Dynamic Management of Multikernel Multithread Accelerators Using Dynamic Partial Reconfiguration

Alfonso Rodríguez, Juan Valverde, Eduardo de la Torre and Teresa Riesgo

Abstract—**Ever demanding systems with restricted resources face increasingly complex applications. Additionally, changeable environments modify working conditions over time. Therefore, a dynamic resource management is required in order to provide adaptation capabilities. By using ARTICo^3, a bus-based architecture with reconfigurable slots, this adaptation is accomplished in three different but dependent areas: Consumption, Confidentiality and fault tolerance, and Computation. The proposed resource management strategies rely on an architecture and a model of computation that make execution configuration to be application-independent, but context-aware, since a CUDA-like execution model is used. The inherent and explicit application-level parallelism of multithreaded CUDA kernels is used to generate hardware accelerators that act as thread blocks. Despite other modes of operation provided by the ARTICo^3 architecture, like module redundancy or dual-rail operation to mitigate Side-Channel Attacks, these thread blocks are dynamically managed and their execution is scheduled using a multiobjective optimization algorithm.**

Keywords—Dynamic and Partial Reconfiguration, Dynamic Resource Management, FPGA, Parallel Computing.

I. Introduction

In the last few years, technology trends have increased significantly application complexity. Some of these applications require intensive data-processing capabilities. High Performance Computing (HPC) tries to find solutions for all these applications, and parallel computing in particular is showing several promising results.

Parallel computing appears as an alternative to those approaches in which performance has stalled mainly due to technology limitations. These limitations include, but are not limited to, resource availability, power consumption and system frequency scaling. However, parallel computing has some important drawbacks, being design complexity the most relevant one. Parallel models of computation, as well as their programming paradigms are, by far, more difficult to understand and implement in actual systems, especially for those who come from pure software backgrounds.

In an attempt to minimize the impact on engineers and developers, different alternatives have been proposed. All these approaches share a common starting point: a high-level programming language, which can be considered as basic knowledge in any engineering degree nowadays. NVIDIA CUDA (Compute Unified Device Architecture), for instance, provides intuitive integration within any C/C++ programming environment, since parallel sections, i.e. kernels, are declared and invoked as if they were common functions. Another important trend uses algorithm description in C/C++/SystemC to generate specific hardware modules by means of High Level Synthesis (HLS). In addition, some important device vendors have released development suites that allow designers to generate custom hardware peripherals without even requiring previous Hardware Description Language (HDL) knowledge, e.g. Xilinx Vivado HLS.

CUDA-based parallel computing provides a flexible multiprocessor architecture in which concurrent execution is achieved by means of multiple software threads. Traditional FPGA (Field Programmable Gate Array) static hardware-acceleration approaches, on the other hand, might show some important limitations, mainly due to resource availability bottlenecks. In addition, these systems must have self-reconfiguration capabilities, in order to remain as flexible as any software alternative. Hence, different tasks can be performed using the same hardware resources, only changing their configuration. Thus, a dynamic resource management strategy is required in order to best optimize resource usage while meeting other internal or external requirements, such as changing operating conditions.

In this paper, a CUDA-like Execution Model as well as Dynamic Partial Reconfiguration (DPR) are combined with a dynamic resource manager that is capable of boosting performance while keeping hardware implementation advantages. This module manages the internal resources of an embedded-system oriented architecture called ARTICo^3, and is designed so that not only internal information, e.g. power consumption, available resources, but also external conditions, e.g. Side-Channel Attack (SCA) protection needs, fault-tolerant operation, or even parallel-processing requirements, are taken into account when dynamically mapping tasks into hardware resources.

The rest of this work-in-progress paper is organized as follows. Section II reviews the current trends and state-of-the-art research lines. Section III presents an overview of the target virtual architecture in which dynamic resource management is required. Then, the execution model of the ARTICo^3 architecture and its relationship with CUDA is analyzed in

Section IV. The dynamic resource manager is presented in Section V, followed by the conclusions in Section VI.

