High-Performance CUDA Kernel Execution on FPGAs

Alexandros Papakonstantinou[1], Karthik Gururaj[2], John A. Stratton[1], Deming Chen[1], Jason Cong[2], Wen-Mei W. Hwu[1]
[1] Electrical & Computer Engineering Dept., University of Illinois, Urbana-Champaign, IL, USA
{apapako2, stratton, dchen, hwu} @ Illinois.edu
[2] Computer Science Dept., University of California, Los Angeles, CA, USA
{ karthikg, cong} @ cs.ucla.edu

ABSTRACT

In this work, we propose a new FPGA design flow that combines the CUDA programming model from Nvidia with the state of the art high-level synthesis tool AutoPilot from AutoESL, to efficiently map the exposed parallelism in CUDA kernels onto reconfigurable devices. The use of the CUDA programming model offers the advantage of a common programming interface for exploiting parallelism on two very different types of accelerators – FPGAs and GPUs. Moreover, by leveraging the advanced synthesis capabilities of AutoPilot we enable efficient exploitation of the FPGA configurability for application specific acceleration. Our flow is based on a compilation process that transforms the SPMD CUDA thread blocks into high-concurrency AutoPilot-C code. We provide an overview of our CUDA-to-FPGA flow and demonstrate the highly competitive performance of the generated multi-core accelerators.

Categories and Subject Descriptors

D.3.3 [**Computer Systems Organization**]: Performance of Systems– *design studies.*

General Terms

Performance, Design, Languages.

Keywords

High performance computing, high-level synthesis, coarse-grained parallelism, FPGA, GPU, CUDA programming model.

1. INTRODUCTION

The computing industry's shift from higher operating frequencies to wider parallelism has led to renewed interest in alternative compute devices, such as GPUs and FPGAs. GPUs depend on the use of a large number of processing cores to handle the intensive compute load with high-degree of data-level parallelism, such as imaging tasks, while FPGAs have gained popularity in the high performance community due to their flexibility to efficiently exploit application specific parallelism patterns. A significant hurdle in exploiting the application parallelism on these parallel processing devices is the lack of good parallelizing compilers which can identify tasks that can run in parallel on the different compute cores. In the GPU domain, this issue has been addressed by proposing new programming APIs that let the programmer express both fine-grained and coarse-grained parallelism of the application. With the recent popularity of CUDA, a wide range of applications now have their performance sensitive kernels accelerated with GPU execution. In the FPGA domain, even though a number of newly emerging high-level synthesis tools aim to raise the level of abstraction for hardware design, the problem of extracting coarse grained parallelism remains. In this work, we propose a new FPGA design flow which combines the CUDA programming model with the state of the art high-level synthesis tool, AutoPilot [4], to efficiently target CUDA kernels to FPGA. The design flow makes a large body of existing and new CUDA applications available to FPGA acceleration. Furthermore, it enables application specific acceleration on FPGA driven by a high-abstraction programming model.

Our CUDA-to-FPGA flow (Fig. 1) is based on a code transformation process, FCUDA (currently targeting the AutoPilot HLS tool), which is guided by preprocessor pragma directives that are inserted by the FPGA programmer into the CUDA code. These directives guide FCUDA translating the expressed parallelism of the CUDA code into explicitly-expressed coarse-grained parallelism in the generated Autopilot code. The FCUDA pragmas describe various FPGA implementation dimensions which include the number, type and granularity of tasks, the type of task synchronization and scheduling, and the data storage within on- and off-chip memories. AutoPilot, consequently, maps the FCUDA specified tasks onto concurrent cores and generates the corresponding RTL description. AutoPilot's high-level synthesis further uses LLVM's [3] dependence analysis techniques to extract finer grained instruction-level parallelism within each task. Finally we leverage Xilinx FPGA synthesis tools to map the multi-core oriented RTL onto the reconfigurable fabric. We demonstrate that the FPGA accelerators generated by our FPGA design flow can efficiently exploit the computational resources of top-tier FPGAs in a customized fashion and provide better performance compared to the GPU implementation for a range of applications.

2. FCUDA PHILOSOPHY

The parallelism is expressed in CUDA as fine-granularity threads that are further bunched into coarse-granularity thread-blocks. Even though the thread-level parallelism is interesting, the thread-blocks offer many more benefits for an efficient multi-core implementation on FPGA. This is due to the fact that thread-blocks in CUDA code are independent since they modify different data sets and do not need synchronization. Conversely, CUDA threads within a thread-block usually reference shared data which may result in synchronization overhead or memory access conflicts. Concurrency in CUDA is activated by invoking a single

This work is partially supported by GSRC under the FCRP program

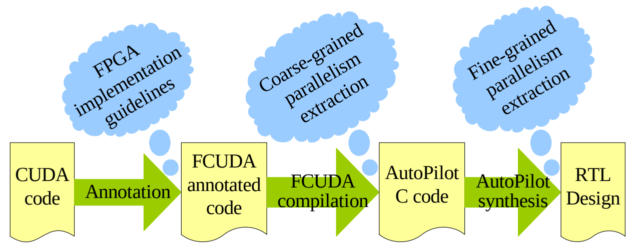
FPGA implementation guidelines
Coarse-grained parallelism extraction
Fine-grained parallelism extraction
CUDA code
Annotation
FCUDA annotated code
FCUDA compilation
AutoPilot C code
AutoPilot synthesis
RTL Design

