FASTCUDA: Open Source FPGA Accelerator & Hardware-Software Codesign Toolset for CUDA Kernels

Iakovos Mavroidis[1], Ioannis Mavroidis[1], Ioannis Papaefstathiou[1], Luciano Lavagno[2], Mihai Lazarescu[2], Eduardo de la Torre[3], Florian Schäfer[4]
[1]Microprocessor and Hardware Lab, Technical University of Crete, Chania, Greece
[2]Department of Electronics, Politecnico di Torino, Torino, Italy
[3]Centre of Industrial Electronics, Universidad Politécnica de Madrid, Madrid, Spain
[4]FSResult GmbH i.G., Germany
http://www.fastcuda.eu/

Abstract

Using FPGAs as hardware accelerators that communicate with a central CPU is becoming a common practice in the embedded design world but there is no standard methodology and toolset to facilitate this path yet. On the other hand, languages such as CUDA and OpenCL provide standard development environments for Graphical Processing Unit (GPU) programming. FASTCUDA is a platform that provides the necessary software toolset, hardware architecture, and design methodology to efficiently adapt the CUDA approach into a new FPGA design flow. With FASTCUDA, the CUDA kernels of a CUDA-based application are partitioned into two groups with minimal user intervention: those that are compiled and executed in parallel software, and those that are synthesized and implemented in hardware. A modern low power FPGA can provide the processing power (via numerous embedded micro-CPUs) and the logic capacity for both the software and hardware implementations of the CUDA kernels. This paper describes the system requirements and the architectural decisions behind the FASTCUDA approach.

1. Introduction

The ever increasing design complexity, where embedded systems consist of several complex components, some of which are implemented in software and others in hardware, makes the task of a designer more and more difficult. Tools that can combine the flexibility and low cost of software solutions with the performance and power characteristics of hardware approaches are becoming imperative in the embedded design world.

In order to solve today's challenges of high-complex embedded system designs, a number of approaches have been proposed. Hardware-software codesign is the first big step and an essential enabling technology towards this end. Electronic System Level (ESL) design is the next big step which addresses the complexity problem by elevating design to a higher level of abstraction, resulting in a more predictable and productive design process. Finally, parallel hardware platforms such as Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) are becoming very popular within PC-based heterogeneous systems for speeding up numerous compute-intensive applications.

FASTCUDA combines all above approaches by enabling hardware-software codesign and ESL design methodologies onto a low power parallel FPGA-based platform.

CUDA[1] is a data parallel programming model that supports several key abstractions (thread blocks, hierarchical memory and barrier synchronization) for allowing efficient applications development. In CUDA, the routines of an application are split into two groups: those that can benefit from a multi-threaded parallel execution and those that can not. The first group of routines, called the "CUDA kernels", are written in standard C/C++ using special annotations and constructs to specify the parallelism and the memory hierarchy. The second group of routines, called the "CUDA host program" are written in standard C/C++.

Execution starts with the CUDA host program running single-threaded on the host CPU. Whenever a CUDA kernel is invoked, the host CPU dispatches the execution of the kernel to an accelerator (separate device) that supports parallel execution of multiple threads. Traditionally these are Nvidia's GPUs or other multi-core platforms. However, we believe that even higher acceleration can be obtained if a CUDA kernel is synthesized into hardware and mapped onto an FPGA for execution. Therefore, FASTCUDA employs a hybrid approach: it uses an FPGA-based accelerator for executing the time critical CUDA kernels and a multi-core processor for executing the CUDA kernels that could not fit in the FPGA fabric.

FASTCUDA is a design methodology and accompanying toolset that allows CUDA programs to be executed efficiently on a shared memory, multi-core CPU communicating with an FPGA-based accelerator. A modern FPGA provides all required resources; multiple embedded micro-CPUs for the CUDA host program and the CUDA kernels that will be executed on the multi-core processor, referred to as "SW kernels" in the rest of the paper, and large logic capacity for the CUDA kernels that will be implemented on the FPGA accelerator, referred to as "HW kernels" in the rest of the text. Toward this end FASTCUDA will not develop everything from scratch but it will join numerous on-going efforts in industry and academia to create a unified efficient open-source framework.

