CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–12
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Fine-grain Data-management Directory For OpenMP 4.0 And OpenACC

Julien Jaeger[1], Patrick Carribault[1] and Marc Pérache[1]

[1] *CEA, DAM, DIF, F-91297, Arpajon, France*

SUMMARY

Today's trend to use accelerators in heterogeneous systems forces a paradigm shift in programming models. The use of low-level APIs for accelerator programming is tedious and not intuitive for casual programmers. To tackle this problem, recent approaches focused on high-level directive-based models, with a standardization effort made with OpenACC and the directives for accelerator in the latest OpenMP 4.0 release. The pragmas for data management automatically handle data exchange between the host and the device. To keep the runtime simple and efficient, severe restrictions hinder the use of these pragmas. To address this issue, we propose the design for a directory, along with a reduced runtime ABI, to handle correctly data management in these standards. A few improvements to our directory allow a more flexible use of data management pragmas, with negligible overhead. Our design fits a multi-accelerator system. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Heterogeneous directory, OpenMP 4.0, OpenACC, Automatic tranfers, accelerator memory coherence

1. INTRODUCTION

In the last decade, High Performance Computing moved towards multi-core and many-core architectures. Scientists must not only deal with all the multiple parallelism levels, but also with the programmability difference of available accelerators. While some languages are specific to an accelerator or constructor (e.g., NVIDIA's CUDA), efforts have been made to offer an interface to write codes for CPUs and GPUs in a common way, first with the OpenCL language, and then with the use of directives, allowing the user to add pragmas in their code to work on accelerators.

Several sets of directives for accelerators (CAPS Entreprise's `hmpp`, PGI directives among others) were merged to provide a more generic set through the OpenACC API and the directives for accelerator in the OpenMP 4.0 API. Usually when programming with accelerators (including GPUs) one has to manage data in memory very carefully. No coherence is implemented between the host and the accelerators, and sometimes also inside an accelerator. This was changed by the use of directives to handle computations on accelerators. Different actions are triggered whether the concerned data are already allocated on the accelerator or not. Hence, there is a need to store the information about the data transferred to an accelerator. However, to maintain a simple and efficient runtime for data tracking and transfer, some restrictions are defined in the two APIs. One of the

*Correspondence to: E-mail: julien.jaeger@cea.fr main restrictions is the impossibility to specify different subsets of the same array in one directive or in nested data environments.

In this paper, we propose a directory scheme to handle the memory management of the two APIs. Some extensions were added to the directory to allow more flexibility when handling subarrays in the data clauses. With these modifications, one can use multiple parts of the same array in nested data environment, keeping the coherence in the accelerator memory between all the subparts. Our directory is designed to work with several accelerator devices attached to the same host.

This paper is a revised and extended version of a previous one presented at the *HeteroPar* workshop [12]. The paper is organized as follows: in Section 2, we detail the data management and its restrictions in the two APIs. Our extensions are described on three motivating situations in Section 3. Section 4 presents the related work around directories and directives for accelerator programming. The design of our directory and the different algorithms implemented in the runtime are described in Section 5. In Section 6, we present our newest contribution to our directory, which is the support of directives for unstructured data lifetime proposed in OpenACC 2.0. Section 7 presents overhead results of our implementation, including comparison to a vendor OpenACC implementation, along with some performance improvements due to reduce transfers. We conclude in Section 8.

2. MEMORY MANAGEMENT

In the APIs, the user specifies which data should be available on the accelerator in directives. Through different clauses, one can choose the action to be performed on each data. Each API has its own clauses, and its own restrictions on the use of the clauses and directives. In this section, the data management part of two APIs are described, along with their specific restrictions.

OpenACC 2.0. The OpenACC API 2.0 was released in July 2013. To the original directives for data management were added directives for unstructured data lifetime. Since the first version of the API, a data environment was defined by the scope of a `data` construct (directives `acc data`, or `acc parallel` and `acc kernel` if data clauses are involved). A `data` construct includes several data clauses to specify if the data need to be transfered from, or to, the host (*copyin*, *copyout*), both from and back to the host (*copy*), are already present (*present*) or need to be allocated on the accelerator (*create*). More complex clauses fuse the behavior of the present clause with the four other. If the data are in the accelerator memory, they will be used without additional treatment. Otherwise, the concerned data are allocated and/or transfered from/to the accelerator memory (*pcreate*, *pcopyout*, *pcopyin*, *pcopy*). Data specified in clauses are alive only in the scope of their data environment.

If one need to update accelerator data with the values on the CPU inside a data environment, a stand-alone directive `update` allows to update data which are already defined on the accelerator.

In the version 2.0 of the API were introduced directives for unstructured data management. With these directives, the scope of the specified data is not statically defined by the scope of the directives, but dynamically with the insertions of `enter data` and `exit data` directives. A data in `enter data` is alive and valid on the accelerator until the end of the program, or until reaching a directive `exit data` with the same data. Hence, the data lifetimes are not perfectly nested, and might overlap some structured data lifetimes. A directive `exit data` can only end the data lifetime from a directive `enter data`. It can not interact with the data specified in structured data management directives.

