A Code Optimization Framework for Performance Portability of GPU Kernels onto Custom Accelerators

Alexandros Papakonstantinou, Deming Chen and Wen-Mei W. Hwu
Electrical and Computer Engineering Department
University of Illinois at Urbana-Champaign
{apapako2, dchen, w-hwu}@illinois.edu

Abstract—**The shift toward parallel computing has resulted into a growing interest in computing systems with heterogeneous processing modules. Reconfigurable devices are often employed in such heterogeneous systems due to their low power and parallel processing benefits. An important issue in the programmability of these systems is the need for a single programming interface. Recent works have leveraged parallel programming models in tandem with high-level synthesis (HLS) to facilitate high abstraction parallel programming of FPGAs. Nevertheless, generating efficient custom hardware accelerators depends on the structure of the parallel input code. Code optimized for programmable multicore devices (e.g. GPUs or CPUs) may result in low-performance custom accelerators. In this work we describe a code optimization framework which analyzes and restructures CUDA kernels that were optimized for GPU devices in order to facilitate synthesis of efficient custom accelerators on FPGA. Our experimental results show that the proposed framework can achieve good performance portability.**

I. INTRODUCTION

Parallel processing has permeated nearly every digital computing domain during the last decade. Meanwhile, the importance of applications that apply computationally intensive algorithms on large volumes of data has grown significantly in domains involving simulation, mining or synthesis. Processing throughput is the key performance metric in such domains. Significantly increased throughput is offered by manycore devices (e.g. GPUs) that host hundreds of simple processing cores. FPGAs, are also receiving increased attention in applications that require high throughput, due to the abundant spatial parallelism and the low power advantages they offer. Furthermore, new high-level synthesis (HLS) tools have greatly improved the abstraction level of FPGA programming [1, 2]. On the other hand, traditional applications are best served by latency-oriented CPUs. As a consequence, the need for efficient handling of latency-bound as well as throughput-bound applications is leading toward heterogeneous systems which combine conventional processors with programmable manycore processors or FPGAs. Heterogeneity enables efficient processing of workloads with different latency, throughput and power requirements. In fact, the benefits of heterogeneous systems are driving industry towards higher integration of heterogeneity [3].

An important issue in the design and use of these heterogeneous compute systems is programmability. Several C-style programming models that offer high-abstraction, easy learning curve and parallelism annotation have been proposed. For example, CUDA [4] is a widely used parallel programming model for GPUs, whereas several HLS tools [1,2] support variations of C-style programming models that facilitate parallelism annotation. Equally important is the issue of a single programming model that offers a unified programming interface for systems with heterogeneous processing modules. OpenCL [5] is the outcome of an ongoing industry-initiated effort that targets this goal. In addition, several academic research efforts propose programming model transformation tools [6-9] that port applications across different programming models. Hence, the programmer may use a single programming interface to program heterogeneous compute devices.

A critical factor in either approach is performance portability. That is, achieving efficient and high performance execution on heterogeneous compute modules from a single source code description of an application. Performance portability depends on how well the source code can be mapped onto the target architecture. Code restructuring and architecture-specific optimizations can have a big impact in performance of massively parallel architectures. Previous studies in GPU architectures have shown that poorly optimized code can lead to dramatic performance degradation [10]. Hence, the use of a single programming model needs to be accompanied by a code porting framework that can reorganize the application description to better fit in the target platform. In this work we propose such a framework that leverages code transformations and optimizations to facilitate performance portability of CUDA kernels (optimized for GPUs) onto custom hardware accelerators on FPGAs.

The proposed code optimization framework is designed for FCUDA [11], a HLS-based flow which maps coarse-grained parallelism in CUDA kernels onto spatial parallelism on FPGA devices. Achieving good performance portability in the original FCUDA flow depends on annotation injection and potentially manual code restructuring by the programmer. The architectural differences between GPU and FPGAs may result in significant code restructuring and optimization work in order to achieve good performance portability. Moreover, identifying the required optimizations may be difficult for the average programmer, as well as error prone. Thus, in this work we describe a code optimization framework which leverages advanced compiler analysis techniques in tandem with powerful optimization and code transformation techniques that help achieve high performance portability.