II. RELATED WORK

FPGA-based reconfigurable accelerator schemes have become a common practice in embedded system design. However, the methodology that is used in order to generate these hardware accelerators changes, ranging from custom hardware description in HDL languages to higher abstraction levels, e.g. Electronic System Level (ESL) design.

In order to dynamically handle these hardware-acceleration modules once they have been generated, different dynamic resource management strategies have been proposed. These management policies may range from simple implementations that only take into account resource availability, to complex management schemes that take into account task priorities and requirements, memory usage, dependencies among tasks, etc.

In this section, related work in the fields of Dynamic and Partial Reconfiguration, Parallel-Processing Hardware Generation and Dynamic Resource Management is presented.

A. *Dynamic and Partial Reconfiguration*

Dynamically and partially reconfigurable architectures rely on reconfiguration engines that allow hardware reusability within the same chip [1] [2]. These reconfiguration engines manage the process of modifying the device configuration memory by sending commands and configuration information through a reconfiguration port. In the case of FPGAs, configuration information is stored as configuration files called bitstreams. Depending on the device family, different reconfiguration ports are available. In order to support self-reconfiguration, the reconfiguration ports must be accessible from inside the FPGA, e.g. the ICAP (Internal Configuration Access Port) [3] in most Xilinx modern FPGAs. However, in Zynq devices, there is an additional internal reconfiguration port: the PCAP (Processor Configuration Access Port) [4].

Dynamic and Partial Reconfiguration (DPR) has been a trending topic during the last few years [5]. Many different techniques such as virtual reconfiguration in Flash based FPGAs [6] or slot based architectures [7] have been presented. Several design methodologies have been widely studied to optimize the reconfiguration process like in [8].

Now, using DPR as a tool is opening a huge scope of solutions for many applications, as in the case of this work where DPR is used to provide embedded systems with dynamic resource allocation and reutilization.

B. *Parallel-Processing Hardware Generation*

After almost twenty years of latent activity, HLS tools are now a feasible alternative to generate specific hardware using high-level programming languages. A good example can be found in [9], where an HLS tool that takes a bit-accurate C/C++ algorithm specification is presented. However, there is no such thing as a standard design flow that, taking any high-level code as the starting point, generates hardware accelerators for a given task.

The FASTCUDA Project [10], which stands for Open Source FPGA Accelerator and Hardware-Software Codesign Toolset for CUDA Kernels, appears in order to establish a common framework in which different design methodologies are merged to provide both embedded hardware-software codesign and ESL design. The FASTCUDA toolset is capable of generating both hardware-specific accelerators and parallel software code that runs in a multiprocessor-based system from plain CUDA kernel descriptions, taking advantage of HLS properties and the explicit parallelism of CUDA.

The most remarkable feature of the FASTCUDA toolset regarding hardware accelerator generation is that kernels are not implemented as monolithic modules, as opposed to traditional HLS approaches, but as thread blocks. Therefore, the final implementation has better performance, since processing is not only accelerated with hardware modules, but also parallelized by replicating thread blocks. In addition, these thread blocks can be generated with a variable number of threads, so that a static library of different blocks with the same functionality is available, being the number of threads the only difference among them. However, these thread blocks have to be defined and allocated statically at design time and thus, they are not interchangeable.

C. *Dynamic Resource Management*

Dynamic and Partial Reconfiguration permits hardware changes at run time. However, the required time to reconfigure the logic inside the FPGA device is not negligible. The larger the hardware accelerator is, the more time is needed to finish the reconfiguration process. Therefore, an efficient reconfiguration management strategy needs to be implemented, so that a lot of time can be saved. Some alternatives have been proposed in the literature. In [11], a dynamic reconfiguration manager handles a certain number of reconfigurable slots. Each slot is considered as a slave module that has been defined at design time, and has its own control FSM (Finite State Machine). In addition, a SystemC simulation model is provided, so that dynamically reconfigurable systems can be simulated and analyzed using their VCD (Value Change Dump) outputs.