Note that, while in this project we focus on the CUDA language, the overall approach can fit very well (and the developed toolset can be easily adapted to) the competing, and essentially equivalent, OpenCL standard.

2. Related Work

The use of FPGAs as hardware accelerators began in late 1980's and since then many FPGA-based embedded systems and tools have been developed. While a 10x-100x performance acceleration can usually be provided now by Reconfigurable Computing (RC) the cost in terms of difficulty in the development process increases significantly over time due to the fact that the designs are becoming larger and more complex.

A significant effort by the research community towards an efficient solution to the design tool problem tackles the hardware/software codesign process which provides a unified path from the application description, done in an abstract system model, down to a mix of hardware and software. Many of the tools provide an automated path from a parallel C-like code down to a hardware platform, but in most cases there are many issues that can be resolved only with the intervention of the designer.

FASTCUDA is distinct from other existing platforms since it is the first to provide a unified environment for programming *both* a multi-core processor and an FPGA-based accelerator. In other words, the CUDA kernel code may be flexibly mapped to a multi-processor or to dedicated hardware resources. In this Section we investigate related work to a) multi-core processors, b) high-level synthesis and c) HW-SW communication.

Multi-core processor

FASTCUDA will employ a multi-core processor in order to run the host program and the SW kernels as described in Section 6. That processor may be, for example, an embedded soft processor, a hard high performance processor (for example Xilinx FPGAs incorporated ARM processors [15]), or an external processor. In order to facilitate the software-hardware communication, FASTCUDA architecture employs an embedded multi-core processor. AMD and Intel have followed similar hybrid approaches[3] embedding a CPU and a GPU on the same chip.

Research on multi-core platforms on FPGAs has been performed in the past [4] [9][10]. FASTCUDA will adapt the framework described in [14] which provides a multi-core processor architecture tailored to the requirements of the CUDA kernels. This innovative multi-core processor features data parallel processing, a thread synchronization mechanism, and shared memory access.

High-level synthesis

High level design simulation languages such as SystemC, SystemVerilog, Handel C and Streams-C are usually used by High Level Synthesis tools in order to describe the functionality of the system.

Several commercial tools provide answers to various aspects of the high-level synthesis problem, ranging all the way from sophisticated techniques for untimed to RTL synthesis of C, C++ and SystemC models (Forte Cynthesizer, Cadence CtoSilicon, and Mentor Catapult), to synthesis of a declarative rule-based specification (BlueSpec), to synthesis from Matlab and SystemC to FPGA implementation (Celoxica and AccelChip) up to synthesis of processors from C code (Tensilica Xpress and LisaTek).

However, none of the above tools considers software/hardware codesign in a holistic manner. Some tools, such as Synopsys Platform Architect, or CoFluent Studio, provide some level of hardware/software interfacing and co-simulation. However, they do not support effective design space exploration, due to the need to implement manually the hardware part of the design, which is a lengthy process, usually limiting such exploration to a few design space points.

Recently, there is an increased interest for adapting OpenCL-like and CUDA-like languages to FPGA-based environments since using such languages for FPGAs has a significant time-to-market advantage compared to traditional FPGA development processes. Altera was the first FPGA vendor that announced a development program in order to enable, in the future, the use of OpenCL to program its FPGAs [2].

Moreover, in [5] the authors describe a CUDA to FPGA flow using AutoPilot as a high-level synthesis tool. The authors extend their architecture in [6] proposing a novel high-level synthesis framework which considers different granularities of parallelism for mapping CUDA kernels onto an FPGA-based accelerator. The framework employs a design space search heuristic in tandem with the estimation models as well as design layout information to derive a performance near-optimal configuration. However this work focuses only on the high-level synthesis framework without providing details for the system memory and the communication between the FPGA-based accelerator and the host processor, while the first work relies on a specific commercial tool for the synthesis part.

