OpenMP device constructs 2

Contents

- **How kernels execute on a GPU**
- **Optimisation**
  - Data locality
  - Accelerated libraries
  - Kernel performance
    - vectorisation
    - data layout
    - collapsing loops

The wheel has been invented already

- **There is a large ecosystem of tuned GPU libraries**
  - Cray libsci_acc
    - man into_libsci_acc (after module load craype-accel-nvidia60)
    - BLAS 1,2,3
    - LAPACK
    - ScaLAPACK
    - PBLAS
  - Third party: cuBLAS, MAGMA, cuFFT, etc.

Library Interoperability in OpenMP (v4.5)

```
PROGRAM main

  <stuff>

!$omp target data map(from:a) map(to:b)
  <stuff>

!$omp target data use_device_ptr(a,b)
  CALL library(a,b)
!$omp end target data

  <stuff>
!$omp end target data

  <stuff>

END PROGRAM main
```

- **use_device_ptr exposes accel. memory address**
  - Must synchronise before and after
  - CUDA kernel written as usual

Library Interoperability in OpenACC

```
PROGRAM main
  <stuff>

!$acc data pcopyout(a) pcopyin(b)
 <stuff>

!$acc wait
!$acc host_data use_device(a,b)
  CALL library(a,b)
!$acc end host_data
!$acc wait

 <stuff>
!$acc end data

  <stuff>
END PROGRAM main
```

- **use_device exposes accel. memory address**
  - Must synchronise before and after
  - CUDA kernel written as usual

Kernel scheduling

- **Once data movements have been optimised**
  - the next step is to improve the kernel scheduling
    - i.e. how the kernels actually execute on the accelerator
- **For this, need to understand how the hardware works**
  - we'll concentrate on Nvidia GPUs for this
- **A kernel executes as a large number of parallel threads**
  - the iterations of the loopnest are divided between these threads
    - This is called "partitioning" of the loops
  - need to understand how these threads are executed on the GPU

How a CUDA kernel is executed on a GPU

- **The threads are logically divided into sets**
  - called a grid of threadblocks in CUDA
    - user must specify the size of these sets (number of threads per threadblock)
  - each threadblock executes independently on a different piece of hardware
    - Symmetric Multiprocessor (SM), which is (almost) like a single vector processor

- **Within a threadblock**
  - The threads are logically divided into sets of 32 threads called warps
    - threads within a warp execute in lockstep (i.e. as SIMD vector instructions)
  - so the threadblock executes as a sequence of warps

- **The CUDA programming model does not make warps explicit**
  - the user only specifies the number of threads per threadblock
  - but they are crucial to understand kernel performance

OpenMP accelerator threads

- **OpenMP target region threads follow a similar model**
  - teams directive generates a league of threadteams
  - a threadteam maps exactly onto a CUDA threadblock
    - each threadteam executes on a single SM
    - the league of threadteams is distributed across the SMs
  - the threadteam will execute as a set of SIMD instructions
    - each SIMD instruction maps exactly onto a CUDA warp
    - multiple SIMD instructions are needed to execute threadblock
    - warps are not made explicit in OpenMP

OpenACC accelerator threads

- **OpenACC threads follow one of two models, either:**
  - parallel directive generates a set of gangs of vectors
  - a gang maps exactly onto a CUDA threadblock
  - a vector maps exactly onto the CUDA threads within a threadblock
- **Or, you can expose three levels of parallelism**
  - parallel directive generates a set of gangs of workers of vectors
  - a gang maps exactly onto a CUDA threadblock
  - a worker maps exactly onto a CUDA warp
  - a vector is a set of exactly 32 threads making up a warp

Loop partitioning (scheduling)

- **When a loopnest is partitioned by the compiler**
  - The loop iterations are divided across a set of threads
  - The threads are then logically grouped to match the hardware requirements
  - There are many different ways for the compiler to do this
    - How many and which loop(s) in loopnest should be partitioned?
    - How many threads (and how divided into threadblocks and threads per threadblock)?
    - How should the iterations be divided?
- **If user gives no further instruction, the compiler makes a decision**
  - You can see what this was from the compiler feedback
  - You can then over-ride this decision (at the next compilation)
    - by using further clauses and directives

