FCUDA – CUDA to FPGA High Level Synthesis

Nguyen Quoc Duy Tan
School of Computer Engineering

Asst Prof Kyle Rupnow
School of Computer Engineering

Abstract – This paper serves as a record of debugging the existing CUDA-to-FPGA tool flow (FCUDA) with various benchmarks in order to improve its robustness and efficiency. The paper starts with a brief introduction about the tool, its infrastructure, impact and role to the computing world nowadays. The body of the paper mainly consists of the methodology the author used when testing FCUDA. It also notes all the bugs the author has possibly found and proposed some work-able solutions to them. Different existing FCUDA versions will also be discussed. Finally, it concludes with some points for future work and enhancement.

Keywords – CUDA; FPGA; FCUDA; benchmark

1 INTRODUCTION

Big data plays a critical role to the technological advancement and a vital point of human future. The data we now have to deal with is far more complicated as well as abundant compared to the one back in the early days of computing. For example, people might use big data in weather forecasting, astronomy, finance, biology, and so on. Therefore, many supercomputing centers have been flourished around the world to which their jobs are processing the massive amount of data within an allowable time, resource usage and power efficiency. In fact, supercomputers are built heavily on layers of heterogeneous system where various kinds of platforms, like CPU, GPU, FPGA or other custom accelerators, integrated to solve complex problem in high performance computing domain. Yet it brings a challenge to application developers, such as scientists who deal directly with the data of their domains in order to write a portable code which is run-able across different devices. Therefore, a unified computing model is necessary to minimize the learning curve of application developers and gives them the time they need to solve their actual problem. In other words, we, as tool developers, would want to remove their concerns on many low-level details of different platforms or how to amalgamate them by building developing tools that assist them to achieve such goals.

CUDA (Nvidia) is a very popular programming model that helps user to exploit the power of GPU computing. CUDA implements a SIMT (single-instruction, multiple-thread) model that enables parallelization of many threads to execute a single instruction which spans over a large dataset. CUDA is one example of a developing tool which bridges the gap between programming in CPU versus programming in GPU. Thus, an application developer can develop his application on an integrated GPU-CPU system without having to worry about whether his code will be run-able or not.

There is another hardware system that constantly gains its popularity among the high performance computing domain: FPGA. FPGA is a customable programming device which means it can be synthesized into different configurations many times, in contrast with ASIC which is only specific for an application and cannot be re-programmable. Moreover, FPGA can explore the power of computing based on generating parallel custom cores. It is also favored by its less power consumption, comparing to GPU. Nevertheless, FPGA is not used as widely as GPU in high performance computing. The complexity of hardware programming model, like Verilog, or VHDL is one of the main barrier that scare application developers, which are not hardware experts, out of using FPGA for their problem. Seeing this issue, people in academic has currently engaged aggressively in High Level Synthesis research which offers a programmer an opportunity to write high level languages, like C/C++ or SystemC to do programming on FPGA without the necessity of acquiring hardware expertise.

A research work from UIUC has combined both of the tools above to produce FCUDA: a CUDA-to-FPGA tool flow which is meant to write a CUDA code that can be ported onto FPGA. Hence, many CUDA applications can now possibly run onto FPGA. FCUDA is not a new programming model. In fact, it is just like a source-to-source compiler tool which translates CUDA code to a C code. That C code then is imported to High Level Synthesis tool to generate RTL code which FPGA can read and execute. The application developer now only needs to write a good CUDA code and then run FCUDA to translate his application to RTL on which can run FPGA. FCUDA provides an excellent opportunity to programming application on systems that integrate both FPGAs and GPUs. It also opens the possibility of porting many existing GPU applications onto FPGA.

2 MAIN CONTENT

2.1 CUDA-TO-FPGA MAPPING

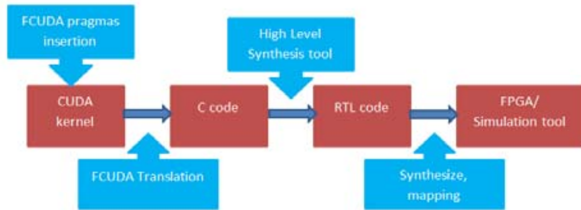

Fig 1 CUDA to FPGA flow

Figure 1 describes the CUDA – to – FPGA tool flow. It takes CUDA kernel as an input with some manual modifications from a user. Those changes can possibly be adding some specific FCUDA pragma annotations. Therefore, FCUDA will translate the CUDA kernel based on those pragmas to a C file. The C file will be ported to a High Level Synthesis tool to generate RTL file. Finally, FPGA can take the RTL file to execute the code and produce output which should match with executing the CUDA code on GPU.

