Experimental evaluation of memory optimizations on an embedded GPU platform

Innocenzo Mungiello
University of Naples Federico II

***Abstract*—This paper presents the experimental evaluation of data mapping techniques in the shared memory of an embedded GPU. The evaluated technique, previously presented in the literature in other contexts, aims at partitioning an array across the shared memory physical banks, so as to increase parallel accesses, resulting in appreciable gains in terms of both performance and energy efficiency. The paper presents the experimental setup used for characterizing physically the behaviour of the platform, allowing a validation and a closer understanding of the evaluated memory mapping technique.**

I. Introduction

Because of the end of Dennard Scaling, since the early 2000s manufacturers have been forced to explore new architectural paradigms, including multi- and many-core solutions as well as Graphics Processing Units (GPUs), now being used for a variety of applications including the embedded domain [1]. In addition, Field-Programmable Gate Arrays (FPGAs) also provide a solution for building customized computing platforms, although they pose non-trivial challenges mostly involving complexity, high-level design as well as interplay with suitable programming models [2], [7], [8], [12], somewhat mitigated by GPUs. A crucial aspect of recent GPU and FPGA architectures is their power-efficiency, i.e., the rate of computation that can be delivered by a system for every watt of power consumed, measured in GFLOPS-per-Watt, today considered a primary performance metric for the scalability of computing platforms.

An interesting observation is that in current computing technologies a major component of power consumption is due to data movement rather than mere processing, which is indeed a widely studied subject in a number of different contexts [11], [5] In particular, looking at today's GPUs, the power contribution of data movement compared to processing can be as high as 85%. This work thus addresses the impact on performance and energy efficiency of memory optimization techniques for heterogeneous architectures, particularly GPUs, in an embedded setting. In particular, the paper focuses on the experimental evaluation of data mapping approaches previously developed in the area of non-uniform memory architectures and transferred to the case of GPUs. The emphasis is on the multi-bank organization of the on-chip GPU shared memory, where the whole shared addressing space is partitioned in a cyclic way and is potentially subject to an access conflict problem. We evaluate a data layout transformation impacting access patterns, which enables a significant gain in terms of both performance and energy efficiency, as confirmed by an experimental set-up used for performing measures on a physical embedded platform.

The paper is structured as follows. Section II discusses the background and the motivation of this work. Section III recapitulates the approach adopted for data partitioning. Section IV shows the set-up used for the experimental evaluation. Section V concludes the paper with some final remarks.

II. Background

Memory mapping has traditionally been an important optimization problem for high-performance parallel systems [11]. Today, these issues are increasingly affecting a much wider range of platforms. In fact, many medium/high-end embedded systems are now based on parallel compute architectures while, at the opposite end of the spectrum, large datacenters currently play a central role for popular cloud-based applications, with a whole range of new disparate challenges, from architecture optimization to security as well as workflow management and validation [19], [17], [29], [32]. Although here we are fundamentally interested in the embedded architecture level, all such platforms are characterized by inherently the same issue concerning the memory infrastructure organization, i.e. the fact that, at the low-level, they are based on non-uniform memory access (NUMA) which, depending on the application access patterns, may be critical to the overall performance. In fact, the NUMA model reflects a scenario where multiple independent processing cores/nodes with local memory modules are connected by some form of interconnect, causing the access time to depend on the location relative to the processor placing the access operation [6]. A closely related concept, distributed shared memory (DSM), is a form of memory architecture where physically separate memories can be addressed as one logically shared address space. DSM systems combine the best features of shared-memory and distributed-memory machines. They support the convenient shared-memory programming model on scalable distributed-memory hardware, exposing a simpler abstraction for data passing to the application programmer. Furthermore, many distributed parallel applications execute in phases, where each computation phase is preceded by a data-exchange phase. The time needed for the data-exchange phase is often dictated by the throughput limitations of the communication system. Distributed shared memory algorithms typically move data on demand as they are being accessed, eliminating the data-exchange phase, spreading the communication load over a longer period of time, and allowing for a greater degree of concurrency. Also, the total amount of memory may be increased proportionally, reducing paging and swapping activity [26], [31]. However, although many DSM systems have been proposed and implemented (see Bal et al. [16], Bershad et al. [20], Chase et al. [21], Dasgupta et al. [22], Fleisch and Popek [23], Li and Hudak [26], Minnich and Farber [27], and Kirk L. Johnson et al. [33]), achieving good performance on DSM systems for a sizable class of applications has proven to be a major challenge [15]. One of

