Data-Management Directory for OpenMP 4.0 and OpenACC

Julien Jaeger, Patrick Carribault, and Marc Pérache

CEA, DAM, DIF, F-91297, Arpajon, France
`firstname.lastname@cea.fr`

Abstract. 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 on OpenMP 4.0 release candidate. The pragmas for data management induce some coherence issues in the accelerator memory for code correctnesse. 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. Our design fits a multi-accelerator system. Also, with our directory, we propose a way to handle correctly pragmas on partially overlapping data intervals.

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 into one set of directives through the OpenACC API and the directives for accelerator in the OpenMP 4.0 release candidate. 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 informations about the data transferred to an accelerator.

OpenACC 2.0. The OpenACC API was released in November 2011 with a draft for the second version appearing one year later. In the scope of our work, we will focus on the data management part. The bounds of `data` constructs define *data environments*. A `data` construct includes several clauses to specify if the data need to be transfered from (or back to) the host (*copyin*, *copyout*), is already present (*present*) or need to be allocated on the accelerator (*create*). More complex clauses are used to allocate and copy the data only if they are not already in the accelerator memory (*pcreate*, *pcopyout*, *pcopyin*). Data specified in clauses are alive only in the scope of their data environment.

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 the accelerator. The different states (*alloc*, *to*, *from*, *tofrom*) have the same behavior as some clauses in OpenACC (resp. *pcreate*, *pcopyin*, *pcopyout*, *pcopy*).

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

In this paper, we propose a directory scheme to handle all the memory management hidden in the directives for accelerators. Our directory is designed to work with several accelerator devices attached to the same host. Moreover, we suggest a simple extension to the two APIs allowing to correctly use data management pragmas on partially overlapping intervals.

The paper is organized as follows: in Section 2, we present three motivating examples to detail the important problems to tackle while managing data with these directives, and to exhibit how should work the use of pragmas on partially overlapping intervals. Section 3 presents the previous work done with directories, and the few efforts already done to propose open implementations of the OpenACC standard. The design of our directory and the different algorithms implemented in the runtime are described in Section 4. Finally, some overhead results of our implementation are shown in Section 5 before concluding in Section 6.

2 Motivating Examples

Our directory is motivated by three major problems. The first one is how to correctly deal with different subparts of the same array spread across nested data environments. The second problem occurs when different pragmas deal with covering subparts of an array accessed through an alias pointer. The last one concerns the use of partially overlapping intervals in nested data environment.

Separate ranges of the same array. When dealing with nested data environments, it is possible that pragmas spread across these data environments handle subparts of the same arrays. Figure 1 shows an example of an OpenACC program with this behavior. The first data environment requests the elements from 0 to 19 of arrays a to be available on the accelerator device, while the second one requests an other subpart of array a. The two subparts of the array are alive, and they are used in the same computation. If the two subparts are allocated in distinct locations of the accelerator memory, the only way to have valid results is to change how we access one of the subparts, using a new pointer. Rewriting the memory accesses can be done on simple codes like the one in Figure 1, but it is impossible to automatically apply it on a more complex code with nested function calls. To have a correct computation, all elements of a should be in the same memory space, with $a[50]$ being fifty elements further than $a[0]$. The pragmas related to the same memory zone have to be matched together inside a common memory area.

```
#pragma acc data
    copyin(b[0:20],a[0:20])
    copyout(a[0:20])
  #pragma acc parallel for
    for (i=0; i<20; i++)
      a[i]+=b[i];
  c=&(a[25]);
  #pragma acc data
    copyin(a[50:20], c[0:20])
    #pragma acc parallel for
      for (i=0; i<20; i++)
        a[i]+=a[i+25]+a[i+50]
```

Fig. 1. OpenACC code using distinct parts of the same array with an alias pointer.