Loop partitioning with OpenMP

- **teams directive generates league of threadteams**
- **distribute directive partitions the loop iterations**
  - Can use composite directive target teams distribute

- **Completing the partitioning**
  - The OpenMP standard is vague on how distribute should work
    - could only partition at the threadblock level
      - then need nested parallel, do/for, simd directives to complete the partitioning
    - could partition over all levels of parallelism (CCE behaviour)
      - parallel directive allowed, but only with one thread
      - simd directive can optionally be used to tune the scheduling

Loop partitioning with OpenACC

- **parallel directive generates gangs of workers/vectors**
- **loop directive partitions the loop iterations**
  - Can use composite directive parallel loop
- **These allow full partitioning of up to three nested loops**
- **Can optionally use gang, worker, vector clauses**
  - Added to top-level loop directive or on additional ones
  - Allows developer to tune the loop scheduling

Why warps matter (1): vectorisation

- **SMs execute warps as a stream of vector instructions**
  - The vectors are 32 slots wide
  - Every slot does the same operation at the same time, but with different data
  - If all 32 threads in a warp want to do the same thing at the same time, then can execute this as a single vector instruction
  - If not, will need multiple (sequential) vector instructions (masking out results from some slots each time)
- **Kernels should be written to take advantage of this**
  - Wherever possible, arrange that groups of 32 consecutively-numbered threads are doing the same thing at the same time
  - This is called "vectorising" the code

Kernel vectorisation

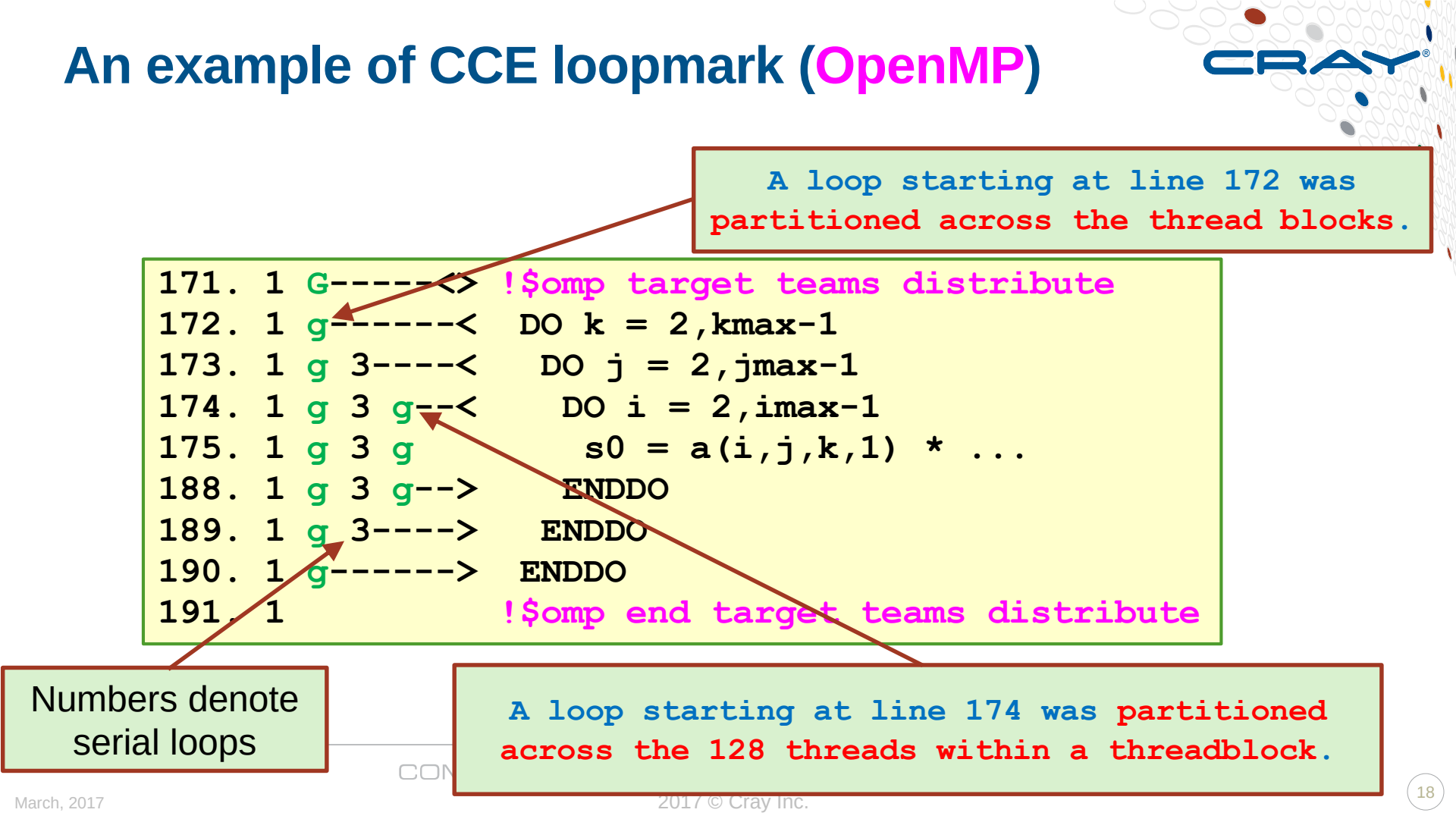
- **The first optimisation:**
  - using data regions to minimise data movement
- **The second optimisation:**
  - make sure all the kernels vectorise
  - meaning the compiler is efficiently using vector instructions on GPU

