DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

DATA-MANAGEMENT DIRECTORY FOR OPENMP 4.0 AND OPENACC

Heteropar'2013

Julien Jaeger, Patrick Carribault, Marc Pérache

CEA, DAM, DIF F-91297 ARPAJON, FRANCE

26 AUGUST 2013

OPENACC INITIATIVE

PGI

CRAY
THE SUPERCOMPUTER COMPANY

- Directive-based approach for programming heterogeneous many-core hardware for C and FORTRAN applications.
- CAPS entreprise, CRAY, Nvidia and PGI initiative.
- Open Standard.
- API released in Nov 2011, second version finalized in June 2013.
- Compilers available for implementation: CRAY, PGI and CAPS (and accULL).
- www.openacc.org

OPENMP 4.O DIRECTIVES FOR ACCELERATOR

- Subgroup created in 2008 working on accelerator directives.
- Extend OpenMP programming model to accelerators
- Open Standard
- First technical report in Nov 2012, OpenMP 4.0 released in July 2013.
- Not available yet.
- www.openmp.org

COMMON PRINCIPLES

- Use directives to specify parts of the code to be executed on an accelerator.
- Data management realized through specific directives/clauses.
- Static allocation or simple cases: compilers may automatically find necessary tranfers.
- Most cases with dynamic allocation: data transfers must be specified by users.
- The scope of a directive with data clauses defines a *Data environment.*
- Data specified in data clauses are alive in the accelerator memory only in its data environment.
- Data environments can be nested.
- No partial overlap allowed.

DATA CLAUSES (1/3)

```
int a[100];

#pragma acc data create(a[0:100])
{
   compute_loop();
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*100);

   compute_loop();
```

```
#pragma acc data copyin(a[0:100])
{
   compute_loop();
}
```

```
acc_a = Allocate(ACC_MEM, sizeof(int)*100);
Transfer(h->acc, &a, &acc_a, sizeof(int)*100);
   compute_loop();
```

```
#pragma acc data copyout(a[20:100])
{
   compute_loop();
}
```

```
acc_a = Allocate(ACC_MEM, sizeof(int)*100);

   compute_loop();
Transfer(acc->h, &a[20], &acc_a, sizeof(int)*100);
```

DATA CLAUSES (2/3)

```
int a[100];

#pragma acc data copy(a[0:100])
{
   compute_loop();
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*100);
Transfer(h->acc, &a, &acc_a, sizeof(int)*100);
   compute_loop();
Transfer(acc->h, &a, &acc_a, sizeof(int)*100);
```

```
#pragma acc data present(a[0:100])
{
   compute_loop();
}
```

```
assert(acc_a is in the accelerator memory)

    compute_loop();
```

DATA CLAUSES (3/3)

- Other clauses for data transfer are derived combining the *present* clause with the other ones.
- ***Present_or_copyin():*** *copyin()* performed if data are not already on accelerator, otherwise using the accelerator data (without any transfer).
- ***Present_or_copyout() :*** same as *pcopyin()* but with *copyout().*
- ***Present_or_copy ():*** combination of *pcopyin()* and *pcopyout().*
- ***Present_or_create()*** : same as *pcopyin()* but with *create().*

OpenACC v2.0	OpenMP v4.0
#pragma acc data	#pragma target data
present_or_copy(list)	*map(tofrom: list)*
present_or_copyin(list)	*map(to: list)*
present_or_copyout(list)	*map(from: list)*
present_or_create(list)	*map(alloc: list)*

DATA PLACEMENT IN ACCELERATOR MEMORY

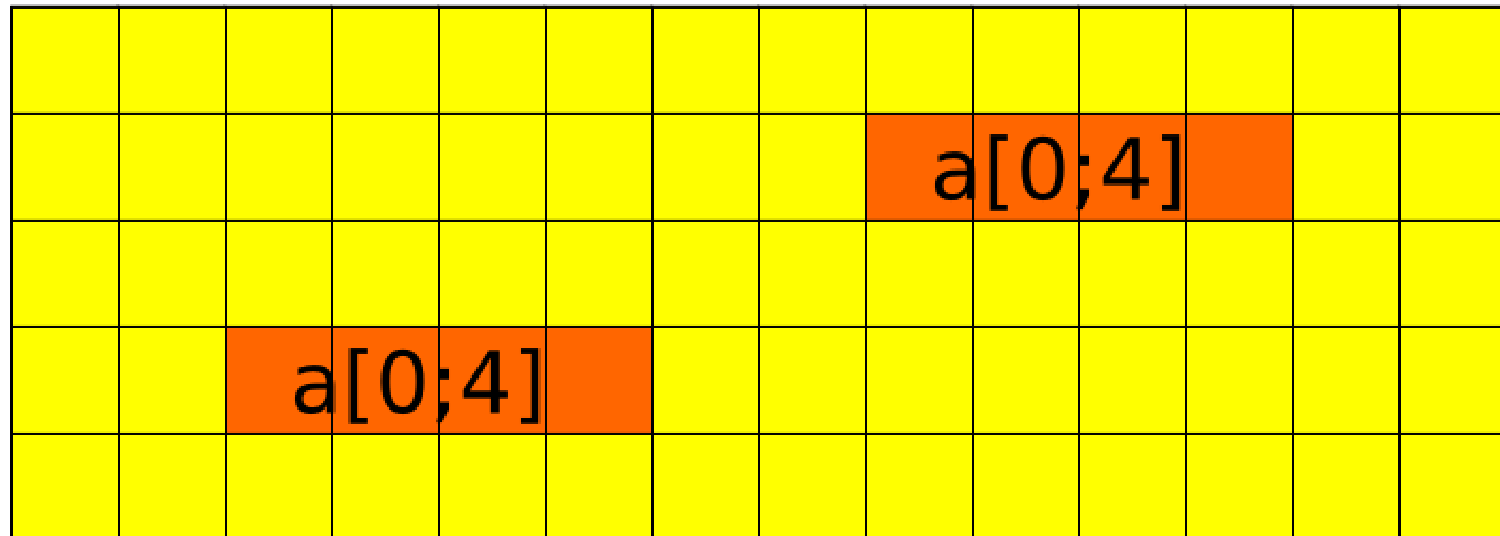
NESTED COPY ON SAME INTERVAL

```
int a[100];

#pragma acc data copy(a[0:4])
{
 #pragma acc data copy(a[0:4])
 {
   compute_loop();
 }
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a, &acc_a, sizeof(int)*4);
acc_a2 = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a, &acc_a2, sizeof(int)*4);
   compute_loop();
Transfer(acc->h, &a, &acc_a2, sizeof(int)*4);
Transfer(acc->h, &a, &acc_a, sizeof(int)*4);
```



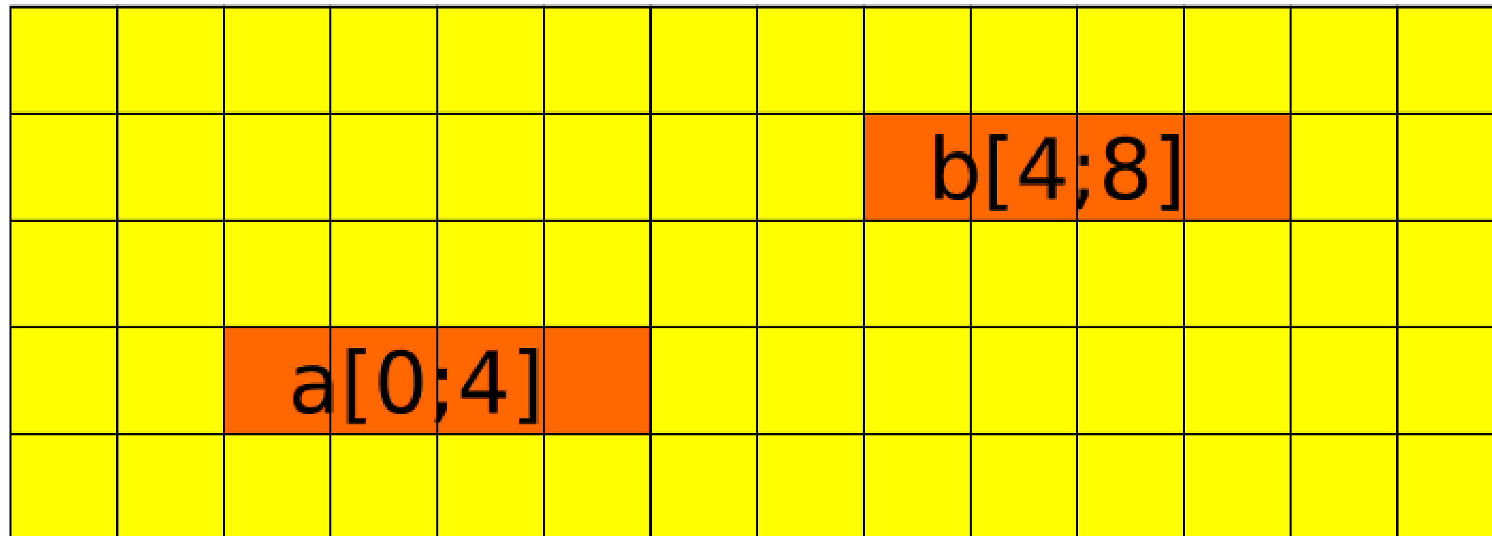
NESTED COPY ON DIFFERENT INTERVALS

```
int a[100];

#pragma acc data copy(a[0:4])
{
 #pragma acc data copy(b[4:4])
 {
   compute_loop();
 }
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a, &acc_a, sizeof(int)*4);
acc_b = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &b[4], &acc_b, sizeof(int)*4);
   compute_loop();
Transfer(acc->h, &b[4], &acc_b, sizeof(int)*4);
Transfer(acc->h, &a, &acc_a, sizeof(int)*4);
```