Covering ranges and aliasing. Another specific case is when aliasing is use within a computation. This alias pointer can be used in a clause to specify a subpart of a data range already declared in an englobing data environment. The two pragmas should be matched to not reallocate and transfer the data already present on the accelerator. Also, if distinct subparts of an array are merged in the same memory allocation, another problem may appear. In the code example displayed in Figure 1, an alias pointer is used to refer to another subpart of the array a. This new code is valid, since we transfer all the data required by the computations to the accelerator. However, all the data pointers in the different pragmas do not have the same reference address. If intervals are not matched, one might end up with a totally different location for $c[0:20]$ than between $a[0:20]$ and $a[50:20]$, ending with a false code on the accelerator. To link this interval to the other clauses, one first has to fuse the two pragmas on address a, creating a new range from $a[0]$ to $a[70]$, and then find that the intersection with the pragma on address c is not empty. Thus, it is necessary to check the covering range of intervals without the same basic addresses.

Overlapping intervals. In OpenMP 4.0 and OpenACC, using an interval in a data environment which partially overlaps another interval previously defined in an englobing environment is not valid. However, it is possible to handle such cases, considering that the overlapping interval can be divided in multiple pieces: the parts that are already present on the accelerator, and the parts that need to be allocated. Like for the case with separate ranges on the same arrays, a reallocation will be performed on the accelerator to have enough memory to reunite the subparts of the overlapping interval. We suggest two schemes to initialize in this new memory space the data that correspond to parts already present on the accelerator: either use these previous data (as with a `present_or` clause on the subpart), or use the corresponding data from the host memory (as with an update clause on the subpart).

3 Related Work

Directory-based coherency protocols have emerged in the late 70's as a concurrent to *snoopy cache* protocols. Tang *et al.* [16] 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, 11, 7, 18]), 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 [15] to cardiac cell modeling [10]. 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 [8], StarPU [2], FLAME [13] or StarSs [12], 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 ([9, 17, 14]). 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 mentionned how the memory allocations and transfers are handled.

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

4 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 4.2. 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 4.3.

4.1 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*.

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 tulpe also embedded 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. As several reallocations may occur, the memory address on the accelerator is updated after each modifications.

To use this directory, we provide a small ABI, based on the different pragmas, presented in Figure 4. When a data environment is opened, functions *new_dataE()* and *read_pragmas()* create the new stage in the stack and allocate the list for the incoming data. For each clause, a corresponding function exists using the same parameters. Once all the clauses are read, *write_pragmas* applies the fusions then inserts the new tuples in the stack. Finally, when the data environment is closed *end_dataE()* transfers the data and delete the remaining structures.

4.2 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 2, 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 tuple is compared to the other existing *original tuples*. If the interval is completely included in the previous tuples and the clause is a *present_or* clause, then it is deleted, as we will use the existing data and no transfer will occur. If a partial overlap is detected, an error is occured. This is the default behavior according to the APIs. If one wants to handle the case of partial overlap in the runtime, then the interval is decomposed in several subparts, being either totally present in the accelerator memory, or totally absent. Once no other overlap can be found, remaining tuples are inserted in the *original* list at the current level of the stack.

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. We suggest to begin with the fusion of tuples based on their basic virtual address on the host, otherwise some covering range might be missed (as explained in Section 2). The fusion on same basic address is realized between all the tuples in the original list of the current level and the artificial list of the previous one.

The resulting fused tuples are inserted in the artificial list of the current level. At this point, a new tuple state is either ALLOC if an allocation is necessary (i.e. the range of the *artificial tuple* is not included in one of its ancestors), or ALIAS otherwise. In Figure 1, the two intervals on a will be fused in a new interval $a[0 : 70]$, stored in the artificial list of the current level. Since the resulting range is greater than each of its ancestors, the state of this new tuple is ALLOC. An artificial tuple keeps the list of its direct ancestors fused to build it.

```
    INPUT: Pl: a list of pragma, D: the stack for the current device.
    INPUT: L, the nested level of the current data environment.
for (each pragma p in Pl) do
    new interval i ← read(pragma p)
    if (i is not totally included in data previously transfered) then
        D.L.originalItvList ← 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 ∪ D.(L-1).artificialItvList)) do
    new interval ni ← tmpi
    ni.ancestors ← *tmpi
    D.L.artificialItvList ← D.L.artificialItvList ∪ ni
for (each interval s,t in D.L.artificialItvList such as s≠t) do
    if (($s.addr_{host}$ = $t.addr_{host}$) or (s.itv ∩ t.itv ≠ ∅)) then
        new interval st ← s ∪ t
        if (st ≠ s and st ≠ t) then
            st.state ← ALLOC
        else
            st.state ← ALIAS
        D.L.artificialItvList ← (D.L.artificialItvList ∪ 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 ⊂ 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)
```