II. RELATED WORK

The need for a single programming model across different compute platforms has motivated several previous research efforts. The popularity of C across different compute systems makes this programming model a natural choice for providing a single programming interface across different compute platforms such as FPGAs and GPUs. Diniz et al. [6], propose a HLS flow which takes C code as input and outputs RTL code that exposes loop iteration parallelism. Baskaran et al. [7], leverage the polyhedral model to convert parallelism in C loop nests into multithreaded CUDA kernels. Their framework also identifies off-chip memory data blocks with high reuse and generates data transfers to move data to faster on-chip memories. The OpenMP programming interface is a parallel programming model that is widely used in conventional multicore processors with shared memory spaces. The transformation framework in [8] describes how the different OpenMP pragmas are interpreted during VHDL generation, but it does not deal with memory space mapping. On the other hand, the OpenMP-to-CUDA framework proposed in

[9] transforms the directive-annotated parallelism into parallel multi-threaded kernels, in addition to providing memory space transformations and optimizations to support the migration from a shared memory space (in OpenMP) to a multi-memory space architecture (in CUDA). The OpenMP programming model is also used in the optimizing compiler of the Cell processor [12] to provide a single programming interface to the processor's PPE and SPE cores while supporting a single memory space abstraction. As described in [12], the compiler can orchestrate DMA transfers between the different memory spaces, while a compiler-controlled cache scheme takes advantage of temporal and spatial data access locality.

A slightly different approach is used in the FCUDA framework [11], which adopts a popular GPU programming interface, namely CUDA, to exploit the spatial parallelism of FPGAs for massively data parallel kernels. FCUDA, leverages user injected annotations to handle the compute and data transfer tasks of the kernels and performs a sequence of transformations in the source code to map the grid of CUDA threadblocks onto a grid of custom cores. In this work we enhance the code transformation techniques of FCUDA with new optimizations that help improve both compute and data transfer performance while saving the programmer from the burden of identifying what annotations to inject and in certain cases manually restructuring the kernel code to achieve good performance portability. The proposed optimizations take into account the architectural characteristics of FPGAs and restructure the code accordingly. The following section describes some of the characteristics that are considered in the code optimization framework.

III. FCUDA FRAMEWORK BACKGROUND

CUDA provides a parallel programming interface which allows concise expression of SPMD parallelism. The parallel tasks of applications expressed in CUDA are decomposed into a large number of fine-grained threads which are grouped into blocks (*thread-blocks*) that may execute completely independently (i.e. free of any inter-block synchronization). Threads within each block may be synchronized, through the use of corresponding CUDA primitives. Parallelism is exposed in FCUDA primarily at the thread-block level by mapping thread-blocks onto custom parallel *cores* generated by the HLS engine. The threads in each thread-block are folded into a nest of *tDim* loops (*thread-loop*), where *tDim* is the thread dimensionality in the thread-block (i.e. between 1 and 3). Thread level parallelism may be exposed through thread-loop unrolling before RTL generation. FCUDA decomposes each kernel into compute and data-transfer tasks with the aid of user-injected annotations that mark the compute and transfer regions of the kernel. These annotations also carry related information, such as the size of data block transfer and the type of transfer (i.e. burst or simple). Organizing data transfers into bursts is critical for achieving good performance across a wide range of CUDA applications which process large volumes of data that are stored in off-chip memories. Efficient utilization of bursts depends on the organization of data fetches along contiguous address sequences. This is important for execution performance on both GPU and FPGA platforms. In the GPU platform, this is often referred to as *data coalescing* and is achieved by organizing off-chip accesses of neighboring threads (in terms of thread id) to neighboring memory addresses. The GPU architecture, consequently, exploits the DDR memory burst feature along with the instant context switch capability between different thread-blocks in order to achieve high off-chip bandwidth accesses. In the FCUDA flow, on the other hand, thread-blocks execution is determined statically at compile time (i.e. no context switches) and threads usually execute in smaller warps (i.e. less concurrent threads per block). Hence, data transfer bursts correspond to block-wide sequences (rather than warp-wide), which are generated by the FCUDA transformation framework according to the user annotations.

The proposed optimization framework, enhances the FCUDA flow, by introducing advanced analysis and transformation techniques that i) remove the burden of annotation injection from the user (eliminating, thus, the human error factor) and ii) enable performance portability across heterogeneous platforms through code restructuring optimizations (hence, making the single programming model meaningful in heterogeneous environments).