To map CUDA onto FPGA, we have to understand the CUDA programming model as well as its memory space. CUDA employs SIMT model which hundreds or thousands of threads concurrently execute instructions. These threads are organized into thread blocks, and thread blocks comprise grid. Threads within thread block can share data via shared memory, whereas there is no communication between thread blocks. Therefore, there must be a synchronization enforcing between threads of a thread block. We can use a For Loop over threads to enforce this synchronization in C code, here we call it ThreadLoop. The paper [4] gives a detail explanation of mapping CUDA code to C code for multi-core system. Besides, CUDA memory space consists of several layers: global memory (also contains constant and texture memory), shared memory, registers. Global memory is visible to all threads of all blocks while shared memory is accessed by threads within a block and each thread block is allocated a distinct shared memory space. Registers is used by a thread and cannot be seen by others. FCUDA's memory space is somewhat similar since it divides largely into 2 parts: off-chip memory (DRAM) and on-chip memory (BRAM). Therefore, we can map off-chip memory of FPGA to global memory from CUDA and on-chip memory of FPGA to shared memory and registers from CUDA [1]

2.2 FPGA PARALLELISM

Exploiting parallelism on FPGA is very important because we not only need to concern about the possibility of porting a CUDA application onto FPGA, but also we have to make sure that the performance of the application on FPGA is competitive to CUDA as well. Fortunately, FPGA has abundant parallel custom engines to let a user takes advantage of parallelism on FPGA.

FCUDA provides several knobs to turn on parallelism extraction on FPGA: by unrolling ThreadLoop, array partitioning, procedure replication (cores). A user can specify which parallelism techniques he would like to use through the injected FCUDA pragma annotations. Unrolling ThreadLoop unrolls the ThreadLoop to a degree specified by user to enable instruction-level parallelism if the instructions do not depend on each other. Partitioning array will divide a chosen array into several subarrays mapping onto different BRAM locations to allow concurrent access (note that multiple accesses to the same BRAM location will cause a serialization of the accesses, hence degrade the performance). Finally, procedure replication will facilitate task-level parallelism provided that these tasks are dependent free. A decent High Level Synthesis tool will then automatically translate those replications into cores which executes concurrently.

The paper [2] uses Design Space Exploration techniques to give a framework of how to determine the best configuration of unrolling, array partitioning, and cores to achieve highest possible performance for each benchmark.

2.3 FPGA INFRASTRUCTURE

The infrastructure of FCUDA is largely built on top of Cetus, a source-to-source compiler which is a research work from Purdue University [8]. FCUDA is extended by adding more compiler passes (layers) to make the tool to be able to parse CUDA syntax as well as doing essential code transformations to generate a C file that is compatible with High Level Synthesis tool like Vivado HLS. FCUDA goes through all its passes sequentially, each pass plays a role in examining the CUDA kernel input and does specific tasks. Below are some essential passes which provides the most important functionalities of FCUDA:

+ *StreamInsertion*: convert a program with a stream pragma into a new kernel instantiating the constant memory buffer as shared memory buffer.

+ *MakeArrayInCompute*: convert variables which are used in different ThreadLoop or different COMPUTE tasks into array. In other words, we privatizes those variables for their corresponding threadIdx.

+ *SplitFCUDATask*: use the user-inserted FCUDA annotation to split the kernel into computation and communication tasks. Each FCUDA annotation task is converted into independent procedure which is called from kernel function.

+ *SerializeThreads*: add ThreadLoop around thread-dependent parts of kernel procedure or task procedures.

+ *UnrollThreadLoops*: unroll ThreadLoop based on "unroll" factor from the FCUDA pragma annotations.

+ *PartitionArrays*: partition arrays based on "mpart" factor from the FCUDA pragma annotations.

+ *WrapBlockIdxLoop*: create blockIdx loop over CUDA statements.

+ *DuplicateForFCUDA*: duplicate the task function calls based on "core" factor from the FCUDA pragma annotations to enable concurrent execution.