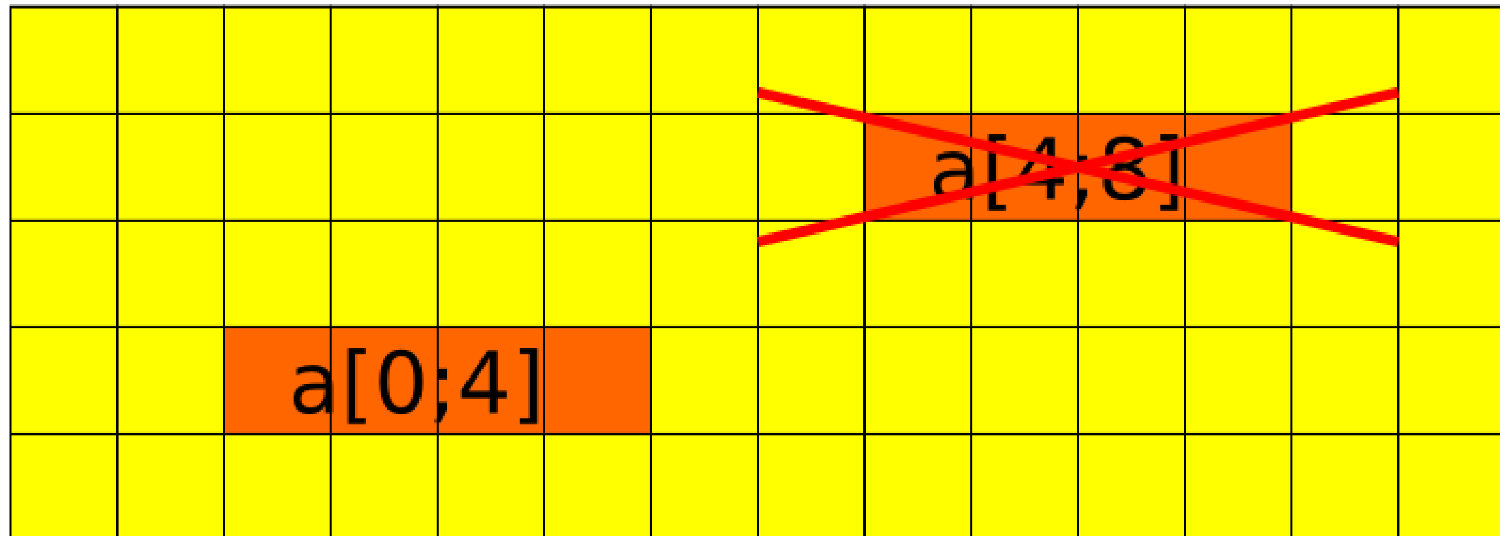
NESTED COPY ON SUBPARTS OF ARRAY

```
int a[100];

#pragma acc data copy(a[0:4])
{
 #pragma acc data copy(a[4:4])
 {
   b[0] = a[0] + a[4];
 }
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a, &acc_a, sizeof(int)*4);
acc_a2 = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a[4], &acc_a2, sizeof(int)*4);
    b[0] = a[0] + a[4];
Transfer(acc->h, &a[4], &acc_a2, sizeof(int)*4);
Transfer(acc->h, &a, &acc_a, sizeof(int)*4);
```



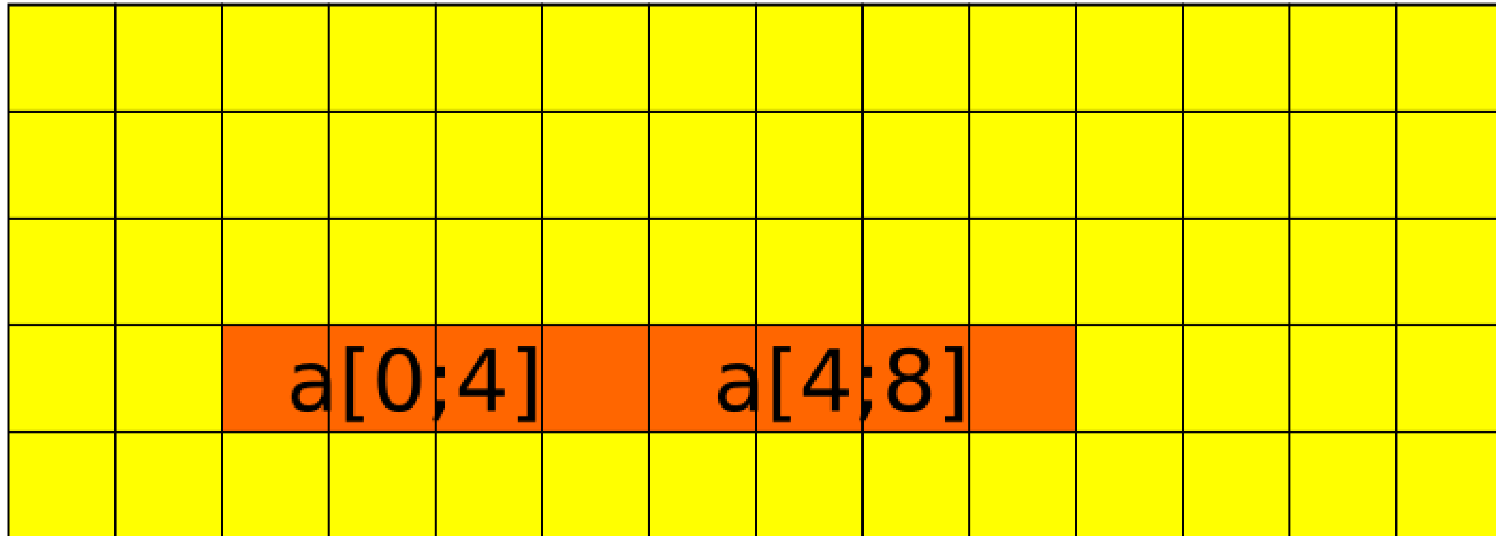
NESTED COPY ON SUBPARTS OF ARRAY

```
int a[100];

#pragma acc data copy(a[0:4])
{
 #pragma acc data copy(a[4:4])
 {
   b[0] = a[0] + a[4];
 }
}
```

```
int a[100];

acc_a = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a, &acc_a, sizeof(int)*4);
acc_a2 = Allocate(ACC_MEM, sizeof(int)*4);
Transfer(h->acc, &a[4], &acc_a2, sizeof(int)*4);
   b[0] = a[0] + a[4];
Transfer(acc->h, &a[4], &acc_a2, sizeof(int)*4);
Transfer(acc->h, &a, &acc_a, sizeof(int)*4);
```



DIRECTORY BEHAVIOR

DE LA RECHERCHE À L'INDUSTRIE

cea

PURPOSE OF THE DIRECTORY

- Automatically handle all the coherence:
  - Check subparts of same array.
  - Allocate another memory area to merge the subparts, only if necessary.
  - Data have to be copied to the newly allocated memory.
- If subparts are not adjacent, the empty space is still allocated.
  - For a subpart in empty allocated space, no reallocation.
- When closing a nested data environment, memory is deallocated.
  - Data declared in enclosing data environment should be copied back.