- **How can I tell if there is a problem?**
  - see if a kernel is "surprisingly slow" on accelerator
    - profile the code executing on the CPU; note the relative ordering (by time) of routines
    - profile the code executing on the accelerator; compare the ordering to that from the CPU
    - look for any anomalously-slow kernels
  - examine the compiler commentary

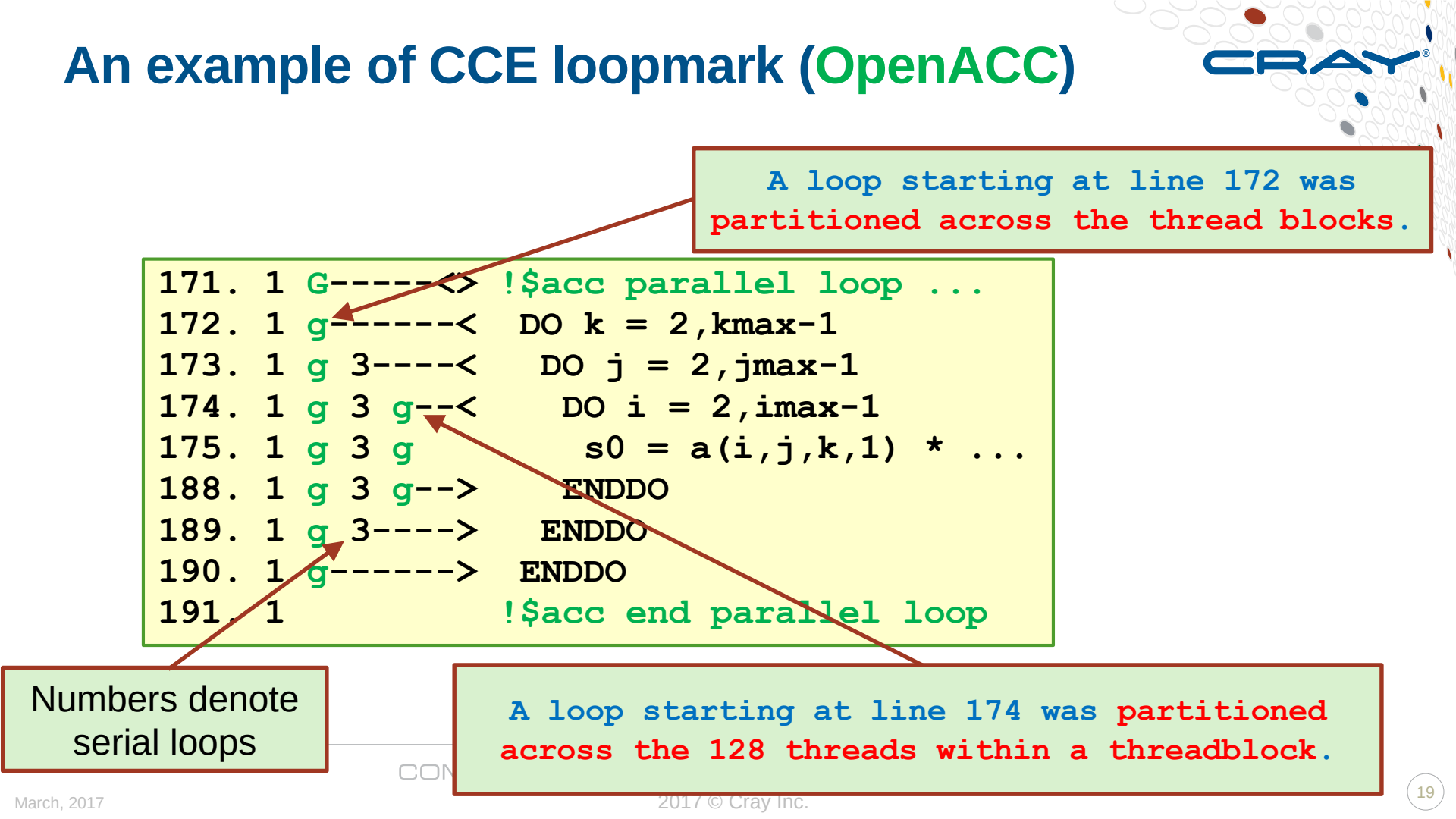
Vectorisation and loopmark

- **Always generate the compiler commentary**
  - CCE: compile with flag -hlist=a to generate *.lst loopmark file
- **Should see loop iterations divided both:**
  - over threadblocks and over threads within a threadblock
- **A single loop:**
  - should be partitioned across both levels of parallelism
    - look for this in the CCE loopmark: Gg
- **A loopnest:**
  - Two (maybe three) loops should be separately partitioned
    - look for G and 2 (or maybe 3) g-s (possibly some numbers in between)

An example of CCE loopmark (OpenMP)



An example of CCE loopmark (OpenACC)



Why warps matter (2): coalescing

- **Data is loaded from (and stored to) memory in chunks**
  - 32 contiguous addresses in memory
  - A warp is 32 threads executing the same instruction on different data
  - If a warp processes a contiguous block of 32 elements from memory,
    - then can service this with a single vector memory operation
  - If not, need multiple operations (with some data "wasted", i.e. unprocessed)
- **Kernels should be written to take advantage of this**
  - Wherever possible, arrange that warps process contiguous blocks of memory.
  - Vectorised should index the fastest-moving dimension of the data arrays
