Efficient Compilation of CUDA Kernels for High-Performance Computing on FPGAs

ALEXANDROS PAPAKONSTANTINOU, University of Illinois at Urbana-Champaign
KARTHIK GURURAJ, University of California, Los Angeles
JOHN A. STRATTON and DEMING CHEN, University of Illinois at Urbana-Champaign
JASON CONG, University of California, Los Angeles
WEN-MEI W. HWU, University of Illinois at Urbana-Champaign

The rise of multicore architectures across all computing domains has opened the door to heterogeneous multiprocessors, where processors of different compute characteristics can be combined to effectively boost the performance per watt of different application kernels. GPUs, in particular, are becoming very popular for speeding up compute-intensive kernels of scientific, imaging, and simulation applications. New programming models that facilitate parallel processing on heterogeneous systems containing GPUs are spreading rapidly in the computing community. By leveraging these investments, the developers of other accelerators have an opportunity to significantly reduce the programming effort by supporting those accelerator models already gaining popularity. In this work, we adapt one such language, the CUDA programming model, into a new FPGA design flow called FCUDA, which efficiently maps the coarse- and fine-grained parallelism exposed in CUDA onto the reconfigurable fabric. Our CUDA-to-FPGA flow employs AutoPilot, an advanced high-level synthesis tool (available from Xilinx) which enables high-abstraction FPGA programming. FCUDA is based on a source-to-source compilation that transforms the SIMT (Single Instruction, Multiple Thread) CUDA code into task-level parallel C code for AutoPilot. We describe the details of our CUDA-to-FPGA flow and demonstrate the highly competitive performance of the resulting customized FPGA multicore accelerators. To the best of our knowledge, this is the first CUDA-to-FPGA flow to demonstrate the applicability and potential advantage of using the CUDA programming model for high-performance computing in FPGAs.

Categories and Subject Descriptors: C.3 [**Special Purpose and Application-Based Systems**]: Real time and embedded systems; C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms: Design, Performance

Additional Key Words and Phrases: FPGA, high-level synthesis, parallel programming model, high-performance computing, source-to-source compiler, heterogeneous compute systems

ACM Reference Format:
Papakonstantinou, A., Gururaj, K., Stratton, J. A., Chen, D., Cong, J., and Hwu, W.-M. W. 2013. Efficient compilation of CUDA kernels for high-performance computing on FPGAs. ACM Trans. Embedd. Comput. Syst. 13, 2, Article 25 (September 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2514641.2514652

Authors' addresses: A. Papakonstantinou (corresponding author), Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, IL; email: apapako02@illinois.edu; K. Gururaj, Computer Science Department, University of California, Los Angeles, CA; J. A. Stratton and D. Chen, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, IL; J. Cong, Computer Science Department, University of California, Los Angeles, CA; W.-M. W. Hwu, Electrical and Computer Engineering Department, University of Illinois at Urbana-Champaign, IL.

DOI: http://dx.doi.org/10.1145/2514641.2514652

1. INTRODUCTION

Parallel processing, once exclusively applied in supercomputing servers and clusters, has permeated nearly every digital computing domain during the last decade: from PCs and laptops to cell phones and from compute clusters in the cloud to gaming and networking devices. Democratization of parallel computing was driven by the power wall encountered in traditional single-core processors, and it was enabled by the continued shrinking of transistor feature size that rendered Chip MultiProcessors (CMP) feasible. Meanwhile, the importance of parallel processing in a growing set of applications that leverage computationally heavy algorithms, such as simulation, mining, or synthesis has underlined the need for on-chip concurrency at a granularity coarser than instruction level. One of the strategies for achieving higher on-chip concurrency is based on increasing the percentage of on-chip silicon real estate devoted to compute modules. In other words, instantiating more, though simpler, cores to efficiently execute applications with massive parallelism and regular structure (i.e., predictable control-flow and data access patterns). This philosophy has been adopted in the architecture of multicore devices such as the Cell-BE [IBM 2006], the TILE [TILERA 2012] family, or the GeForce [NVIDIA 2012b] family as well as FPGAs, which offer reconfigurable spatial parallelism. Large caches, complex branch predictors, and dynamic instruction schedulers are usually not employed in such devices, which can provide up to one to two orders of performance boost for applications with inherently massive parallelism.

Despite the shared philosophy of achieving a high degree of parallelism using simpler cores, these devices possess very diverse characteristics which render them optimal for different types of applications and different usage scenarios. For example, the IBM cell is a MultiProcessor Systems-on-Chip (MPSoC) based on a set of heterogeneous cores. Thus, it can serve either as a stand-alone multiprocessor or a multicore accelerator, though at the cost of lower on-chip concurrency (i.e., it has fewer cores compared to other types of multicore accelerators). GPUs, on the other hand, consist of hundreds of processing cores clustered into Streaming Multiprocessors (SMs) that can handle kernels with high degree of data-level parallelism. Launching the SMs requires a host processor, though. As a result of the diversity in the multicore space, a growing interest in heterogeneous compute systems either at the chip [IBM 2006; AMD 2012] or the cluster level [Showerman et al. 2009] is observed. Especially in High-Performance Computing (HPC), heterogeneity has been gaining great momentum. In this work we focus on high-performance acceleration on FPGAs that offer flexible and power-efficient application-specific parallelism through their reconfigurable fabric.

Unfortunately, the performance and power advantages of the different multicore devices are often offset by the programmability challenges involved in the parallel programming models they support. First, migration into the parallel programming mindset involves an associated cost that a wide range of programmers and scientists who enjoy the object-oriented features and simplicity of sequential languages such as C/C++ and Java are not willing to pay. Second, achieving efficient concurrency in several of these programming models entails partial understanding of the underlying hardware architecture, thus restricting adoption. This is especially true for FPGA programming, where the traditional programming interfaces are VHDL and Verilog, which describe computation at the Register Transfer Level (RTL). Democratization of parallel computing has also been hindered by domain-specific programming interfaces (e.g., OpenGL and DirectX programming models used in GPU devices) that require application morphing into graphics-style computation. Fortunately, the parallel programming interface space has dramatically changed during the last years (e.g., new advanced high-level synthesis tools for FPGAs, new programming models such as CUDA [NVIDIA 2012a] and OpenCL [Khronos 2011]), easing the path toward parallel

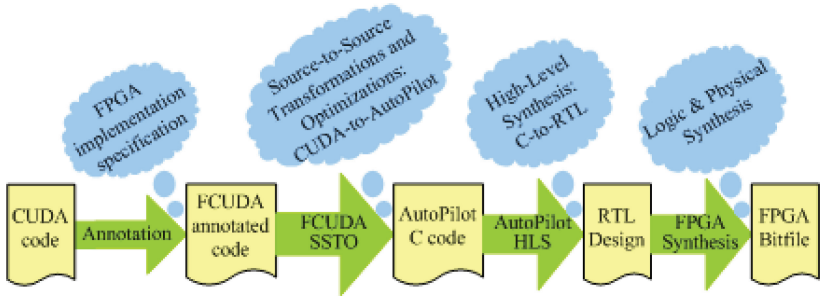


Fig. 1. FCUDA flow.

computing. Nevertheless, achieving high performance in an efficient manner still remains a challenge. The use of different parallel programming models by heterogeneous accelerators complicates the efficient utilization of the devices available in heterogeneous compute clusters, which reduces productivity and restricts optimal mapping of kernels to accelerators. In this work we describe the FCUDA framework which aims to address both: (i) the programmability efficiency of FPGAs for massively parallel application kernels and (ii) providing a common programming interface with GPUs and CPUs.