DE LA RECHERCHE À L'INDUSTRIE

cea

CONTRIBUTION

- Design of a directory to handle automatically memory coherence on accelerators when transferring subparts of same arrays alive concurrently.

- Handle complexities of memory management hidden behind directives for accelerators (OpenACC 2.0 and OpenMP 4.0).

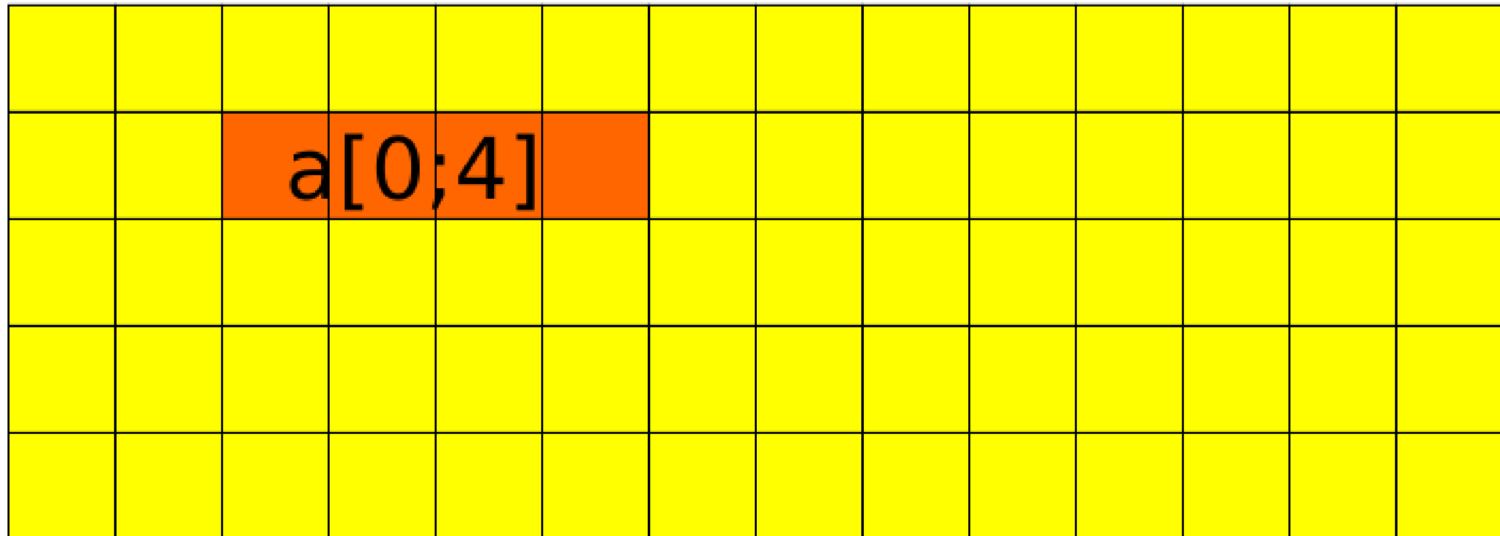
- Provide a simple ABI, mapped on the directives for accelerator, to use the directory.

NESTED COPY ON SUBPARTS OF ARRAY (1)

```
int a[100];

#pragma acc data copyin(a[0:4])
{
 #pragma acc data copyin(a[4:4])
 {
   compute_loop();
 }
}
```

- First allocation performed on device for first pragma.
- Data are transfered from the host to the newly allocated area on the device.

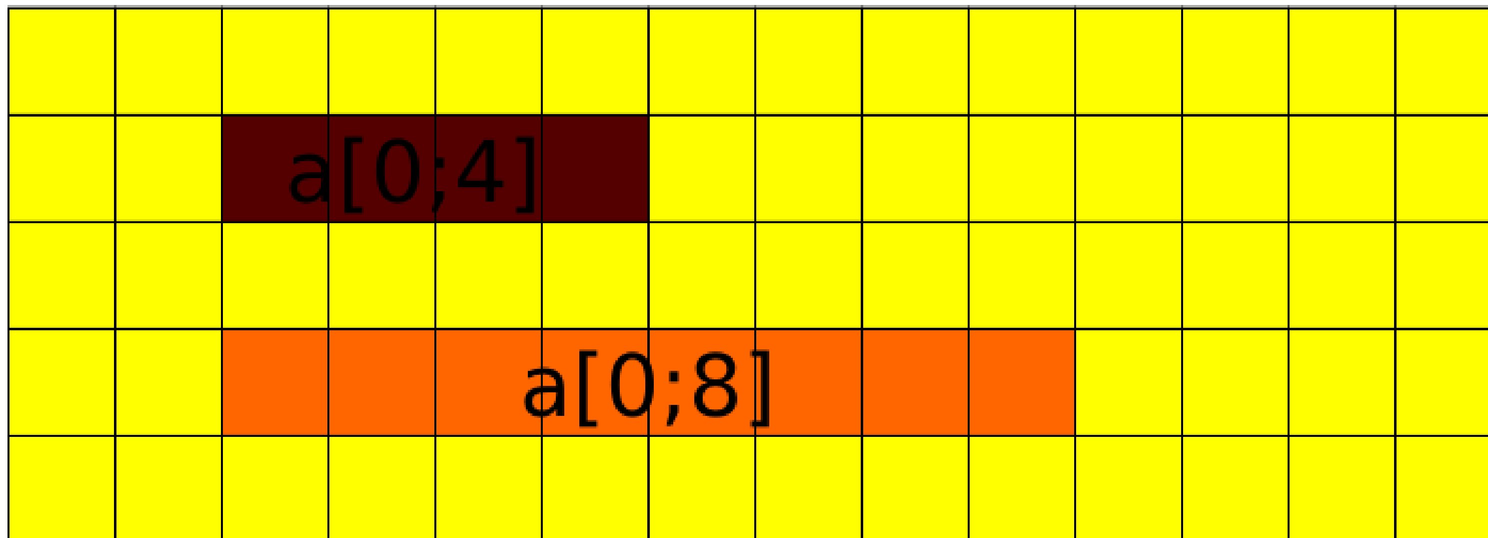


NESTED COPY ON SUBPARTS OF ARRAY (2)

```
int a[100];

#pragma acc data copyin(a[0:4])
{
 #pragma acc data copyin(a[4:4])
 {
   compute_loop();
 }
}
```

- The second pragma use data referenced by same basic pointer address as an enclosing pragma.
- Concerned data ranges are mapped together ➔ A new allocation for the whole range is performed.
- Data already present in device memory are transfered to newly allocated area.

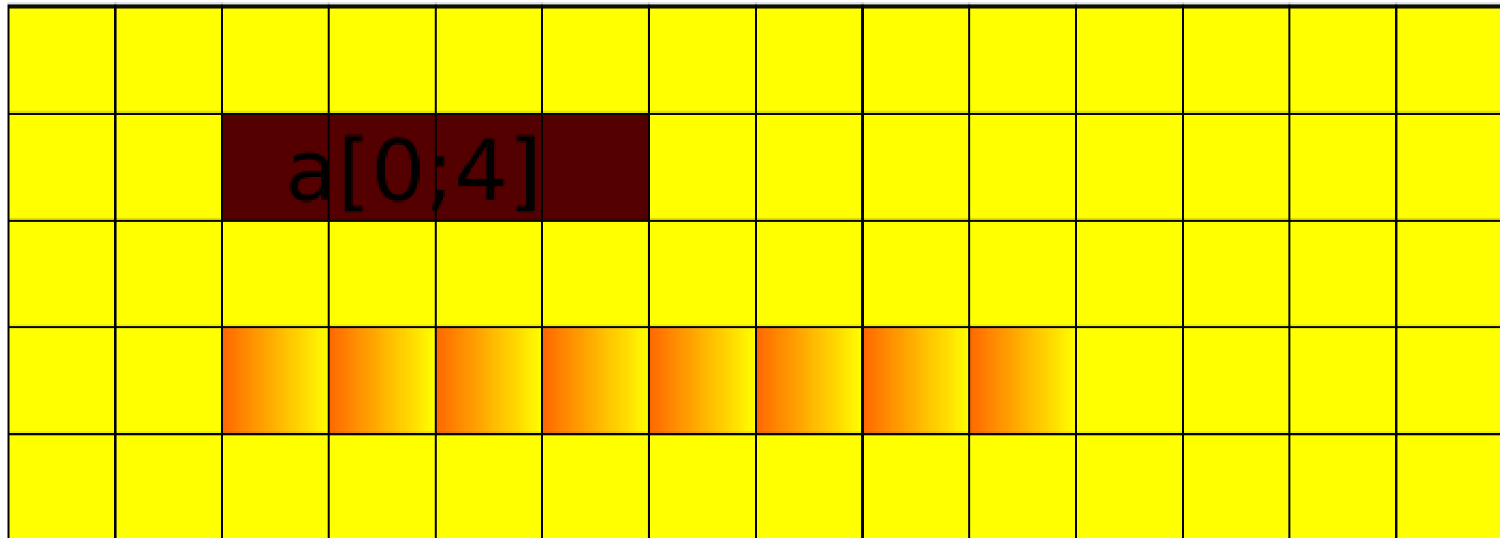


NESTED COPY ON SUBPARTS OF ARRAY (3)

```
int a[100];

#pragma acc data copyin(a[0:4])
{
 #pragma acc data copyin(a[4:4])
 {
   compute_loop();
 }
}
```

- No need for the whole range anymore ➔ it will be deallocated.
- Data from enclosing data environment might still be used.
- Necessary to copy back the updated data to the previous memory area before deallocating.