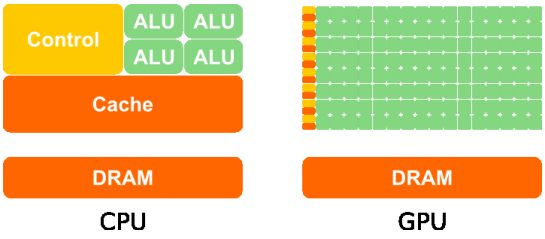

Fig. 1. CPU architecture vs. GPU architecture

the key problems in building an efficient software DSM system is to reduce the amount of communication needed to keep the distributed memories consistent. Often, the proposed solutions result in a trade-off between performance and consistency models, with the aim of enhancing the concurrency available in the distributed shared memories [37]. Another problem is to avoid *access conflicts* to physically different memory banks from multiple threads/processes running concurrently. This problem can impact greatly the performance of the system, especially in distributed systems, since it causes serialized accesses and a significant interconnect overhead. A large number of works addressed this problem, e.g. Das et al. [28] considered the star-template access on two specific host topologies, tori and hypercubes, enabling conflict-free mappings using an optimal or provably good number of memory modules. Monchiero et al. [30] propose a mechanism for data allocation on a distributed shared memory space, dynamically managed by an on-chip hardware memory management unit. Sung et al. [10] present automatic data layout transformation as an effective compile-time performance optimization for memory-bound structured grid applications.

A. Memory in heterogeneous parallel systems

As implied by the above introduction, traditional solutions for memory mapping optimization in NUMA contexts such as parallel computer/datacenters might play a key role also in today's embedded platforms. In fact, many of such computing platforms are increasingly being provided with high-end GPU and/or FPGA units, where parallel processing elements access simultaneously several independent memory banks through complex interconnects. This potentially provides an opportunity for improving the memory bandwidth available to the application, provided that one adopts suitable memory partitioning strategies based on the actual access patterns [5]. In particular, GPUs represent today a major paradigm shift in computing architecture focusing on increasing the execution throughput of parallel applications. A current representative instance of this trend is the NVIDIA GeForce GTX680 graphics processing unit (GPU) with 16384 threads, executing in a large number of simple, in-order pipelines. As of 2012, the ratio between many-core GPUs and multi-core CPUs for peak floating-point calculation throughput was about 10 to 1 [4], mainly due to the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1. This approach is natively supported by the Compute Unified Device Architecture (CUDA) programming model, introduced by NVIDIA in 2007, and is reflected by more recent programming models such as OpenCL (now at version 2.0), OpenACC, C++ AMP and OpenMP 4.0.

For performance purposes, the on-chip shared memory of a GPU device is normally divided in banks, which can be accessed in a parallel way from all the threads in a warp. The number of banks is strictly dependent on the architecture. As an example, in an NVIDIA Kepler architecture the number of banks is 32 and each bank has a word of 8 bytes. Data allocated in the shared memory are cyclically distributed over the banks in two ways:

- **4-byte access**: Successive 4-byte words are mapped to successive banks. The memory can be seen as made of 32 banks, each 4-byte wide. If the data that we use is 8 bytes wide, the access mode becomes an 8-byte access.
- **8-byte access**: Successive 8-byte words are mapped to successive banks.

We can easily compute the bank where a data is mapped to, as follows:

- (8B word index) mod 32;
- (4B word index) mod $(32 \cdot 2)$;
- (byte address) mod $(32 \cdot 8)$

Figure 2 shows an example of data mapping on shared memory with both modes. In this example the data are 4 byte-word index and, for simplicity, we show only four banks.

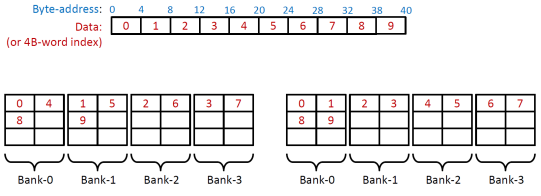

Fig. 2. Comparing bank mode mappings. On the left-hand side we have a 4-byte access. On the right-hand side we have an 8-byte access.