The recent introduction of the CUDA programming model [NVIDIA 2012a] by Nvidia marked a significant milestone toward leveraging the parallel compute capacity of GPUs for nongraphics applications. CUDA provides a programming model that facilitates parallelism expression across a hierarchy of thread groups. The thread hierarchy is defined within procedures that are called kernels and contain the inherently parallel sections of an application. Kernels are executed on the manycore GPU device whereas the rest of the code is executed on the CPU. In this work we explore the use of CUDA for programming FPGAs. We propose the FCUDA flow (Figure 1) which is designed to efficiently map the coarse- and fine-grained parallelism expressed in CUDA kernels onto the reconfigurable fabric. The proposed flow combines source-code transformations and optimizations with High-Level Synthesis (HLS) to enable high-performance acceleration with high programming abstraction. The input CUDA code is initially restructured by a novel Source-to-Source Transformation and Optimization (SSTO) engine, implemented with the Cetus compiler [Lee et al. 2003], and subsequently fed to an advanced high-level synthesis tool called AutoPilot[1] [Zhang et al. 2008; Cong et al. 2011], which generates RTL output. A recent independent evaluation study [BDTI 2010] on the usability and quality of results of AutoPilot has concluded that the quality of the generated RTL is comparable to hand-written RTL.

The SSTO engine performs two main types of transformations: (i) data communication and compute optimizations and (ii) parallelism mapping transformations. The first are based on analysis of the kernel dataflow followed by data communication and computation reorganization. This set of transformations aims to enable efficient mapping of the kernel computation and data communication onto the FPGA hardware. The latter expose the parallelism inferred in the CUDA kernels in the generated AutoPilot-C descriptions which are converted by the HLS engine into parallel Processing Engines (PEs) at Register Transfer Level (RTL).

[1]AutoPilot was developed by a startup company called AutoESL and was based on the xPilot HLS system [Chen et al. 2005] developed at UCLA. AutoESL was acquired by Xilinx in early 2011 and the AutoPilot tool is now available from Xilinx.

The use of CUDA for mapping compute-intensive and highly parallel kernels onto FPGAs offers three main advantages. First, it provides a C-styled API for expressing coarse-grained parallelism in a very concise fashion. Thus, the programmer does not have to incur a steep learning curve or excessive additional programming effort to express parallelism (expressing massive parallelism directly in AutoPilot-C can incur significant additional effort from the programmer). Second, the CUDA-to-FPGA flow shrinks the programming effort in heterogeneous compute clusters with GPUs and FPGAs by enabling a common programming model. This simplifies application development and enables efficient evaluation of alternative kernel mappings onto the heterogeneous acceleration devices by eliminating time-consuming application porting tasks. Third, the wide adoption of the CUDA programming model and its popularity render a large body of existing applications available to FPGA acceleration.

The main contributions in this article are summarized in the following.

—We describe a novel CUDA-to-FPGA programming flow which combines FPGA-specific source-to-source optimizations and transformations with high-level synthesis.
—We show that the CUDA programming model, though designed for the GPU computing domain, can be leveraged in the FPGA computing domain to efficiently generate customized parallel compute architectures on the reconfigurable fabric.
—We provide a set of experimental results that show the performance benefits of the source-to-source optimizations implemented in FCUDA.
—We use several integer CUDA kernels to compare execution performance between FCUDA-configured FPGAs and GPUs and we provide insight on the comparison results based on the kernel characteristics.

In the next section we discuss FPGA computing platform features along with previous related work. Section 3 describes the characteristics of the CUDA and AutoPilot programming models and provides insight to the suitability of the CUDA API for programming FPGAs. The FCUDA translation details are presented in Sections 4 and 5, while Section 6 describes experimental results and shows that the proposed flow can efficiently exploit the computational resources of FPGAs in a customized fashion. Finally, Section 7 concludes the article and discusses future work.

2. RECONFIGURABLE COMPUTING

2.1. The FPGA Platform

As silicon process technology keeps enabling smaller feature sizes, the transistor capacity of FPGAs is increasing dramatically. Modern FPGAs, fabricated with the latest 28nm process technology, host a heterogeneous set of hard IPs (e.g., PLLs, ADCs, PCIe interfaces, general-purpose processors, and DSPs) along with millions of reconfigurable logic cells and thousands of distributed memories (e.g., BRAMs in Xilinx Virtex-5 devices [Xilinx 2012]). Their abundant compute and memory storage capacity makes FPGAs attractive for the implementation of compute-intensive and complex applications [Che et al. 2008; Cong and Zou 2008; Cho et al. 2009], whereas the hard IP modules offer compute (e.g., CPUs, DSPs) and data communication (e.g., PCIe, Gbps transceivers) efficiency, enabling high-performance System-on-Chip (SoC) implementations [He et al. 2009]. One of the main benefits of hardware reconfigurability is increased flexibility with regard to leveraging different types of application-specific parallelism, for example, coarse- and fine-grained, data- and task-level, and versatile pipelining. Moreover, parallelism can be leveraged across FPGA devices such as in the HC-1 Application-Specific Instruction Processor (ASIP) [Convey 2011] which combines a multicore CPU with multi-FPGA-based custom instruction accelerators. The

potential of multi-FPGA systems to leverage massive parallelism has been also exploited in the recently launched Novo-G supercomputer [CHREC 2012], which hosts 192 reconfigurable devices.

Power is undeniably becoming the most critical metric of systems in all application domains from mobile devices to cloud clusters. FPGAs offer a significant advantage in power consumption over CPUs and GPUs. Williams et al. [2008] showed that the computational density per watt in FPGAs is much higher than in GPUs. The 192-FPGA Novo-G [CHREC 2012] consumes almost three orders of magnitude less power compared to Opteron-based Jaguar and Cell-based Roadrunner supercomputers, while delivering comparable performance for bioinformatics-related applications. In this work we evaluate the FCUDA framework and demonstrate that for certain types of applications, the CUDA-to-FPGA implementation can offer competitive performance to the GPU at a fraction of the energy consumption.

2.2. Programmability

Despite the power and application customization advantages, FPGAs have not enjoyed wide adoption due to the programmability challenges they pose. Traditionally, FPGA programming required RTL and hardware design skills. Fortunately, the advent of several academic [Gajski 2003; Chen et al. 2005; Diniz et al. 2005; Gupta et al. 2004] and commercial [Impulse 2003; Zhang et al. 2008; Mentor 2012] high-level synthesis tools has helped raise the abstraction level of the programming model. Most of these tools generate RTL descriptions from popular High-Level programming Languages (HLLs). The inherent sequential flow of execution in traditional programming languages restricts parallelism extraction at granularities coarser than loop iterations [Diniz et al. 2005]. Thus the abundant spatial parallelism of FPGAs may not be fully exploited. To overcome this problem, various HLS frameworks have resorted to the introduction of new parallel programming models [Hormati et al. 2008; Huang et al. 2008] or language extensions for coarse-grained parallelism annotation [Impulse 2003; Zhang et al. 2008]. However, expressing massive coarse-grained parallelism in several of the proposed parallel programming models may incur significant learning and/or programming effort. A new data-parallel programming model, OpenCL [Khronos 2011], has been recently proposed for programming heterogeneous parallel systems. Initial efforts to use OpenCL for FPGA programming include OpenRCL [Lin et al. 2010] (SIMD multicore on FPGA) and SOpenCL [Owaida et al. 2011] (MIMD multicore on FPGA). However, both of these works use core templates rather than HLS to customize cores for each application. Moreover, neither of these works compares FPGA performance with other devices. In this work we propose the use of CUDA for concise expression of coarse-grained parallelism in inherently parallel and compute-intensive kernels. The popularity of CUDA demonstrates its benefits in terms of learning and programming effort. Moreover, by adopting CUDA we enable a common programming interface for two widely used heterogeneous types of accelerators, namely GPUs and FPGAs.

