CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced topics in OpenACC

CSCS-USI Summer School 2014

Markus Wetzstein
wetzstein@cscs.ch

Motivation and Outlook

there surely has to be more to OpenACC...???

...yes, indeed:

- **tuning workshare constructs**
- **reduction operations**
- **asynchronous operations**
- **interoperability with CUDA and libraries**

I have an existing OpenMP code, how easy is it to port to OpenACC?

...fairly easy, BUT:

important differences between OpenMP and OpenACC (although it all looks very similar)

Parallel region with multiple workshared loops ?

- **often used construct in OpenMP**
- **thread creation/ destruction @ begin/end of parallel region**
- **loops are workshared**
- **code between loops is executed redundantly on each thread**

what about OpenACC ?

```
!$omp parallel shared(…) &
!$omp& private(…) […]

!$omp do
do i=istart,iend
   …
enddo
[some code, not workshared]
!$omp do
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$omp end parallel
```

Use of parallel and loop

- **OpenACC parallel region can span several loops**
  BUT
- **OpenACC ≠ OpenMP although visually very similar**
- **each loop executed by the same sets of threads**
- **no synchronization between these and none with code in between !**
- **no global barrier mechanism inside parallel region**
- **very easy to create race conditions**

```
!$acc parallel shared(…) &
!$acc& private(…) […]

!$acc loop
do i=istart,iend
   …
enddo
[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$acc end parallel
```

Use of parallel and loop (2)

```
!$acc parallel shared(…)
private(…) […]

!$acc loop
do i=istart,iend
   …
enddo

[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo

!$acc end parallel
```

```
!$omp parallel shared(…) &
!$omp& private(…) […]

!$omp do
do i=istart,iend
   …
enddo
!$omp end do nowait
[some code, not workshared]
!$omp do
do j=jstart,jend
   do k=kstart,kend
      …
   enddo
enddo
!$omp end do nowait
!$omp end parallel
```

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Use of parallel and loop: recommendations

- **only use composite `parallel loop` at first**
  ➔ get correct code
- **understand `loop` by itself as corresponding to having an implicit `nowait` clause in OpenMP (without the option to have a `wait`!)**
- **carefully separate directives as a later performance tuning step:**
- only if you're sure the loops can be independent kernels and have no race conditions !
- consider explicitly using `async` clause instead
  ➔ slightly more complicated, but enhanced code clarity

```
!$acc parallel shared(…) &
!$acc& private(…) […]

!$acc loop
do i=istart,iend
   a(i)=b(i)*const + c(i)
enddo
[some code, not workshared]
!$acc loop
do j=jstart,jend
   do k=kstart,kend
      x(j,k)=y(j,k)+z(j,k)
   enddo
enddo
!$acc end parallel
```

✓ loops independent!
✓ potentially slightly faster than having two separate `parallel loop` clauses

Kernels vs parallel regions

- **what is the difference between `kernels` and `parallel` ?**
- very similar usage
- have different historic origins (`kernels` ➔PGI, `parallel` ➔Cray)
- OpenACC standard not very helpful to understand when to use what

```
#pragma acc parallel loop […]
for(i=istart ;i<=iend; i++) {
   …
}
#pragma acc kernels loop […]
for(i=istart ;i<=iend; i++) {
   …
}
```

in common:

- **both define a region to be accelerated**

differences:

- **different levels of obligation to compiler**

`parallel`	`kernels`
1 kernel	≥1 kernel(s)
must be accelerated	can be accelerated
tuning clauses	no tuning clauses

Kernels vs parallel regions (2)

- compiler will automatically analyze all loops inside `kernels`
- BUT: with kernels, first loop is guaranteed to have finished before second starts

```
#pragma acc kernels […] {
for(i=istart ;i<iend; i++) {
   a[i]=b[i]*c[i];
}
for(i=istart+1 ;i<iend-1; i++) {
   d[i]= 0.5*(a[i-1]+a[i+1]);
}
} //acc kernels
```

What to use…?

- `parallel` offers greater control
- `kernels` maybe better to initially explore parallelism

suggestion:

- don’t mix them unless you’re really aware of the subtle differences

Tuning parallel execution

```
[parallel] loop [gang] [worker] [vector]
```

- **parallel execution structured into hierarchy:**
  gang $\longrightarrow$ worker $\longrightarrow$ vector
- **code is executed in parallel with current level of parallelism until a new level is opened (gang: redundant execution)**
- **optional, compilers define defaults (possibly using heuristics)**
- **only allowed on `loop` directive**

```
!$acc parallel loop […] gang
do i=istart,iend
!$acc loop worker
   do j=jstart,jend
!$acc loop vector
      do k=kstart,kend
         …
      enddo
   enddo
enddo
!$acc end parallel
```

OpenACC	CUDA
gang	threadblock
worker	warp of threads
vector	threads

when to use it, and why?

THIS is not a very efficient use case !