The 4-byte access is the default mode. We can change it by using the CUDA function *cudaDeviceSetSharedMemConfig*(`param`) where `param` can be:

`cudaSharedMemBankSizeEightByte` , or

`cudaSharedMemBankSizeFourByte`.

The access mode can affect the performance of a kernel. In fact, allowing all the threads in a warp to fetch data in parallel from this multi-bank memory can lead to great performance improvements, but it requires explicit, bank-aware organization of the data layout. There are three main mechanisms for shared memory access which guarantee improved performance:

- **Unicast**: each thread in a warp tries to access a different location stored in a different bank.
- **Multicast**: One or more groups of threads *in a warp* try to access the same location stored in one of the banks. The other threads perform a unicast-style access.
- **Broadcast**: Every thread in a warp will access exactly the same location, obviously stored in the same bank.

These three mechanisms are enabled by an interconnection network which links each core of a Streaming Multiprocessor to the shared memory. By using this interconnection network and performing one of these access patterns, data can be retrieved at low latency as they were stored in the registers. If the pattern is different from those described above, shared memory performance may easily decrease. In particular, in the case where two or more threads in the *same* warp try to access *different words* stored in the same bank, the interconnection network is no more able to provide the required data to all the threads in parallel. This situation, called *bank conflict*, is a major problem related to the use of the on-chip shared memory. In particular, if two threads try to access different words stored in the same bank, we have a 2-way bank conflict. If three threads try to access different words stored in the same bank, a 3-way bank conflict occurs, and so on. The worst case involves all 32 threads in a warp trying to access different words stored in the same bank, causing a 32-way bank conflict. Whenever a conflict occurs, it is resolved by serializing the accesses in time. As an example, the serialization in a 2-way scenario leads to doubled latency and can increase the energy consumption considerably.

III. Data layout transformation

As already mentioned above, heterogeneous platforms provide some form of customizability that can be effectively exploited to improve performance and power efficiency. While FPGAs offer the possibility to jointly customize the memory infrastructure architecture and the application task mapping, by using one of several approaches in the literature [5], [14], GPUs have still enough flexibility to expose the physical shared memory bank structure to the programmer, enabling bank mapping to be tailored on the application access patterns. Here we shortly review a literature approach to data mapping in parallel architectures [11], that can be effectively applied to the case of the GPU shared memory, as shown in this paper.

The results presented here apply to affine static control parts (SCoPs), i.e., code segments in performance-critical loops where loop bounds, conditionals, and subscripts of memory references are affine functions of the surrounding loop iterators and of constant parameters possibly unknown at compile-time. For each reference to an array A in the loop nest, call *memory access function* a correspondence F associating each element of A with a value of the iteration vector $\overrightarrow{v}$, which is the vector having as elements the indices of the loop nest containing the reference. Since the subscripts in SCoP code are affine functions, F can always be expressed as $F = \mathbf{F} \cdot \overrightarrow{v} + \overrightarrow{c}$, where $\mathbf{F}$ is an integer matrix and $\overrightarrow{c}$ is a constant displacement.

Furthermore, we need a mathematical formulation of the bank mapping in the GPU shared memory. The cyclic scheme described in the previous section can be seen as a special case of an allocation expressed as a *modular mapping* function $\sigma\left(\overrightarrow{l}\right) = \mathbf{M} \cdot \overrightarrow{l} \mod \overrightarrow{m}$, associating each index $\overrightarrow{l}$ of an array element having p components with the corresponding bank [11]. $\mathbf{M}$ is a $p \times n$ integer matrix, $\overrightarrow{m}$ is p-dimensional array of integer moduli[1], and the modulo operation is component-wise. We regard the physical banks making up the GPU shared memory as a linear array, hence ($p = 1$). Assume that we have a bi-dimensional array to allocate and let $\begin{bmatrix} x \\ y \end{bmatrix}$ be the indices of its elements. The mapping problem can thus be expressed as:

$$Bank(x, y) = M \cdot \begin{bmatrix} x \\ y \end{bmatrix} mod\, \overrightarrow{m}$$

where $\overrightarrow{m}$ is in fact mono-dimensional and coincides with the number of available banks, denoted $banks$. The value of this constant depends on the specific GPU architecture. For instance, $banks = 32$ for the NVIDIA Kepler family.