In [8] the authors describe an OpenCL to FPGA flow where the proposed architecture decouples data accesses and computations by using blocks with explicit FIFO channels that produce and consume data elements. Reconfigurable links are formed from the outputs of producing functional units to the inputs of the next consuming functional units. FASTCUDA also decouples data accesses from computations in a simpler and more efficient way.

HW-SW communication

The authors in [11] describe how to communicate and instantiate a routine implemented in hardware using user directives. In a similar way, CUDA supports memory allocation and data transfer routines for instantiating a CUDA kernel. Data transfers between the host processor executing the CUDA host program and the accelerator executing the CUDA kernels can be accelerated by using both pinned and paged host memory buffers [12][13]. FASTCUDA provides an even faster communication mechanism between the multi-core processor and the FPGA accelerator through a shared memory infrastructure. This can significantly boost the performance since no data transfers are required in order to execute a HW kernel (Section 7).

3. The FASTCUDA Approach

Today's complex systems employ both software and hardware implementations of components. General purpose CPUs, or more specialized processors such as GPUs, running the software components, will routinely interact with special purpose ASICs or FPGAs that implement time-critical functions in hardware. In these systems the separation of duties between software and hardware is usually very clear.

FASTCUDA aims to bring software and hardware closer together, interacting and cooperating for the execution of a common source code. As a proof of concept FASTCUDA will focus on source codes written in CUDA.

```
//kernel
__global__ void vectorAdd(float *A, float *B, float *C) {
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
# define N 100
#define M N*sizeof(int)
//host program
main() {
int A[N], B[N], C[N];
...
//copy input vectors from host memory to device memory
cudaMemcpy( d_A, A, M, cudaMemcpyHostToDevice);
cudaMemcpy( d_B, B, M, cudaMemcpyHostToDevice);
// kernel invocation
vectorAdd<<<1,N>>>(d_A, d_B, d_C);
//copy output vectors from device memory to host memory
cudaMemcpy(C, d_C, M, cudaMemcpyDeviceToHost );
...
}
```

Figure 1. Example CUDA code

CUDA is a single instruction multiple threads (SIMT) architecture and programming model initially developed by Nvidia for its GPUs. Figure 1 shows an example CUDA code that adds two arrays, A and B, into a resulting array C. The addition is performed in a CUDA "kernel" that runs in parallel across multiple cores in a SIMT fashion. The CUDA kernels are invoked by the CUDA "host program" which runs serially on a single core.

Each kernel implicitly describes multiple CUDA threads that are organized in groups, called "thread-blocks". Thread-blocks are further organized into a grid structure. Threads within a thread-block are executed by a single "streaming multiprocessor" inside a GPU and are synchronized and share data through a fast and small private memory of the streaming multiprocessor, called "shared memory". On the other hand, synchronization between threads belonging to different thread-blocks is not supported. However, a slow and large "global memory", is accessible by all thread-blocks. Similar to a GPU, FASTCUDA employs two separate memory spaces (global and local) as well as a similar mapping of the block-threads onto the FPGA resources as described below.

Bringing software and hardware close together, FASTCUDA will accelerate the execution of CUDA programs by running some of the kernels in hardware. A modern state-of-the-art FPGA will provide all required resources; multiple embedded micro-CPUs for the host program and the SW kernels, and logic capacity for the HW kernels.

Figure 2 shows a block diagram of the overall system. A multi-core processor, consisting of multiple embedded cores (configurable small processors), is used to run the host program serially and the SW kernels in parallel. Threads belonging to the same CUDA thread-block are executed by the same core. The HW kernels are partitioned into thread-blocks, and synthesized and implemented inside an "Accelerator" block. Each thread-block has a local private memory while the global shared memory can be accessed by any thread following the philosophy of the CUDA model. This is more elaborated in Sections 6 and 7.