    - left-most in Fortran, right-most in C
  - This ensures the minimum number of (expensive) memory operations
  - This is called "coalescing" the memory accesses

Helping vectorisation (1)

- **Every kernel should have a vectorised loop**
  - "partitioned across threads within a threadblock"
  - generally want to vectorise the innermost loop
  - usually labels fastest-moving array index
    - to enable coalescing
- **If not vectorised, can we make it vectorise?**
  - Can loop iterations be computed in any order?
  - No? Then can we rewrite code?
    - avoid loop-carried dependencies
    - e.g. buffer packing: calculate rather than increment
    - these rewrites will probably perform better on CPU

Replace:

```
i = 0
DO y = 2,N-1
   i = i+1
   buffer(i)=a(2,y)
ENDDO
buffsize = i
```

By:

```
DO y = 2,N-1
   buffer(y-1)=a(2,y)
ENDDO
buffsize = N-2
```

Helping vectorisation (OpenMP)

- **Can loop iterations be computed in any order?**
- **Yes? Then we should guide the compiler**
  - put simd directive above this loop
  - check the code is still correct (as well as running faster)
    - the compiler might not be vectorising the loop for a good reason
- **Advice:**
  - Remember to repeat any reduction clauses on simd directives

Helping vectorisation (OpenACC)

- **Can loop iterations be computed in any order?**
- **Yes? Then we should guide the compiler**
  - put "acc loop vector" directive above this loop
  - check the code is still correct (as well as running faster)
    - the compiler might not be vectorising the loop for a good reason
- **Advice:**
  - Remember to repeat any reduction clauses on new loop directives

Changing the vectorisation (OpenMP)