2.3. Performance

There has been considerable previous research on the comparison of performance between different types of compute devices. Che et al. [2008] compare CPU, GPU, and FPGA platforms' performance for three types of applications: Gaussian elimination, DES, and Needleman-Wunsch. They show that FPGAs achieve better latencies in terms of cycles, especially for applications with bit-level and custom bitwidth operations. However, their work does not take into account clock frequency. Frequency is a performance variable in the FPGA platform and it depends on the RTL design (i.e., pipelining) and the physical implementation (place and route). Williams et al. [2008] define Computational Density (CD) and Computational Density per Watt (CDW) for

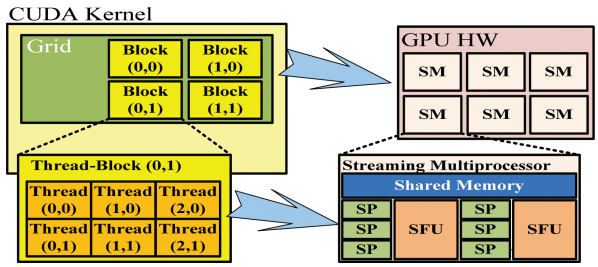


Fig. 2. CUDA programming model.

fixed and reconfigurable multicore devices. They compare these metrics for FPGAs, GPUs, and multicore CPUs and show that FPGAs offer higher computational densities for bit operations and 16-bit integer arithmetic (up to 16X and 2.7X respectively) over GPUs but may not compete as well for wider bidwidths, such as 32-bit integer and single-precision floating-point operations (0.98X and 0.34X respectively). In their work, FPGA frequency is determined based on the frequency of a single ALU, not accounting for interconnection delays between different ALUs. Random number generators for commonly used distributions are implemented by Thomas et al. [2009] for four distinct compute platforms: multicore CPUs, GPUs, 2D core grids, and FPGAs. They explore different algorithms and they conclude that none of the used algorithms is optimal across all devices. In this work we map data-parallel CUDA kernels onto parallel custom processing engines on the FPGA. The performance of the FCUDA-generated custom accelerators is evaluated by taking into account both cycle latency and clock frequency.

3. OVERVIEW OF PROGRAMMING MODELS IN FCUDA

3.1. CUDA C

The CUDA programming model, developed by Nvidia, offers a simple interface for executing general-purpose applications on Nvidia GPUs. Specifically, CUDA is designed to expose parallelism on the SIMT (Single Instruction, Multiple Thread) architecture of CUDA-capable GPUs [NVIDIA 2012b]. CUDA C is based on a set of extensions to the C programming language which entail code distinction between host (i.e., CPU) and GPU code. The GPU code is organized into procedures called *kernels* which contain the embarrassingly parallel parts of applications and are invoked from the host code. Each kernel implicitly describes thousands of CUDA threads that are organized in groups called *threadblocks*. Threadblocks are further organized into a *grid* structure (Figure 2). The number of threadblocks per grid and threads per threadblock are specified in the host code, whereas built-in variables (i.e., *threadIdx*, *blockIdx*) may be used in the kernel to specify the computation performed by each thread in the SIMT architecture. It is the programmer's responsibility to partition the computation into parallel coarse-grained tasks (threadblocks) that consist of finer-grained subtasks (threads) that can execute in parallel. The proposed FCUDA flow maps the parallelism contained in the hierarchy of threads and threadblocks of the kernel onto spatial hardware parallelism on the FPGA.

CUDA extends C with synchronization directives that control how threads within a threadblock execute with respect to each other (i.e., synchronization points impose a bulk-synchronous type of parallelism). Conversely, threadblocks may execute independently in any parallel or sequential fashion. In recent updates of the CUDA platform, atomic operations and fence directives can be used to enforce the order of memory accesses either at threadblock or grid level. The two granularities of CUDA parallelism are also represented in the SIMT architecture (Figure 2), where Streaming Processors (SPs) are clustered in Streaming Multiprocessors (SMs). Each threadblock is assigned

to one SM and its corresponding threads are executed on the SPs of the SM in parallel thread groups called *warps*. The SIMT architecture executes warps in a SIMD (Single Instruction, Multiple Data) fashion when warp threads converge on the same control flow. On the other hand, control-flow divergent threads within a warp execute sequentially, limiting the amount of exploited concurrency from the SIMT hardware. The FCUDA framework generates Processing Engines (PEs) at threadblock granularity with custom warp size parallelism that is determined by the programmer and performance-versus-resource-budget trade-offs.

The CUDA programming model entails multiple memory spaces with diverse characteristics. In terms of accessibility, memory spaces can be distinguished into thread-private (e.g., SP-allocated registers), threadblock-private (e.g., SM-allocated on-chip memory), and global (e.g., off-chip DDR memory). Each SP is allocated a set of registers out of a pool of SM registers according to the kernel's variable use. The SM-allocated memory is called *shared memory* and it is visible by all the threads within the threadblock assigned to the corresponding SM (Figure 2). In terms of globally visible memory spaces, CUDA specifies one read-write (*global memory*) space and two read-only (*constant* and *texture memory*) spaces. Registers and shared memory incur low access latency but have limited storage capacity (similarly to CPU register files and tightly coupled scratchpad memories). The three globally visible memory spaces are optimized for different access patterns and data volumes. In the FPGA platform we leverage two main memory structures: off-chip DDR and on-chip BRAMs and registers. The visibility and accessibility of the data stored on these memories can be customized arbitrarily depending on the application's characteristics.

3.2. AutoPilot-C

AutoPilot is an advanced commercial HLS tool which takes C/C++ code and generates an equivalent RTL description in VHDL, Verilog, and SystemC. The C/C++ input is restricted to a *synthesizable subset* of the language. The main features not supported in high-level synthesis include dynamic memory allocation, recursive functions, and, unsurprisingly, the standard file/io library calls. The input code may be annotated by user-injected directives that enable automatic application of different transformations. AutoPilot converts each C procedure into a separate RTL module. Each RTL module consists of the datapath that realizes the functionality of the corresponding C procedure along with FSM logic that implements control-flow scheduling and datapath pipelining. Procedure calls are converted to RTL module instantiations, thus transforming the procedure call graph of the application into a hierarchical RTL structure. A pair of *start/done* interface signals is attached to the FSM logic to signal the beginning and end of the module's operation, facilitating inter-module synchronization.

AutoPilot leverages the LLVM compiler infrastructure [LLVM 2007] to perform code transformations and optimizations before its backend RTL generator translates the described functionality into datapath and FSM logic. Some transformations and optimizations are performed by default whereas others are triggered by user-injected directives. In particular, AutoPilot will automatically attempt to extract parallelism both at instruction level and task level (i.e., multiple sequential procedure calls may be converted to concurrent RTL modules, if proven not data dependent). On the other hand transformations such as loop unrolling, loop pipelining, and loop fusion can be enabled by user-injected directives as long as data-dependence analysis permits them (Figure 3).

With regard to data storage, AutoPilot distinguishes between two main storage types: on-chip and off-chip. On-chip storage needs to be statically allocated and thus it is suitable for scalar variables (mapped onto FPGA slice registers) and constant size arrays and structures (mapped onto FPGA BRAMs). Off-chip storage can be inferred through C pointers along with corresponding user-injected directives and its size does

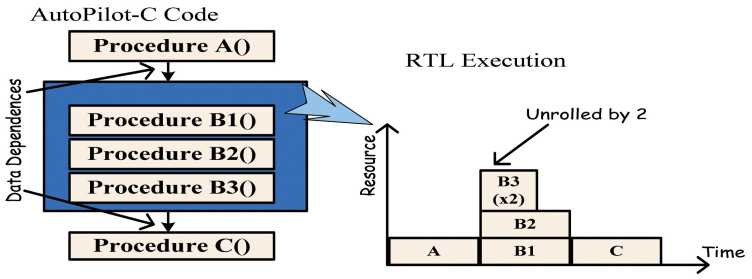


Fig. 3. AutoPilot-C programming model.

not need to be statically defined (it's the programmer responsibility to honor off-chip memory access bounds). In the FPGA platform, the on-chip BRAMs may be organized into multiple independent address spaces. AutoPilot maps each nonscalar variable onto a set of BRAMs (with sufficient aggregate storage capacity) which are only accessible by the PEs that correspond to procedures that reference the nonscalar variable.