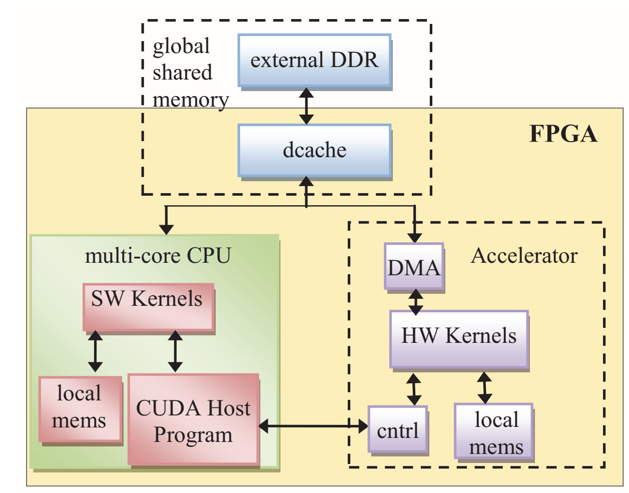

Figure 2. FASTCUDA Block Diagram

For our prototype version, we will be using the Xilinx Virtex-6 FPGA with 500MB of external DDR memory placed on a Xilinx ML605 evaluation board [17], and the multi-core processor will consist of an array of Xilinx Microblaze CPUs. However, the final product should use faster embedded processors such as the ARM Cortex-A9 MPCore.

4. FASTCUDA "Compilation"

The process that we use in FASTCUDA in order to "compile" a CUDA source code for execution onto our prototype platform is depicted in Figure 3.

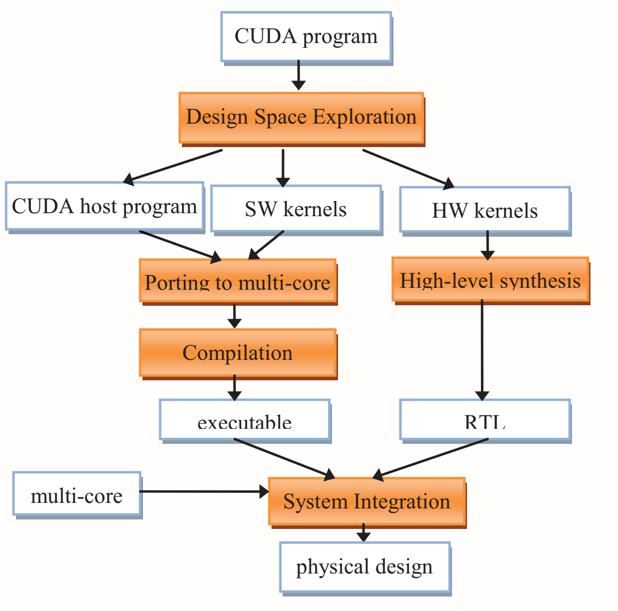

Figure 3. FASTCUDA Design Flow

The FASTCUDA toolset is responsible for automating most of this process, thus minimizing user intervention. The following sections will discuss in greater detail the various steps of this process.

5. Design Space Exploration

This first step basically needs to decide how to make the best use of the available FPGA resources for a given CUDA program; it needs to answer the following questions:

- What percentage of the FPGA real estate should be allocated to the multi-core processor for the SW kernels, and what percentage should be allocated to the Accelerator for the HW kernels?
- Which kernels should be run in software and which in hardware?
- What area-speed tradeoff is best for each of the HW kernels?
- What is the optimal configuration (number of cores, cache sizes, memory banks, etc.) for the multi-core processor?

In order to make these decisions each kernel is first carefully examined and run through several simulation and synthesis runs. The simulation tool provides runtime estimates for the execution of each kernel in software, for several configurations of the multi-core processor (with varying cache sizes, memory banks, etc.). The synthesis tool provides latency estimates for the execution of each kernel in hardware, with varying hardware footprints (i.e. trading area for speed).