An example of matrix M is:

$$Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \mod banks$$

where N is equal to the size of the array along the x dimension. Table I provides an example for a 52×52 array, highlighting the cyclic scheme followed by data allocation.

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	···	51
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	···	19
1	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	···	7
2	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	···	27
3	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	···	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
51	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	···	15

TABLE I. Mapping of a 52 x 52 array

In essence, avoiding conflicts requires the threads of a GPU warp to access all different banks. We associate each thread with a bi-dimensional identifier (t_x, t_y). Based on the formulation above, each thread (t_x, t_y) needs to access element $F(t_x, t_y)$ and hence bank $Bank(F(t_x, t_y))$. As we are looking into a single warp, there is no need to introduce block identifiers (as intended in CUDA). As an example, a 2×16 warp accessing the previous array clearly incurs bank conflicts, as highlighted in the following Table II.

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	···	51
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	···	19
1	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	···	7
2	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	···	27

TABLE II. Example of a conflict

The repetition of the values $0, 1, 2, 3$ causes here a 2-way conflict. To avoid conflicts, no repetitions must occur in the rectangular domain. Equivalently, the bank mapping function corresponding to the memory reference in the threads must be injective in the rectangular domain covered by the warp.

We consider the class of transformations to SCoP code that change the memory access function by multiplying its expression by a matrix $\mathbf{T}$ [3]:

$$T = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

which also implies changing the layout in memory of the locations concurrently accessed by the threads in a warp. A new allocation can be defined as:

$$Bank(x, y) = \begin{bmatrix} 1 & N \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \mod banks =$$

[1] In general, the set of banks may have a dimensionality equal to p, so that σ returns a p-dimensional bank index. Modular mappings can change the dimensionality of the data address.

$$[\ a + c \cdot N \quad b + d \cdot N\ ] \cdot \begin{bmatrix} x \\ y \end{bmatrix} \bmod banks$$

Matrix **M** can now be written as

$$M = [\ a + c \cdot N \quad b + d \cdot N\ ]$$

In case the transformation matrix **T** results in an injective function over the rectangular domain identified by the dimension of a warp, the transformation ensures *conflict-free accesses*. We relied on the formal treatment in [3] to check whether a given transformation induced by **T** is injective.

Below we exemplify the procedure used to address the problem of bank conflicts. We consider an algorithm performing matrix multiplication. In our quantitative experiments, we choose 52×52 tile dimensions. Consequently, the code snippet is as follows:

```
__shared__ double AS[2704];
__shared__ double BS[2704];
     //Calculate the row index of the C element and A
     int Row = blockIdx.x*blockDim.x+threadIdx.x;

     //Calculate the column index of C an B
     int Col = blockIdx.y*blockDim.y+threadIdx.y;

     if ((Row < WIDTH) && (Col < WIDTH)){
          double Cvalue = 0;
          // each thread computes one element
          // of the block sub-matrix
     #pragma unroll
          for (int k = 0; k < WIDTH; k++)
            Cvalue += AS[(Row*WIDTH)+k]*BS[k*WIDTH+Col];
          C[Row*WIDTH+Col] = Cvalue;
     }
```

We can solve the problem by selecting a specific access performed by a warp. In this case, as an example, we define a warp of 32×1 threads[2]. Without applying any transformation, the accesses to matrix `AS` incur a 4-way conflict. For example, locations `AS[208]`/`AS[624]`, corresponding to iterations with `k=0`, `Col=0`, and `Row=4`/`Row=12`, respectively, executed by concurrent threads in a warp, both access bank $208 \bmod 32 = 624 \bmod 32 = 16$. For matrix `BS`, on the other hand, we have a broadcast access, i.e., all 32 threads access at same bank, with no conflict.