Tuning parallel execution (2)

Explicitly using multiple levels of parallelism:

- **loop iterations must be data independent (except `reductions`)**
- **usage: indirect indexing, …**
- **`worker` and `vector` loops have an implied barrier at end of loop**

```
!$acc parallel loop […] gang
do i=istart,iend
   inew=index_list(i)
!$acc loop worker
   do j=jstart,jend
      jnew=index_list2(j)
!$acc loop vector
      do k=kstart,kend
         a(k,j,i)=b(k,jnew)+c(k,inew)
      enddo
   enddo
enddo
!$acc end parallel
```

```
loop [collapse(Nlevels)]
```

- **specifies how many tightly nested loops are associated with a `loop` construct**
- **without `collapse` a `loop` construct only affects the immediately following loop**

```
!$acc parallel loop […] collapse(3) &
!$acc& gang worker vector
do i=istart,iend
   do j=jstart,jend
     do k=kstart,kend
         a(k,j,i)=b(k,j)+c(k,i)
      enddo
   enddo
enddo
!$acc end parallel
```

Tuning parallel execution (3)

```
parallel [num_gangs(N1)] [num_workers(N2)] [vector_length(N3)]
```

- **`num_gangs`**: nr. of gangs to use for parallel region (integer)
- **`num_workers`**: nr. of workers to use for **`worker`** loops (integer)
- **`vector_length`**: nr. of threads to use for **`vector`** loop (integer)

- **binds to `parallel`, not `loop`**
- **if omitted, compiler chooses itself**
- **`vector_length`: compiler might allow only certain values**
  e.g. Cray: 1, 64, 128 (default), 256, 512, 1024
- **Cray only allows:**
  either `num_workers` (fixes **`vector_length`**`=32`)
  or `vector_length` (fixes **`num_workers=vector_length/32`**)

Tuning parallel execution (4)

Some suggestions:

- **explicitly using `worker` as separate level often not very useful** (in current implementations)
- **tightly nested loops: try if `collapse` improves performance**
- **tuning `num_gangs|num_workers|vector_length`**
  - is time consuming
  - optimal choice depends on actual loop
  - focus on expensive loops
- **to debug kernel by running single thread, use:**

```
#pragma acc parallel num_gangs(1) vector_length(1)
```

Reduction operations

```
[parallel | loop] reduction(operator:variable-list)
```

- **OpenACC reductions very similar to OpenMP**
- **`reduction` only allowed for scalars**
  arrays: rewrite to use temporary scalars inside loop nest for reduction

```
#pragma acc parallel loop
        reduction(+:t)
for(i=istart;i<=iend;i++) {
   t=t + a[i] – b[i];
}
```

- **reduction variable is `private` to each thread**
- **combine result over all threads**
  e.g. sum, max, min, logical and
- **careful: reduction over `gang`s only done at end of parallel construct !**

Reduction operations (2)

C / C++		Fortran	
operator	initialization	operator	initialization
`+`	0	`+`	0
`*`	1	`*`	1
`max`	least	`max`	least
`min`	largest	`min`	largest
`&`	`~0`	`iand`	all bits on
`|`	0	`ior`	0
`^`	0	`ieor`	0
`&&`	1	`.and.`	`.true.`
`||`	0	`.or.`	`.false.`
		`.eqv.`	`.true.`
		`.neqv.`	`.false.`

initialization automatically by compiler (based on operation)

```
#pragma acc parallel […] {
[some code]
#pragma acc loop reduction(+:t)
        gang vector collapse(2)
for(i=istart;i<=iend;i++) {
   for(j=jstart;j<=jend;j++) {
      t = t + a[j,i] – b[j,i];
   }
}
// reduction of t is INCOMPLETE
// using t here=race condition
} // end acc parallel

// using t here is OK
```

be careful with reductions over gangs before exiting parallel region

Asynchronous operations

- **GPUs have more than one queue (CUDA: stream) into which operations can be entered**
  in hardware: Nvidia Fermi 16, Kepler 32 (with better hardware to overlap those), nr. of logical queues even much higher
- **operations in different queues can be executed concurrently**
- **CPU can continue execution immediately after adding an operation to a queue ➔ no need to wait for completion of actual operation**
- **potential performance gains from:**
- overlapping data transfer with computation on GPU
- overlapping data transfer with computation on CPU
- expose more parallelism to the GPU (e.g. multiple kernels and data transfers at the same time)

List of asynchronous clauses / directives

```
wait[(handle-list)] [async(handle)]
```

- synchronisation directive
- `handle`: non-negative integer denoting the queue
- `handle-list`: list of handles, can only be used with `wait`
- `wait`: wait until all asynchronous operations have completed
- `wait(handle-list)`: wait until all asynchronous operations in the queues specified by `handle-list` have completed
- `wait async(handle)`: enters the synchronisation into the queue `handle`

```
[parallel | kernel | enter data | exit data | update]  [async[(handle)]]
[wait[(handle-list)]]
```

- `async`: enters the operation into a default queue
- `async(handle)`: enters the operation into the queue `handle`
- `wait`: operation starts after all asynchronous operations have completed
- `wait(handle-list)`: operation starts after all asynchronous operations in the queues specified by `handle-list` have completed
- combinations possible, e.g. `parallel wait async(1)` enter parallel region into queue 1, but don’t execute it until all asynchronous operations have completed