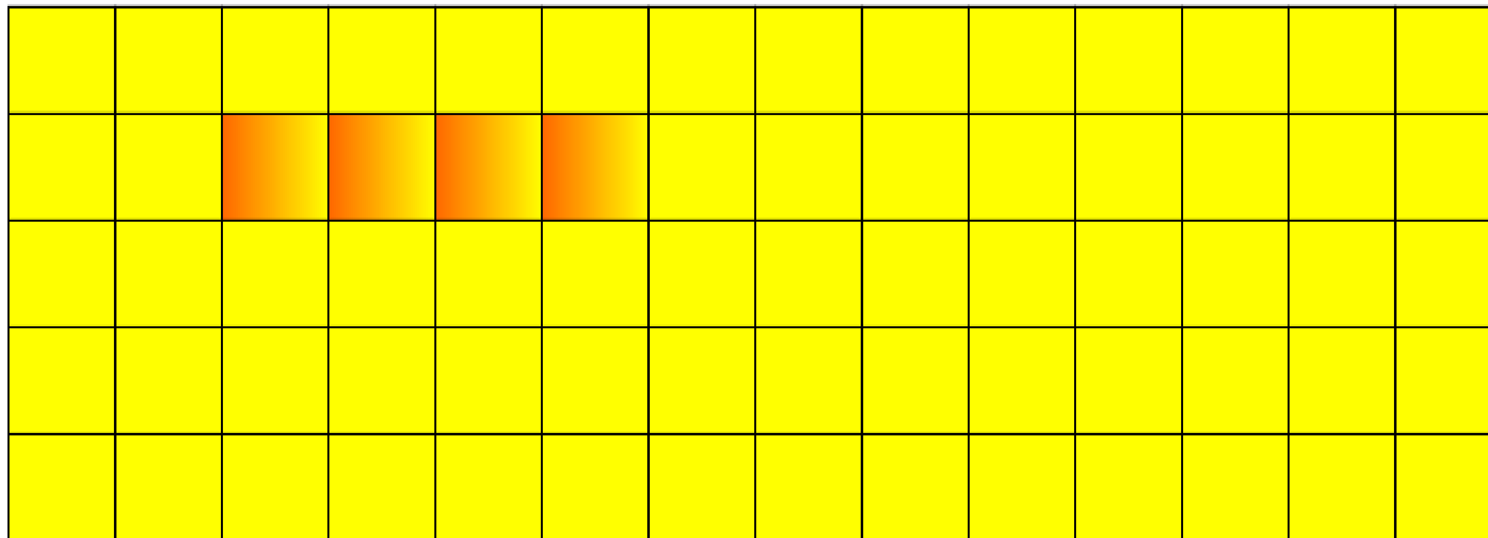


NESTED COPY ON SUBPARTS OF ARRAY (4)

```
Int a[100];

#pragma acc data copyin(a[0:4])
{
 #pragma acc data copyin(a[4:4])
 {
   compute_loop();
 }
}
```

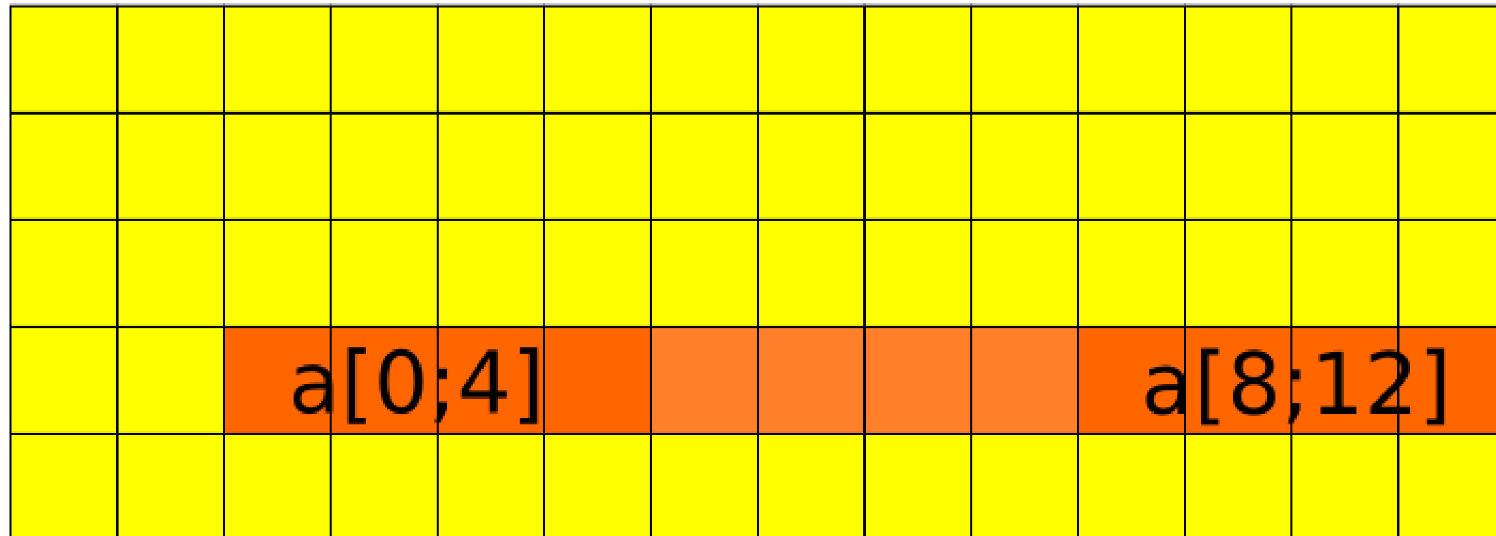
- The data is not referenced anymore.
- In the example, no need to copy back data range to the host memory.
- Memory space is just deallocated.



NESTED COPY ON ALIASED SUBPARTS OF ARRAY

```
#pragma acc data copyin(a[0:4],a[8:4])
{
 b = &a[4];
 #pragma acc data copyin(b[0:4])
 {
   compute_loop();
 }
}
```

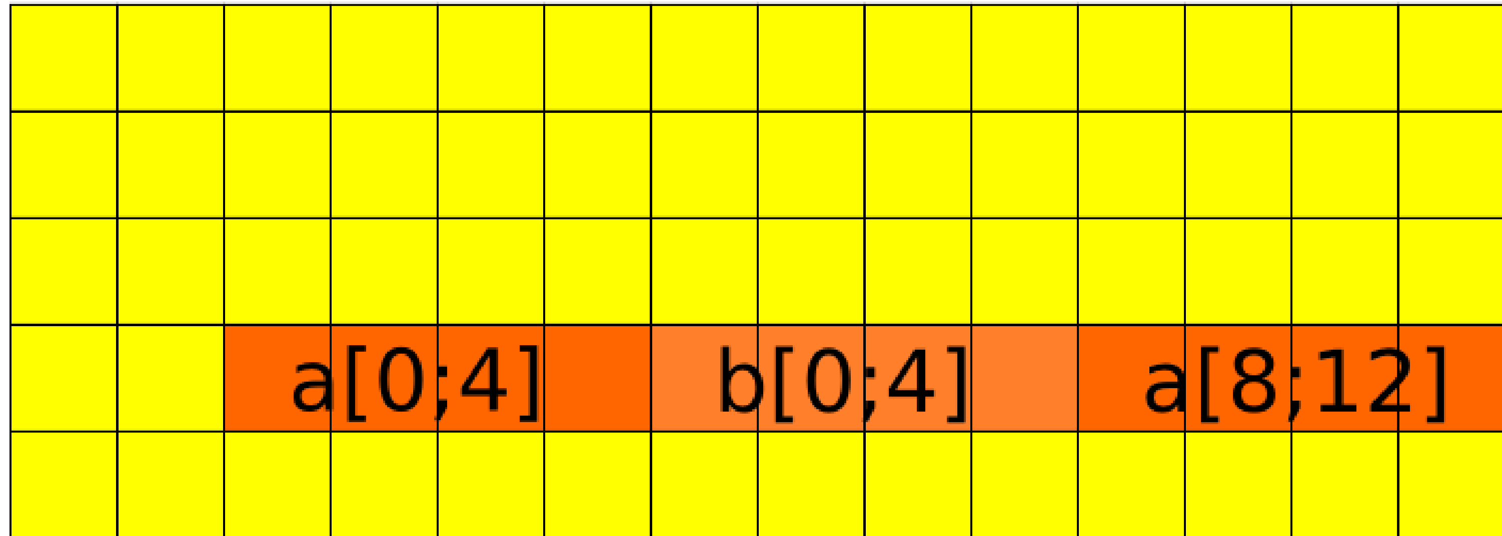
- The two ranges in first pragma are fused in one block ➔ one allocation covering the two ranges.
- Subpart of array is referenced with alias pointer.
- Overlap exists between allocated range and aliased data ➔ mapped together.