OpenMP API 4.0. OpenMP 4.0 specification including directives for accelerators was released in July 2013. One can understand from the document that data management is nearly the same as with OpenACC. The bounds of a *target data* construct defines a *data environment*. A *target data* construct will use the clause *map* with states to specify the data management between the host and

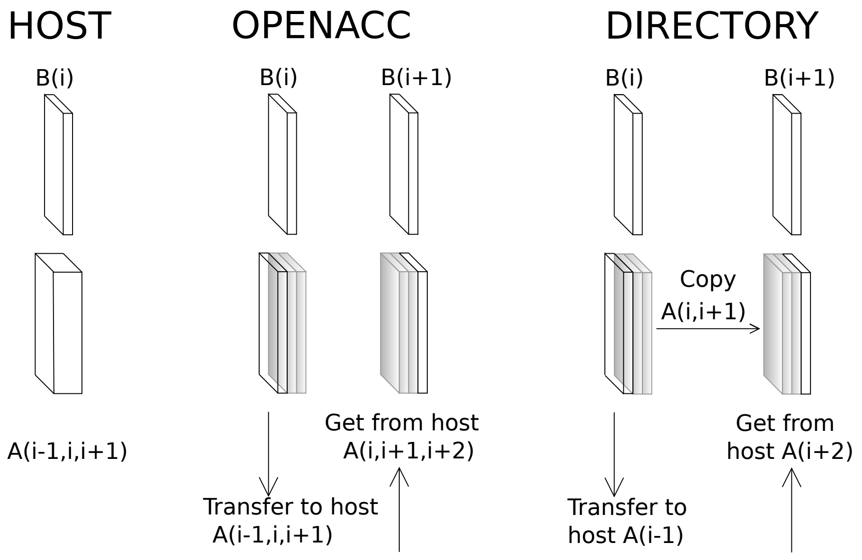


Figure 1. Necessary transfers to compute a plane in a 3D 7-point stencil.

the accelerator. The different states (*alloc*, *to*, *from*, *tofrom*) have the same behavior as some clauses in OpenACC (respectively *pcreate*, *pcopyin*, *pcopyout*, *pcopy*). So far, the OpenMP 4.0 specification does not contain unstructured data management, and has no directives allocating and transferring a data regardless to its presence or absence in the accelerator memory.

Restrictions: To handle data safely in the directives, some restrictions are issued in the APIs.

For OpenACC, a present clause in a nested data environment can check only the existence of a subarray in the accelerator memory if the subarray is a valid subset of the subarray specified in the enclosing data environment. No partial overlap is allowed. In OpenMP 4.0, this restriction is even stronger as the two subarrays must be the same.

For all the clauses checking if the data already exist in the accelerator memory, two distinct parts of the same array cannot be specified in nested data environment if they are referenced with the same pointer or if they overlap. Also, in a directive, a variable can appear only once.

With all these restrictions, the user is not able to use several distinct subsets of the same array in a nest of data environment. Our directory aims to tackle these restrictions to allow more flexibility when dealing with subarrays in data environments.

In the following of this paper, we use OpenACC terms to refer to the two APIs. For example, *pcopyin* will refer to *pcopyin* in OpenACC and *map(to)* in OpenMP.

3. MOTIVATING SITUATIONS

Our directory is motivated by one main extension: being able to correctly deal with different subparts of the same array spread across nested data environments. This extension rises another concern: if two subparts of the same array are mapped together in the accelerator memory, what about an alias pointer referencing an address between these two subparts? Finally, we present a motivating example on a 3D stencil showing the transfers that can be saved with our directory.

3.1. Subparts of the same array.

Considering the two APIs, when dealing with nested data environments, it is not possible to specify subparts of the same array in the same directives, or in nested data environment. This can hinder the use of the directives, as with nested calls of functions in a real-life simulation program with thousands of lines of code, it might be very difficult to know if the different subparts needed for the kernel have already been transferred. The easy solution is to completely allocate and/or transfer the whole array when the first subpart is required. However, it can use a huge part of the accelerator memory, and possibly for a long time. If the memory is a critical resource, it can be useful to allocate the different subparts of the array only when they are required in the nesting of data environment. To have a correct computation, the pragmas related to the same memory zone have to be matched together inside a common memory area. This removes the constrain on the use of subparts of the same array in the same directory, or in nested data environments.

To be thorough, we will now deal with the case when an alias pointer can be used in a clause to specify a subpart of a data range already declared in an enclosing data environment. The two pragmas should be matched to not reallocate and transfer the data already present on the accelerator. However, the memory area pointed by the alias pointer may not overlap other transferred data, but may point to an area between two other subparts of the same array using the same base address. To gather the three subparts in the same allocation, it is necessary to fuse all the intervals with the same base address first, before check the covering range of intervals without the same basic addresses.