Fig. 2. Algorithm to apply when reading the pragmas of a new data environment.

A second fusion based on covering ranges is performed on these new tuples. The memory interval of tuples are compared. If there is an overlap, the two tuples are fused together in a new tuple. Their corresponding list of ancestors are also fused to become the ancestor list of the new tuple. The state of this tuple is ALLOC if its data range is not included in one of the tuples fused to build it, or if at least one of the fused tuples was already in a ALLOC state. Otherwise, the state of this tuple is ALIAS. In Figure 1, array c has a common part with the new artificial interval $a[0 : 70]$. The result of their fusion has the same range of data. One of its ancestors has a ALLOC state, so the state of the new tuple will also be ALLOC.

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, and to perform Actions are performed only for the tuples with a `ALLOC` state. A memory range corresponding to the data range of the tuple is allocated on the accelerator device. For all the ancestors of the tuple, a copy takes place, either from host to device if the state of the ancestor is `(P)COPYIN`, or from another part of the accelerator memory if the state of the ancestor is `ALLOC` or `ALIAS`.

```
    INPUT: L, the nested level of the current data environment.
    INPUT: D, the directory for the current device.
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(2×lv) $\leftarrow$ D(2×lv) / {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}
```

Fig. 3. Algorithm to apply when closing a data environment.

4.3 Data environment exit.

Deleting a stack level in the directory is lighter than the reading of new pragmas. The algorithm for exiting a data environment is shown in Figure 3. When a data environment is closed, the list of *artificial tuples* is unstacked. For each tuple of this list with a `ALLOC` state, actions will be performed on the different subparts of the interval, depending on its corresponding ancestors state. If the ancestor has a `COPYOUT` state, it is an *original tuple* requesting to copy the subpart back to the host memory. Data transfer occurs from the accelerator address specified by the ancestor tuple to the host address given by the same ancestor tuple. On the other hand, an ancestor with a `ALLOC` state means

OpenACC	OpenMP 4.0	ABI

# pragma acc data	**# pragma** target data	new_dataE(); 	
		read_pragma(); . . . write_pragma();

clauses *pcopy*, *pcopyin*
pcopyout, *pcreate*
	clauses *map(tofrom), map(to)*
	map(from) map(alloc)
		pragma_pcopy(), pragma_pcopyin()
		pragma_pcopyout(), pragma_pcreate()

clauses *copy*, *copyin*
copyout, textitcreate
	no correspondance
	in OpenMP 4.0
		pragma_copy(), pragma_copyin()
		pragma_copyout(), pragma_create()

# pragma acc update	**# pragma** target update	pragma_update()
end of data environment	end of data region	end_dataE();

Fig. 4. ABI for the use of the designed directory scheme.

that a previous allocated space exists in a previous data environment. To keep the data coherence between the interleaved data environment, a data copy corresponding to this subpart is necessary between the current memory space and the memory space pointed by the ancestor. Once all the copies are performed, the top stage of the stack is deleted. The new level at the top of the stack contains the two lists for the enclosing (and now current) data environment.

5 Experimental results

To evaluate our method, we measured the overhead of using our directory through our directory's ABI (shown in Figure 4), compared to using the CUDA API directly. The CUDA kernels are the same, and only the memory transactions (allocations, transfers) with the GPU are handled either using CUDA functions or our ABI. We present the results for two applications on an Intel Xeon Westmere-EP E5620 with a GPU Nvidia Fermi M2090.