NESTED COPY ON ALIASED SUBPARTS OF ARRAY

```
#pragma acc data copyin(a[0:4],a[8:4])
{
 b = &a[4];
 #pragma acc data copyin(b[0:4])
 {
   compute_loop();
 }
}
```

- The two ranges in first pragma are fused in one block ➔ one allocation covering the two ranges.
- Subpart of array is referenced with alias pointer.
- Overlap exists between allocated range and aliased data ➔ mapped together.



DIRECTORY IMPLEMENTATION

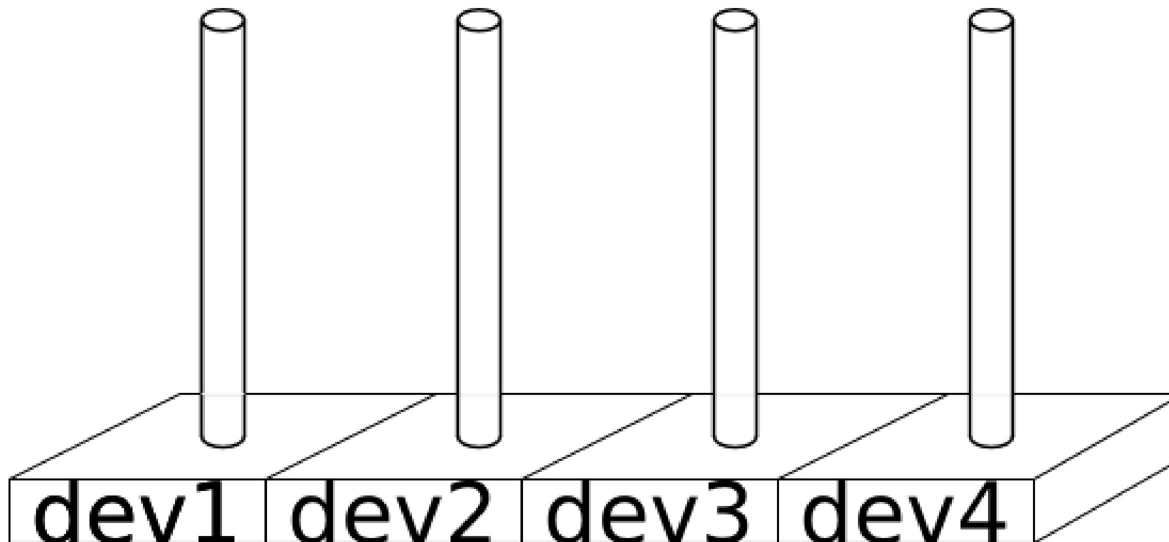
DESIGN OF THE DIRECTORY

- A stack is used to represent the nesting of data environments:
  - New data environment is the current one.
  - Exiting a nested data environment, fallback to the enclosing one.
- A data environment is composed of two lists:
  - List of *original tuples*: coming from the data clauses (if not present).
  - List of *fused tuples*: result of fusion between related clauses (same basic address or memory address overlap)
- The list of original tuples represent the transfers between host and accelerator.
- The list of fused tuples represent the memory to be allocated on the accelerator.

