Automatic run-time mapping of polyhedral computations to heterogeneous devices with memory-size restrictions

Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos
Departamento de Informática
Edif. Tecn. de la Información, Universidad de Valladolid,
Campus Miguel Delibes, 47011 Valladolid, Spain
E-mail: {yuri.torres, arturo, diego}@infor.uva.es

***Abstract*—Tools that aim to automatically map parallel computations to heterogeneous and hierarchical systems try to divide the whole computation in parts with computational loads adjusted to the capabilities of the target devices. Some parts are executed in node cores, while others are executed in accelerator devices. Each part requires one or more data-structure pieces that should be allocated in the device memory during the computation.**

In this paper we present a model that allows such automatic mapping tools to transparently assign computations to heterogeneous devices with different memory size restrictions. The model requires the programmer to specify the access patterns of the computation threads in a simple abstract form. This information is used at run-time to determine the second-level partition of the computation assigned to a device, ensuring that the data pieces required by each sub-part fit in the target device memory, and that the number of kernels launched is minimal. We present experimental results with a prototype implementation of the model that works for regular polyhedral expressions. We show how it works for different example applications and access patterns, transparently executing big computations in devices with different memory size restrictions.

***Keywords*—Heterogeneous devices, Polyhedral model, Memory-size restrictions, Automatic mapping tools**

I. Introduction

Heterogeneous systems can be built with very different hardware devices (CPU-cores, accelerators) grouped in several nodes and interconnected in a distributed environment. Portable codes for such systems should implement parallel algorithms, while abstracting them from the mapping activities that adapt the computation to the platform. Thus, the programming model should encapsulate the mapping techniques and the CPU/accelerator synchronization with appropriate abstractions.

Taking into account the memory size limitations of heterogeneous target devices is an additional challenge. Currently, many approaches do not focus in this problem, working with fixed sized middle-grain tasks [1], or assuming that the tasks fit, or are generated to fit into the devices [2], [3]. Other approaches simply advise to add more computation devices to allow finer partitions [4]. A simple way to tackle the problem is to generate more distributed processes than system nodes, mapping several of them to the same device [5]. In this way, each process is responsible for a smaller part of the computation. When enough processes are launched, the parts are small enough to fit in any target device. However, this leads to more costly inter-process communications and scalability problems. A more sophisticated approach is to consider the device memory limitations while creating the high-level partition [6]. This approach highly complicates the whole partitioning activity.

An associated problem for memory-restrictions-aware systems is to find a proper representation of the parallel computation that allows the system to locate, and measure the size, of the data portions required by a generic part of the computation. This information is needed for both generating a balanced partition, and mapping the parts adequately [6], even for libraries that make transparent the node to device communication [7].

In this work we propose a solution to allow a hidden layer to: (1) Split an arbitrarily large computation in parts that fit the memory limitations of an assigned target device; (2) transparently launch the partial kernels generated; and (3) move the required pieces of the data structures between the main node memory and the target device memory when needed. We present a model of parallel applications that allows to express access patterns to data structures in terms of the thread index domains. These expressions allow the system to automatically compute the memory requirements of a computation part (a block of threads). We introduce a generic algorithm for regular polyhedral computations to compute at run-time an appropriate partitioning of the thread space that minimizes the number of kernels launched, ensuring that the pieces of the data structures needed by each kernel fit into the target device memory.

We also present examples of how to represent with this model parallel kernels and applications. Experimental results obtained with a prototype implementation of the model show its feasibility.

The rest of the paper is organized as follows: Section II describes the proposed approach and how it integrates in a previous run-time mapping framework. Section III describes the model for parallel computations and access patterns. Section IV presents an algorithm for computing a partition with memory size limitations. Section V shows experimental results with a prototype implementation, while Sect. VI presents our conclusions.

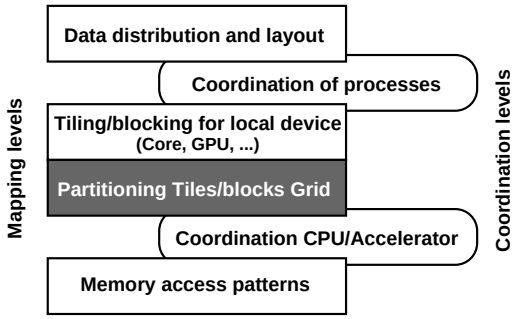


Fig. 1. Mapping/Coordination levels. The new level of automatic partitioning is highlighted with a dark-grey shadowed box.