3.2. Motivating example

If one consider a 7-point 3D stencil, the six direct neighbours are required to update a cell. Thus, to completely update a plane in the 3D stencil, the two surrounding planes are required. The left part of Figure 1 displays this behavior in an example where two arrays are used, one for the previous computed stencil iteration, and one for the current iteration. When a very large stencil is computed on an accelerator, the whole array may not fit in memory. Then the computation is performed in parts, with only the corresponding data transferred on the accelerator device.

Since it is not possible with OpenACC and OpenMP 4.0 to transfer only subparts of the same array, to compute a whole plane required for three planes to be pulled from the host memory. Once the current plane is updated, then the three planes are forgotten. A new triplet of plane is then loaded to the accelerator memory to perform the next slice of the stencil, as shown in the middle of Figure 1. However, of those three planes, two could be reused for the update of the next slice.

With our directory, one can only transfer the elements not already present in the accelerator memory, and reuse the data brought by the previous computations. Some copies happen in the accelerator memory, but fewer data are transferred between the host and the device (in this example, only a third of the transfers required with OpenACC still occur with our directory). This behavior is displayed in the right part of Figure 1.

4. RELATED WORK

Directory-based coherency protocols emerged in the late 70's as a concurrent to *snoopy cache* protocols. Tang *et al.* [25] and Censier *et al.* [4] set the base of the directory-based protocols ([5]). The latest proposed a *Full-map directory*, which stores for each block in global memory its status in every caches. Later approaches aimed to reduce the memory footprint of this full-map directory [1, 6]. The multiplication of cores and caches on the same node brought new interest to directory-based cache coherency. Studies were performed to work on coarse-grain tracking to identify group of blocks which do not require coherence protocol [3, 18, 8, 28], allowing some trade-off between performance and accuracy.

Caches protocols were previously used on GPUs for performance issues. A broad range of codes uses a software cache to improve their execution time, from a simple sum-products [23] to cardiac cell modeling [14]. Some of the most known frameworks optimizing and scheduling codes on GPUs use some software cache to have better performance. The frameworks looking for work on all available devices, like XKaapi [9], StarPU [2], FLAME [20] or StarSs [19], often rely on directory schemes to keep the data coherency between the different memories. However, these frameworks check and transfer blocks of data that will be ultimately used by a device. If only a subpart is available in the device memory, the whole block will still be updated, as no scheme allows to maintain the data coherency in a single memory between several transfers on related data. Moreover, the data decomposition is handled by the runtime and the user can not specify a finer granularity.

If the OpenACC standard release was praised, it spawned very few research papers on the subject [13, 26, 22]. To the best of our knowledge, the only open implementation for OpenACC that can be found in scientific publications is `accULL`, and neither the `accULL` publications nor the vendor implementation and documentation of OpenACC compilers mentioned how the memory allocations and transfers are handled.

The firsts tests with the OpenACC standard showed promising results. If the performance figures of OpenACC codes did not catch-up with those of CUDA or OpenCL codes, the effort-performance ratio of OpenACC has been greatly highlighted [26, 11, 7]. Since, OpenACC has been used to optimize simulation programs [13, 17, 7], sometimes coupled with other runtimes (OpenMP [27]). With the spread of users, numerous studies comparing the different existing compilers for OpenACC were issued [21, 15, 16], while academic source-to-source compilers appeared [22, 24]. However, none of these studies or implementations discuss the restrictions embedded in the OpenACC API.

Our work was inspired by the directory-based coherence protocols. If keeping the data location between several devices was already shown useful to improve performance, the idea of some sort of data coherence on a single accelerator memory is rather new and is induced by the directives for data management on recent APIs. The automatic treatment of the copy needs fundamental ideas of directory-based coherence protocol: which data are on which accelerators, and with which state.

5. DIRECTORY IMPLEMENTATION

The purpose of the directory is to store the information of all host data located on accelerators. These data intervals are represented by tuples in our directory. The tuples are the basic block of the directory, embedding all the necessary informations for a given data interval (length, element size, host and accelerator memory pointers, state).

In its design, the directory needs to handle correctly the opening and closure of data environment for all available accelerators. When a new data environment opens, new clauses will be read, and new tuples will be added at the top of the chosen accelerator handler. As we described earlier in Section 2, some fusion between intervals might be necessary before any allocation and copy. All the actions to be performed when entering a new data environment are detailed in Section 5.2. When closing a data environment, all the clauses embedded in the current data environment should be closed and erased. This procedure is described in Section 5.3.

5.1. Directory design

We designed our directory as a table of stacks, one for each accelerator. The stack allows to represent easily the nesting of data environments, as the newest data environment is always the current one, and we should revert to the enclosing one when the current one is closed. A data environment is represented as a level of the stack, which is composed of two list of tuples. The first list of tuples are the tuples created directly from the incoming clauses of the current data environment. They are called *original tuples*. The second list stores the final tuples after applying the different fusion between the *original tuples* and the already existing tuples. We refer later to these tuples as *artificial tuples*. Figure 2 displays a representation of our directory.