- **If the inner loop is vectorising but performance is still bad**
- **Is the inner loop really the one to vectorise in this case?**
  - in this kernel, perhaps we know mmax is small
    - or perhaps CrayPAT loop-level profiling told us this
  - so we should vectorise the i-loop
- **Put simd directive above i-loop**
  - m-loop now not partitioned
  - executed "redundantly" by every thread
  - this may also help performance
    - t is now an i-loop private scalar
    - rather than a reduction variable

```
DO i = 1,N
  t = 0
  DO m = 1,mmax
    t = t + c(m,i)
  ENDDO
  a(i) = t
ENDDO
```

Changing the vectorisation (OpenACC)

- **If the inner loop is vectorising but performance is still bad**
- **Is the inner loop really the one to vectorise in this case?**
  - in this kernel, perhaps we know mmax is small
    - or perhaps CrayPAT loop-level profiling told us this
  - so we should vectorise the i-loop
- **Put vector clause above i-loop**
  - m-loop now not partitioned
  - executed "redundantly" by every thread
  - this may also help performance
    - t is now an i-loop private scalar
    - rather than a reduction variable

```
DO i = 1,N
  t = 0
  DO m = 1,mmax
    t = t + c(m,i)
  ENDDO
  a(i) = t
ENDDO
```

An aside on this example

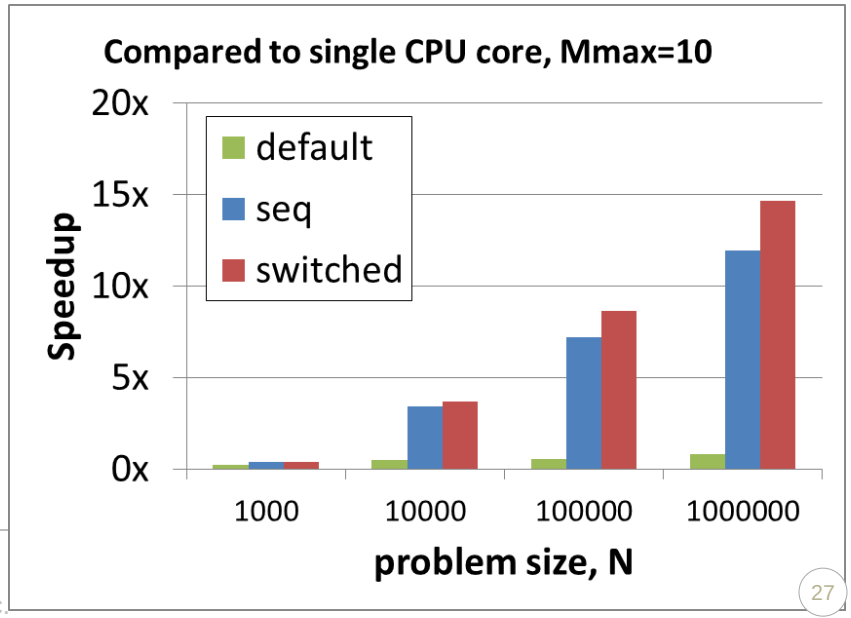
```
DO i = 1,N
  t = 0
  DO m = 1,mmax
    t = t + c(m,i)
  ENDDO
  a(i) = t
ENDDO
```

- **We forced vectorisation of the i-loop**
  - because mmax was too small
    - small loops will not give the GPU enough work
- **Performance is still likely to be bad**
  - Memory accesses to c are no longer coalesced
  - The vectorised loop does not index the fastest-moving array dimension
    - 32 threads execute together as a warp (vector), each with different i-values
    - each warp needs 32 elements of data, e.g. c(m,1:32)
    - this is not a contiguous chunk of memory, so need multiple vector loads per warp
  - loads/stores from global memory are slow, so performance will suffer
    - this is a hardware feature, not a limitation of the programming model

Improving coalescing

- **Consider re-ordering arrays**
  - Keep vectorising i-loop
    - as mmax small
  - Make i fastest index of new array ct
  - ct(1:32,m) is a contiguous chunk
  - Now get coalesced loads
- **Requires refactoring code**
  - may be much work (but maybe not)
  - may also get better CPU performance
    - e.g. with AVX instructions

```
DO i = 1,N
  t = 0
  DO m = 1,mmax
    t = t + ct(i,m)
  ENDDO
  a(i) = t
ENDDO
```