II. Run-time approach for mapping

A. Hitmap run-time mapping framework

In a previous work [5] we proposed a programming approach and framework based on: (1) Several layers of plug-in modules that encapsulate mapping functions; and (2) functionalities to build the coordination (synchronization and communication) structures of an algorithms, which are transparently adapted at run-time in terms of the results of the mapping functions. The approach was incorporated into Hitmap [8], [9], a parallel programming library where partition policies are implemented through a set of plug-ins with a common interface.

Figure 1 shows the original mapping and coordination layers (white boxes). There are two levels of partition. The first one is designed to encapsulate coarse-grain mapping techniques, appropriate for distributed-memory nodes. At this level logical processes are assigned to processing nodes, or accelerator devices. Coordination patterns are built with high-level point-to-point, or collective communications, using the results automatically generated at run-time by the partition strategies. Thus, if partition or distributed topology details change, the communication structure will reflect the changes automatically.

Given the computation part in a logical process, assigned to a target device, the second mapping level allows to compute a proper middle-grain blocking partition. The mapping plug-ins at this level encapsulate heterogeneous policies to generate appropriate tiling sizes for CPU cores, thread-block geometry for GPU devices, etc. The coordination, data movement between the CPU and accelerators, and kernel launch, is automatized by a run-time system, using the second-level partition results.

The programmer naturally introduces a third level of mapping inside the kernel code by implementing specific, thread-level memory access patterns.

B. Integration approach

In the previous mapping approach, the computation partitioning is done top-down. The whole computation is first split and coordinated among logical processes in a distributed-memory environment. Load balancing techniques can be used at this level to adapt the amount of computation of each part to the computation power and characteristics of each device assigned to a process.

As we mentioned in the previous section, although device memory restrictions can be considered at this level in the partition policies, these policies would become much more complicated. For huge computations, they will lead to the creation of a higher number of logical processes, with the associated penalties for coordination and communication.

Our solution is to keep using simple partition policies at the highest level, that do not take into account the memory restrictions of heterogeneous accelerator devices attached to the system nodes. Then, we introduce a hidden abstraction layer that splits the computation in several parts which memory requirements fit the device limits. This layer is applied after determining the appropriate tile or block geometry (see the dark shaded box in Fig. 1). To keep the optimizations obtained by the tiling/blocking techniques of the upper layer, this new internal partition uses as basic mapping elements the tiles or blocks. Sections of the grid of tiles/blocks are going to be sequentially launched to the device as separate kernels.

In general, due to communication costs between the main node and the device memories, the partition of a computation should be minimal. Besides this, when launching a subpart of a computation, the exact pieces of data structures accessed by each part are determined by the application algorithm, and the design details of the parallel solution. In our approach, we introduce a simple abstraction to help the programmer express the access patterns of the threads to any data structure involved. Thus, the system can automatically derive expressions to compute at run-time the exact memory requirements, and the exact locations of data pieces needed for a given computation subpart (a section of the tiles/blocks grid).

III. Model for parallel computations and accesses patterns

A. Polyhedral domain spaces

We define a *domain* D as a collection of n-tuples of integer numbers that define a space of n-dimensional indexes. For dense arrays, the index domain is a subspace of Z^n, defined by a rectangular parallelotope. In this work we also allow *strided* domains, where the parallelotopes are defined by its dimensional limits, and a stride value for each dimension. A *Signature* is a 3-tuple of integer numbers $S = (b, e, s) : b, e, s \in \mathbb{Z}$ representing a subset of integer numbers where the begin or lower limit is b, the end or upper limit is e, and the elements are selected using the stride value s. We denote this subset of integer numbers as the *range* of the signature $\breve{S}$.

$$S = (b, e, s);\ \breve{S} = \{x \in \mathbb{Z} : x \geq b,\ x \leq e,\ (x - b) \mod s = 0\}$$
$$D < S_0, \ldots, S_n >= \{(p_0, \ldots, p_n) : p_i \in \breve{S}_i\}$$

Domains are used in this work to represent the index space of a data structure, a set of indexed threads, the geometry of a tile/block of threads, a grid of tiles/blocks, or a superblock geometry (a subdomain of a grid of tiles/blocks).