IV. PERFORMANCE PORTABILITY FRAMEWORK

A. Framework Environment and Goals

The FCUDA flow is based on Cetus [13], a source-to-source parallelizing compiler (available as open source) which offers several tools for parallelism analysis and optimization. The proposed performance portability framework leverages the original FCUDA transformations in combination with existing/modified and newly introduced optimizations (currently under implementation within the Cetus compiler framework). With regard to RTL generation, FCUDA uses the commercial HLS tool, AutoPilot [1], which takes as input the transformed source code produced by the Cetus compiler. Thus, even though some of the code transformation in Cetus may be customized for AutoPilot, the main optimization concepts are applicable to other HLS tools also.

As mentioned in the previous section, the FCUDA mapping of CUDA kernels on the reconfigurable fabric is based on kernel decomposition into compute and data-transfer tasks, thus aiming to disentangle compute throughput from off-chip data transfer throughput. Moreover, the decomposition enables separate optimization of the two types of tasks. Once tasks are identified they are abstracted into separate procedures (i.e. outlined) that may be called from the kernel procedure. With regard to data transfer tasks the goals of the optimization framework include i) data coalescing analysis for burst generation, ii) efficient BRAM resource allocation and iii) redundant off-chip data transfer elimination. On the other hand, compute optimizations aim to restructure kernel compute and data flow through elimination of interleaved off-chip data transfers, redundant synchronization and other control flow overhead. Thus, the HLS engine can better extract parallelism at the instruction level and the loop iteration level.

The proposed code optimization framework takes as input CUDA code along with a set of optional user-injected kernel annotations. These annotations may include i) the value of the Num_{BRAM} parameter, which specifies the maximum number of BRAMs allowed for the hardware implementation of each threadblock (i.e. custom core) and ii) the value ranges of the kernel input parameters and the CUDA implicit variables (e.g. thread-block and block-grid size vectors). Let us note that these annotations are not related to the aforementioned annotations used in the original FCUDA flow. Parameter Num_{BRAM} helps constrain the BRAM usage per each custom core, whereas the kernel input parameter ranges may facilitate kernel optimizations that depend on parameters whose values can not be statically determined. This way, the user can assist the optimization of the kernel by communicating to the framework domain-specific input constraints. The input parameter ranges may be annotated as bounded range section (BRS) descriptors (i.e. d=(0:100)) [14] or combinations of symbolic expressions with BRS descriptors (i.e. d=2^n, n=(5:8)). Due to space limitation, we discuss the analyses and transformations used in the proposed code optimization

framework for the case of a single kernel procedure which has been flattened through inlining of its callee procedures. Nevertheless, the techniques employed in the framework can support inter-procedural analysis and optimization of kernels with complex multilevel call graphs.

B. Framework Optimizations

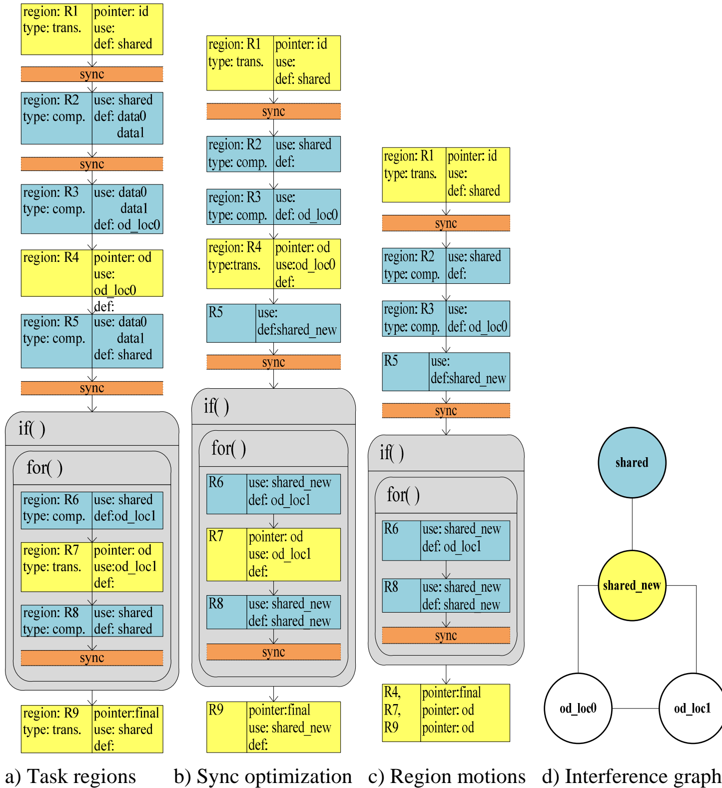
1) Disentangle compute and data transfer parts: During this phase, the kernel is scanned for statements that include both compute and data-transfer tasks. Such statements are identified by their reference to global memory space (GMS) elements through global memory (GM) pointer/array variables (prior to this step, all off-chip data references are converted into array-based accesses enabling unified handling) and the fact that they entail operations other than just a simple copy. The outcome of this transformation is the replacement of the original statement by a simple copy statement (i.e. data transfer) and one or more compute statements. For example the mixed-task statement (*od*: global memory, *shared*: on-chip memory):

```
od[g_wpos] = (shared[idata0] - shared[idata1]) * INV_SQRT_2;
```