Another example that includes a more complex approach can be found in [12], where a layered dynamic resource manager for FPGA-based reconfigurable systems is presented. The lower layer corresponds to the reconfiguration engine, which has been implemented in hardware. The upper layer, on the contrary, is a set of software application drivers that can be used in the main application which is running in an embedded microprocessor. This layer performs all the required scheduling tasks using both design-time and run-time information.

Dynamic resource management strategies are, in general, focused on optimizing resource utilization and achieving high-performance parallel processing [13]. Therefore, the vast majority of approaches in the literature might not be useful in environments with changing conditions or with specific requirements that change over time. Since no solution is available yet, a completely different alternative has to be proposed in order to deal with those systems. In this work, a combined solution is proposed, including a system that takes advantage of the CUDA execution model and DPR features to accelerate parallel computations, while at the same time being aware of the changing conditions regarding available power budget, or dependability needs. It is important to highlight that the amount of allocated resources changes according to the working conditions, and independently of the application code.

III. ARTICo^3 VIRTUAL ARCHITECTURE

In order to fully exploit dynamic resource management capabilities, an embedded architecture called ARTICo^3 is used. ARTICo^3 (Reconfigurable Architecture for an Intelligent Management of Consumption, Computation, Confidentiality, fault tolerance and security) is a bus-based virtual architecture capable of adapting the use of resources according to external and internal conditions. The architecture is able to calculate new working points in real time depending on the required computing performance, available power budget and levels of security and confidentiality.

The working point is defined by three coordinates representing the combination of the three different types of strategies to be implemented: Consumption, Confidentiality and fault tolerance, and Computing. Therefore, the solution scope resembles a cube where the outer vertices, i.e. those that are not on the axes, will be, in general not included. This is because, in most cases, the highest levels of any strategy are not compatible with the highest levels of the others. Hence, the working point will be placed in a solution scope that resembles the shape shown in **Fig. 2**. Notice that, since ARTICo^3 is a general purpose virtual architecture, the Pareto optimal solutions (represented as a blue surface in the figure) might change their shape and distribution depending on the target hardware platform. However, these Pareto optimal points will

Fig. 2 ARTICo^3 solution scope

always be placed inside the cube. In addition, any point that is non-Pareto optimal, i.e. those that fall within the volume generated by the intersection of the Pareto optimal surface and the coordinate surfaces that contain the axes, might be a valid solution, which means that the system could operate on that working point even though it is not optimal if the available resources do not permit a better solution.

The Resource Manager unit is the one in charge of changing this working point in real time to organize the available resources taking into account both time and space availability. It is important to highlight that the design of the Resource Manager is highly dependent on the platform where the architecture is used, as well as on the number of strategies included in each one of the three axes. Strategies to boost computation performance include parallel execution of

Fig. 1 ARTICo^3 virtual architecture block diagram

hardware accelerators, optimized burst transactions to move data between memory units or task execution overlapping. Strategies to enhance dependability (confidentiality, fault tolerance and security), on the other hand, include Double or Triple Modular Redundancy (DMR and TMR), the use of encryption and hash modules or dual-rail mitigation techniques against Side-Channel Attacks, i.e. logic replication with exactly the opposite behavior of the original module in order to mask data-dependent leakages.

The ARTICo^3 architecture is divided into two different regions inside the FPGA: static and dynamic. The static region hosts all hardware units that are not changed in real time. On the contrary, the dynamic region is divided into different slots where hardware accelerators can be dynamically loaded from an external library or using partial remote hardware configuration.

The distribution of the dynamic reconfigurable resources in the dynamic region as well as the way data are delivered to the accelerators is defined by every new working point in order to find the best available solution in a multiobjective way. As it will be explained in the following sections, in order to establish an analogy with the CUDA execution model, during the rest of the document, a kernel is referred to as a combination of one or more hardware accelerators, while each copy of them, which processes data in parallel will be called a block. At the same time, inside each block, one or more threads can be found. Threads within a block may share data, while blocks among themselves cannot. Whenever a block is replicated to provide the system with hardware redundancy, blocks will be called replicas.