B. Parallel computations

A data structure or tile T is a map between elements of a domain and data elements of a given type: $T : D \rightarrow dataType$. We denote with $d(T)$ the Domain of a tile.

We define a *Parallel Computation* $P < D, f, T_0, \ldots, T_m >$ as a collection of threads manipulating data in one or more data structures or tiles $T_0, \ldots, T_m$. The domain of the computation D defines the number and indexes of the threads to be executed. The computation is the application of the function f (or collection of statements) by each thread on data elements. A *Polyhedral Computation* is a parallel computation where its domain D can be expressed as a parallelotope, and where the function f uses affine expressions on the thread indexes to locate and access data elements in any data structure T_i.

C. Access patterns

An *Access Pattern* AP is a set of access expressions. An *Access Expression* represents a domain transformation $A : D, \mathbb{Z}^n \rightarrow D$. It is a tuple of n *Signature Functions* $A = (A_0, \ldots, A_n)$. Each signature function maps a signature, and one domain element, to another signature: $A_i : S, \mathbb{Z}^n \rightarrow S$.

Affine Access Expressions are those whose signature functions determine the resulting signatures using affine expressions in terms of the input domain element $\vec{x} \in D$, to compute the begin and end elements of the new signature and the resulting stride is proportional to the original one.

$$\begin{aligned} A_i < \vec{a}_b, b_b, \vec{a}_e, b_e, c > (S, \vec{x}) = (b', e', s') : \\ b' = \vec{a}_b \cdot \vec{x} + b_b, \\ e' = \vec{a}_e \cdot \vec{x} + b_e, \\ s' = c \times s \end{aligned}$$

In some real parallel computations one dimension of a data structure is fully traversed by any thread. We model this special behavior using infinity values in the signature function to refer to the limits of the input signature. If $b_b = -\infty$, then $b' = b$. If $b_e = \infty$, then $e' = e$.

D. Union of domains

The union of generic domains expressed by signatures, cannot always be expressed themselves by signatures. As an example, consider the situation where there is a gap between their extremes, such in $S = (2, 100, 2), S' = (250, 300, 2)$, or when the strides are not compatible, such in $S = (2, 100, 2), S' = (2, 100, 3)$.

We define the *Signature coarse union* operator $\sqcup$ as: $S \sqcup S' = (b'', e'', s'') : b'' = \min(b, b'), e'' = \max(e, e'), s'' =$ m.c.d.(s, s'). We can also extent the operator definition to n-dimensional domains. The *Domain coarse union* of two domains is calculated applying the signature coarse operator to each pair of signatures with the same index: $D \sqcup D' = (S_0 \sqcup S'_0, \ldots, S_n \sqcup S'_n)$. The application of this operator to merge two strided parallelotope domains generates another strided parallelotope that can be expressed with signatures, with minimal number of extra added elements.

E. Domain transformations

We define a *Domain transformation* $\Gamma : D, AP, D \rightarrow D$ as the coarse union of the domains obtained applying each access pattern to each element of the second domain, using as reference the first domain, or data-structure domain.

$$\Gamma(D, AP, D') = \sqcup\{A(D, \vec{x})\} \ \forall \vec{x} \in D' \wedge \forall A \in AP$$

We call *Regular access expressions* to those that for two given input domain elements $\vec{x}, \vec{y}$, the signatures $A_i(D, \vec{x}) = (b, e, s)$ are a translation of the signatures $A_i(D, \vec{y}) = (b', e', s')$ such that $\forall i$: (1) $b' = b, e' = e, s' = s$, or (2) $b' = b + (y_i - x_i), e' = e + (y_i - x_i), s' = s$. A *Regular access pattern* is a pattern with only regular access expressions. Memory requirements of regular access patterns grow linearly when the threads space grows in only one dimension.

IV. Partition of regular computations for heterogeneous devices with memory limitations

This section presents a general algorithm that, given a polyhedral parallel computation with regular access patterns, determines how to split in regular parts the grid of tiles/blocks of threads, in such a way that the number of parts is minimal, and the memory requirements of each part does not exceed an arbitrary memory limit. To introduce the basic concept we present first the special case for 1-dimensional domains. Then, we present the solution for 2-dimensional domains. Algorithms for higher dimensions can be deduced from these ones.