The design space exploration tool uses these area and performance estimates, along with its full knowledge of the underlying platform's resources and available configurations, to heuristically search for the best answers to the questions listed above. User experience can be used to guide the tool, e.g. by restricting the search to a smaller set with the most "interesting" multi-core configurations.

6. Multi-Core Processor

The CUDA host program as well as the SW kernels (the subset of the kernels determined by the design space exploration tool) will run in software on the multi-core processor of Figure 2. In this section we first review the architecture of this processor, and then we discuss the required process in order to port the CUDA source code to this architecture.

6.1. Architecture

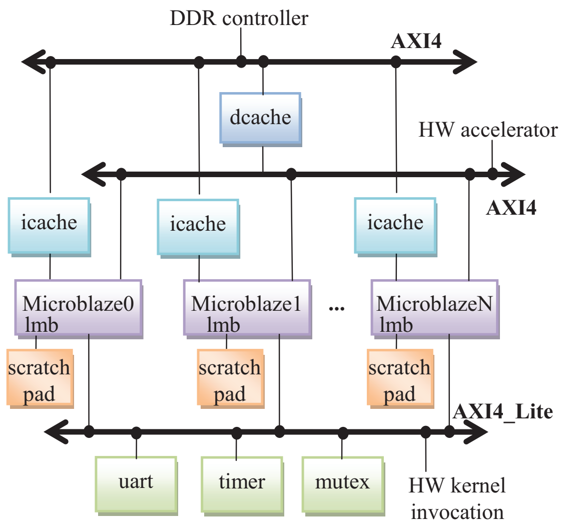

Figure 4. Multi-Core Processor Architecture

Figure 4 shows our prototype's multi-core processor architecture. It uses Xilinx Microblaze soft cores (configurable small processors) with separate instruction caches and a shared data cache all communicating through two AXI4-based buses [16]. FASTCUDA follows a similar mapping of the threads with a GPU. Each core executes a thread-block which can use the core's scratchpad memory as a private local memory. All the threads from any thread-block can access the global shared memory which can also be accessed by the HW accelerator (notice the connection on the AXI4 bus in the Figure).

The AXI4_Lite bus is used for the communication between the multi-core processor and the Accelerator block that is running the HW kernels. A simple handshake protocol is employed to pass the arguments and trigger a specific HW kernel to start running, which will then respond back when it has finished running.

Lastly, the *timer* and *mutex* blocks on the AXI4_Lite bus are a requirement for the symmetric multiprocessing (SMP) support of the runtime running on the processor as we will explain in the following section.

Notice that the number of cores, as well as the data cache size and organization (single or multiple banks), the configuration of the Microblazes, and other configuration parameters, are application-dependent and are determined by the design space exploration tool of the previous section according to the requirements of the CUDA application.

6.2. Implementing CUDA kernels on the multi-processor

The OS-level software running on our multi-core processor is a modified version of the Xilinx kernel "Xilkernel". Xilkernel supports POSIX threads, mutexes and semaphores, but was meant to run on a single core, thus having no support for a SMP environment like ours. We consequently had to add SMP support to Xilkernel (see [14] for a description of the methodology on how to do this).

CUDA kernels are supposed to run on SIMT devices (i.e. GPUs), which are drastically different from our multi-core processor. Thus, the next step is to port the CUDA kernels to run on top of the multi-core multi-threaded environment provided by our modified Xilkernel, using MCUDA[7].

The compilation process that results in a single executable code is shown in the following Figure.

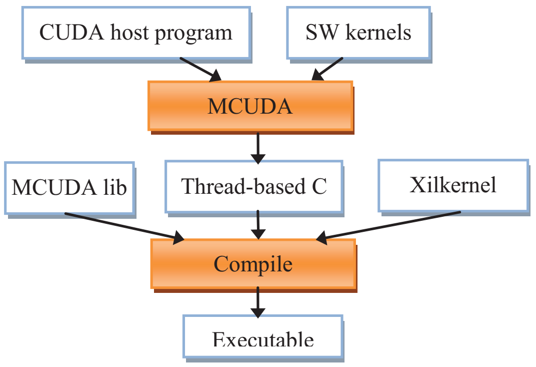

Figure 5. Software Porting Process

MCUDA transforms the CUDA code into thread-based C code that uses the MCUDA library in order to create a pool of threads and coordinate the operations of the threads as well as to provide the basic CUDA runtime functionality for kernel invocation and data movements.