3.3. Programming Model Translation Advantages

As compute systems become increasingly heterogeneous with different types of parallel processing accelerators, FCUDA offers an inter-programming model translation tool for efficient kernel portability across GPUs and FPGAs. In this work we facilitate translation of CUDA C into AutoPilot-C. Even though the CUDA programming model is closely linked to the GPU architecture, it offers a good model for programming other platforms such as FPGAs. This conclusion can be reinforced by comparing CUDA with the recently introduced OpenCL [Khronos 2011] parallel programming model which offers a common API for programming heterogeneous devices. OpenCL is based on similar thread group hierarchies, user-exposed memory spaces, and thread synchronization concepts.

The FCUDA flow combines the advantages of the CUDA programming model with the advanced high-level synthesis engine of AutoPilot. The CUDA C programming model provides high abstraction and incurs a low learning curve while enabling parallelism expression in a very concise manner (i.e., enables higher programming productivity compared to AutoPilot-C). FCUDA uses source-to-source transformations and optimizations to convert the CUDA threadblock and thread parallelism into procedure and loop iteration parallelism in AutoPilot's C programming model. By leveraging high-level source-to-source transformations rather than low-level IR translation (e.g., from CUDA's assembly-like PTX IR to RTL), FCUDA can efficiently exploit different levels of coarse-grained parallelism while leveraging existing HLS tools. Furthermore, an important benefit of leveraging CUDA for FPGA programming is the distinction of on-chip and off-chip memory spaces in the CUDA C programming model. This fits well with the memory view within hardware synthesis flows. Finally, the transformations entailed in FCUDA automate the cumbersome task of replication and interconnection of parallel Processing Engines (PEs) along with their associated on-chip memory buffers and the data transfers from/to off-chip memories.

4. FCUDA FRAMEWORK

The proposed CUDA-to-FPGA flow (Figure 1) is based on a Source-to-Source Transformation and Optimization (SSTO) engine which processes the input CUDA code and generates C code for AutoPilot. During the SSTO processing, the implicit parallelism in the SIMT programming model of CUDA is explicitly exposed and restructured to better fit to the reconfigurable fabric. The transformations and optimizations applied on the input code are partially controlled by the parameter values of the user-injected

annotations (Figure 1). Following the FCUDA SSTO phase, AutoPilot extracts fine-grained Instruction-Level Parallelism (ILP) from the transformed code and maps the SSTO-exposed coarse-grained parallelism onto parallel PEs at RTL level. The RTL description is pipelined according to the programmer-specified clock period. Finally, the FCUDA flow leverages the Xilinx ISE logic and physical synthesis tools to map the design onto the reconfigurable fabric. In the following subsections we discuss the overall philosophy of the SSTO engine followed by an overview of the user-injected annotation semantics and their effect in the FCUDA flow.

4.1. CUDA C to Autopilot-C Translation Philosophy

The FCUDA translation aims to: (i) convert the implicit workload hierarchies (i.e., threads and threadblocks) of the CUDA programming model into explicit AutoPilot-C work items, (ii) expose the coarse-grained parallelism and synchronization restrictions of the kernel in the AutoPilot-C programming model, and (iii) generate AutoPilot-C code that can be converted into high-performance RTL implementations. Hence, it is necessary to judiciously map the coarse-grained parallelism (expressed as a hierarchy of thread groups in CUDA) onto the spatial parallelism of the reconfigurable fabric. Moreover, the FCUDA translation implements code restructuring optimizations (e.g., kernel decomposition into compute and data-transfer tasks) to facilitate better performance onto the reconfigurable architecture.

In the CUDA programming model, coarse-grained parallelism is organized in two logical hierarchy levels: threadblocks and threads (Figure 2). The FCUDA SSTO engine converts the logical thread hierarchies into explicit work items. Figure 4 depicts this conversion for the *coulombic potential* (cp) kernel. The logical thread hierarchies in CUDA are expressed via the built-in dim3 vectors (i.e., C structs comprising 3 integer values) *threadIdx* and *blockIdx* (Figure 4(a)). In AutoPilot-C we use for-loop nests to express the workload of these thread-group hierarchies. At grid level, threadblock workloads are materialized through threadblock loops (Figure 4(b)). Similarly, at threadblock level we materialize thread workloads through thread loops (Figure 4(c)). Coarse-grained parallelism can then be exposed by applying loop unroll-and-jam [Aho et al. 2006] transformations on the threadblock and thread loops. Since CUDA threadblocks constitute subtasks that can be executed independently, threadblock-level parallelism is used in FCUDA for the generation of concurrently executing Processing Engines (PEs) on the FPGA. Each PE is statically scheduled to execute a disjoint subset of threadblocks in the kernel grid. On the other hand, threads within a threadblock execute cooperatively (in terms of synchronization and data sharing) to solve the parent threadblock subproblem and their parallelism is leveraged in FCUDA to reduce the PE execution latency. Threads are also scheduled to execute in *warps* of size equal to the thread loop unroll degree. The independent execution of threadblocks facilitates more flexible mapping onto the spatial parallelism, but thread parallelism offers more opportunity for resource sharing. Moreover, the two types of parallelism affect differently the execution frequency of the FPGA implementation. In Section 6 we will discuss the performance impact of the two types of parallelism exposure and we will evaluate different parallelism configurations.

Despite their flexible architecture, FPGAs bear inherent architectural differences with respect to GPUs, which need to be considered during kernel code optimization to achieve optimum performance. For example, FPGAs contain blocks of on-chip memory (BRAMs) which are better suited for use as scratchpad memories rather than caches. In addition, it is preferred to avoid complex hardware for dynamic threadblock context switching as well as dynamic coalescing of concurrent off-chip memory accesses on the FPGA. Hence, the FCUDA SSTO engine implements static coalescing by aggregating all the off-chip accesses into block transfers (provided memory accesses can be

```
1  __constant__ int4 atominfo[MXATMS];
2  #pragma FCUDA GRID x_gdim=32 y_gdim=32 pe=6 cluster=3x3
3  #pragma FCUDA BLOCK x_bdim=16 y_bdim=16
4  #pragma FCUDA SYNC type=sequential
5  #pragma FCUDA COMPUTE name=compute unroll=4 part=2 ptype=block array=energyval
6  #pragma FCUDA TRANSFER name=crd type=stream io=0 global=atominfo size=2000 gsize=40000
7  #pragma FCUDA TRANSFER name=read type=burst io=0  global=energygrid size=16 tlp=y
8  #pragma FCUDA TRANSFER name=write type=burst io=1  global=energygrid size=16 tlp=y
9  __global__ void cenergy(int numatoms, int gridspacing, int * energygrid) {    // Kernel
10   unsigned int xindex  = (blockIdx.x * blockDim.x) + threadIdx.x;
11   unsigned int yindex  = (blockIdx.y * blockDim.y) + threadIdx.y;
12   unsigned int outaddr = (gridDim.x * blockDim.x) * yindex + xindex;
13   int coorx = gridspacing * xindex;
14   int coory = gridspacing * yindex;
15   int atomid;
16   int energyval=0;
17   for (atomid=0; atomid<numatoms; atomid++) {
18     int dx = coorx - atominfo[atomid].x;
19     int dy = coory - atominfo[atomid].y;
20     int r_1 = dx*dx + dy*dy + atominfo[atomid].z;
21     energyval += atominfo[atomid].w * r_1;
22   }
23   energygrid[outaddr] += energyval;
24 }
```