In the tuple, the host interval is stored using four values: its basic pointer (a_h), its first element (elt), its length and the size ($size$) of an element. To map it quickly to its accelerator memory, the tuple also embeds the accelerator address of the first element of the interval (a_{acc}). Hence, when copying the first element of an interval from the host to the device, one has to copy the data from the location $a_h + elt * size$ to the address a_{acc} on the accelerator . Copying the data directly to a_{acc} avoid the unnecessary allocation of element a_h to $a_h + elt * size$ on the accelerator memory.

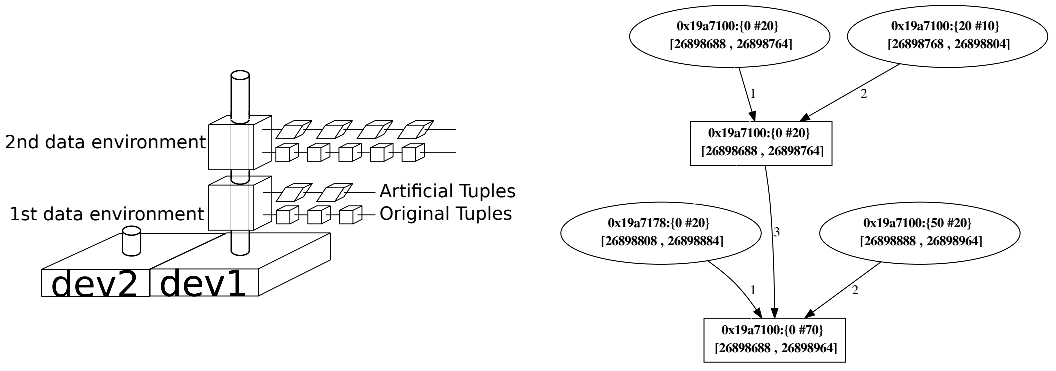

2nd data environment
1st data environment
Artificial Tuples
Original Tuples
dev2
dev1
0x19a7100:{0 #20} [26898688 , 26898764]
0x19a7100:{20 #10} [26898768 , 26898804]
1
2
0x19a7100:{0 #20} [26898688 , 26898764]
3
0x19a7178:{0 #20} [26898808 , 26898884]
0x19a7100:{50 #20} [26898888 , 26898964]
1
2
0x19a7100:{0 #70} [26898688 , 26898964]

Figure 2. **Left**: Schematic representation of the Directory with two devices, and two nested data environment on the first device. The first data environment receives three original tuples, which are fused into two artificial tuples. The second receives five original tuples, which are fused into four artificial tuples. **Right**: Close-up on the some tuples matched and fused in two nested data environment. The bottom left bubble is an alias pointer (different base address) matched to the other tuple due to covering addresses.

```
IN CDl: a list of couple clauses+data.
IN D: the stack for the current device.
IN L: the nested level of the current data environment.

for (each couple cd in CDl) do
    new interval i $\leftarrow$ read(cd)
    if (i is not totally included in data previously transfered) then
        D.L.originalItvList $\leftarrow$ D.L.originalItvList $\cup$ {i}
    else
        if (i is partially included in data previously transfered) then
            Do actions for partially covering intervals
for (each interval tmpi in (D.L.originalItvList $\cup$ D.(L-1).artificialItvList)) do
    new interval ni $\leftarrow$ tmpi
    ni.ancestors $\leftarrow$ *tmpi
    D.L.artificialItvList $\leftarrow$ D.L.artificialItvList $\cup$ {ni}
for (each interval s,t in D.L.artificialItvList such as s$\neq$t) do
    if ((s.addr$_{host}$ = t.addr$_{host}$) OR (s.itv $\cap$ t.itv $\neq \emptyset$)) then
        new interval st $\leftarrow$ s $\cup$ t
        if (st $\neq$ s AND st $\neq$ t) then
            st.state $\leftarrow$ ALLOC
        else
            st.state $\leftarrow$ ALIAS
        D.L.artificialItvList $\leftarrow$ (D.L.artificialItvList $\cup$ st) / {s,t}
for (each interval a in D.L.artificialItvList) do
    if (a.state = ALLOC) then
        a.addr$_{acc}$ = Allocate$_{acc}$(a.range)
    for (each interval anc in a.ancestorList) do
        if (anc $\subset$ D.L.originalItvList) then
            if (anc.state = COPYIN OR anc.state = PCOPYIN) then
                copy$_{host \rightarrow acc}$(anc.addr$_{host}$, a.addr$_{acc}$, anc.range)
        else
            copy$_{acc \rightarrow acc}$(anc.addr$_{acc}$, a.addr$_{acc}$, anc.range)
```

Figure 3. Algorithm to apply when reading the clauses of a new data environment.