2.4 FCUDA VERSIONS

FCUDA version in the early days [1, 2] requires user to annotate FCUDA-specific pragmas around statements in a CUDA kernel (note that FCUDA only translates CUDA kernel, not the entire CUDA program which would mean to include the CPU C code of which plays the role initializing, allocating, and invoking CUDA kernels also. FCUDA only targets CUDA kernel as an input). The pragmas will guide FCUDA to split the CUDA kernel into several tasks, for instances: COMP (compute) task, or COMM (communication or transfer task). Generating multiple tasks will result in better execution latency since tasks can be overlapped or run concurrently if data independence among tasks is ensured (coarse-grain parallelism). Later on, the authors of FCUDA inserted some additional passes which build hierarchical region graph of the kernel and implemented several sophisticated code motion techniques which arranged the statements within the CUDA kernel [3]. The aim is to improve the throughput of the generated code as well as eliminating the necessity of manual user intervention by wrapping FCUDA pragmas around tasks. It removes the user’s burden of having to scan through a CUDA kernel and adds FCUDA pragma annotations. An application developer can just write a CUDA kernel, run the FCUDA script, and it will generate the C code automatically.

The automation of FCUDA is far from perfect and there is a current progress which involves a lot of work to improve it. Therefore, this paper will focus on debugging the old FCUDA version which still requires user to inject FCUDA pragma annotations.

2.5 FCUDA PRAGMA INSERTION - HOWTO

There is not a concrete algorithm to identify compute task or transfer tasks of a CUDA kernel. However, there are some common patterns which are widely recognizable and applicable to most of CUDA kernels.

Firstly, we need to identify global memory pointers and their usages within the CUDA kernel. Global memory pointers could be introduced through the kernel's parameter list, or constant variables. If there is an assignment statement of a global memory pointer and a shared memory pointer, we will convert it to transfer task. If there is not, we can add new shared memory pointer to the CUDA kernel and replace all the statements which use global pointer by the new shared pointer. This could potentially boost the performance because we restrain from accessing global pointer, which is off-chip memory in FPGA, which incurs long latency.

Secondly, if we cannot be certain whether a section of statements is a transfer task, we can safely assume it as compute task. Note the potential problem of privatizing variables (MakeArrayInCompute) used inside the task to decide whether to split the task into different compute tasks.

Finally, we can do some arrangements over the statements of the CUDA kernel input to separate compute sections from transfer section. The ideal task flow should be:

fetch (from off-chip memory to on-chip memory) → compute → write (from on-chip memory to off-chip memory)

2.6 FCUDA TESTING

Debugging FCUDA means translating a CUDA kernel to a C file, after that we compile and execute the C file to produce an output. If that output matches the CUDA's output, we can conclude that FCUDA functions well with the CUDA kernel input.

The author has used some CUDA applications from several benchmarks suites including Parboil bechmark (cp) [5], Rodinia benchmark (hotspot) [6] and CUDA SDK (matmul, dwt, fwt) [7].

It is deemed unnecessary and impossible to include every debugging detail within the limitation scope of this paper. Therefore, the user chooses to address some notable bugs as well as proposed solutions to them.

1. For transfer task, in some cases we may want to add an offset to the on-chip memory pointer as FCUDA does not seem to take into account of that offset during translating assignment statement to memcpy. The workaround is, we can possibly create a new on-chip memory pointer which has address offset by an exact offset value from the original on-chip memory pointer and do transfer task with that new on-chip memory pointer. However, in the long term, we would want the tool to handle it automatically for us so we do not need to add extra pointer. Hence, we need to add some code to FCUDA so that it can take into account of the offset on the on-chip memory pointer (benchmark dwt).

2. Right now, Privatizing scalar variables (MakeArrayInCompute) only relies on whether the variables span in different tasks. There are some cases in which we want to privatize a variable which uses in different ThreadLoop within a task, especially when a variable is firstly defined in a ThreadLoop and later used by another ThreadLoop, and the values of that variable are different for different threads. Because we use ForLoop to enforce the synchronization among threads, careful investigation of data flow should be done to make sure of data integrity (benchmark dwt, benchmark hotspot)

3. There seems to be a mixing between using unroll index in UnrollThreadLoops pass and PartitionArrays. Therefore, the author uses an extra variable 'isArrayPartitioning' to check if an array is going to be partitioned or not. Note that UnrollThreadLoops occurs before PartitionArrays. If it is, we proceed as like usual (associating unroll indices to partitioned arrays). If it is not, we do not proceed and keep the old unroll indices (benchmark matmul).

4. There is a minor issue that leads to compilation error. Because this benchmark has 2 kernels, we will encounter a case that both of the kernels define 2 same macros

   #define BLOCKDIM_X ....

   This leads to redefinition error. To avoid this, we simply associate the macro with the name of the kernel, so we will have a statement like:

   #define BLOCKDIM_X_[benchmark_name] ...