(a) coulombic potential (cp) CUDA kernel

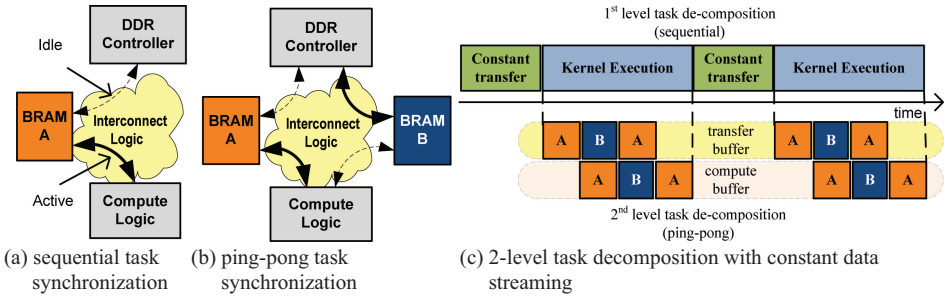
```
1  void cenergy(int numatoms, int gridspacing, int * energygrid, dim3 blockDim, dim3 gridDim) {
2    …
3    for(tbID = clID*peNum; tbID < tbNum; tbID+=clNum*peNum) {   // Threadblock loop
4      for(bIdx = tbID; bIdx< bIdx+peNum; bIdx++) {
5          …
6          cenergy_read(energygrid, outaddr, energygrid_local, ...);          // TRANSFER task call
7          cenergy_compute(energygrid_local, ...);                              // COMPUTE task call
8          cenergy_write(energygrid, outaddr, energygrid_local, ...);         // TRANSFER task call
9    } }
10 }
```

(b) threadblock loop and compute/transfer task outlining in kernel procedure

```
1  void cenergy_read(energygrid, outaddr, energygrid_local) {                  // TRANSFER Task proc.
2    for(threadIdx.y = 0; threadIdx.y < blockDim.y; threadIdx.y++)             // Thread-loop
3      memcpy(energygrid+outaddr, energygrid_local[], ...);                    // Burst
4  }
5
6  void cenergy_compute(energygrid_local, ...) {                               // COMPUTE Task proc.
7    for(threadIdx.y = 0; threadIdx.y < blockDim.y; threadIdx.y++) {           // Thread-loop
8      for(threadIdx.x = 0; threadIdx.x < blockDim.x; threadIdx.x++) {
9        ...
10   }}
11 }
12
13 void cenergy_write(energygrid, outaddr, energygrid_local) {                 // TRANSFER task
14   for(threadIdx.y = 0; threadIdx.y < blockDim.y; threadIdx.y++)             // Thread-loop
15     memcpy(energygrid_local[], energygrid + outaddr, ...);                  // BURST
16 }
```

(c) compute and data-transfer task procedures

Fig. 4. Converting CUDA C to AutoPilot-C.



(a) sequential task synchronization (b) ping-pong task synchronization (c) 2-level task decomposition with constant data streaming

Fig. 5. Task synchronization schemes.

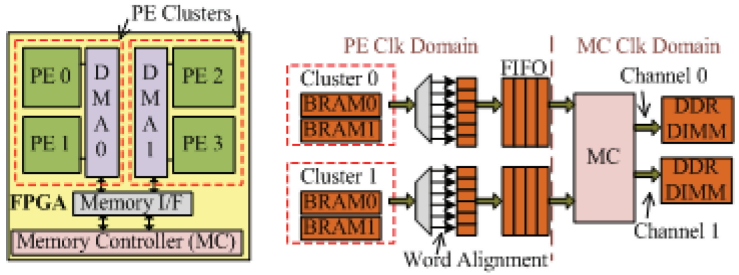
coalesced). In particular, the kernel is decomposed into *compute tasks* and *data-transfer tasks* through procedural abstraction (i.e., the inverse of function inlining) as depicted in Figure 4(b). The off-chip memory accesses are then converted into *memcpy* calls (Figure 4(c)) which are synthesized by AutoPilot into burst transfers facilitating high utilization of the off-chip memory bandwidth. Note that coarse-grained parallelism exposure at grid level through unroll-and-jam of the threadblock loop will result in multiple compute task calls which AutoPilot will translate into concurrent PEs at RTL level (task-level parallelism exposure described in Section 3.2).

The restructuring of the kernel into compute and data-transfer tasks offers potential benefits beyond higher off-chip memory bandwidth utilization. It also enables compute and data-transfer overlap at a coarser granularity (i.e., task granularity) for more efficient kernel execution. By leveraging AutoPilot's procedure-level parallelism the FCUDA SSTO engine can arrange the execution of data-transfer and compute tasks in an overlapped fashion (Figure 5(b)). This implements the *ping-pong task synchronization* scheme at the cost of more BRAM resources (i.e., twice as many BRAMs are utilized). Tasks communicate through double BRAM buffers in a pipelined fashion where the data producing/consuming task interchangeably writes/reads to/from one of the two intermediate BRAM buffers (see lower part of Figure 5(c)). Alternatively, the *sequential task synchronization scheme* (Figure 5(a)) schedules tasks in an interleaving fashion. The latter scheme may be preferred for implementations on FPGAs with low BRAM count or for kernels with very small data-transfer volumes.

4.2. FCUDA Annotation Directives

As discussed in previous sections, the FCUDA SSTO-driven translation leverages programmer-injected annotations that guide the transformations and optimizations applied to the kernel. The annotations comprise pragma directives which specify parameter values for the transformations and optimizations applied to the kernel. A set of annotation directives may be attached to every kernel by insertion before the kernel procedure declaration without affecting compilation of the kernel by other compilers (Figure 4(a)). The Cetus compiler [Lee et al. 2003] has been extended to parse FCUDA annotation directives which are introduced in the rest of this section.

The *SYNC* directive contains the *type* clause which specifies the task synchronization scheme (sequential or ping-pong). *COMPUTE* and *TRANSFER* directives guide—as the names imply—the transformations and optimizations applied on the compute and data-transfer tasks. The *name* clause contains the basic seed used to form the task name (each task name consists of the basic seed along with the kernel procedure name and a PE ID). Other implementation information specified by the *COMPUTE* directive includes degree of thread loop unrolling (*unroll*), degree of array partitioning (*part*), array partitioning scheme (*ptype*), and arrays to be partitioned (*array*). The array



(a) PE clustering (b) custom memory interface

Fig. 6. FCUDA memory interface architecture.

partitioning scheme can be *block* based or *cyclic* based. On the other hand, *TRANSFER* directives specify the direction of the transfer (*io*) and the off-chip memory array pointer (*global*) as well as the block burst size (*size*), the off-chip array size (*gsize*), the thread loop index (*tlp*), and the type of transfer (*type*). The type of a transfer specifies whether the transfer corresponds to a regular burst transfer or a constant streaming transfer (see Section 5.2.3) whereas *tlp* denotes the threadblock dimension to use in the thread loop (see Section 5.1). Each *TRANSFER* directive corresponds to a separate transfer statement in the kernel. The order of the TRANSFER directives needs to match the static order of the transfer statements in the kernel code (transfer statements in callee procedures should be treated as inlined into the kernel code). The *GRID* directive uses clauses *x_gdim* and *y_gdim* to specify the CUDA grid dimensions (*y_gdim* clause is optional and may be omitted for single-dimension grids). Grid dimensions are used during the setup of the threadblock loop's upper bounds. The *GRID* directive also specifies the number of processing engines per cluster (*pe*) along with the clustering scheme (*cluster*). Clustering refers to the physical grouping of PEs on the FPGA die (Figure 6(a)). Each PE cluster contains one or more DMA engines for the communication of its PEs' memories with the memory controller(s). The interface of the FCUDA-generated design to the memory controller(s) can currently be materialized in two ways: (i) a PLB (Processor Local Bus) interconnection scheme supported by AutoPilot- and Xilinx-embedded design tools (EDK) or (ii) a parallel and scalable interconnection scheme which can offer high bandwidth on-chip communication at lower resource cost (Figure 6(b)). Note that the clustering concept can be also used for multi-FPGA implementations where each cluster corresponds to an FPGA device (though this is not currently supported in FCUDA). Finally the *BLOCK* directive declares the threadblock dimensions (*x_bdim*, *y_bdim* and *z_bdim*), through which the programmer can resize the threadblock dimensions and consequently the on-chip buffers' dimensions to better fit on the BRAMs.