DIRECTORY BEHAVIOR ON SMALL EXAMPLE(1)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
 {
   compute_loop();
 }
```

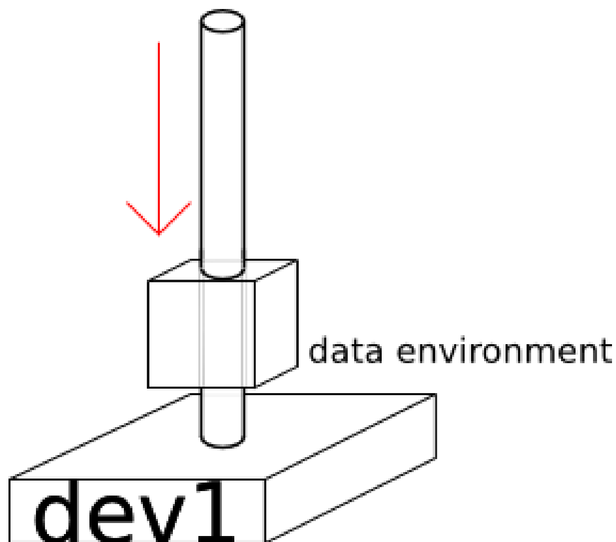
- The directory is structured as a set of stack.
- Each stack represents a device available.
- Data environment are stacked and unstacked for the selected device.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(2)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

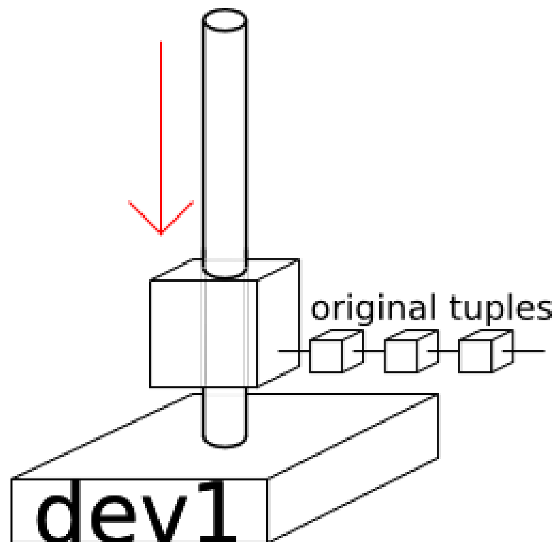
- When a pragma is read, a data environment is opened, and a new structure is stacked.
- Clauses are stored in original tuples.
- Tuples are checked then fused.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(2)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

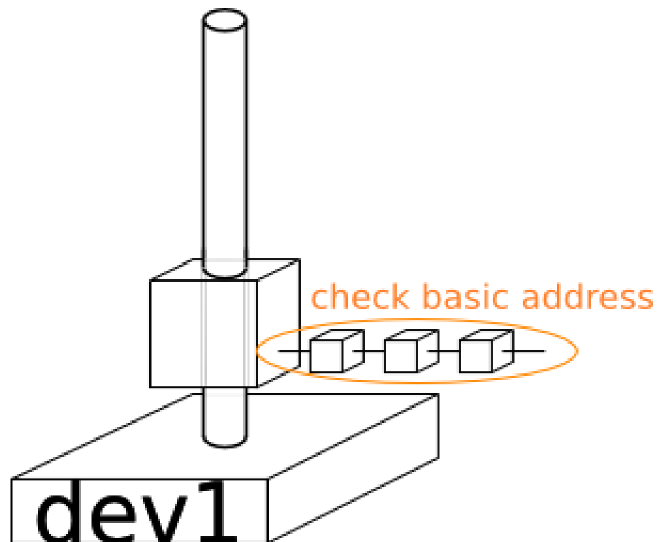
- When a pragma is read, a data environment is opened, and a new structure is stacked.
- Clauses are stored in original tuples.
- Tuples are checked then fused.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(2)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

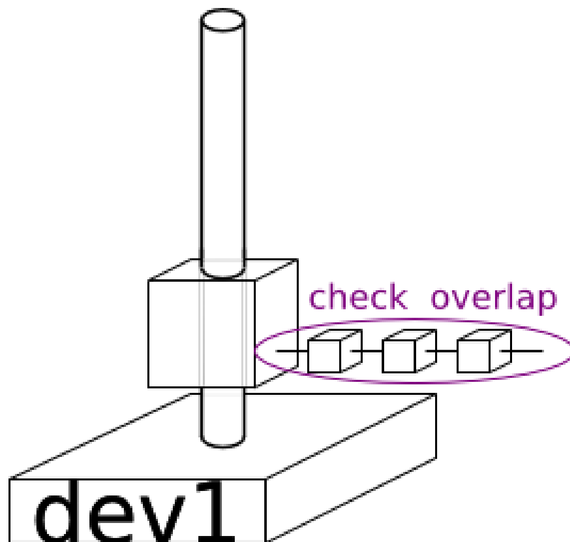
- When a pragma is read, a data environment is opened, and a new structure is stacked.
- Clauses are stored in original tuples.
- Tuples are checked then fused.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(2)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

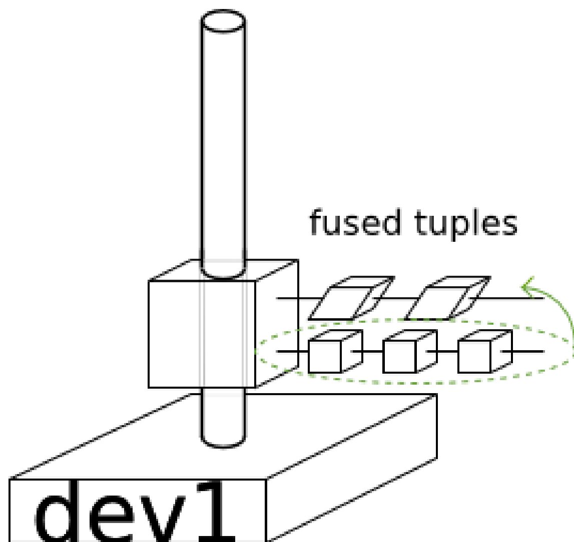
- When a pragma is read, a data environment is opened, and a new structure is stacked.
- Clauses are stored in original tuples.
- Tuples are checked then fused.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(2)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

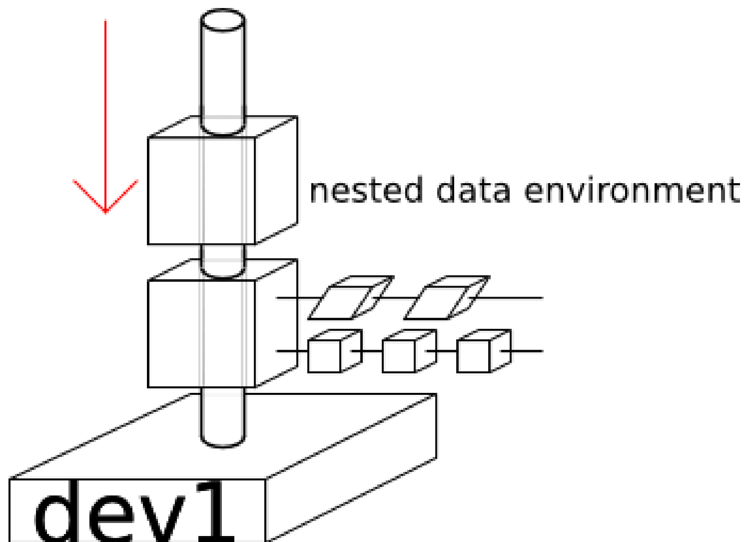
- When a pragma is read, a data environment is opened, and a new structure is stacked.
- Clauses are stored in original tuples.
- Tuples are checked then fused.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

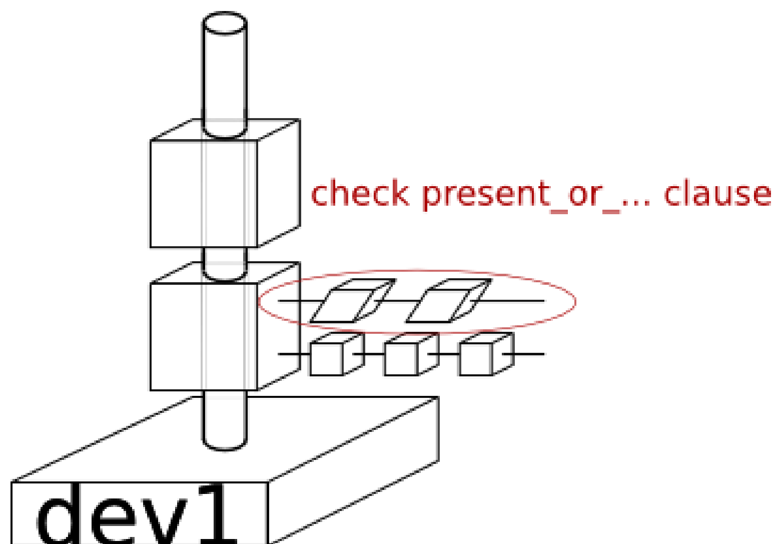
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occurs between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

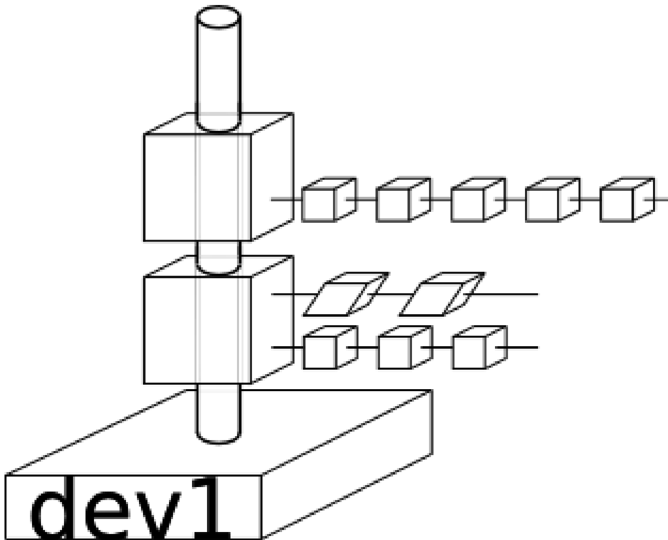
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occurs between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

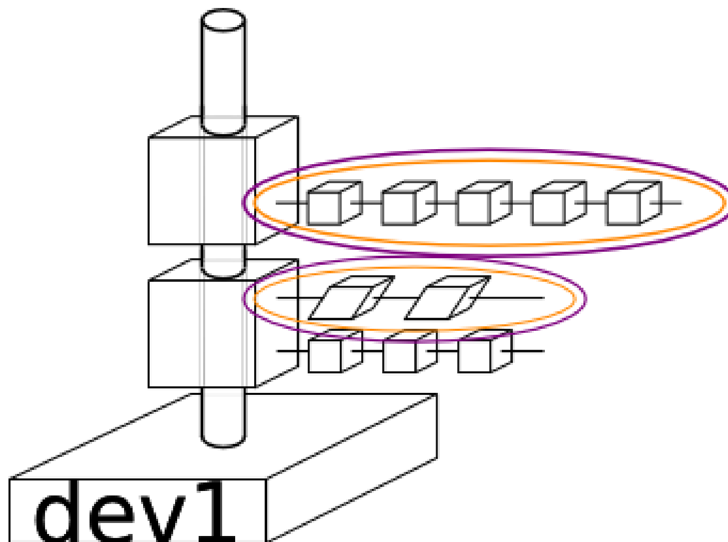
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occurs between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

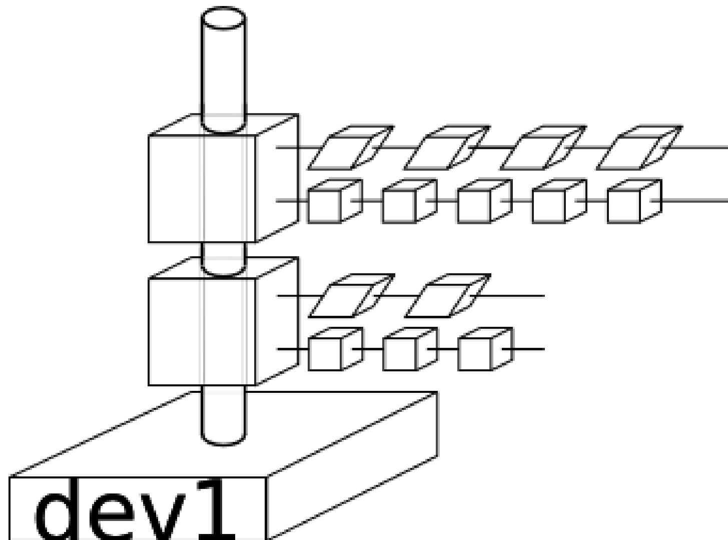
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occurs between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

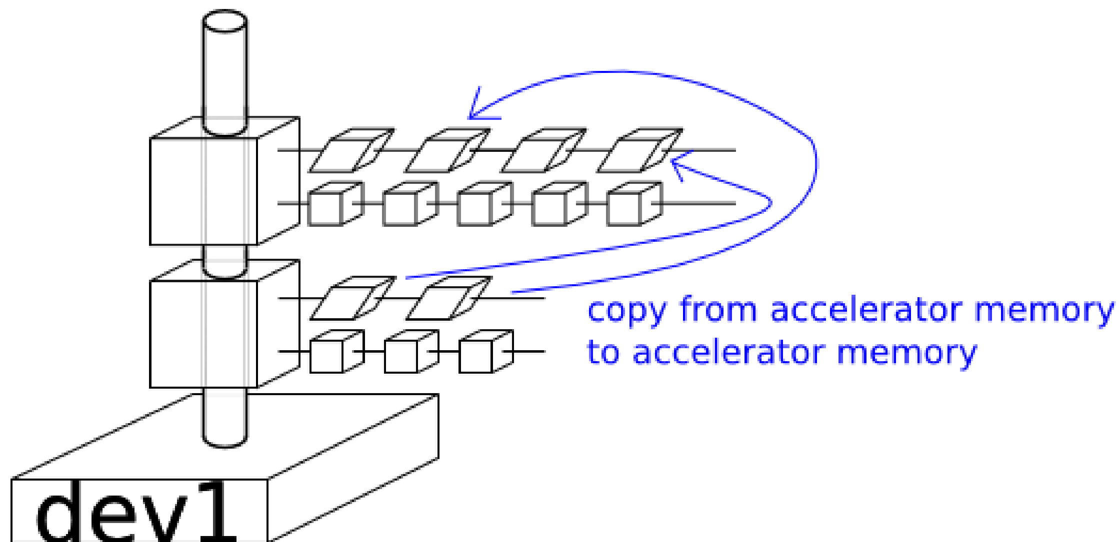
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occurs between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(3)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

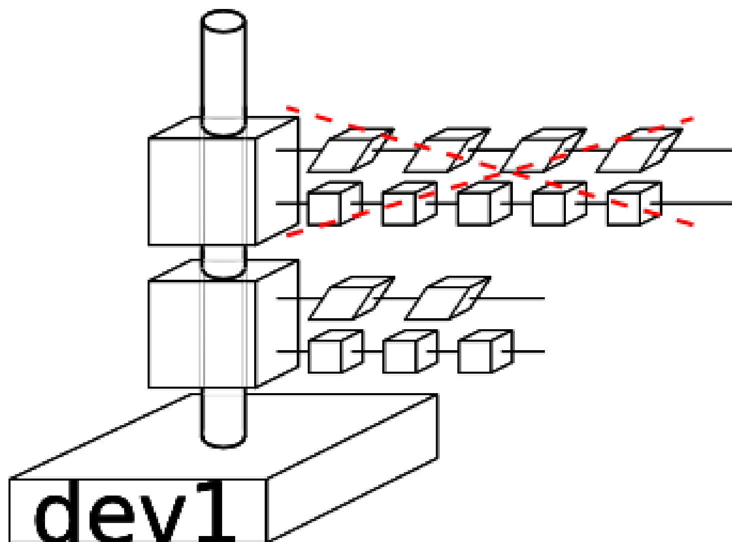
- Before creating original tuples, presence is checked in case of « present_or_... » clause.
- Fusion occures between original tuples and fused tuples of previous structure.
- Copy between previously allocated accelerator memory and the newly allocated area.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(4)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

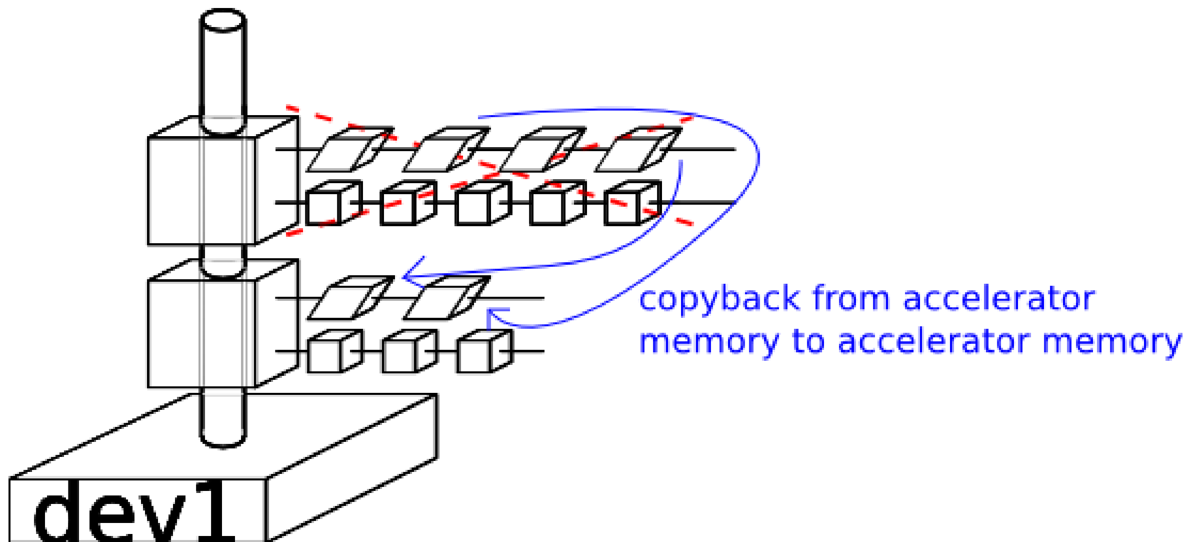
- When closing a nested data environment, concerned data are copied back to the previously allocated accelerator memory areas.
- Once transfers are performed, tuples are deleted.
- The structure is unstacked. The nested data environment is the new current structure.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(4)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

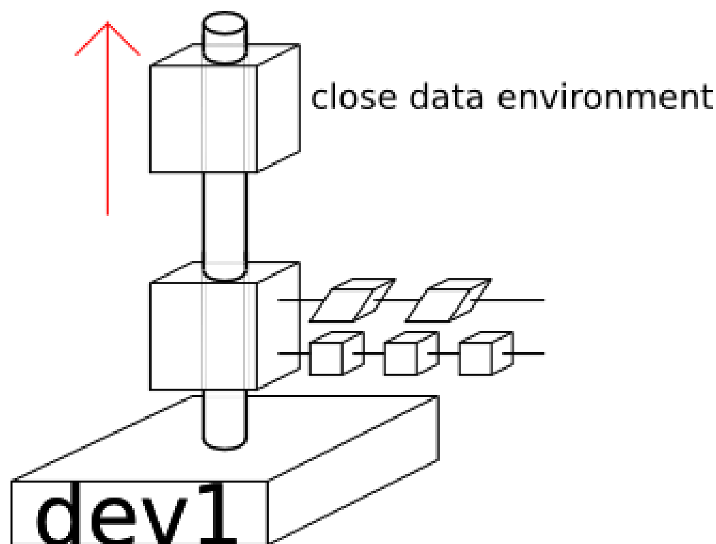
- When closing a nested data environment, concerned data are copied back to the previously allocated accelerator memory areas.
- Once transfers are performed, tuples are deleted.
- The structure is unstacked. The nested data environment is the new current structure.



DIRECTORY BEHAVIOR ON SMALL EXAMPLE(4)