As several reallocations may occur, the memory address on the accelerator is updated after each modification.

```
IN D: the stack for the current device.
IN L: the nested level of the current data environment.

for (each interval t in D.L.artificialItvList)
    if(t.state = ALLOC
        for (each interval a in t.ancestorList)
            if (a $\subset$ D.L.originalItvList)
                if (a.state = COPYOUT)
                    copy$_{ac \to h}$(t.addr$_{ac}$ + a.first$_h$ - t.first$_h$, a.addr$_h$, a.range)
                D.L.originalItvList) $\leftarrow$ D.L.originalItvList / {a}
            else
                copy$_{ac \to ac}$(t.addr$_{ac}$ + a.first$_h$ - t.first$_h$, addr$_{ac}$, a.range)
    D.L.artificialItvList $\leftarrow$ D.L.artificialItvList / {t}
```

Figure 4. Algorithm to apply when closing a data environment.

5.2. *Entering a new data environment.*

The insertion of a new tuple in the directory is a critical moment. A lot of informations must be checked (possible fusion, allocation) and updated. The operations to perform at the creation of a new tuple, forming the algorithm in Figure 3, can be decomposed in three main steps: the decoding of data clauses, the fusion of tuples, and the allocation and copy of data intervals.

The first step is the decoding of the clauses in the data environment. Each clause is translated in a tuple, its state being the type of the clause (*copyout, pcopyin...*). The next step is the fusion of intervals, either due to a common reference address on the host, or due to some covering range between the tuples. The resulting fused tuples are inserted in the artificial list of the current level.

The final step consists in the allocation of data on the accelerator according to the list of *artificial tuples* at the current level of the stack.

list for the artificial tuples, result of the fusions. The main distinction is that original tuples are directly allocated and transferred to the accelerator memory. This is due to the exiting of unstructured data management that differs slightly from the closing of nested data environment. The same mechanisms for handling subarrays are performed inside the list (see algorithm in Figure 5).

5.3. *Data environment exit.*

Deleting a stack level in the directory is lighter than the reading of new pragmas. The algorithm 4 shows the steps for exiting a data environment. When a data environment is closed, the two lists for the current level are deleted, and the appropriates transfers are performed either to the host memory if requested by the pragma, or to another part of the accelerator memory to maintain the coherence. The new level at the top of the stack contains the two lists for the enclosing (and now current) data environment. All the steps for the two presented algorithms are thoroughly described in [12].

6. SUPPORT OF UNSTRUCTURED DATA LIFETIME

In the version 2.0 of the OpenACC API were introduced directives for unstructured data management. As presented in Section 2, these directives induce data lifetimes which are not perfectly nested, and might overlap with structured directives.

Unstructured data lifetime is not directly compatible with the directory presented in this paper, as the stack structure used to implement the directory directly emulates the nested behavior of the original directives in OpenACC 1.0 (and OpenMP 4.0).

To allow the use of directives for unstructured data lifetime, a simple extension was applied to the directory. Since only data embedded in the pragma `data begin` can be closed with `data`

```
IN PDl: a list of pair clauses+data.
IN UL: the lists for unstructured data lifetime.

for (each pair cd in PDl) do
    new interval i $\leftarrow$ read(cd)
    if (i is not totally included in data previously transfered) then
        UL.originalItvList $\leftarrow$ UL.originalItvList $\cup$ {i}
        newItvList $\leftarrow$ newItvList $\cup$ {i}
        i.addr$_{acc}$ = Allocate$_{acc}$(i.range)
        if (i.state = COPYIN OR i.state = PCOPYIN) then
            copy$_{host \to acc}$(i.addr$_{host}$, i.addr$_{acc}$, i.range)
    else
        if (i is partially included in data previously transfered) then
            Do actions for partially covering intervals
for (each interval s in (UL.artificialItvList) do
    for (each interval t in (newItvList) do
        if ((s.addr$_{host}$ = t.addr$_{host}$) OR (s.itv $\cap$ t.itv $\neq \emptyset$)) then
            new interval st $\leftarrow$ s $\cup$ t
            if (st $\neq$ s AND st $\neq$ t) then
                st.addr$_{acc}$ = Allocate$_{acc}$(st.range)
                copy$_{acc \to acc}$(s.addr$_{acc}$, st.addr$_{acc}$, s.range)
                copy$_{acc \to acc}$(t.addr$_{acc}$, st.addr$_{acc}$, t.range)
                st.ancestorList $\leftarrow$ s.ancestorList $\cup$ {t}
                UL.artificialItvList $\leftarrow$ (UL.artificialItvList $\cup$ st) / {s}
            else
                s.ancestorList $\leftarrow$ s.ancestorList $\cup$ t
            UL.artificialItvList $\leftarrow$ (UL.artificialItvList $\cup$ st) / {s,t}
            newItvList $\leftarrow$ newItvList / t
for (each interval t in (newItvList) do
    UL.artificialItvList $\leftarrow$ UL.artificialItvList $\cup$ t
    t.ancestorList $\leftarrow$ t
    newItvList $\leftarrow$ newItvList / t
```

Figure 5. Algorithm to apply when reading a `enter data` directive.