The main components of the ARTICo^3 architecture (Fig. 1) are the following:

- Data Shuffler: This is a data dispatcher module. It can dispatch data in different ways depending on the operation mode. In particular, for replicas, coalesced data access is provided by this shuffler.
- Resource Manager: this is the module in charge of organizing hardware resources to find the best solution possible. In most cases, the best solution does not only pursue one objective.
- Kernel Wrapper for IP designers: this is the wrapper where the processing kernels must be included to be used within the architecture.

Depending on the requirements of the tasks, the Data Shuffler module can work in six different operation modes:

- Mode 0 (replica single): a single piece of data is delivered in parallel to two or three identical copies of the same thread block, i.e. two or three replicas, and the result is voted.
- Mode 1 (replica single + Side-Channel Attack protection): same operation as the previous mode but the voting process takes into account the negative nature of the dual-rail-enabled replica.
- Mode 2 (replica burst): same operation as Mode 0 but data are delivered using burst transactions.
- Mode 3 (replica burst + Side-Channel Attack protection): same operation as Mode 1 but data are delivered using burst transactions.
- Mode 4 (block burst): data are transferred using burst transactions to different thread blocks. Parallel processing capabilities are achieved since different data are processed in each block.
- Mode 5 (reduction mode: block burst + arithmetic operation): same operation as Mode 4 but whenever data are collected, an arithmetic operation is performed on the whole burst transaction.

When higher computing performance is required, the architecture is able to load different blocks of a kernel with different threads to work in parallel. Data are then delivered using burst transactions, which are then handled by the shuffler module so blocks can start working in cascade, overlapping their processing capabilities. Results are sent back to memory in different ways depending on the operation mode in which the shuffler unit is working (modes 4 and 5). In mode 4, the shuffler module will just organize data to be sent with a burst transaction to memory. In mode 5, on the other hand, the shuffler unit can perform a given operation (addition, maximum/minimum detection, etc.) in such a way that results are partially reduced.

Configuration aspects such as the number of blocks of the same kernel to be loaded, kernels execution order, or the number of threads per block, need to be defined by the resource manager, which will then implement different policies. Some parameters, such as kernel termination times (assuming the termination time for a thread is known), can be also taken into account, allowing the resource manager to assign task or kernel priorities.

On the contrary, there will be occasions where not only high performance computing is required but also fault tolerance or protection against Side-Channel Attacks. To comply with fault tolerance requirements, different replicas of the same accelerator can be loaded to have hardware redundancy. In modes 0 and 2, data are delivered strictly in parallel to the different replicas and results are voted in the shuffler module, where in case of finding any problem, an error code is reported. Side-Channel Attack protection can be also performed with the ARTICo^3 architecture by loading an accelerator and its negative copy. In modes 1 and 3, data are also delivered in parallel so the negative copy can mask the noise effect of the original one. Results are also compared in the shuffler module and sent back to memory.

IV. Execution Model

CUDA-based GPGPU parallel computing targets potentially parallelizable computations, which are often referred to as kernels. Each kernel is subdivided in a certain number of fixed-size blocks of concurrent-execution elements, the so-called threads. In the CUDA execution model [14], application-level parallelism is mapped transparently into GPGPU hardware resources, providing not only high application performance but also inherent thread-level scalability.

A. *The CUDA Execution Model*

Any CUDA device presents a common architectural hardware element called streaming multiprocessor (SM). The replication of this block is what makes the aforementioned mapping process work, since each thread block is executed on a single SM. SMs cannot share data and therefore, thread blocks are independent, thus complying with parallel algorithm division requirements. SMs are built using an SIMD (Single Instruction Multiple Data) approach. The total amount of SIMD cores per SM is determined by the internal architecture of the GPGPU device. For instance, a Fermi SM features 32 CUDA processors [15], whereas a Kepler SMX features 192 CUDA cores [16].