```
#pragma acc data copyin(a[0:4],a[8:4],c[0:4])
{
e = &a[4];
#pragma acc data
copyin(a[12:4],b[0:2],c[4:8],d[8:8],e[0:4])
  {
   compute_loop();
  }
}
```

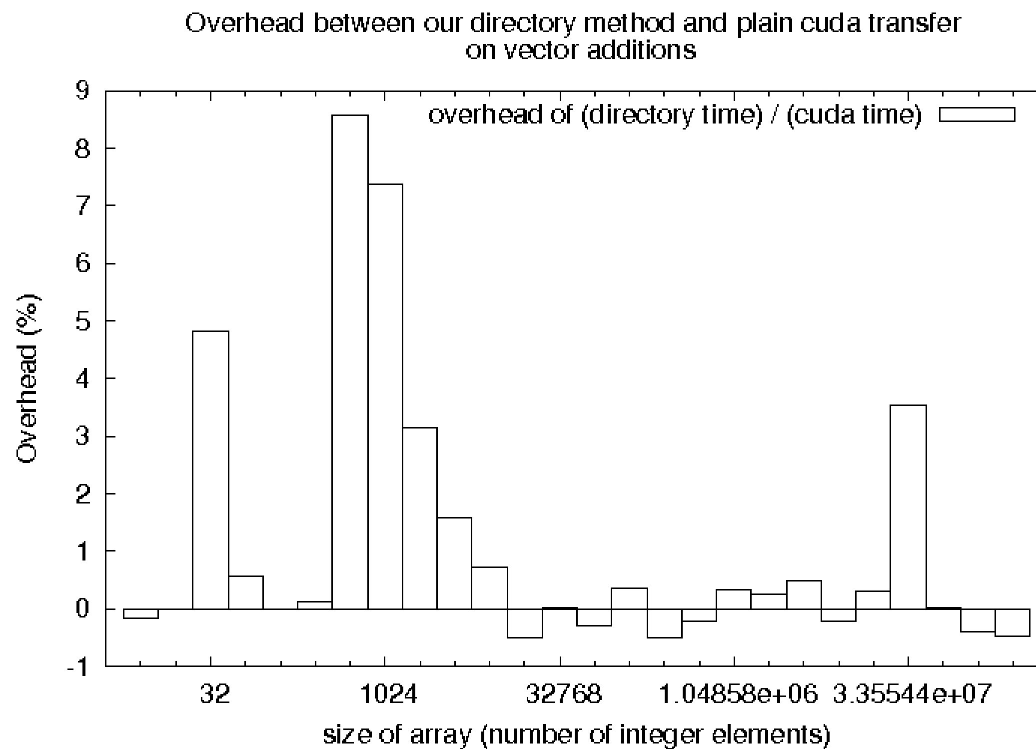
- When closing a nested data environment, concerned data are copied back to the previously allocated accelerator memory areas.
- Once transfers are performed, tuples are deleted.
- The structure is unstacked. The nested data environment is the new current structure.



RESULTS

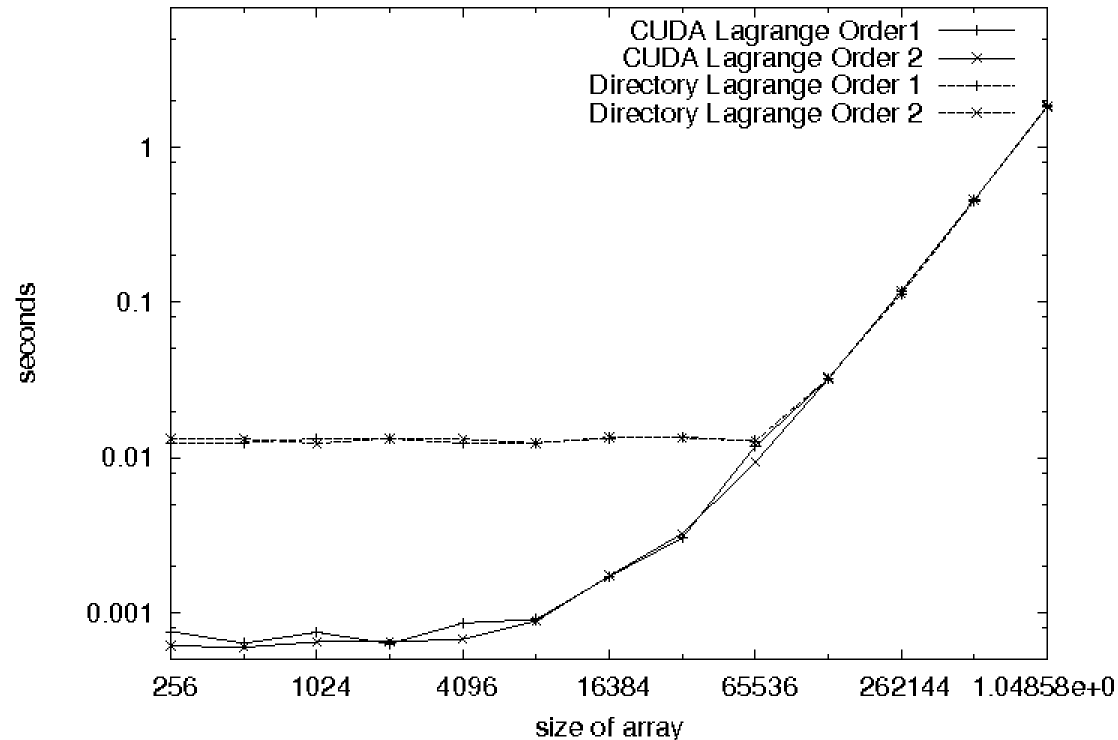
OVERHEAD MEASURED ON 3 VECTOR ADDITIONS

- Time overhead measured between plain cuda transfers and our directory.
- 3 vector additions with dependencies: 1st on the acc, 2nd on the host, 3rd on the acc.
- Overhead less than 10%, and less than 5% on large sizes.



OVERHEAD MEASURED ON HYDRODYNAMIC MINI-APP

- Same method to measure overhead, on real mono-material hydrodynamic mini-app: GAD.
- More arrays ➔ more element to parse in each list.
- Constant overhead of 12ms, negligible with sizes >65536.



CONCLUSION

CONCLUSION

- Automatically managing transfers between host and accelerators is not hard...
- …but induced memory coherence is more challenging.
- Directives for accelerator need this coherence to merge subparts of arrays transfered separately but alive concurrently.
- To map subparts of arrays, basic address checking may not be enough.
- Aliasing forces to check overlap between allocated ranges.
- Our directory handle the problem with a simple ABI mapped on the different pragmas.
- Our overhead is not important on simple cases, but should be more deeply evaluated.

FUTURE WORKS

- Deeper evaluation of our implementation.
- Support more runtime (opencl backend in progress…).
- Optimization of algorithms / implementation.
- Robustness

Q & A

DIRECTORY ABI

- Simple ABI to use the directory.
- ABI design mapped on directives, especially data clauses.
- Functions arguments are the same than clauses (pointer, first element, size, *sizeof*)

OpenACC v2.0	OpenMP v4.0	ABI
#pragma acc data	#pragma target data	new_dataE(); read_pragma(); … write_pragma();
clauses *pcopy, pcopyin, pcopyout, pcreate*	**Clauses** *map(tofrom, to, from, alloc)*	*pragma_pcopy(); pragma_pcopyin()…*
clauses *copy, copyin, copyout, create*	No correspondance in OpenMP 4.0	*pragma_copy(); pragma_copyin()…*
#pragma acc update	#pragma target update	*pragma_update();*
End of *data environment*	End of *data region*	*end_dataE();*

NESTED COPY ON SUBPARTS OF ARRAY