`end`, a simple list per device (which will be referred later as *unstructured list*) is enough to store the different intervals handled with these new constructs. Like a level of the stack, this list will have two distinct entities: a list for original tuples, and a However, this new list does not interact with the stack structure, and no coherence is implemented between them. So if different subparts of an array are handled in the two types of directives (structured and unstructured data management), they will not be merged in a same allocation to maintain the coherence.

When a `exit data` directive is encountered, the artificial tuples are swapped to find which one includes the specified intervals. The artificial tuple is destroyed, along with the concerned original tuple, after the data from the searched interval are transferred back to the host, and the other data are copied to new memory locations regrouping the remaining original tuples.

7. EXPERIMENTAL RESULTS

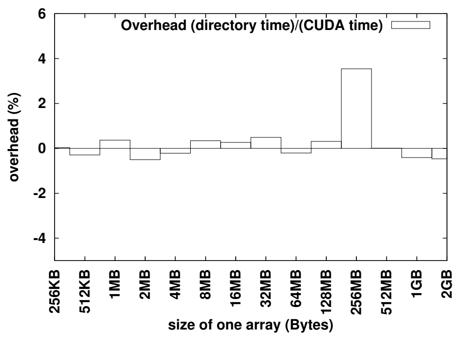
To evaluate our method, we measured the overhead of using our directory through our directory's ABI , compared to using the CUDA API directly, or automatic transfer with an OpenACC implementation.

For the comparison with CUDA, the CUDA kernels are the same, and only the memory transactions (allocations, transfers) with the GPU are handled either using CUDA functions (CUDAMalloc, CUDAFree, CUDAMemcpy) or or our ABI.

```
IN PDl: a list of pair clauses+data.
IN UL: the lists for unstructured data lifetime.

for (each pair cd in PDl) do
    new interval i ← read(cd)
    tmpList ← tmpList ∪ {i}
for (each interval s in (tmpList) do
    for (each interval t in (UL.artifialItvList) do
        for (each interval a in t.ancestorList) do
            if a.itv = s.itv then
                removeList ← removeList ∪ t
                if s.state = COPYOUT then
                    copy$_{acc \rightarrow host}$(t.addr$_{acc}$, s.addr$_{host}$, s.range)
                    t.ancestorList ← t.ancestorList / a
                    UL.originalList ← UL.originalList / a
for (each interval t in (removeList) do
    for (each interval a in t.ancestorList) do
    Apply fusion as in enter data
    UL.artificialList ← UL.artificialList / t
```

Figure 6. Algorithm to apply when reading a `exit data` directive.



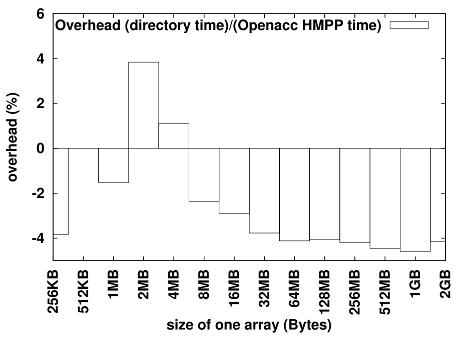


Figure 7. Overhead on a succession of vector additions alternating between a CPU and a GPU. Left: comparison between data management with directory, and plain CUDA transfers and allocations. Right: comparison between data management with directory, and OpenACC directives handled with HMPP 3.3.0.

For the comparison with OpenACC, we used the CAPS Enterprise compiler HMPP version 3.3.0 which supports the OpenACC API 1.0. In both cases, the kernels are generated automatically through OpenACC directives. Like the CUDA versions, only the memory transactions are handled either using directly OpenACC directives inside the code (letting the HMPP compiler to handle the pragmas), or replacing the directives by our ABI functions.

We present the results for three applications for the comparison with CUDA, and three applications with OpenACC. All tests were performed on a machine composed with an Intel Xeon Westmere-EP E5620 2.4GHz , combined with a GPU Nvidia Fermi M2090, with 512 thread processors for a total of 666 GFlops peak and 6 GB of memory fed through GDDR5 bus at 177GB/s.

Small benchmarks. The first applications are a test with three successive vector additions, and another with three successive matrix multiplications. The first operation is performed on the GPU, the second one on the CPU using results from the first operation, and the last one is performed again on the GPU using data produced by the computation on the CPU. This forces to have exchange of data at each step between the host and the device. The time of the whole routine was measured