To simplify the presentation, in the following algorithms we assume that the thread index space has stride 1, and starts at 1, for all dimensions. It is straightforward to extend the algorithm to use generic thread index domains with any stride or starting positions.

A. Inputs/Outputs

The algorithms have the following parameters:

Input:	The device memory limit $devLim \in \mathbb{N}$.
Input:	The dimensional sizes of the grid of tiles/blocks $\vec{g} \in \mathbb{N}^n$.
Input:	The dimensional sizes of the tile/block $\vec{b} \in \mathbb{N}^n$.
Input:	A collection of data structures or tiles $T_0, \ldots, T_m$.
Input:	A collection of access patterns, one for each tile $AP_0, \ldots AP_m$.
Output:	The number of blocks in each dimension that will form a subpart $\vec{r} \in \mathbb{N}^n$.

B. Algorithm for 1-dimensional spaces

The algorithm is based on determining the linear increasing rate of memory requirements when more blocks are grouped together, and represent it with a line equation. Substituting the device memory limit into the equation, we can obtain the higher number of blocks which memory requirements fits in the available space.

1. $B_1 = ((1, b, 1)), B_2 = ((1, 2 \times b, 1))$
2. $s_1 = \sum_i |\Gamma(d(T_i), AP_i, B_1)|, \ s_2 = \sum_i |\Gamma(d(T_i), AP_i, B_2)|$
3. Compute $\alpha, \beta, \gamma : 0 = \alpha x + \beta y + \gamma$ is the line equation that contains both $(1, s_1)$ and $(2, s_2)$.
4. Return $r = \lfloor -(\beta \cdot devLim + \gamma)/\alpha \rfloor$

C. Algorithm for 2-dimensional spaces and beyond

For two dimensional spaces we obtain a plane equation for the memory requirements of three samples of block groups. Substituting the device memory limit into the equation, we obtain a line equation. The points of this equation determine

Vector addition

1. $\forall i \in d(\vec{z})$
1.1. $z_i = x_i + y_i$
2. Return $\vec{z}$

Cellular automata

1. for i=1…t
1.1. $A' = A$
1.2. $\forall (i,j) \in d(A)$
1.2.1. $A(i,j) = (A'(i-1,j) + A'(i+1,j) + A'(i,j-1) + A'(i,j+1))/4$
2. Return A

Matrix-matrix multiplication

1. $C = 0$
1. $\forall (i,j) \in d(C)$
1.1. $\forall\ k \in [0, m-1]$
1.1.1. $C(i,j) = C(i,j) + A(i,k) \times B(k,i)$
2. Return C

Fig. 2. Algorithms for the three study cases.

the best candidates for the solution. These candidates are checked to determine which one leads to less number of parts due to better alignment of multiples of the new superblock sizes with the grid dimensions.

1. $B_1 = ((1, b_0, 1), (1, b_1, 1)), B_2 = ((1, b_0, 1), (1, 2 \times b_1, 1)), B_3 = ((1, 2 \times b_0, 1), (1, b_1, 1))$
2. $s_1 = \sum_i |\Gamma(d(T_i), AP_i, B_1)|$,
   $s_2 = \sum_i |\Gamma(d(T_i), AP_i, B_2)|$,
   $s_3 = \sum_i |\Gamma(d(T_i), AP_i, B_3)|$,
3. Compute $\alpha, \beta, \gamma, \delta : 0 = \alpha x + \beta y + \gamma z + \delta$ is the plane equation that contains $(1, 1, s_1)$, $(1, 2, s_2)$, and $(2, 1, s_3)$.
4. Substitute $z = devLim$ to obtain a line equation $0 = \alpha x + \beta y + \delta'$.
5. $\forall\ \vec{r} = (r_0, r_1) : r_0 = \lfloor q_0 \rfloor, r_1 = \lfloor q_1 \rfloor : 0 = \alpha q_0 + \beta q_1 + \delta'$
5.1. Compute $k(\vec{r}) = \lceil g_0/r_0 \rceil \times \lceil g_1/r_1 \rceil$
6. Return $\vec{r}$ with the minimum value of $k(\vec{r})$.