Figure 1. CUDA-to-FPGA flow

CUDA kernel with built-in variables that identify the threads and thread-blocks. The number of threads and the number of thread-blocks are specified by additional parameters in the call to the kernel. On the other hand, parallelism in the C code for FPGA synthesis by AutoPilot is expressed by explicitly executing multiple function calls in parallel, provided that they do not share registers or on-chip memory data. Each function is mapped onto an allocated set of compute and storage resources on the configurable device, which we refer to as a *core*, which matches the thread-blocks well. Therefore, the main goal of the FCUDA source-to-source translation is to convert thread-blocks into parallel functions by packing all the threads of each thread-block into a C function for AutoPilot. Furthermore, each CUDA kernel is transformed to explicitly expressed parallelism by generating multiple calls of the C function in the output code for AutoPilot, which can then map each function call onto a separate computing core on the FPGA.

The threads within each C function can be scheduled by AutoPilot to be executed either in a parallel or serial fashion. Nevertheless, due to the aforementioned advantages of coarser-granularity parallelism, resource allocation for fine-grained threads is done only after the specified number of cores has been generated, provided that available resources still exist. As a consequence the set of threads that are executed concurrently on the FPGA is usually different from the corresponding set of threads on the GPU. Nevertheless, parallelism in both devices is mainly limited by the number of available cores.

Another important characteristic of the FCUDA philosophy is to decouple off-chip data transfers from the rest of the thread-block operations. The main goal is to avoid long latency references which may impact the efficiency of the multi-core execution. This is particularly important in the absence of context switching. Moreover, by aggregating all of the off-chip accesses into DMA burst transfers from/to on-chip BRAMs, the off-chip memory bandwidth can be utilized more efficiently.

3. FCUDA COMPILATION

The FCUDA compilation is based on the Cetus source-to-source compiler framework [2] and it consists of two main phases. The front-end phase is focused on the intra-block thread ordering semantics to ensure that the serialization of the threads as well as the split of the computation and communication tasks adhere to the CUDA program semantics. It extends the techniques proposed in MCUDA [1] to parse and leverage FCUDA pragma directives along with the regular CUDA synchronization primitives.

The back-end phase of FCUDA deals with the explicit expression of the CUDA kernels in the C code for AutoPilot according to the details annotated in the FCUDA pragma directives. This phase also handles the on-chip memory resource allocation for each core and enforces the synchronization scheme imposed by the programmer through the FCUDA pragma parameters.

4. RESULTS

In this work we targeted the Xilinx XC5VFX200T device for our performance evaluations. Virtex5 FPGAs are fabricated in 65nm CMOS technology and can be clocked at frequencies of up to 550MHz. These features render them good candidates for making meaningful comparisons with most of the currently used GPU devices. The GPU device used for the comparisons was Nvidia G80 [5] with 16 SM units and 128 cores. Figure 2 compares the FPGA and GPU performance for different integer bit-width versions of two CUDA kernels: Matrix Multiplication (MATMUL) and Coulombic Potential (CP). The FPGA performance results are based on the assumption that the device is connected to the off-chip memory with a high-bandwidth bus (e.g. FSB). All the results are normalized with respect to the 32-bit GPU performance. The GPU latencies do not include the data communication from/to the CPU.

As our results show, the generated multi-core accelerators on Virtex-5 FPGAs can outperform G80, especially in the case of smaller bitwidths, where application specific customization can adapt the datapath of the cores and use the freed resources for instantiating more cores. The number of instantiated cores on the FPGA is determined by the most heavily utilized resource (LUTs, BRAMs or DSPs). Availability of low-utilization resources can be used to extract thread-level parallelism. For example, in the case of 16-bit and 8-bit MATMUL kernels where BRAM is the core limiting resource, the remaining LUT and DSP resources are used to allow two threads in each thread-block to execute concurrently, which offers an extra speedup of 2X.

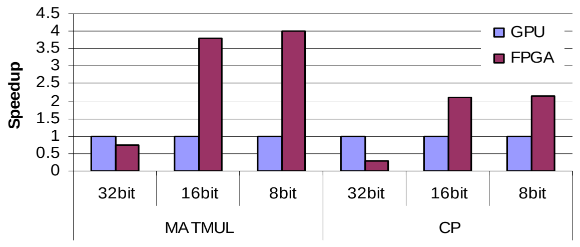

Figure 2. GPU – FPGA performance comparison

5. REFERENCES

[1] J. Stratton et. al. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *21st Int. Workshop on Languages and Compilers for Parallel Computing*, 2008.

[2] S. Lee, T. Johnson, and R. Eigenmann. Cetus - An extensible compiler infrastructure for source-to-source transformation. *16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2003)*. 2003.

[3] LLVM compiler, http://www.llvm.org

[4] AutoESL, http://www.autoesl.com/.

[5] http://www.nvidia.com/page/geforce_8800.html