A transformation that can be used to solve the conflicts in matrix `AS` for this warp is

$$T = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$$

identified by enumerating feasible transformation solutions and checking their injectivity by the formal procedure in [3]. This transformation leads to a modified layout of matrix `AS` in memory along with the transformed code below:

```
__shared__ double AS[2805];
__shared__ double BS[2704];
     //Calculate the row index of the C element and A
     int Row = blockIdx.y*blockDim.y+threadIdx.y;

     //Calculate the column index of C an B
     int Col = blockIdx.x*blockDim.x+threadIdx.x;

     if ((Row < WIDTH) && (Col < WIDTH)){
          double Cvalue = 0;
          // each thread computes one element
          // of the block sub-matrix
     #pragma unroll
          for (int k = 0; k < WIDTH; k++)
            Cvalue += AS[(Row*53)+k*2]*BS[k*WIDTH+Col];
          C[Row*WIDTH+Col] = Cvalue;
     }
```

This transformation completely clears the conflict problem. As an example, the previous two accesses performed by iterations with `k=0`, `Col=0`, and `Row=4`/`Row=12` now become `AS[212]`/`AS[636]`, accessing banks $212 \bmod 32 = 20$ and $636 \bmod 32 = 28$, respectively, but the formally proved injectivity condition ensures that this occurs for any thread pair in the rectangular warp. Notice that the conflict-free condition comes at the cost of stretching the memory region covered by matrix `AS`, requiring interleaved placement of other data structures for efficient utilization of the shared memory. Furthermore, the number of arithmetic operations involved in address computation might increase, making the trade-offs with time and energy efficiency less obvious. The experimental evaluation carried out in the next section provides some insights about the overall benefit of the adopted technique.

IV. Experimental Evaluation

In this section we present a set-up used to carry out the experimental evaluation of the above optimization technique and collect performance/power data from a physical platform.

A. System Overview

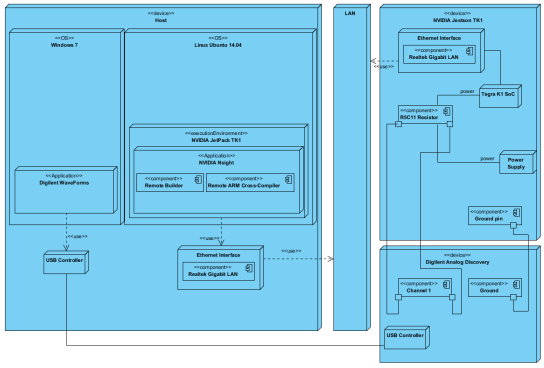

Fig. 3. Overview of the experimental environment

The diagram shown in Figure 3 is a schematic representation of our environment. We have a host PC running a WMware Virtual Machine with Ubuntu 14.04 and the JetPack installed on it. In this environment, we use *NVIDIA Nsight* to write CUDA code and, then, to compile and remotely run it on a Jetson TK1 development board. On the same machine we have a Windows Operating System with *Digilent WaveForms* application installed on it. We use this application as a data logger. The data are collected by the *Digilent Analog Discovery* suitably connected through Channel 1 wire probes to the *R5C11* resistor, available on the Jetson board in order to control the power consumption of the overall platform, by measuring the voltage across this resistor. Figure 4 shows the real system used in our experimental setup.

[2]We can actually define a *block*, not a warp. But if the block is composed by only 32 threads, then, we are defining the shape of the warp too.

Fig. 4. Experimental setup

B. Results

In this section we discuss the transformation results. By using the system described in IV-A it was possible to evaluate the impact of the transformation on power consumption and execution time. Figure 5 shows the results collected using the Analog Discovery.

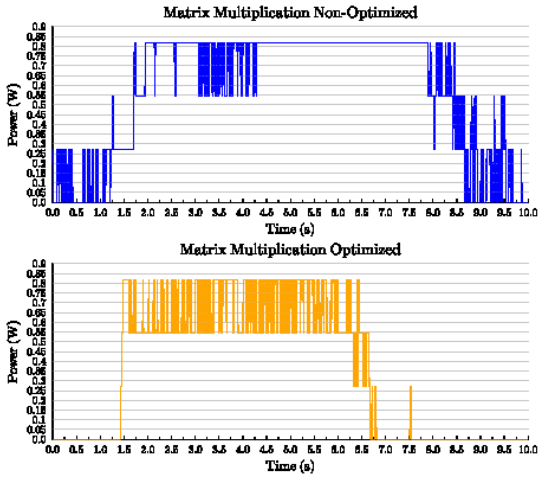

Fig. 5. Power measurements

The instrument does not allow us to differentiate the consumption, in terms of watt, of the two versions, as both versions reach a value of around $0.80\,W$. On the other hand, we can appreciate that the execution time is drastically reduced. Figure 5 also shows that the optimized kernel takes about 5 seconds, instead the non-optimized takes 6.5 seconds. This means that the optimized kernel consumes around $0.80\,W \times 5\,s \simeq 4\,J$, instead the non-optimized consumes around $0.80\,W \times 6.5\,s \simeq 5.2\,J$, totalling a difference of $1.2\,J$, i.e. a gain of 23%.