D. Study cases

In this section we present some examples of regular kernels and applications to show how our model can be used to express different access patterns. The base algorithms for the study cases are presented in Fig. 2.

1) Vector addition: This simple kernel computes $\vec{z} = \vec{y} + \vec{x}$ using one thread to compute the result of each z_i element. It uses a 1-dimensional thread space of as many threads as elements in the arrays. The access pattern for this kernel have a single access expression:

$$A_0 < 1, 0, 1, 0, 1 >$$

Thus, the resulting signature

$$S' = A_0(S, \vec{x}) = (1 \times x_0 + 0, 1 \times x_0 + 0, 1) = (x_0, x_0, 1)$$

contains only one point in its range $\breve{S}' = \vec{x}$.

2) Stencil program: Cellular automata: This is an example of an stencil application in a two dimensional array space. It implements a PDE solver to compute the heat distribution is a 2-dimensional discretized space using the Jacobi method. The application has a step loop that applies a stencil computation, computing the new value of a matrix position using the old values of its four neighbors. There is only one input/output parameter, a matrix A.

The thread domain is the same as the matrix index domain. Each thread compute one matrix position. All threads synchronize on each i loop step.

The access pattern for this kernel can be expressed with one access expression for each matrix access, or in a compact form with only one expression:

$$A = (A_0 < 1, -1, 1, 1, 1 >, A_1 < 1, -1, 1, 1, 1 >)$$

Thus, the resulting signatures are

$$S'_0 = A_0(S_0, \vec{x}) = (x_0 - 1, x_0 + 1, 1)$$
$$S'_1 = A_1(S_1, \vec{x}) = (x_1 - 1, x_1 + 1, 1)$$

This compact form directly includes in the access pattern result the four *corner* elements that are not really accessed. However, the resulting domain is a parallelotope. When the pattern is applied to a subset of the thread index space, the amount of added data is negligible, and the parallelotope shape conveniently simplifies the movement of data between node and device memories.

Note that, for threads in the limits of the thread domain, the resulting accessed pattern exceeds the limits of the original matrix. To avoid the use of costly conditional evaluations in the fine-grain threads, the A matrix should be extended with *ghost borders*, or the thread index space should be reduced by one element on each border.

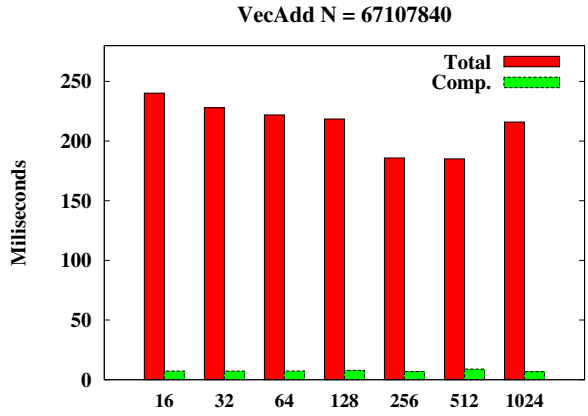
3) Matrix multiplication: In all the previous examples the resulting domains do not need to take into account the domain description of the data structures. Thus, the input signatures on the access expressions are simply ignored.

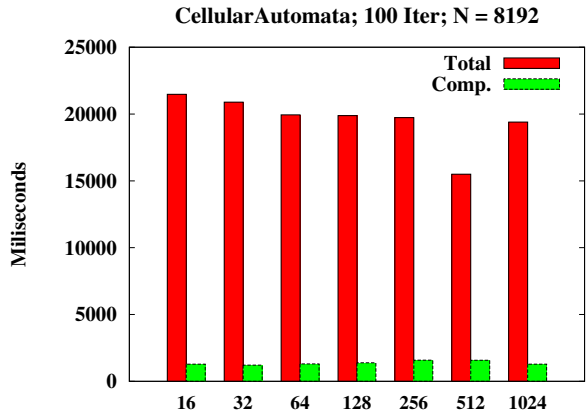
This study case is a direct implementation of the classical matrix-matrix multiplication $C_{n,n} = A_{n,m} \times B_{m.n}$, with three loops. It implements a fine-grain parallelization of the first two loops. Each thread executes the third loop to compute one position of the resulting matrix.