It's all vectorising, but still performing badly

- **Profile the code and start where it takes most time**
  - check the slowest thing really is compute kernels
  - if it really is GPU compute kernels...
- **GPUs need lots of parallel tasks to work well**
  - check that there really are enough loop iterations (beware of test cases)
- **Then look at loop scheduling using clauses**
- **Then might need to consider more extreme measures**
  - source code changes
  - handcoding CUDA kernels

Tuning the OpenMP partitioning

- **Before we alter the schedule, there are two easy tuning choices**
  - number of threads per threadblock; total number of threadblocks
- **Do this on a per-kernel basis using clauses on team directive**
  - This is optional, unlike in CUDA
- **thread_limit(<value>)**
  - changes the number of threads per threadblock
  - CCE-specific details:
    - value needs to be fixed at compile time
    - allowed values are: 1, 32, 64, 128, 256, 512, 1024 (default value is 128)
- **num_teams(<value>)**
  - changes the number of threadblocks
  - the compiler will make a default choice of sufficient blocks

Tuning the OpenACC partitioning

- **Before we alter the schedule, there are two easy tuning choices**
  - number of threads per threadblock; total number of threadblocks
- **Do this on a per-kernel basis using clauses on parallel directive**
  - This is optional, unlike in CUDA
- **vector_length(<value>)**
  - changes the number of threads per threadblock
  - CCE-specific details:
    - value needs to be fixed at compile time
    - allowed values are: 1, 32, 64, 128, 256, 512, 1024 (default value is 128)
- **num_gangs(<value>)**
  - changes the number of threadblocks
  - the compiler will make a default choice of sufficient blocks

Advanced loop scheduling

- **Loop schedules (usually) are limited by the loop bounds**
  - one loop's iterations are divided over threadblocks
  - another loop's iterations are divided over threads in threadblock

- **This has limitations, for instance**
  - "tall, skinny" loopnests (j=1:big; i=1:small) won't schedule well
    - inner loop is too small
      - if less than 32 iterations won't even fill a warp, so wasted SIMT
  - "short, fat" loopnests (j=1:small; i=1:big) also not good
    - outer loop is too small
      - want lots of threadblocks to swap amongst SMs

```
DO j = 1,Nj
  DO i = 1,Ni
    :
  ENDDO
ENDDO
```

- **In both cases there are enough total iterations to keep the GPU busy**
  - but division of iterations between the two loops is non-optimal for the GPU

collapse clause

- A way of increasing loop scheduling flexibility
- Merges iterations of two or more loops together
  - Then applies scheduling to composite loop (automatic or explicit scheduling)
- Both "tall, skinny", "short, fat" examples will benefit from collapse(2)
  - collapse the j and i loops into a single iteration space, effectively: `DO ij=1,small*big`
  - then divide the total iterations over all levels of parallelism on the GPU

- Things to look for
  - the compiler may use this automatically (look for C in CCE loopmark feedback)
  - no guarantee that it is faster [index rediscovery uses expensive integer divisions]
    - e.g. `j = INT(ij/big); i = ij - j*big`
  - you can only collapse perfectly nested loops

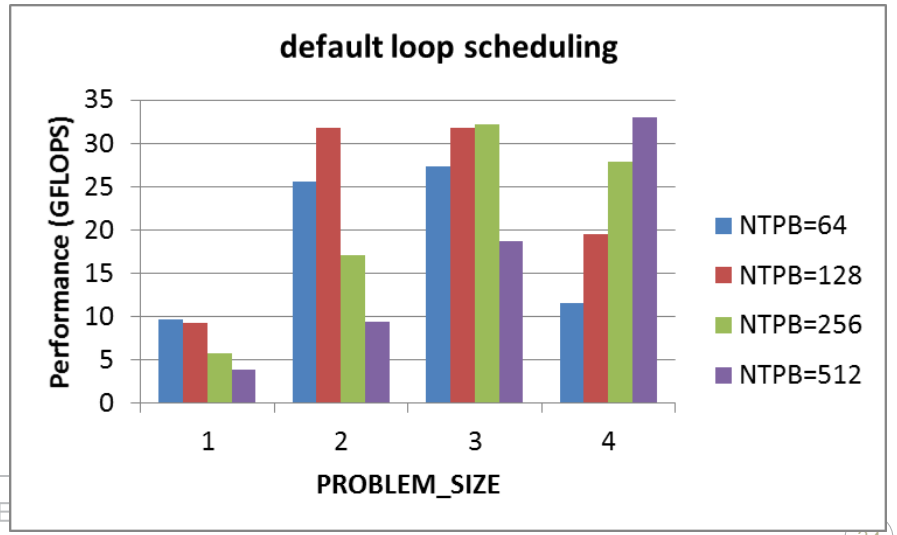
- Advice: generally best to fully collapse loopnest