5. FCUDA TRANSFORMATION AND OPTIMIZATION ALGORITHMS

In this section we first present an overview of the transformation flow followed by more detailed discussion of some of the key transformations and optimizations.

5.1. FCUDA Translation Overview

A high-level overview of the FCUDA transformation sequence is depicted in Algorithm 1. The first two transformation steps, *constant_memory_streaming()* and *global_memory_buffering()*, map the CUDA memory spaces onto FPGA memories. The *constant_memory_streaming()* step allocates BRAMs for buffering constant memory data and organizes constant data prefetching to the constant data buffers in a streaming fashion. The *global_memory_buffering()* step, on the other hand, handles BRAM

allocation for nonconstant global memory data. The algorithm and implementation details of both memory space mapping functions are described in Section 5.2.

Subsequently, *create_kernel_tasks()* splits the kernel into data-transfer and compute tasks. This pass entails task-statement coalescing code motions and Common-Subexpression-Elimination (CSE) optimizations along with procedural abstraction (i.e., compute/transfer task outlining) transformation (see Section 5.3). The generated compute and data-transfer task procedures are subsequently optimized (lines 6–8 in Algorithm 1) before manipulating the kernel procedure (last three steps in Algorithm 1). First, thread loop generation and fission is performed in the compute tasks. A task-wide nested thread loop of degree equal to the number of nonunit thread dimensions in the *BLOCK* directive (Section 4.2) is wrapped around the compute code (Figure 4(c)). Loop fission is based on the algorithm proposed by Stratton et al. [2008] to enforce correct thread synchronization and functionality in the presence of CUDA *__syncthreads()* directives or other control-flow statements (e.g., loop statements and *break/continue* statements). In a nutshell, loop fission recursively splits the task-wide thread loop into smaller thread loops at the boundaries of control-flow statements. Data-transfer tasks employ thread loop generation in a similar fashion (though, no loop fission is required) with the only difference that the loop nest contains loops only for the threadblock dimensions included in the *tlp* clause of the corresponding *TRANSFER* annotation directive (Figure 4(c)). This way the user can hint across which threadblock dimensions to create burst transfers for hard to analyze nonlinear addressing patterns. Subsequently, unroll-and-jam optimization is applied to thread loops in compute tasks by *unroll_thread_loop()* according to the value of the *unroll* clause in the COMPUTE directive (no unroll is inferred by a unit value). Thread loop unrolling helps expose the thread parallelism in CUDA as long as array accesses do not result in a performance bottleneck due to low BRAM port parallelism. This can be dealt with through array partitioning (cyclic or block based) by *partition_arrays()* which facilitates port parallelism enhancement through array splitting into multiple independent BRAM buffers. The generated array partitions are allocated evenly to the unrolled threads in compute tasks (e.g., for $unroll = 4$ and $part = 2$, two threads share one array partition). Array partitioning is applied in the current framework only when the referenced array elements by each thread can be contained in a cyclic or block partition. The degree of partitioning is controlled by the *part* clause in the *COMPUTE* directive and should

ALGORITHM 1: *FCUDA_Compilation(G_{AST})*

```
    /* Sequence of FCUDA Transformation and Optimization passes on the CUDA abstract
       syntax tree graph, G_AST*/
    K ← CUDA kernels
1   for each kernel ϵ K do
2     constant_memory_streaming()
3     global_memory_buffering ()
4     create_kernel_tasks()
5     create_thread_loop()
6     unroll_thread_loop()
7     partition_arrays()
8     create_threadblock_loop()
9     build_task_synchronization()
10    unroll_threadblock_loop()
11
```

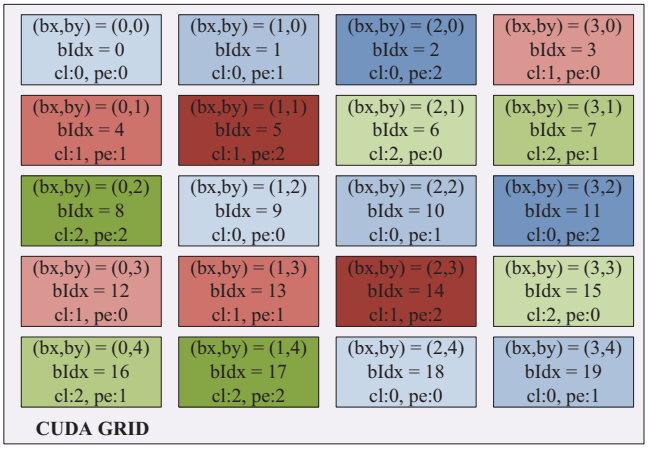


Fig. 7. Threadblock mapping onto PEs and clusters (PE#:3, Cluster#:3).

not exceed the unroll degree (no partition is inferred by a unit value). Partitioning affects transfer tasks, which need to be updated with the partitioned local arrays.

Having completed the manipulation of the compute and transfer tasks, the FCUDA flow works on the kernel procedure by explicitly materializing the grid of threadblocks in the *create_threadblock_loop()* flow step. A blockthread loop is wrapped around the kernel code to materialize the threadblocks for one PE cluster. In the current framework implementation, PE clusters are designed to be identical and are only differentiated by the cluster-ID value *clID* (Figure 4(b)). FCUDA SSTO engine and AutoPilot work on a single cluster which is automatically replicated in the top-level FPGA design (during interconnection with the memory controller interface) to create the total number of PE clusters specified in the *cluster* clause of the *GRID* directive. The threadblock loop determines the scheduling of threadblock execution as well as their mapping onto PEs and clusters. Threadblocks are mapped onto PEs in a block-cyclic fashion (Figrue 7) according to the number of PE clusters (*clNum*), the number of PEs per cluster (*peNum*), and the number of total threadblocks (*tbNum*), where *clNum*, *peNum*, and *tbNum* are calculated based on the values of the *GRID* clauses *cluster*, *pe*, *x_gdim* and *y_gdim*. The block-cyclic mapping is done on a flattened grid of threadblocks where the ID of each threadblock (*bIdx*) is defined as: $bIdx = blockIdx.y^{*}x\_gdim+blockIdx.x$. For kernels with 2D grids, the flattened threadblock ID, *bIdx* (Figure 4(b)) is translated back to the corresponding 2D grid coordinates (i.e., $blockIdx.y = bIdx \text{ floor } x\_gdim$ and $blockIdx.x = bIdx \bmod x\_gdim$) used within the threadblock loop code.