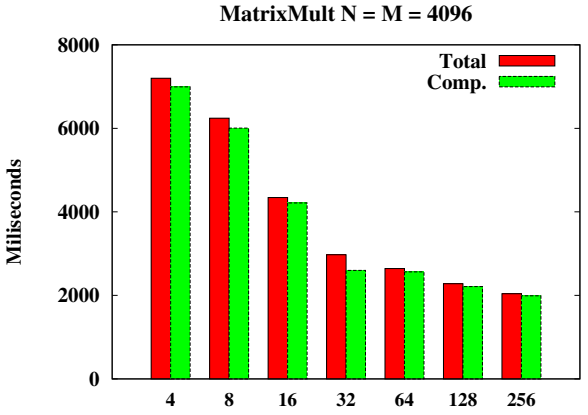
There are three different access patterns for this application, one for each matrix. Each pattern has a single access expression:

For matrix A: $(A_0 < 0, -\infty, 0, +\infty, 1 >, A_1 < 1, 0, 1, 0, 1 >)$
For matrix B: $(A_0 < 1, 0, 1, 0, 1 >, A_1 < 0, -\infty, 0, +\infty, 1 >)$
For matrix C: $(A_0 < 1, 0, 1, 0, 1 >, A_1 < 1, 0, 1, 0, 1 >)$

This access patterns indicate that each thread accesses to a full row of the A matrix, a full column of the B matrix, and one element of the C matrix, with the same indexes as the thread.







Vector Addition

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	.	.	49	25	13	7	3	2	1
Kernel size	.	.	.	.	16	32	64	128	256	512	767

Cellular Automata

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	.	.	48	24	13	7	4	2	1
Kernel size	.	.	.	.	11	21	43	85	170	241	512

MM Multiplication

Memory limit MBs	1	2	4	8	16	32	64	128	256	512	1024
#Kernels	.	.	1103	433	128	43	19	9	1	.	.
Kernel size	.	.	0.4	1.2	4	12	28	60	192	.	.

Fig. 3. Execution times for: (a) Vector addition; (c) Stencil computation; (d) Matrix-matrix multiplication. The tics in the x-axis indicate the value of the memory-size-limit parameter. The table shows for each program and each memory-size-limit value, the number of sub-kernels generated by our system for this case, and the memory size actually used.

V. Experimental study

We have developed a prototype implementation of the algorithms presented in Sect. IV. The implementation uses Hitmap, a library for automatic partition and mapping of parallel applications using hierarchical tiling arrays, that was briefly described in Sect. II. Our prototype layer implements the automatic computation of the best partition, the transparent movement of the required portions of the data structures to/from the target device memory, and the sequential execution of each part as a different kernel. The hidden layer is integrated in a new kernel launching function, that receives one access pattern specification along with each tile parameter.

We have implemented the three study cases presented above using the new tools. The codes are similar to the original ones, with expressions of the access patterns for each data structure involved in the computation. We have tested the prototype with a GPU target device, manually changing the memory-size-limit parameter to simulate different scenarios.

Our experimental platform is a GForce GTX 680 (Kepler, 2048 MB GDDR5) NVIDIA GPU device. The host machine is a 64-bits Intel(R) Core(TM) i7 CPU 960 3.20GHz, with a global memory of 6 GB DDR3. It runs an UBUNTU desktop 10.10 (64 bits) operating system. The applications have been developed using CUDA 4.2 toolkit and the 295.41 64-bit driver.

The use of integer or float data element lead to practically the same execution time in CUDA. Thus, we select integer as data type. The data structures size chosen for each benchmarks are different, in order to obtain stable execution time values. The number of items are the following: (1) Vector addition: $n = 67\,107\,840$; (2) cellular automata: $n = m = 8\,192$, and (3) matrix multiplication $n = m = 4\,096$. These sizes are multiple of the selected threadBlock size to avoid any padding operation.

To simulate results for different kinds of devices, we decided to manually change the memory-size-limit parameter. We have selected values that are powers of two in the range of 1 to 1024 Mbytes. For each kernel there is a different range of this parameter that leads to a feasible number of sub-kernels with a reasonable kernel size. Figure 3 shows the execution times (in milliseconds) obtained for some memory-size-limit parameter values. The first bar indicates the total execution time, while the second bar indicates the time devoted to real computation. The rest of the time is spent in node-device communications.