Tuning performance with the loop schedule

- **Tuning the loop schedule is main performance tuning**
  - first make sure you've minimised data transfers
  - then make sure the default schedule is sensible
    - loop(s) partitioned both ACROSS and WITHIN threadblocks
  - then think about using tuning clauses to improve performance
- **The next few slides give a tuning Case Study**
  - The scalar Himeno code that we discussed earlier
  - Applying tuning clauses to the jacobi stencil kernel

Scalar Himeno performance

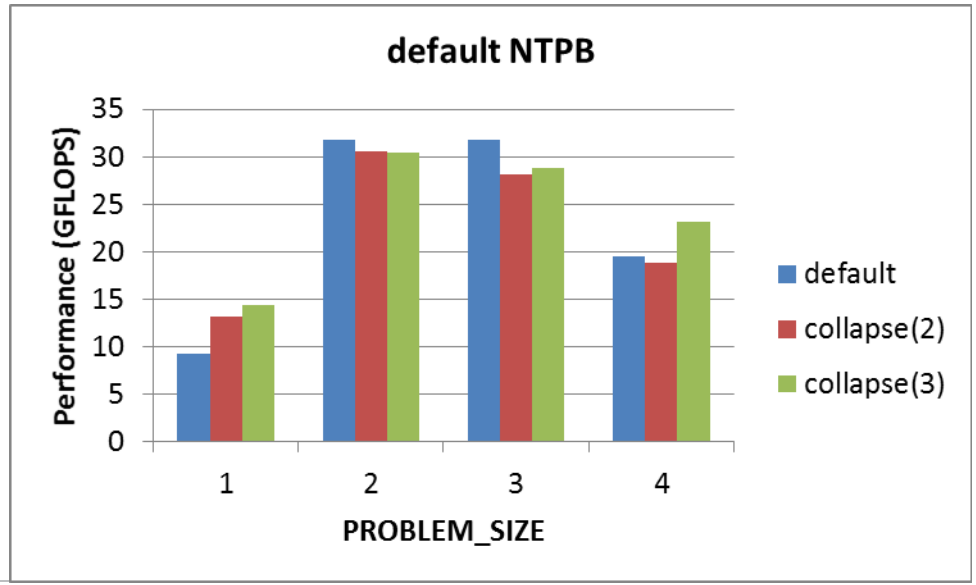
- **PROBLEM_SIZE=1,2,3,4**
  - For a parallel problem, this simulates strong scaling choices
  - Here we look at all 4 options (all double precision)
- **threads per threadblock choices: NTPB=64,128,256,512**
  - Easy tuning choice, but needs a recompile with CCE
    - easiest to compile with -DNTPB=<value>
  - CCE defaults to 128
- **This has a BIG effect**
  - best NTPB relative to default:
  - PROBLEM_SIZE=1: + 4%
  - PROBLEM_SIZE=2: no change
  - PROBLEM_SIZE=3: + 1%
  - PROBLEM_SIZE=4: +69%



Scalar Himeno performance

- PROBLEM_SIZE=1,2,3,4
- 3 scheduling choices for jacobi stencil kernel
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
- NTPB is default