is converted into the following two statements, whereas a new intermediate compute storage buffer (*od_loc0*) is introduced:

```
od_loc0 = (shared[idata0] - shared[idata1]) * INV_SQRT_2;
od[g_wpos] = od_loc0;  // simple copy
```

2) Hierarchical control flow graph generation: Having separated the compute from the data transfer tasks at the statement level, the initial compute and data-transfer regions are annotated and a hierarchical control flow graph is built. The control flow graph is similar to conventional control flow graphs constructed with basic blocks, with the only difference that instead of basic blocks it comprises task-regions. A task-region is characterized by its statements' type (i.e. compute or data-transfer) and its bounds may be determined by i) thread-loop invariant control flow statements ii) FCUDA synchronization (sync) directives or iii) statements of differing type (Fig. 1(a)).



a) Task regions b) Sync optimization c) Region motions d) Interference graph

Figure 1. Dwt kernel example

3) Thread-loop iteration space adjustment: In each task-region containing thread-loop dependent control flow structures, symbolic analysis tools [14] are used to adjust the thread-loop iteration space accordingly (initially the thread-loop iteration space is equivalent with the threadIdx space). Subsequently each task-region is annotated with the adjusted iteration space.

4) Global memory access analysis and annotation: Every data transfer to/from GM is examined to identify the range and contiguity characteristics of the accessed address space in GM. Symbolic expression analysis in tandem with range analysis are employed to generate BRS descriptor sets [14] which are used to annotate each GM access statement (no aliasing between GM pointers is allowed). Subsequently, the BRS sets are used to detect access range overlap between pairs of accesses (referring to the same global variable). The contiguity attribute is used to convert data transfer statements to data transfer bursts (represented with *memcpy()* calls in AutoPilot's programming model) across contiguous address ranges. On the other hand, address range overlap information is used for eliminating redundant off-chip accesses. This may be feasible for pairs of off-chip accesses where one of the accesses dominates the other and they have a read-after-read (RAR) or read-after-write (RAW) relationship (with no intermediate overlapping write accesses). When both of these conditions hold and the *meet* operation of the two sets generates a BRS set that is equivalent to the wider of the two sets (i.e. one address range is a subset of the other address range), a global memory dependence edge (GMDE) is added from the dominator to the post-dominator statement. The GMDE is used in two code optimizations: i) in data transfer statement code motions to endure that no GMS dependences are broken and ii) in the elimination of redundant read transfers (when data is available in on-chip memories). The second optimization can be also used for data streaming between custom cores of sequential kernels (note that in this work we consider only single kernel applications).