The results show that, as expected, for the kernels with low computational load per thread (vector addition and cellular automata), the ratio of communication vs. computation is very high, being very small in the remaining cases. When communication times dominate the total execution time, we observe a trend to reduced communication times for particular memory restrictions. This effect can be explained by the fact that the PCI Express bus works faster for memory transactions of particular sizes. Thus, when the subkernels generated require memory sizes that fit well in the PCI bus, the communication times are reduced. This information can be exploited by a library to split the communication in proper block sizes [10].

For the unidimensional example, vector addition, we can

see that the algorithm generates kernels that fit the memory limit almost perfectly. However, this is not the case for 2-dimensional problems. In the current implementation, the stage 5 of the 2-dimensional algorithm has not been yet implemented, leading to suboptimal partition results. However, the performance results show the same trends when manually selecting the best candidate.

The intensive reutilization of caches by the concurrent dot products in matrix multiplication application, leads to reduced total execution times when the kernels have bigger sizes.

The results show that the hidden layer does not impose a substantial overhead on the execution of the whole computation, and it can take away the burden of considering memory-size restrictions from upper mapping layers. Moreover, a deeper research on the information provided by the access patterns may also leads to detect situations where the system can get profit of the artificial automatic partition of the kernels to improve performance results.

VI. Conclusions

In this paper we present a model of parallel computations that allows to build a transparent mapping layer that divides and executes a computation taking into account the memory restrictions of the assigned device. The model requires the programmer to specify the access patterns of the computation threads in a simple abstract form. This information is used at run-time to compute the pieces of data-structures required by a generic partition, and to determine the best partition that ensures that each subpart fits in the device memory.

We discuss an implementation of this concept into an automatic mapping tool that allows to apply high-level distributions in heterogeneous devices without the need to take into account the memory limitations of the target devices. Our experimental results show that feasibility of the solution proposed.

Future work includes a further study of the opportunities to deal with more irregular access patterns at run-time, adding to the model considerations about optimal kernel sizes, and analyzing memory transactions cost between node and target devices.

Acknowledgments

The authors would like to thank Prof. Murray Cole for many fruitful discussions. This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (Mogecopp project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E); and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

References

[1] C. de la Lama, P. Toharia, J. Bosque, and O. Robles, "Static multi-device load balancing for opencl," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, july 2012, pp. 675 –682.

[2] A. Binotto, C. Pereira, and D. Fellner, "Towards dynamic reconfigurable load-balancing for hybrid desktop platforms," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*, April 2010, pp. 1 –4.

[3] E. Burrows and M. Haveraaen, "A hardware independent parallel programming model," *Journal of Logic and Algebraic Programming*, vol. 78, pp. 519–538, 2009.

[4] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Employing multiple cuda devices to accelerate ltl model checking," in *Proc. ICPADS'2010*, dec. 2010, pp. 259 –266.

[5] Y. Torres, A. Gonzalez-Escribano, and D. Llanos, "Encapsulated synchronization and load-balance in heterogeneous programming," in *Euro-Par 2012 Parallel Processing*, ser. LNCS. Springer Berlin Heidelberg, 2012, vol. 7484, pp. 502–513.

[6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Proc. ACM PPoPP'08*. New York, NY, USA: ACM, 2008, pp. 1–10.

[7] A. Aji, J. Dinan, D. Buntinas, P. Balaji, W. chun Feng, K. Bisset, and R. Thakur, "Mpi-acc: An integrated and extensible approach to data movement in accelerator-based systems," in *Proc. HPCC-ICESS'2012*, june 2012, pp. 647 –654.

[8] C. de Blas Cartón, A. Gonzalez-Escribano, and D. R. Llanos, "Effortless and Efficient Distributed Data-Partitioning in Linear Algebra," in *Proc. HPCC'2011*. IEEE, Sep. 2010, pp. 89–97.

[9] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Automatic Data Partitioning Applied to Multigrid PDE Solvers," in *PDP'2011*. IEEE, Feb. 2011, pp. 239–246.

[10] F. Song and J. Dongarra, "A scalable framework for heterogeneous gpu-based clusters," in *Proc. ACM SPAA'2012*. New York, NY, USA: ACM, 2012, pp. 91–100.