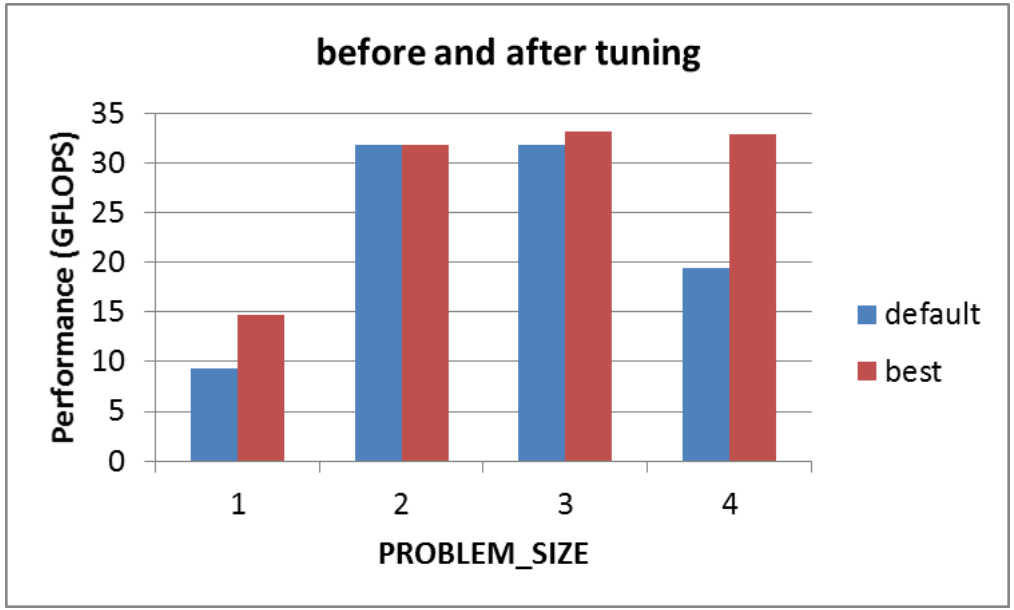
- Effect also quite big
  - PROBLEM_SIZE=1: +55%
  - PROBLEM_SIZE=2: no change
  - PROBLEM_SIZE=3: no change
  - PROBLEM_SIZE=4: +19%



Scalar Himeno performance

- PROBLEM_SIZE=1,2,3,4
- 3 scheduling choices for jacobi stencil kernel
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
- optimal NTPB

- Final improvements
  - PROBLEM_SIZE=1: +60%
  - PROBLEM_SIZE=2: + 1%
  - PROBLEM_SIZE=3: + 4%
  - PROBLEM_SIZE=4: +69%



Scalar Himeno tuning conclusions

- **Tuning can have a big effect, relative to default**
  - It is worth doing, but only after you optimise the data locality
    - data region gave 44x speedup; kernel tuning gave less than 2x
  - We gained something at almost every problem size
  - Only explored NTPB and basic scheduling (collapse)
    - only change was directive clauses; CPU version same

- **General conclusions for scalar Himeno:**
  - Larger NTPB suited larger problem sizes
    - 64 best for PS1; 128 best for PS2; 256 best for PS3; 512 best for PS4
  - collapse(3) was best for small problems (PS1,4)

Aside: Will my code accelerate well?

- **Computation should be based around loopnests processing arrays**
- **Loops should have defined tripcounts**
  - either at compile- or run-time
  - while loops will not be easy to port
    - they are hard to execute on a GPU
- **Data structures should ideally be simple arrays**
  - derived types, pointer arrays, linked lists etc. may stretch compiler capabilities
- **Loopnests should have a large total number of iterations**
  - at least measured in the thousands
  - even more is better
  - less will execute, but inefficiently
- **The loops should span as much code as possible**
  - maybe with some loops very high up the callchain
- **Loopnest kernels should not be too branched**
  - one or two nested IF-statements is fine
  - too many will lead to slow execution on many accelerators
- **The code can be task-based**
  - but each task should contain a suitable loopnest

Conclusions

- **Optimisation path:**
  - minimise data movements
  - use libraries if possible
  - ensure vectorisation of correct loop
  - consider data reordering
  - look at collapsing perfectly nested loops

- **Only then consider the more advanced optimisations**
  - e.g. hand-coded CUDA

Tuning code performance

- **Remember the Golden Rules of performance tuning:**
  - always profile the code yourself
    - always verify claims like "this is always the slow routine" as codes and computers change
  - optimise the real problem running on the production system
    - a small testcase running on a laptop will have a very different profile
  - optimise the right parts of the code
    - the bits that take the most time, even if these are not the exciting bits of the code
    - e.g. it might not be GPU compute; it might be comms (MPI), I/O...
  - keep on profiling
    - the balance of CPU/GPU/comms/IO will change as you go
    - refocus your efforts appropriately
  - Keep on checking for correctness
  - Know when to stop (and when to start again)