The function *build_task_synchronization()* implements the synchronization of compute and transfer tasks according to the *type* clause of the *SYNC* directive. More details on how the *ping-pong* synchronization scheme is applied are discussed in Section 5.4 (nothing needs to be done for the *sequential* scheme. Finally function *unroll_threadblock_loop()* fully unrolls the inner loop of the threadblock loop nest (facilitated by *bIdx* induction variable in Figure 4(b)) to expose the threadblock parallelism specified by the *pe* clause in the *COMPUTE* directive. Compute and transfer task calls are replicated within the outer threadblock loop and a *blockIdx* vector for each threadblock is calculated and passed to the corresponding task through its parameter list.

5.2. CUDA Memory Space Mapping

As discussed in Section 3.2, CUDA exposes to the programmer different memory spaces: (i) registers, (ii) shared memory, (iii) global memory, (iv) constant memory, and

(v) texture memory. Each memory space has different attributes in terms of access latency and accessibility which affect its usage scenarios and consequently how they are leveraged in FCUDA (texture memory is not currently supported in the FCUDA framework).

5.2.1. CUDA Constant Memory. Constant memory is used to store read-only data that are visible to all threads. In the CUDA architecture a small portion of global memory is reserved for constant data, and access latency is improved through SM-private caches that take advantage of temporal and spatial access locality across all SM threads. On the FPGA, FCUDA facilitates access latency optimization through prefetching constant data on BRAMs. This is achieved by allocating constant on-chip buffer space for each PE (sharing constant buffers across multiple PES would result in long, high fanout interconnections and memory port access bottlenecks) and decoupling constant off-chip memory transfers from the rest of the kernel (Figure 8(a), 8(b)). In particular, a new array is introduced to denote the constant buffer and all constant memory accesses are replaced by local buffer accesses (*atominfo_local*). Furthermore, a new procedure for constant data streaming is created (*cenergy_crd()*) containing a data-transfer task, followed by a kernel compute task. The two tasks may be wrapped in a constant data stream loop (e.g., *strm_count* loop in Figure 8(a)) when the *size* value is a fraction of the *gsize* value in the corresponding *TRANSFER* directive (Figure 4(a)). Thus, through these clauses the programmer can control the constant buffer allocation on the FPGA to: (i) adjust constant array sizes according to the FPGA BRAM sizes, or/and (ii) facilitate multiple kernel calls without multiple invocations from the host CPU (the DDR part allocated for constant data on FPGAs is not limited by the constant cache memory organization as in GPUs). Sequential or ping-pong task synchronization schemes can be also applied on constant data-transfer and kernel tasks independently of the synchronization scheme at the subkernel task level. Figure 5(c) depicts the case where the sequential synchronization is used in the first level (i.e., constant data-transfer and kernel tasks) while the ping-pong scheme is used at the second level (i.e., data-transfer and compute tasks).

5.2.2. CUDA Global Memory. Most of the massively data-parallel CUDA kernels leverage huge input and output datasets that cannot fit in on-chip memory. Significant acceleration of these compute-intensive data-parallel kernels is contingent on efficient data communication between the device and global memory. In the CUDA programming model it is the programmer's responsibility to organize data transfers across warps of threads in a coalesced way in order to achieve maximum off-chip bandwidth utilization. However, warp sizes implemented in FCUDA may be smaller than GPU warp sizes. In addition, hiding off-chip memory latency through fast context switches is not feasible. FCUDA deals with these issues by decoupling global memory references from computation, which facilitates data coalescing across wide blocks of memory. Burst transfers can consequently be generated to achieve high utilization of the off-chip bandwidth (Figure 4(c)).

The first step toward decoupling global memory accesses from computation is described in Algorithm 2. The SSTO first scans the kernel for statements that contain global memory accesses (line 6). Selection of global memory accesses is facilitated by the *TRANSFER* directives through the array names linked to the *global* clause (Figure 4(a)) and the order of the pragma annotations (hence, the programmer can control which of the off-chip accesses need to be handled). Subsequently, the selected statements are filtered with respect to whether they contain compute operations (lines 8–9). If the containing statement describes a simple data transfer without any compute operations, nothing needs to be done. Otherwise, the compute part is disentangled from the data-transfer part by introducing new variables (lines 13, 18), to buffer the output of the computation in on-chip BRAMs (lines 14, 19). Thus the initial statement

```
1 void cenergy_cstream_crd(int numatoms,int atominfo_local[MXATMS],atominfo,strm_count) {
2     memcpy (atominfo_local[0], atominfo+strm_count, numatoms * sizeof(int4));          // Burst
3 }
4 void cenergy_cstream(int numatoms, int totalatoms, int gridspacing, int * energygrid) {
5   int4 atominfo_local0[MXATMS];
6   int4 atominfo_local1[MXATMS];
7 int pingpong=0;
8 cenergy_cstream_crd(numatoms, atominfo_local0, atominfo,strm_count);   // Transfer task (constant)
9   for(int strm_count = 0; strm_count < totalatoms; strm_count+=numatoms) {   // Constant stream loop
10    if(pingpong) {
11      cenergy(numatoms,gridspacing,energygrid,atominfo_local1[MXATMS]);  // Compute task(kernel)
12      cenergy_cstream_crd(numatoms,atominfo_local0,atominfo,strm_count); // Transfer task(constant)
13      pingpong=0; }
14    else {
15      cenergy(numatoms,gridspacing,energygrid,atominfo_local0[MXATMS]);  // Compute task(kernel)
16      cenergy_cstream_crd(numatoms,atominfo_local1,atominfo,strm_count); // Transfer task(constant)
17      pingpong=1; }
18  }}
```

(a) constant data streaming us ing ping-pong task synchronization

```
1  void cenergy(int numatoms, int gridspacing, int * energygrid,int4 atominfo_local[MAXATOMS]) {
2    unsigned int xindex  = (blockIdx.x * blockDim.x) + threadIdx.x;
3    unsigned int yindex  = (blockIdx.y * blockDim.y) + threadIdx.y;
4    unsigned int outaddr = (gridDim.x * blockDim.x) * yindex + xindex;
5    int coorx = gridspacing * xindex;
6    int coory = gridspacing * yindex;
7    int atomid;
8    int energyval[blockDim.y][blockDim.x];
9    energyval[threadIdx.y][threadIdx.x]=0;
10   for (atomid=0; atomid<numatoms; atomid++) {
11     int dx = coorx - atominfo_local[atomid].x;
12     int dy = coory - atominfo_local[atomid].y;
13     int r_1 = dx*dx + dy*dy + atominfo_local[atomid].z;
14     energyval[threadIdx.y][threadIdx.x] += atominfo_local[atomid].w * r_1;  }
15   int energygrid_local[blockDim.y][blockDim.x];
16   energygrid_local[threadIdx.y][threadIdx.x] = energygrid[outaddr];                  // TRANSFER
17   energygrid_local[threadIdx.y][threadIdx.x] += energyval[threadIdx.y][threadIdx.x];
18   energygrid[outaddr] = energygrid_local[threadIdx.y][threadIdx.x];  }               // TRANSFER
```

(b) statement decomposition to compute and transfer operations

```
1  void cenergy(int numatoms, int gridspacing, int * energygrid,int4 atominfo_local[MAXATOMS]) {
2    unsigned int xindex  = (blockIdx.x * blockDim.x) + threadIdx.x;
3    unsigned int yindex  = (blockIdx.y * blockDim.y) + threadIdx.y;
4    unsigned int outaddr = (gridDim.x * blockDim.x) * yindex + xindex;
5    int energygrid_local[blockDim.y][blockDim.x];
6    energygrid_local[threadIdx.y][threadIdx.x] = energygrid[outaddr];                  // TRANSFER
7    int coorx = gridspacing * xindex;
8    int coory = gridspacing * yindex;
9    int atomid;
10   int energyval[blockDim.y][blockDim.x];
11   energyval[threadIdx.y][threadIdx.x]=0;
12   for (atomid=0; atomid<numatoms; atomid++) {
13       int dx = coorx - atominfo_local[atomid].x;
14       int dy = coory - atominfo_local[atomid].y;
15       int r_1 = dx*dx + dy*dy + atominfo_local[atomid].z;
16       energyval[threadIdx.y][threadIdx.x] += atominfo_local[atomid].w * r_1;  }
17   energygrid_local[threadIdx.y][threadIdx.x] += energyval[threadIdx.y][threadIdx.x];
18   energygrid[outaddr] = energygrid_local[threadIdx.y][threadIdx.x];  }               // TRANSFER
```

(c) task statement coalescing

Fig. 8. SSTO transformations/optimizations in coulombic potential (cp) kernel.