As mentioned in the previous section, the transformation solves the conflict issue, but at the cost of increasing the arithmetic operations to be performed and the amount of shared memory to allocate. Table III shows the differences, in terms of number of instructions executed by the two kernels. The difference is 1040000 instructions, i.e. a 3.84% overhead. As for the amount of memory to allocate, the optimized kernel requires 808 bytes, or 3.73% more. In terms of performance per watt, the kernel executes 56243200 floating point operations in double precision. So, for the non-optimized kernel we have a value of 10.816 MFLOPS/watt (referred to the subset of the GPU actually used), while the optimized one has a value of 14.0608 MFLOPS/watt, i.e. we have an increase of 30% with this transformation.

Kernel	Number of Instructions
MatMul_52_Optimized	28070952
MatMul_52_No_Optimized	27030952

TABLE III. COMPARISON OF THE NUMBER OF INSTRUCTIONS

V. Conclusions and Future Work

GPUs have become extremely important for today's HPC and Cloud Computing as they provide an effective answer for increasingly stringent energy constraints. This work presented a practical experience centered around the evaluation of an optimization technique for GPU on-chip memory. The experimental results collected pointed out the significant incidence that such techniques may have on both execution time and energy efficiency. As a part of our future work, we plan to build a complete heterogeneous platform pairing embedded GPUs with FPGA units for a range of applications including multimedia, testing, and security [25], [24], [34], allowing an extensive evaluation of optimization techniques combining special-purpose acceleration and software heterogeneous programming.

The presentation of this work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - *MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems*.

References

[1] D. Hallmans, M. Asberg, and T. Nolte, "Towards using the Graphics Processing Unit (GPU) for embedded systems", IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA), Sept. 2012.

[2] P. Coussy and A. Morawiec, *High-Level Synthesis from Algorithm to Digital Circuit*, Springer, 2008.

[3] Darte, Alain, Michele Dion, and Yves Robert, "A characterization of one-to-one modular mappings", Parallel Processing Letters 6.01 (1996): 145-157.

[4] Kirk, David B., and W. Hwu Wen-mei, *Programming massively parallel processors: a hands-on approach*, Newnes, 2012.

[5] A. Cilardo and L. Gallo, "Improving multibank memory access parallelism with lattice-based partitioning," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 45:1–45:25, Jan. 2015.

[6] Manchanda, Nakul, and Karan Anand. "Non-Uniform Memory Access (NUMA)", New York University (2010).

[7] A. Cilardo, L. Gallo, and N. Mazzocca, "Design space exploration for high-level synthesis of multi-threaded applications," *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1171–1183, 2013.

[8] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca, "Efficient and scalable OpenMP-based system-level design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 988–991.

[9] Hagersten, Erik, Anders Landin, and Seif Haridi, "DDM-a cache-only memory architecture", Computer 25.9 (1992): 44-54.

[10] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010.

[11] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," IEEE Transactions on Computers, vol. 54, no. 10, p. 12421257, 2005.

[12] A. Cilardo, E. Fusella, L. Gallo, and A. Mazzeo, "Exploiting concurrency for the automated synthesis of MPSoC interconnects," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, p. 57, 2015.

[13] Nieplocha, Jaroslaw, Robert J. Harrison, and Richard J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers", The Journal of Supercomputing 10.2 (1996): 169-189.

[14] A. Cilardo, D. Socci, and N. Mazzocca, "ASP-based optimized mapping in a Simulink-to-MPSoC design flow," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 108 – 118, 2014.

[15] Carter, John B., John K. Bennett, and Willy Zwaenepoel, "Techniques for reducing consistency-related communication in distributed shared-memory systems", ACM Transactions on Computer Systems (TOCS) 13.3 (1995): 205-243.

