived type data structures
- Very old Fortran often not supported
  - “Data”, goto (improving), equivalence, mysterious subroutines
- “Just because you can do a thing doesn’t mean you should”

**Much of accelerator refactoring benefits the CPU**

# “Performance Portability”

# “Performance Portability”



# “Performance Portability”

- Identical code will never perform optimally on all platforms
  - CPU vector length: 256 bits (8 “vector threads”)
    - Heavily cache-based
  - KNL vector length: 512 bits  x  2 (16-32 “vector threads”)
    - Moderately cache-based, some latency/bandwidth hiding
  - GPU vector length: 65,536 bit (2048 “GPU vector threads”)
    - Less cache-based, heavy on latency/bandwidth hiding

# “Performance Portability”

- Identical code will never perform optimally on all platforms
  - CPU vector length: 256 bits (8 “vector threads”)
    - Heavily cache-based
  - KNL vector length: 512 bits x 2 (16-32 “vector threads”)
    - Moderately cache-based, some latency/bandwidth hiding
  - GPU vector length: 65,536 bit (2048 “GPU vector threads”)
    - Less cache-based, heavy on latency/bandwidth hiding
- Directives inherently balance performance & maintainability

# “Performance Portability”

- Identical code will never perform optimally on all platforms
  - CPU vector length: 256 bits (8 “vector threads”)
    - Heavily cache-based
  - KNL vector length: 512 bits x 2 (16-32 “vector threads”)
    - Moderately cache-based, some latency/bandwidth hiding
  - GPU vector length: 65,536 bit (2048 “GPU vector threads”)
    - Less cache-based, heavy on latency/bandwidth hiding
- Directives inherently balance performance & maintainability
- Often best to branch the code, but at the lowest level possible
- Similar looking code is easier to maintain

# Things That Can Help Performance Portability

- CPU's / MIC's being able to handle if-statements in vector units
- The ability to use nested "omp do" and "acc loop" in the same vector / thread context
- GPUs fixing their "register explosion" problem with long kernels
- CPU's & MIC's allowing users to prioritize / specify data for cache
- All compilers implementing automatic directive-based tiling
- GPU implementations improve performance of manually strip-mined loops (as opposed to having to be collapsed)

# Bugs Happen

- OpenACC & OpenMP 4.x are still maturing
- Large codebases are likely to encounter bugs

# Bugs Happen

- OpenACC & OpenMP 4.x are still maturing
- Large codebases are likely to encounter bugs
- Poor performance is a bug

# Bugs Happen

- OpenACC & OpenMP 4.x are still maturing
- Large codebases are likely to encounter bugs
- Poor performance is a bug
- A feature you rely on heavily that isn’t supported is a bug

# How To Deal With Compiler Bugs

# How To Deal With Compiler Bugs

- Report, Report, Report !!!
  - The very moment you find a bug, check in a commit and tag it
  - Try to reproduce in a smaller, more maintainable code
  - Helpful to maintain a “mini-app” to make this quicker

# How To Deal With Compiler Bugs

- Report, Report, Report !!!
  - The very moment you find a bug, check in a commit and tag it
  - Try to reproduce in a smaller, more maintainable code
  - Helpful to maintain a “mini-app” to make this quicker
- Be Proactive
  - Send vendors small code samples that you care about
  - Provide comparison points if applicable
  - Get in touch with vendor reps so they’re aware of your code

# How To Deal With Compiler Bugs

- Report, Report, Report !!!
  - The very moment you find a bug, check in a commit and tag it
  - Try to reproduce in a smaller, more maintainable code
  - Helpful to maintain a “mini-app” to make this quicker
- Be Proactive
  - Send vendors small code samples that you care about
  - Provide comparison points if applicable
  - Get in touch with vendor reps so they’re aware of your code
- Be kind: Compiler developers are people too

# A Sociological Experiment

# A Sociological Experiment

- OpenMP and OpenACC are as much sociology as technical
  - App. Developer: “I won’t use it because it isn’t mature.”
  - Compiler Developer: “It immature because you won’t use it.”
- You determine when it’s appropriate to try things out
  - But “mature” is not a well-defined idea
  - Accelerator directives will always have room for improvement

# A Sociological Experiment

- OpenMP and OpenACC are as much sociology as technical
  - App. Developer: “I won’t use it because it isn’t mature.”
  - Compiler Developer: “It immature because you won’t use it.”
- You determine when it’s appropriate to try things out
  - But “mature” is not a well-defined idea
  - Accelerator directives will always have room for improvement

**We <u>all</u> benefit when you engage compiler developers**

**https://developer.nvidia.com/accelerated-computing-developer**

# Refactoring: Lots Work Per Node

- Often have to stage data to Accelerator's smaller RAM

# Refactoring: Lots Work Per Node

- Often have to stage data to Accelerator's smaller RAM
- Usually, long outer loop over many routines

```
do ie = 1 , nelements
 tmp1 = routine1(data1(:,:,ie))
 [Intermittent work]
 tmp2 = routine2(tmp1)
 [Intermittent work]
 data3(:,:,ie) = routine3(tmp1,tmp2)
enddo
```

# Refactoring: Lots Work Per Node

- Often have to stage data to Accelerator's smaller RAM
- Usually, long outer loop over many routines
- On GPUs, long kernels → register pressure → poor performance

```
do ie = 1 , nelements
 tmp1 = routine1(data1(:,:,ie))
 [Intermittent work]
 tmp2 = routine2(tmp1)
 [Intermittent work]
 data3(:,:,ie) = routine3(tmp1,tmp2)
enddo
```

# Refactoring: Lots Work Per Node

- Often have to stage data to Accelerator's smaller RAM
- Usually, long outer loop over many routines
- On GPUs, long kernels → register pressure → poor performance
- Have to break up loops and turn local temps into globals

```
do ie = 1 , nelements
 glob1(:,:,ie) = &
                 routine1(data1(:,:,ie))
 [Intermittent work]
enddo
do ie = 1 , nelements
 glob2(:,:,ie) = &
                 routine2(glob1(:,:,ie))
 [Intermittent work]
enddo
do ie = 1 , nelements
 data3(:,:,ie) =  &
               routine3(glob1(:,:,ie),&
                        glob2(:,:,ie))
enddo
```