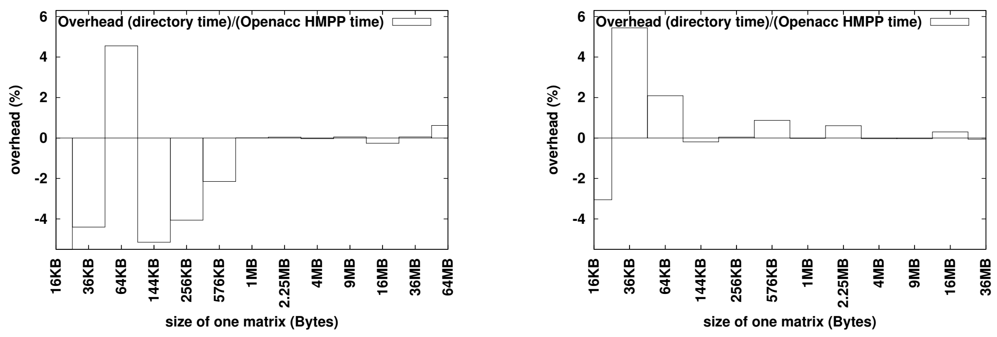


Figure 8. Overhead on a succession of matrix multiplications alternating between a CPU and a GPU. Left: comparison between data management with directory, and plain CUDA transfers and allocations. Right: comparison between data management with directory, and OpenACC directives handled with HMPP 3.3.0.

(allocation and data copy on the GPU before the first operation, the necessary transfers between the different step, and the copy and deallocation of data after the computation).

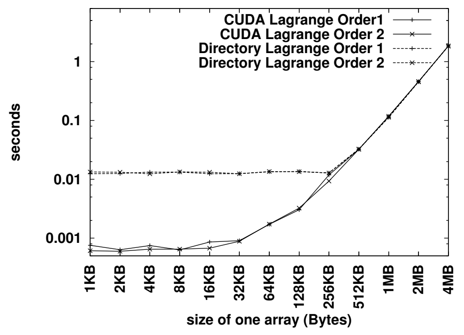
The results for the vector additions are presented in Figure 7. When compared to plain CUDA transfers, we observe that the overhead is very low (within 5% with meaningful data volume) for a small number of data transfers. Here, the overhead is due to the allocation of tuples in the directory.

When compared to OpenACC with HMPP 3.3.0, the overhead is still under 5%, and we observe that for the larger sizes, we are faster than the code generated with HMPP. These results highly depend on the code generation performed by the compiler for the OpenACC directives. We were not able to perform the same test with the latest version of the HMPP compiler, nor with other compilers handling OpenACC, to confirm the low overheads we observed compared to this version of CAPS compiler and the CUDA version.

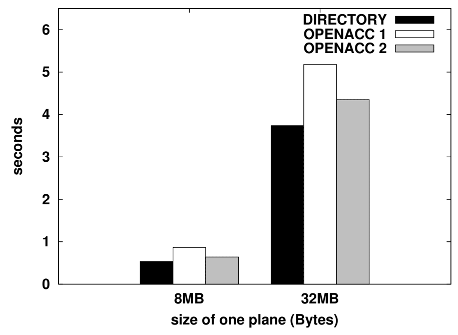
The results for the matrix multiplications are presented in Figure 8. As with the vector additions, the results show a low overhead (under 6%), and very similar performance results for the greater sizes when comparing with OpenACC and CAPS compiler. We were not able to try larger matrices as the OpenACC runtime always failed with square matrices sizes above 4096 integers.

7-point 3D Stencil To show the benefit of our directory, we implemented the stencil application described in Section 3.2. Again, the kernels are generated with the OpenACC compiler, and only the data management parts is either control with HMPP compiler or our Directory. Compared to the transfers required by OpenACC, our implementation need to move three times less data with the same kernel implementation. This leads to a performance improvement of 28% on large sizes, as shown in Figure 9(b) (black bar is for our directory, and the white bar labelled OpenACC 1 for this version). As our directory requires more memory (for copies inside the accelerator memory) than the OpenACC data management, we tested another OpenACC implementation to use the same amount of memory. This implementation updates two planes in the kernel, and loads six slices of the stencil, where our directory requires the space for seven slices at one point to update one plane. The resulting OpenACC code performs half memory transfers, but still transfers more data than our directory. Despite performing more transfers, the version using our directory outperforms the new OpenACC version and is 15% faster (see the grey bar labelled OpenACC 2 in Figure 9(b)). However, with sizes less than 1MB, the amount of data to transfer is not enough to overweight the time lost in the initialization of the transfers.

Real-life application. Another set of preliminary benchmarks was realized on a mono-material hydrodynamic mini-app, GAD [10]. This code solves mono-material Euler equations relying on a Lagragian phase followed by a projection phase. Here, the number of manipulated arrays is larger.



(a)



(b)

Figure 9. **(a)**: Time consumption of memory management methods on the GAD code. **(b)**: Time consumption of a 3D 7-point Stencil on a GPU, using our Directory or OpenACC for the data transfers.

Twelve arrays are used on the accelerator, with seven of them requiring to transfer the data between the host and the device before and after the computations. This larger number of arrays used on the GPU results in a similar number of tuples in our directory. When adding a new tuple, there is a greater number of existing tuples to compare to. For every step requiring it, swapping the list of tuples becomes heavy. Here the overhead of the directory can clearly be measured. Looking at the results displayed in Figure 9(a), one can easily see an overhead of 12 ms, which has a huge impact for small sizes. With a larger size, this fixed overhead becomes negligible compared to the compute time.