```
int a[100];

#pragma acc data copy(a[0:4])
{
 #pragma acc data copy(a[4:4])
 {
   b[0] = a[0] + a[4];
 }
}
```

```
int a[100];

new_dataE();
  read_pragma();
    copy(&(a[0]), 0, 4, sizeof(a[0]));
  write_pragma();

  new_dataE();
    read_pragma();
      copy(&(a[0]), 4, 4, sizeof(a[0));
    write_pragma();

    a_acc = getDeviceAddress(&(a[0]));

    b[0] = a[0] + a[4];

  end_dataE();
end_dataE();
```

DE LA RECHERCHE À L'INDUSTRIE

cea

PARTIAL OVERLAP

- Not allowed in the specifications.
- Suggestion: in case of partial overlap, separate data interval in mulitple ranges.
- Each ranges is either totally included or totally excluded of englobing clauses.
- For parts not already on the accelerator, apply same coherence as array subparts.
- For parts already in the accelerator memory, specify a behavior:
  - Use the data already on the accelerator: *present_or_* behavior.
  - Update data with host value: *update* behavior.
- We can have new pragmas: *present_and_copy* and *update_and_copy*.

Commissariat à l’énergie atomique et aux énergies alternatives
Centre de Saclay | 91191 Gif-sur-Yvette Cedex
T. +33 (0)1 XX XX XX XX | F. +33 (0)1 XX XX XX XX

Etablissement public à caractère industriel et commercial | R.C.S Paris B 775 685 019

Direction
Département
Service