Thread scheduling in CUDA follows a hierarchical approach. Thread blocks are managed by the global scheduler, which is in charge of allocating one or more blocks to each SM. Scalability is favored, since the global scheduler only needs to know whether an SM is busy or not. Therefore, each SM is responsible for scheduling its internal resources and allocating threads to those resources. The basic unit scheduled within an SM is the so-called warp, which is a block of 32 SIMD threads that run in parallel. More than one warp scheduler is present in each SM, allowing concurrent warp execution.

CUDA GPGPUs cannot be considered, as a whole, as SIMD machines, for it is highly likely that different SMs run different instructions at the same time. This is the reason why CUDA-enabled GPGPUs are classified as SIMT (Single Instruction Multiple Thread). Therefore, thread termination times may be unequal, allowing data-dependent branches within the thread code.

In traditional GPGPU applications, kernels are invoked from the host CPU. Parallel streams are then executed in the device, returning to the host code when finished.

B. *The ARTICo^3 Execution Model*

In an attempt to establish an analogy between ARTICo^3 and CUDA devices, potential parallelizable computations will be called kernels, which can use a number of blocks for execution of all required threads in an application-independent number of blocks, but in a block-dependent number of threads per block. The architecture concept is to migrate the CUDA execution model to ARTICo^3, trying to keep as many advantages as possible, e.g. transparent parallelism mapping and resource allocation or even thread-level scalability, while changing completely the underlying hardware.

In ARTICo^3, CUDA streaming multiprocessors and its SIMT approach are substituted by hardware accelerators in dynamically reconfigurable slots. SMs use their instruction dispatch units to perform different operations using the same hardware elements. However, hardware accelerators are often application-specific modules. Therefore, in order to maintain the flexibility that SMs provide, Dynamic and Partial Reconfiguration is the most feasible alternative to implement dynamic resource management strategies.

Fig. 3 ARTICo^3 execution model

The CUDA execution model, migrated to a reconfigurable architecture is shown in **Fig. 3**. For the sake of generality, it is assumed that the architecture has multitasking capabilities, i.e. more than one process can be executed concurrently from one or more processors. Each process, or task, has its own sequential execution flow. Whenever a task requires parallel processing capabilities to enhance performance, the application makes a request to the resource manager, which can be included in an application-transparent manner by including this call in the kernel invocation function or, with some application-dependence, in a speculative way which permits to advance accelerator programming before the kernel invocation arrives. The resource manager handles the request and allocates hardware resources to be used as parallel blocks of a same kernel. **Fig. 4** shows a possible resource allocation schedule that corresponds to the kernel invocations in **Fig. 3**. At the very beginning, the reconfigurable slots are empty, since no previous kernel invocations have happened. In step 1, the application #1 (blue) requests hardware resources to execute kernel #1 (purple), and the resource manager reconfigures all available slots in order to speed-up the task. Then, the application #2 (orange) requests a higher-priority kernel to be executed, which is kernel #2 (green). The resource manager dynamically reconfigures the slots so that the new kernel has more blocks than the previous one. The execution of kernel #2

Fig. 4 Dynamic Partial Reconfiguration in ARTICo^3

blocks is done as the DPR process is gradually getting more slots allocated for this kernel. Since the resource manager is context-aware, the number of slots allocated for kernel #1 is increased when kernel #2 is almost finished, thus reaching a balanced situation. Eventually, when kernel #2 finishes, the available slots are replaced by new blocks of kernel #1, in order to finish its execution as soon as possible.

V. DYNAMIC RESOURCE MANAGER

The dynamic resource manager module reconfigures the hardware slots taking into account both internal and external conditions, so that the changeable working point requirements are met. The resource manager is also module-aware, which means that slot distributions are known at any time (this information is provided by the shuffler unit).