Asynchronous example 1

```
[prepare array a on CPU]
#pragma acc enter data async(1) copyin(a)
[prepare array b on CPU]
#pragma acc enter data async(2) copyin(b)
#pragma acc parallel loop async(1) present(a)
for(i=istart ;i<iend; i++) {
   a[i]= [some computation on GPU]
}
#pragma acc exit data copyout(a) async(1)
#pragma acc parallel loop async(2) present(b)
for(j=jstart ;j<jend; j++) {
   b[j]= [some computation on GPU]
}
#pragma acc exit data copyout(b) async(2)
[some computation on CPU]
#pragma acc wait
[continue to use updated a,b on CPU]
```

- simple example with two arrays
- update of arrays independent of each other
  - copy data to GPU
  - compute on GPU
  - copy back to CPU
- this approach can be generalized, e.g. for slices of a larger array

Asynchronous example 2

```
REAL::a(Nvec,Nchunks),b(Nvec,Nchunks)
!$acc data create(a,b)
DO j = 1,Nchunks
!$acc update device(a(:,j)) async(j)
!$acc parallel loop async(j)
   DO i = 1,Nvec
      b(i,j) = [function of a(i,j)]
   ENDDO
!$acc update host(b(:,j)) async(j)
ENDDO
!$acc wait
!$acc end data
```

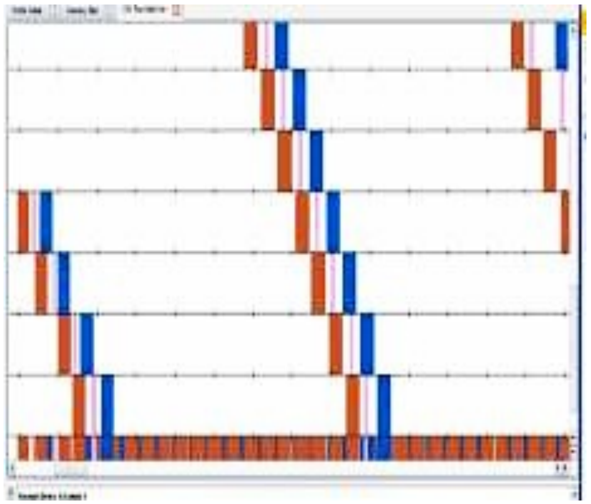
- can overlap 3 streams at once
- use slice number as stream handle
- don't worry if number gets too large
- OpenACC runtime maps it back into allowable range (using MOD function)

Execution times (on Cray XK6):

- CPU: 3.76s
- OpenACC, blocking: 1.10s
- OpenACC, async: 0.34s

NVIDIA Visual profiler:

- time flows left to right
- streams stacked vertically
- only 7 of 16 streams fit in window
- red: data transfer to GPU
- pink: computational on GPU
- blue: data transfer from GPU
- vertical slice shows what is overlapping
- collapsed view at bottom
- async handle modded by number of streams
- so see multiple coloured bars per stream (horizontally)



Recommendations for use of async

- **view it as part of performance tuning**
- **first implement synchronous code, verify it**
- **investigate bottlenecks:**
  - do the kernels need tuning?
  - do the data transfers need tuning?
- **look for data independencies**
  - across kernels
  - between kernels and host code
- **once you have the extent of independent regions, add asynchronous clauses / directives**
- **careful with `async` handles: only integers and easy to confuse if you need many different ones**
  - consider using e.g. named integer constants if reasonably descriptive naming is possible, e.g. to separate different sets of queues from one another

Use of data on GPU in libraries / CUDA

```
!$acc host_data use_device(var-list)
```

- **how to pass a pointer to memory on the GPU to a library, or to a CUDA kernel? E.g. to:**
  - use third party GPU library (e.g. Cray libsci_acc, cuBLAS, cuFFT, …) to process data already held on device
  - use optimized CUDA kernel to process data already held on device
  - use optimized MPI library to transfer data across nodes directly between the GPU memories
- **`host_data` makes a pointer on the device available on the host**
- **nested inside `data` region which put *`var-list`* on the GPU**

Interoperability with CUDA

```
PROGRAM main
  INTEGER :: a(N)
  [stuff]
!$acc data copy(a)
! Populate a(:) on device
! as before
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  [stuff]
END PROGRAM main
```

```
__global__ void dbl_knl(int *c) {
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- **Call CUDA-C wrapper (compiled with nvcc; linked with normal compiler)**
  - must include cudaThreadSynchronize()
  - Before: so asynchronous accelerator kernels definitely finished
  - After: so CUDA kernel definitely finished before we return to the OpenACC
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

Some useful tips at the end…

- **if in doubt, check the OpenACC standard**
- **focus on getting correct code on the GPU first**
- **then start optimizing**
- **focus on data transfers before aiming for a few percent improvement on a kernel**
- **on Cray systems, get detailed info about size of data transfers, kernels launched, etc:**
  environment variable `CRAY_ACC_DEBUG=2`
- **make efficient use of tools provided at your computing centre (e.g. DDT/totalview for debugging)**
  - it might take some time to ‘learn’ using the tool
  - but debugging complex code with printf will cost you much more time
  - same goes for performance analysis !