8. CONCLUSION

In this paper, we outline some restrictions on the data management directives embedded in the two major APIs (OpenACC and OpenMP) for directive based accelerator programming. To help the implementation of runtimes for such APIs, we developed a directory scheme to handle the memory management between a host and accelerators, including the directives for unstructured data lifetime.

As some restrictions may become very difficult to observe in real life programs with multiple nested function calls, we offer some extension in our directory to allow more flexibility when using data management directives. With our extensions, it is now possible to use different subparts of a same array in nested data environments.

A first non-optimized implementation of this directory has been tested, and results about its overhead have been displayed in Section 7 with very promising results. When using our runtime on simple codes, with only few distinct data areas to deal with, no overhead was measured compared to other runtimes. For the test on a real-life code, the measured overhead becomes negligible with huge (realistic) data sets compared to the actual transfer time.

With the support of unstructured data lifetime, we were able to optimize the data movement for a 7-point 3D stencil with a time improvement compared to an OpenACC implementations of 28% when considering the same code, and 15% when optimizing the OpenACC implementations to use as much memory as our directory.

REFERENCES

1. J. Archibald and J. L. Baer. An economical solution to the cache coherence problem. In *11th ISCA*, pages 355–362, New York, NY, USA, 1984. ACM.

2. C. Augonnet and R. Namyst. A unified runtime system for heterogeneous multi-core architectures. In *14th Euro-Par Workshops*, pages 174–183, Berlin, Heidelberg, 2009. Springer.
3. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *32nd ISCA*, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.
4. L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, 1978.
5. D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.
6. Guoying Chen. Slid - a cost-effective and scalable limited-directory scheme for cache coherence. In *Parallel Architectures and Languages Europe '93*, volume 694 of *LNCS*, pages 341–352. Springer Berlin Heidelberg, 1993.
7. Steffen Christgau, Johannes Spazier, Bettina Schnor, Martin Hammitzsch, Andrey Babeyko, and Joachim Waechter. A comparison of cuda and openacc: Accelerating the tsunami simulation easywave. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–5, Feb 2014.
8. B. Cuesta, A. Ros, M.E. Gomez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th ISCA*, pages 93–103, 2011.
9. T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *27th IPDPS*, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.
10. Olivier Heuzé, Stéphane Jaouen, and Hervé Jourdren. Dissipative issue of high-order shock capturing schemes with non-convex equations of state. *Journal of Computational Physics*, 228(3):833 – 860, 2009.
11. T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 136–143, May 2013.
12. Julien Jaeger, Patrick Carribault, and Marc Prache. Data-management directory for openmp 4.0 and openacc. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 168–177. Springer Berlin Heidelberg, 2014.
13. J.M. Levesque, R. Sankaran, and R. Grout. Hybridizing s3d into an exascale application using openacc: An approach for moving to multi-petaflops and beyond. In *SuperComputing 2012*, pages 1–11, 2012.
14. F. V. Lionetti, A. D. McCulloch, and S. B. Baden. Source-to-source optimization of cuda c for gpu accelerated cardiac cell modeling. In *16th Euro-Par*, volume 6271 of *LNCS*, pages 38–49. Springer Berlin Heidelberg, 2010.
15. I. L?opez-Rodriguez. Openacc implementations comparison. 2012.
16. Nicolas Maillard. *Hybrid Parallel Programming-Evaluation of OpenACC*. PhD thesis, UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2012.
17. Pushan Majumdar. Lattice simulations using openacc compilers. *arXiv preprint arXiv:1311.2719*, 2013.
18. A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *32nd ISCA*, pages 234–245, Washington, DC, USA, 2005. IEEE Computer Society.
19. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, August 2009.
20. G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *14th PPoPP*, pages 121–130, New York, NY, USA, 2009. ACM.
21. R. Reyes, I. López, J. J. Fumero, and F. Sande. A preliminary evaluation of openacc implementations. *J. Supercomput.*, 65(3):1063–1075, September 2013.
22. R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. `accULL`: an openacc implementation with cuda and opencl support. In *18th Euro-Par*, pages 871–882, Berlin, Heidelberg, 2012. Springer-Verlag.
23. M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *22nd ICS*, pages 309–318, New York, NY, USA, 2008. ACM.
24. Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source openacc compiler for cuda. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 178–187. Springer Berlin Heidelberg, 2014.
25. C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of national computer conference and exposition '76*, AFIPS '76, pages 749–753, New York, NY, USA, 1976. ACM.
26. S. Wienke, P. Springer, C. Terboven, and D. Mey. Openacc: First experiences with real-world applications. In *18th Euro-Par*, volume 7484 of *LNCS*, pages 859–870. Springer Berlin Heidelberg, 2012.
27. Rengan Xu, S. Chandrasekaran, and B. Chapman. Exploring programming multi-gpus using openmp and openacc-based hybrid model. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1169–1176, May 2013.
28. H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. Spatl: Honey, i shrunk the coherence directory. In *20th PACT*, pages 33–44, 2011.