[16] Bal, Henri E., M. Frans Kaashoek, and Andrew S. Tanenbaum, "Orca: A language for parallel programming of distributed systems", Software Engineering, IEEE Transactions on 18.3 (1992): 190-205.

[17] Y. K. Sinjilawi, M. Q. AL-Nabhan, and E. A. Abu-Shanab, "Addressing security and privacy issues in cloud computing," Journal of Emerging Technologies in Web Intelligence, vol. 5, no. 2, pp. 192199, 2014.

[18] A. Cilardo, D. De Caro, N. Petra, F. Caserta, N. Mazzocca, E. Napoli, and A. Strollo, "High speed speculative multipliers based on speculative carry-save tree," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 61, no. 12, pp. 3426–3435, Dec 2014.

[19] K. Paranjape, S. Hebert, and B. Masson, "Heterogeneous computing in the cloud: Crunching big data and democratizing HPC access for the life sciences," Intel Corporation, Tech. Rep., 2010.

[20] Bershad, Brian N., Matthew J. Zekauskas, and Wayne A. Sawdon, "The Midway distributed shared memory system", IEEE, 1993.

[21] Chase, Jeffery, et al., "The Amber system: Parallel programming on a network of multiprocessors", Vol. 23. No. 5. ACM, 1989.

[22] Dasgupta, Partha. "The design and implementation of the Clouds distributed operating system", 1990.

[23] Fleisch, Brett, and Gerald Popek, "Mirage: A coherent distributed shared memory design", Vol. 23. No. 5. ACM, 1989.

[24] A. Cilardo, "New techniques and tools for application-dependent testing of FPGA-based components," *Industrial Informatics, IEEE Transactions on*, vol. 11, no. 1, pp. 94–103, Feb 2015.

[25] A. Cilardo and N. Mazzocca, "Exploiting vulnerabilities in cryptographic hash functions based on reconfigurable hardware," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 5, pp. 810–820, May 2013.

[26] Li, Kai, and Paul Hudak, "Memory coherence in shared virtual memory systems", ACM Transactions on Computer Systems (TOCS) 7.4 (1989): 321-359.

[27] Minnich, Ronald G., and David J. Farber. "The Mether system: Distributed shared memory for SunOS 4.0", 1993

[28] Das, Sajal K., Irene Finocchi, and Rossella Petreschi, "Conflict-free star-access in parallel memory systems", Journal of Parallel and Distributed Computing 66.11 (2006): 1431-1441.

[29] F. Moscato, V. Vittorini, F. Amato, A. Mazzeo, N. Mazzocca, "Solution workflows for model-based analysis of complex systems", *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 1, pp. 83-95, 2012.

[30] Monchiero, Matteo, et al, "Exploration of distributed shared memory architectures for NoC-based multiprocessors", Journal of Systems Architecture 53.10 (2007): 719-732.

[31] Stumm, Michael, and Songnian Zhou, "Algorithms implementing distributed shared memory", Computer 23.5 (1990): 54-64.

[32] F. Moscato, F. Amato, A. Amato, R. Aversa, "Model-driven engineering of cloud components in MetaMORP(h)OSY", *International Journal of Grid and Utility Computing*, Inderscience Publishers Ltd, vol. 5, no. 2, pp. 107-122, 2014.

[33] Johnson, Kirk Lauritz, M. Frans Kaashoek, and Deborah A. Wallach, "CRL: High-performance all-software distributed shared memory", Vol. 29. No. 5. ACM, 1995.

[34] A. Cilardo, "Exploring the potential of threshold logic for cryptography-related operations," *Computers, IEEE Transactions on*, vol. 60, no. 4, pp. 452–462, April 2011.

[35] ——, "Efficient bit-parallel GF(2^m) multiplier for a large class of irreducible pentanomials," *Computers, IEEE Transactions on*, vol. 58, no. 7, pp. 1001–1008, July 2009.

[36] Cierniak, Michał, and Wei Li, "Unifying data and control transformations for distributed shared-memory machines", Vol. 30. No. 6. ACM, 1995.

[37] Hutto, Phillip W., and Mustaque Ahamad, "Slow memory: Weakening consistency to enhance concurrency in distributed shared memories", 10th IEEE International Conference on Distributed Computing Systems, 1990.