Xilkernel provides the mutex support required by the MCUDA library and the thread-based support required by the multi-threaded SW kernels.

7. Accelerator

In CUDA the host program is usually run on a separate chip from the CUDA kernels; the first is run on a general-purpose CPU and the latter on a GPU. Thus the CUDA programming model assumes that the host and device maintain their own separate memory spaces, referred to as *host memory* and *device memory* respectively. The execution of a kernel, involves a) memory transfers of the input vectors from the host memory to the device memory, b) the kernel execution which uses the input vectors in order to generate the output vectors and c) memory transfers of the output vectors from the device memory to the host memory as shown in Figure 1. The addresses of the input and output vectors are passed as arguments to the CUDA kernel.

In contrast, FASTCUDA runs everything on the same chip, thus favoring a different memory model where all the threads of a kernel and the host program can share a single global memory (see Figure 2). In this model, the HW kernels inside the Accelerator have direct access to the memory in order to read their input vectors and write their output vectors.

7.1. Implementing CUDA kernels in hardware

In FASTCUDA, the code of the HW kernels is pre-processed before it is synthesized. To aid in this pre-processing the programmer is required to use "*#pragma*" directives in order to specify which ones among the kernel arguments are inputs and outputs, as well as their sizes.

The result of translation from CUDA to SystemC is shown in Figure 6. An advanced memory interface, using a SystemC interface called `fcMem`, will be provided to coalesce global memory accesses, like in a modern GPU, in order to better exploit the AXI interface bandwidth.

Note how argument pointer accesses are transformed into reads and writes to and from a base address (A, B and C) and an offset (i) using the global memory port.

```
//SystemC module
SC_MODULE(addMod) {
  sc_in<int> A, B, C, threadIdx_x;
  sc_port<fcMem> sMem, gMem;
  sc_in<bool> clk, start;
  sc_out<bool> done;
  SC_CTOR(addMod) {
    SC_CTHREAD(add, clk);
    reset_signal_is(start);
  }
// kernel
void add() {
  int i = threadIdx_x;
  gMem.writeFloat (C+i,
     gMem.readFloat(A+i) + gMem.readFloat(B+i));
}
```

Figure 6.Example CUDA to SystemC transformation

This simple example does not show the use of shared memory on the GPU, which is generally used to perform computations on fast local data, and which will also be modeled in SystemC as a port implementing the `fcMem` interface. Transfers between global memory and shared memory are managed by CUDA programmers by hand, and thus can be considered akin to sophisticated application-specific DMA engines. Our translation strategy naturally exploits the fast local BRAM (FPGA Block RAM), where the shared CUDA memory is mapped, by converting the

CUDA transfers between global and shared memory into SystemC accesses to global memory and local vectors.

CUDA assumes that the GPU supports three distinct device memories referred to as global, shared and constant memories. These will be implemented as global memory (shared with the host processor and the multi-processor, and implemented in an external DRAM), local per thread-block memory (implemented in BRAM or registers) and constants (translated directly to logic by the synthesis tool).

Any high-level synthesis tool that takes SystemC as input and can perform the synthesis required, such as the AutoPilot tool which has been recently acquired by Xilinx [19], can then be used for synthesis.

The aforementioned flow is depicted in Figure 7.