In order to dynamically adapt the hardware resources to meet the given requirements, the resource manager in ARTICo^3 monitors several system metrics at runtime so as to guide the management process. Examples of internal system metrics include the number of available slots, the number of thread blocks per kernel currently available, execution times per thread block, reconfiguration times, or battery level (in those platforms powered by batteries). Nevertheless, external system metrics have strong dependencies on the target platform. For instance, in a Cyber Physical System (CPS) intelligent node (which is the platform ARTICo^3 was originally conceived to target), the resource manager takes into account the information that is received through the communication interfaces, as well as the information collected by several sensors.

Therefore, the dynamic resource manager in ARTICo^3 differs from other task scheduling strategies, e.g. a regular OS task scheduler, in two main aspects: on the one hand, the intrinsic nature of the tasks (hardware-based vs. software-based); on the other hand, the deeper self-awareness of the resource manager.

In traditional dynamic resource management schemes, the working point is normally located on the Computing vertex (remember that the solution scope triangle vertices are Consumption, Confidentiality and fault tolerance, and Computing). If the kernel library as well as their characteristics, e.g. execution time per thread block, are known at design time, and the system does not have multitasking capabilities, an offline optimization algorithm can be used to obtain the optimal schedule and resource allocation to fulfill high performance requirements. On the contrary, if multitasking is supported by the system, the optimization strategy cannot be performed offline. The optimal solution has to be found at run time using an optimization algorithm.

Assuming that the resource manager has no information about any thread block, i.e. hardware accelerator in a reconfigurable slot, whenever a new task is to be carried out, task dependencies have to be determined. The system will also have to reconfigure at least one thread block of the new kernel so that the execution time per element can be obtained. Once the resource manager is aware of the execution times of each necessary block, the optimization process starts and, also taking into account the time spent during the partial reconfiguration processes, the optimal kernel scheduling can be generated.

Fig. 5 Thread block reconfiguration and execution overlapping to reduce reconfiguration overhead

However, the aforementioned strategies are not enough in the ARTICo^3 architecture, since high performance capabilities are not the only requirement. Therefore, it is necessary to change from a static resource management, in which the number of thread blocks per kernel in the optimal solution is fixed, to a dynamic resource management. The ever changing working conditions, which are task-related, i.e. different tasks might have different positions within the solution scope triangle, make it impossible to implement static optimization algorithms, since the maximum number of available reconfiguration slots might vary over time. Therefore, a multiobjective optimization algorithm is absolutely necessary in order to best establish the tradeoff between the three conditions. The use of Dynamic and Partial Reconfiguration is essential in order to develop good resource allocation and scheduling strategies.

The reconfiguration overhead in ARTICo^3 is expected to be significantly high, since changes are very likely to appear from one evaluation of the working point to the next one, making it absolutely necessary to change the hardware accelerators several times. However, the ARTICo^3 architecture can mask this reconfiguration overhead by overlapping reconfiguration and execution, as it is shown in **Fig. 5**. Parallelizing the reconfiguration process as well as the kernel execution is particularly useful in those hardware platforms where the reconfiguration engine is the bottleneck, mainly due to its low reconfiguration speeds.

The final goal is to build a resource manager capable of changing the traditional CUDA kernel invocation, which has a static allocation of both number of blocks and number of threads per block, into a new invocation process where the aforementioned parameters are dynamically allocated as well. The number of thread blocks per kernel is modified by means of Dynamic and Partial Reconfiguration strategies, whereas the number of threads per thread block is changed using the prebuilt module library, generated with the HLS toolset.

VI. Conclusions

The increasing application complexity enables High Performance Computing as a crucial set of techniques to provide very efficient solutions. Furthermore, systems are no longer working under static conditions and hence, dynamic adaptation is required. Therefore, Dynamic and Partial Reconfiguration can be used to dynamically manage the resources in order to find the optimal solution.

Dynamic Partial Reconfiguration is expected to play an important role in merging FPGA-based hardware acceleration schemes with traditional GPGPU-based parallel computing paradigms. Following a CUDA-like execution model within a custom-made virtual architecture, an efficient use of resources can be achieved while searching throughout the whole solution scope defined by the triangle Consumption, Confidentiality and fault tolerance, and Computing.