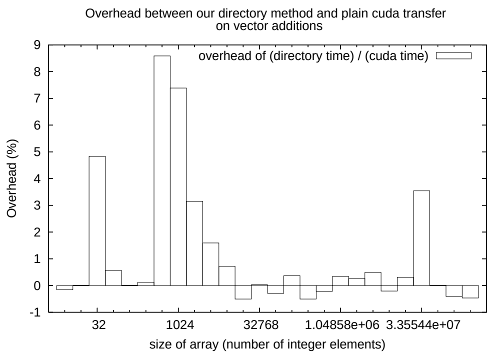

Fig. 5. Overhead between data management with directory compared to plain CUDA allocations and transfers on a succession of vector additions alternating between a CPU and a GPU.

The first application is three successive vector additions. The first addition is performed on the GPU, the second one on the CPU and using results from the first addition, and the last one is performed again on the GPU using data produced by the computation on the CPU. The time of the whole routine was measured (allocation and data copy on the GPU before the first addition, the necessary transfers between the different additions, and the copy and deallocation of data after the computation). The results presented in Figure 5 shows that for a small number of data transfers, the overhead is very low (within 10% for small sizes, and less than 5% with meaningful data volume). Here, the overhead is due to the local allocation of tuples in the directory.

Another set of benchmarks was realized on a mono-material hydrodynamic mini-app, GAD. 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. 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 greater 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 6, 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.

6 Conclusion

In this paper, we outline the data management problem that can be brought by directives for accelerators based on the two major APIs (OpenACC and OpenMP). To produce a valid code, some data locality coherency must occur between the different pragmas handling data. To help the implementation of runtimes for such APIs, we developped a directory scheme to handle the memory management between a host and accelerators. A first non-optimized implementation of this directory has been tested, and results about its overhead have been displayed in Section 5.

The two major APIs proposing directives for accelerators are still evolving. In OpenACC 2.0, the concept of non-nested data environment has been introduced through the use of new pragmas (`data begin` and `data end`). While this evolution is not directly compatible with the directory implementation we proposed, since a stack is used to mimic the nested behaviour of data environment, it is rather easy to include this notion in our directory. Since only clauses embedded in the `data begin` can be closed with `data end`, a simple list is enough to store the different intervals handled with these new constructs. In each step of the stack, after the floor for the *artificial tuples* of the current nested data environment, we can add a new floor, where the *artificial intervals* will be fused to the intervals in the new list. That way, when opening or ending a non-nested data environment, only the new separate structures have to be updated, and the original behaviour of our directory will remain unchanged.

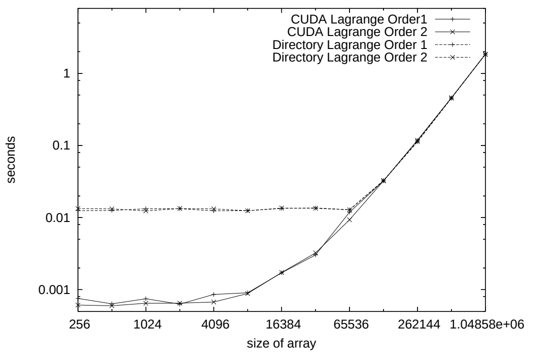

Fig. 6. Time consumption of memory management methods on the GAD code

As an evolution to these APIs, we suggest also to add the support for partially overlapping intervals. In this paper, we explained how it is posssible to handle such cases with this type of directory. New pragmas can be created to this purpose like a `present_and_copyin` (which will use the existing data on parts already present on the accelerator, and copy from the host only the subparts of the interval not present on the accelerator) or `update_and_copy` (which will copy all the interval from the host to the new allocated memory, and updating the parts already present on the accelerator at the clause end).

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. 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.
8. T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for dataflow task programming on heterogeneous architectures. In *27th IPDPS*, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.
9. 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.
10. 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.
11. A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *32nd ISCA*, pages 234–245, Washington, DC, USA, 2005. IEEE Computer Society.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. Spatl: Honey, i shrunk the coherence directory. In *20th PACT*, pages 33–44, 2011.