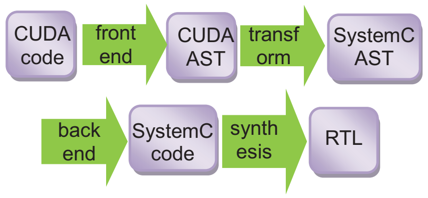

Figure 7.CUDA to FPGA flow

8. Applications

Software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. Most of the CUDA applications mentioned in [18] should run on the FASTCUDA platform. However, FASTCUDA provides performance and power advantages over GPUs only for applications that exhibit some specific characteristics.

A GPU supports a sophisticated memory interface which coalesces accesses to speed up bulk memory transfers. Therefore, memory intensive applications may run faster on a GPU than on the FASTCUDA platform. On the other hand FPGAs can provide enormous parallelism and much lower power and energy consumption when performing the computations of a kernel. Hence, compute intensive applications may run faster and with lower power consumption on FASTCUDA than on a GPU.

9. Conclusion

In this paper we presented a novel open-source framework, FASTCUDA, which aims to bring software and hardware closer together, interacting and cooperating for the execution of a common source code under a unified environment and with minimal user intervention. FASTCUDA will allow CUDA programs to be executed efficiently on a multi-core processor communicating with an FPGA accelerator. A shared memory infrastructure provides a fast communication mechanism between the multi-core processor and the FPGA accelerator. The FASTCUDA framework provides a higher level programming abstraction than traditional FPGA design tools, combining a novel CUDA to FPGA flow that uses a high-level synthesis tool with a CUDA to multi-core compilation flow that employs a source to source translation tool.

Acknowledgments

The research leading to these results has received funding from the European Union 7th Framework Programme [FP7/2007-2013], under the FASTCUDA project (grant agreement n^o 286770).

References

[1] CUDA parallel programming model http://www.nvidia.com/object/cuda_home_new.html

[2] Altera OpenCL Program for FPGAs http://www.altera.com/corporate/news_room/releases/2011/products/nr─opencl.html?GSA_pos=5&WT.oss_r=1&WT.oss=OpenCL

[3] AMD Fusion http://en.wikipedia.org/wiki/AMD_Fusion

[4] Phil James-Roxby, Paul Schumacher, Charlie Ross, "A Single Program Multiple Data Parallel Processing Platform for FPGAs", FCCM'04, pp.302-303

[5] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2009, "High-performance CUDA kernel execution on FPGAs", ICS '09

[6] Alexandros Papakonstantinou, Yun Liang, John A. Stratton, Karthik Gururaj, Deming Chen, Wen mei W. Hwu, and Jason Cong, "Multilevel Granularity Parallelism Synthesis on FPGAs", FCCM'11, pp. 178-185

[7] John A. Stratton, Sam S. Stone, Wen-mei W. Hwu, "MCUDA: An Efficient Implementation of CUDA. Kernels for Multi-Core CPUs", Un. of Illinois, TR'08

[8] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas and Christos Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs", FCCM'11, pp. 186-193

[9] Huerta P., Castillo J, Martinez I. J.," Multi MicroBlaze System for Parallel Computing", ICC'05, pp. 1–6.

[10] Roger Moussali, Nabil Ghanem, Mazen A. R. Saghir, "Supporting Multithreading in Configurable Soft Processor Cores", CASES 2007

[11] Cabrera D., Martorell X., Gaydadjiev GN., Ayguadé E., Jiménez-González D., "OpenMP extensions for FPGA Accelerators", SAMOS'09, pp. 17-24

[12] Vasily Volkov and James W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra", SC'08

[13] Glenn Lupton, Don Thulin, "Accelerating HPC Using GPU's", High Performance Computing Division, 2008

[14] Pablo Huerta, Javier Castillo, Carlos Sánchez, Jose Ignacio Martínez, "Operating System for Symmetric Multiprocessors on FPGA", ReConFig'08, pp.157-162

[15] Xilinx Zynq-7000 family, http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm

[16] ARM AMBA bus protocol http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[17] Xilinx Virtex-6 ML605 evaluation kit http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

[18] CUDA applications http://www.nvidia.com/object/cuda_app_tesla.html

[19] Xilinx acquires AutoESL http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1521536