5) Synchronization and on-chip memory optimizations: The aim of this optimization phase is to eliminate (if feasible) synchronization directives between compute task regions in order to consolidate multiple compute regions into a single region (HLS can find more optimization opportunities in larger compute regions). This optimization is based on dataflow analysis for the *local arrays* used within each kernel. With the term local arrays we refer to both arrays that exist in the original CUDA kernel and correspond to GPU on-chip memories (e.g. shared and constant memory) and arrays that are created in FCUDA as a consequence of applying expansion to scalar variables that are live across explicit synchronization points (*__syncthreads*) or implicit synchronization points (i.e. region boundaries). Eliminating synchronization directives requires dependence testing of array accesses in compute regions across the synchronization point (data-transfer regions do not need to be analyzed, as they enforce synchronization by being abstracted into separate procedures). The proposed optimization framework leverages the range test technique [15] to identify array reference dependencies across regions. In the *dwt* running example depicted in Fig. 1, elimination of the *__syncthreads* primitive between regions *R2* and *R3* (Fig. 1(b)) is based on the application of the range test on the references of arrays *data0* and *data1* in regions *R2* and *R3*, as well as the references of array *shared* in regions *R2* and *R5*. The results of the range test reveal a WAR loop-carried dependence across *R2* and *R5* regions for array *shared*. WAR dependences can be eliminated by array renaming [14], thus, enabling elimination of the synchronization between regions *R2* and *R3*. Moreover, the scalar expansion of variables *data0* and *data1* can be avoided

due to the *R2-R3* synchronization elimination (i.e. number of arrays is reduced by one: $+\text{Array}_{\text{share_new}} - \text{Array}_{\text{data0}} - \text{Array}_{\text{data1}}$). Note that arrays *shared* and *shared_new* should not be bound to the same BRAM. This is taken into account in the following optimization phase during the construction of the interference graph.

6) BRAM binding through graph coloring: Local arrays are stored into BRAMs which are used as scratchpad memories for data fetched/written from/to off-chip memory. Given unlimited BRAM resource, FCUDA pre-fetches all the required off-chip data in the beginning of the kernel execution and delays all of the off-chip memory writes until the completion of the compute execution (Fig. 1(c)). Nevertheless, this could entail a high cost in BRAM resource usage. The proposed optimization framework employs a modified version of k-coloring to bind arrays onto BRAMs. The user can optionally specify the BRAM constraint (number of colors: *k*) during FCUDA invocation. If no BRAM constraint is specified the number of BRAMs is set to the maximum number of overlapping lifetimes after off-chip reads/writes have been percolated toward the beginning/end of the kernel control flow graph (Fig. 1(c)). The k-coloring algorithm (often used in compilers for register allocation and register spilling) is employed in FCUDA as a means of mapping arrays onto BRAM resource and spilling the BRAM contents to off-chip memory when the BRAM resource can not accommodate all live arrays. Initially, the def-use chains of arrays are used to determine the connected live ranges (*webs*) of all the array variables in the kernel [16]. Subsequently, an interference graph $G(V,E)$ is constructed, where nodes in V correspond to webs and edges in E represent interference between the life ranges of the webs they connect (Fig. 1(d)). Interference edges are also added between nodes that correspond to webs of arrays that should not be bound to the same BRAM (e.g. arrays *shared* and *shared_new*). Nodes that correspond to webs of local arrays that were declared in the original kernel are pre-colored as they are not candidates for spilling (e.g. *shared* and *shared_new* in Fig. 1(d)). Moreover, each node is annotated with a BRAM size which corresponds to the number of BRAMs required by the corresponding array. The interference graph nodes are subsequently colored and BRAM spilling determines the location of the data transfer regions in the control flow graph of the kernel (details are omitted due to space limitation). Finally, the data-transfer regions are percolated to the corresponding locations of the control flow graph in the abstract syntax tree (AST) representation of the kernel.

7) Thread-loop unrolling and array partitioning: Thread level parallelism can be exposed through unrolling of the thread-loops inside compute regions. Due to the limited number of ports in BRAMs, array partitioning is often necessary in order to extract the performance benefits of thread-loop unrolling.

Currently, the framework is under ongoing development and provides only partial support of the aforementioned optimizations. Proof of concept is based on evaluation of a few benchmarks (see next section) which were partially optimized through manual manipulations.