   Therefore, we do not get redefinition error again because each kernel has its own unique name (benchmark fwt).

5. A test failure after doing array partitioning is usually due to the execution of a group of thread elements on an array modifies the values of that array's elements which is handled by a different group of thread elements. To prevent it from happening, we can add a conditional statement to guard the section of an array which is managed by a specific group of thread elements, so that only that group of thread elements are allowed to make change to the corresponding assigned portion of the array (benchmark fwt).

6. There is another issue with memcpy used in transfer task. Apparently, if we use normal assignment, we have to run a ThreadLoop of threadIdx.x on an assignment statement, while we don't have to do that when we convert the assignment statement to memcpy, because memcpy will take care of the ThreadLoop by copying a block of consecutive threadIdx.x. If in case some elements of threadIdx.x should not be copied (for instances, there is a control flow statement that only allows assignment on a specific condition), it raises a question on how to handle the situation properly. In the worst case, we could just use a compute task to handle it instead of a transfer task, i.e. a compute task will just use normal assignment. This is an open problem to think of.

   Yet we still need to take care of another seem-not-so-trivial issue: boundary of the memcpy. We have to make sure that we do not step on the memory region of another data structure. We must add some conditional statements to rigorously check the boundary of the off-chip pointer when we do memcpy.

   For example, a very typical transfer task will consist of a memcpy statement like below:

   *memcpy(onchip_ptr, offchip_ptr + X + c * threadIdx.y, burst_size * sizeof(DATATYPE));*

   (burst_size is usually equal to blockDim.x)

   Suppose the size of offchip_ptr is M, that is, anything lies beyond offchip_ptr + M or before offchip_ptr is out-of-range and unacceptable.

   The proposed solution to this matter is:

```
int offset = X + c * threadIdx.y;
int offset1 = M - offset;
if (offset1 < 0)
continue; // out-of-range, unacceptable!
if (offset1 > blockDim.x)
offset1 = blockDim.x;
int offset2 = offset + blockDim.x;
if (offset2 <= 0)
continue; // negative index, unacceptable!
if (offset > 0)
memcpy(onchip_ptr, offchip_ptr + X + c * threadIdx.y, offset1 * sizeof(DATATYPE));
else
memcpy(onchip_ptr + (-offset), offchip_ptr + X + c * threadIdx.y + (-offset), offset2 * sizeof(DATATYPE));
```

   (benchmark hotspot)

3 CONCLUSION AND FUTURE WORK

FCUDA is a promising research. It can explore the possibility of integrating GPUs and FPGAs into one system or achieve performance portability across many platforms. FCUDA is already incorporated into other researches: multi-core FPGA, network-on-chip, etc. Futute work includes doing performance evaluation of the generated FCUDA C code on FPGA simulation tool, and debugging with more benchmarks. The FCUDA team will keep refining FCUDA existing code to be more stable, robust and plan to open-source it in a near future.

4 ACKNOWLEDGMENT

The author would like to give his appreciation Asst/Prof Kyle Rupnow as well as Dr. Swathi Gurumani from ADSC who provide generous support to him when he is doing the project, show him how the academic

environment looks like, and how to conduct various research methodologies in an articulate manner.

We wish to acknowledge the funding support for this project from Nanyang Technological University under the Undergraduate Research Experience on CAmpus (URECA) programme.

REFERENCES

[1] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.M. Hwu, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," Proceedings of IEEE Symposium on Application Specific Processors, July 2009. (Best Paper Award)

[2] A. Papakonstantinou, Y. Liang, J. Stratton, K. Gururaj, D. Chen, W.M. Hwu and J. Cong, "Multilevel Granularity Parallelism Synthesis on FPGAs," Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines, May 2011. (Best Paper Award)

[3] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.M. Hwu, "Efficient Compilation of CUDA Kernels for High-Performance Computing on FPGAs," ACM Transactions on Embedded Computing Systems, Special Issue on Application-Specific Processors, Vol. 13, Issue 2, September 2013.

[4] J. Stratton, S. Stone, and W.M. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," 21st International Workshop on Languages and Compilers for Parallel Computing, 2008.

[5] http://impact.crhc.illinois.edu/Parboil/parboil.aspx

[6] https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators

[7] https://developer.nvidia.com/cuda-downloads

[8] S. Lee, T. Johnson, and R. Eigenmann. "Cetus - An extensible compiler infrastructure for source-to-source transformation," Languages and Compilers for Parallel Computing, 2003.