Acknowledgment

This work was supported by the Spanish Ministry of Economy and Competitiveness under the project DREAMS (Dynamically Reconfigurable Embedded Platforms for Networked Context-Aware Multimedia Systems) with number TEC2011-28666-C04-02.

References

[1] Otero, A.; Morales-Cas, A.; Portilla, J.; de la Torre, E.; Riesgo, T., "A Modular Peripheral to Support Self-Reconfiguration in SoCs," *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on* , vol., no., pp.88,95, 1-3 Sept. 2010

[2] Dondo, J.; Barba, J.; Rincón, F.; Moya, F.; López, J.C.; "Dynamic objects: Supporting fast and easy run-time reconfiguration in FPGAs," *Journal of Systems Architecture-Embedded Systems Design*, vol.59, no.1, pp.1-15, 2013

[3] "7-Series Configuration User Guide UG470 (v1.7)," Xilinx, 2013

[4] "Zynq All Programmable SoC Technical Reference Manual UG585 (v1.7)," Xilinx, 2014

[5] Becker, J.; Hubner, M.; Hettich, G.; Constapel, R.; Eisenmann, J.; Luka, J., "Dynamic and Partial FPGA Exploitation," *Proceedings of the IEEE* , vol.95, no.2, pp.438,452, Feb. 2007

[6] Philipp, F.; Glesner, M., "Mechanisms and Architecture for the Dynamic Reconfiguration of an Advanced Wireless Sensor Node," *Field Programmable Logic and Applications (FPL), 2011 International Conference on* , vol., no., pp.396,398, 5-7 Sept. 2011

[7] Koch, D.; Beckhoff, C.; Teich, J., "ReCoBus-Builder — A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS," *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on* , vol., no., pp.119,124, 8-10 Sept. 2008

[8] Wei He; Otero, A.; de la Torre, E.; Riesgo, T., "Automatic generation of identical routing pairs for FPGA implemented DPL logic," *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on* , vol., no., pp.1,6, 5-7 Dec. 2012

[9] Coussy, P.; Heller, D.; Chavet, C., "High-Level Synthesis: On the path to ESL design," *ASIC (ASICON), 2011 IEEE 9th International Conference on* , vol., no., pp.1098,1101, 25-28 Oct. 2011

[10] Mavroidis, I.; Mavroidis, I.; Papaefstathiou, I.; Lavagno, L.; Lazarescu, M.; de la Torre, E.; Schafer, F., "FASTCUDA: Open Source FPGA Accelerator & Hardware-Software Codesign Toolset for CUDA Kernels," *Digital System Design (DSD), 2012 15th Euromicro Conference on* , vol., no., pp.343,348, 5-8 Sept. 2012

[11] Kuehnle, M.; Brito, A.; Roth, C.; Dagas, K.; Becker, J., "The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip," *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on* , vol., no., pp.13,18, 4-6 July 2011

[12] Cervero, T.; Dondo, J.; Gomez, A.; Pena, X.; Lopez, S.; Rincon, F.; Sarmiento, R.; Lopez, J.C., "A Resource Manager for Dynamically Reconfigurable FPGA-Based Embedded Systems," *Digital System Design (DSD), 2013 Euromicro Conference on* , vol., no., pp.633,640, 4-6 Sept. 2013

[13] Jara-Berrocal, A.; Gordon-Ross, A., "Hardware module reuse and runtime assembly for dynamic management of reconfigurable resources," *Field-Programmable Technology (FPT), 2011 International Conference on* , vol., no., pp.1,6, 12-14 Dec. 2011

[14] R. Farber, "The CUDA Execution Model," in *CUDA Application Design and Development,* 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ch. *4*, pp. *85–108*

[15] "Fermi Architecture White Paper (v1.1)," NVIDIA, 2009

[16] "Kepler GK110 Architecture White Paper (v1.0)," NVIDIA, 2012