V. Experimental Results & Conclusion

Fig. 2 presents preliminary performance results for a small set of kernels: i) Matrix Multiplication (*mm*), ii) Fast Walsh Transform (*fwt1*, *fwt2*), iii) Coulombic Potential (*cp*), and iv) Discreet Wavelet Transform (*dwt*). The FPGA execution latency (for two different off-chip memory bandwidths, namely 16GB/sec and 64GB/sec) is compared with the GPU execution latency. We observe that for equal off-chip bandwidths (i.e. 64GB/s) the custom FPGA accelerator outperforms the GPU device in three out of the five tested kernels. The proposed code optimization framework employs advanced analysis and optimizations techniques that enable high-performance FPGA execution of CUDA kernels that were optimized for the GPU architecture. In our future work we plan to enhance the proposed framework with inter-procedural optimization features for multi-kernel acceleration and optimization space exploration [17]. Moreover, we consider extending the framework infrastructure with data and compute pattern analysis tools that could assist the user in deciding which device to select in a heterogeneous system for a generic kernel (i.e. a kernel which is not optimized for a particular architecture).

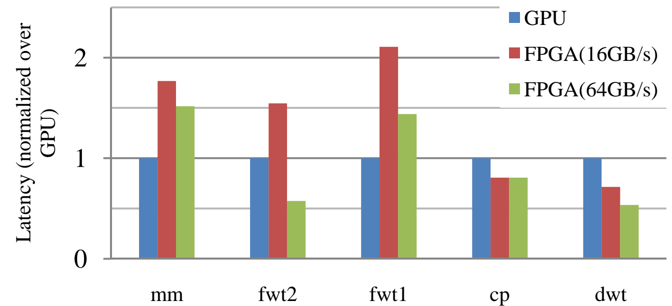


Figure 2. FPGA vs. GPU performance comparison

References

[1] Z. Zhang et al., "Autopilot: a platform-based ESL synthesis system," in High-Level Synthesis: from Algorithm to Digital Circuit, P. Coussy and A. Morawiec, Eds. Netherlands: Springer, 2008.

[2] Impulse Accelerated Technologies, "Impulse CoDeveloper," 2010, www.impusec.com

[3] AMD Fusion family of APUs: Enabling a superior, immersive PC experience," White Paper, AMD, Mar. 2010. [Online]. Available: http://sites.amd.com/us/Documents/48423Bfusion whitepaper WEB.pdf

[4] NVIDIA, "CUDA Zone", 2011, Accessed Mar 2011, http://www.nvidia.com/object/cuda_home_new.html

[5] Khronos, "The OpenCL specification, version: 1.1," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2010.

[6] P. Diniz et al., "Automatic Mapping of C to FPGAs with the DEFACTO compilation and synthesis system," *Microprocessors & Microsystems*, vol. 29(2-3), 2005, pp. 51-62.

[7] M. Baskaran et al., "Automatic C-to-CUDA code generation for affine programs", *Proc. Int'l Conf. Compiler Construction*, CC'10, 2010.

[8] Y.Y. Leow et al., "Generating Hardware from OpenMP Programs," *Proc. IEEE Int'l Conf. Field Programmable Technology* (FPT), 2006.

[9] S. Lee et al., "OpenMPC: Extended OpenMP programming and tuning for GPUs," *Proc. ACM/IEEE Conf. Supercomputing* (SC'10), 2010.

[10] S. Ryoo et al., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *ACM Symp. Principles and Practices of Parallel Programming* (PPoPP'08), 2008.

[11] A. Papakonstantinou et al., "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," *Proc. IEEE Symp. Application Specific Processors* (SASP'09), 2009.

[12] A. Eichenberger et al., "Optimizing Compiler for a CELL processor," *Proc. IEEE Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT'05), 2005.

[13] H. Bae et al., "Cetus: A source-to-source compiler infrastructure for multi-cores," *Proc. Int'l Workshop Compilers for Parallel Computing* (CPC'09), 2009.

[14] R. Allen and K. Kennedy, "Optimizing compilers for modern architectures," Morgan Kaufmann, 2002.

[15] W. Blume et al., "The range test: a dependence test for symbolic, non-linear expressions," *Proc. Conf. Supercomputing* (SC'94), 1994.

[16] S. Muchnick, "Advanced compiler design & implementation", Morgan Kaufmann, 1997.

[17] A. Papakonstantinou et al., "Multilevel granularity parallelism synthesis on FPGAs," *Proc. IEEE Int'l Symposium Field-Programmable Custom Computing Machines* (FCCM'11), 2011.