ALGORITHM 2: Decouple global memory accesses from compute operations

```
1   /* Processing of global memory accesses to ease kernel decomposition into compute
       and data-transfer
2   tasks; G_AST := Abstract Syntax Tree Graph of CUDA code */
3   HandleGlobalMemoryAccesses(G_AST)
4     V ← list of global variables referenced in TRANSFER directives
5     for each v ∈ V do
6       s ← next statement accessing v
7       M ← {v | v ∈ V and getAccess(s, v)}      /* get all accesses of v in s*/
8       If (|M| > 1) handleStatement(s,M)    /* multiple accesses in statement */
9       else if (not isSimpleTransfer(s)) handleStatement(s,M) /* one access in statement */
10  HandleStatement(s,M)
11      {mL, mR} ← separateLhsRhs(M)      /* label accesses based on RHS/LHS location */
12      for each m ∈ mR do
13        m_local ≡ newVariableOfType (m)
14        ss ≡ newStatement(expression(m_local = m)
15        insertBefore(G_AST , s, ss)
16        replaceExpression(RHS(s), m, m_local)      /* substitute m with m_local in RHS */
17      for each m ∈ mL do
18        m_local ≡ newVariableOfType(m)
19        ss ≡ newStatement(expression(m = m_local))
20        insertAfter(G_AST , s, ss)
21        replaceExpression(LHS(s), m, m_local)      /* substitute m with m_local in LHS */
```

is converted to a simple data-transfer assignment between the global memory variable and the newly introduced on-chip memory variable (lines 16, 21). In the *cp* running example, the statement in line 23 of Figure 4(a) is expanded into the set of statements in lines 15–18 of Figure 8(b) and new on-chip storage, *energygrid_local*, is introduced. Subsequently, kernel decomposition into compute and transfer tasks (described in Section 5.3) can be carried out.

5.2.3. CUDA Registers. Registers are used to store scalar and vector (e.g., int4) variables that are not specified as global, shared, or constant memory variables in the kernel. For each register-stored variable, each thread has a private copy in a register allocated to each thread. In fact, registers are organized in memory-like register files with wide data ports to feed all of the concurrently active threads of a threadblock. Implementing registers as arrays with wide ports can severely impact resource utilization and consequently performance on FPGAs (i.e., arrays are materialized with BRAMs, and high BRAM usage per PE limits the number of instantiated PEs). The flexibility of the reconfigurable fabric is used in FCUDA to exploit register sharing among threads. Threads within the same threadblock may share registers when either all the threads compute the same value for the corresponding variable or the lifetime of the variable does not cross synchronization points or task boundaries and the threads are not active concurrently (e.g., in Figure 8(b) the lifetime of *atomid* does not cross synchronization or task boundaries). Otherwise the variable is vectorized and materialized through BRAM memory allocation. (e.g., *energyval* is live in both compute and transfer statements and gets vectorized in Figure 8(b)). Dataflow analysis is employed in the SSTO engine to identify variables that need to be vectorized. Live variable analysis is used to identify variables with lifetimes spanning

more than one task region. Variables with different values across different threads are identified through reaching *threadIdx*-dependent definition analysis (RTDD). RTDD is similar to definition reaching analysis [Aho et al. 2006] with the definition of *GEN*() function being modified to the following.

$$GEN(n) = \{v | v \epsilon \text{ set of variables defined in node } n, \text{ whose definition is dependent on CUDA threadIdx variable or other variables included in } IN(n)\}$$

5.2.4. CUDA Shared Memory. Shared memory variables are threadblock-private (i.e., only threads within the corresponding threadblock have visibility and access), and thus, can be conveniently translated into BRAM-based memories that are accessible only by the PE executing the corresponding threadblock.

5.3. Kernel Decomposition into Compute and Data-Transfer Tasks

After compute and data-transfer operations are disentangled at the statement level (Figure 8(a)), the SSTO engine uses procedural abstraction to outline groups of contiguous compute statements into compute procedures and contiguous off-chip memory references into transfer procedures. However, interleaving between compute and data-transfer statements (e.g., *energygrid* off-chip accesses in Figure 8(a)) may result in multiple small generated tasks, causing performance inefficiency due to frequent task switching (e.g., high thread loop and synchronization redundancy). The SSTO engine employs task-statement coalescing code motions prior to applying procedural abstraction to avoid fragmentation of the kernel into multiple small tasks.

In particular off-chip memory read references are percolated toward the top of the Control-Flow Diagram (CFG), whereas write statements are percolated toward the bottom of the CFG. Both upward and downward statement percolations shift data-transfer statements until they encounter: (i) another data-transfer statement, (ii) a CUDA synchronization directive, or (iii) a data-dependent compute statement. Upward code percolation is done in forward order of data-transfer statements in the CFG, whereas downward code percolation is done in reverse order. Code percolation may shift statements across entire control-flow constructs (e.g., *energygrid* read statement is shifted across *atomid* loop in Figure 8(c)) as long as there are no data dependences or contained synchronization primitives and the dynamic execution characteristics of the statement do not change (i.e., no statement shifts into or out of control-flow bounds).

Upon completion of task-statement coalescing, Common Subexpression Elimination (CSE) is employed to identify expressions common to all threads that can be computed in the kernel procedure, and procedural abstraction of task regions is performed. Procedural abstraction for data-transfer tasks entails conversion of global memory reads and writes to burst transfers. Bursts are represented by *memcpy()* calls (Figure 4(c)). The conversion process facilitates the values of *size* and *tlp* clauses specified by the programmer in the corresponding TRANSFER directive along with data type and array offset information derived from the actual data-transfer statement to build the *memcpy()* call parameters.

5.4. Task Synchronization

Synchronization of the compute and data-transfer tasks is performed by the SSTO engine according to the type clause of the *SYNC* pragma directive. In the current implementation the available SYNC options are *sequential* and *ping-pong* (Figure 5). The former corresponds to a simple task synchronization scheme that does not require any special code manipulation before HLS. It essentially results in the serialization of all the compute and data-transfer tasks of the kernel. The *ping-pong* option selects the ping-pong task synchronization scheme in which two copies of each local array

Table I. CUDA Kernels

Application – Kernel (Suite)	Data sizes (read/write traffic)	Description
Matrix Multiply – mm (SDK)	2048 × 2048 matrices (4GB/16MB)	Computes multiplication of two 2D arrays (used in many applications)
Fast Walsh Transform – fwt1 (SDK)	8M Points (32MB/32MB)	Walsh-Hadamart transform is a generalized Fourier transformation used in various engineering applications
Fast Walsh Transform – fwt2 (SDK)		
Coulombic Potential – cp (Parboil)	512 × 512 Grid, 40000 Atoms (20.625MB/20MB)	Computation of electrostatic potential in a volume containing charged atoms
Discreet Wavelet Transform – dwt (SDK)	128K points (512KB/512KB)	1D DWT for Haar Wavelet and signals

are declared, doubling the amount of inferred BRAM blocks on the FPGA. Moreover, the task procedure calls are replicated in an *if-else* coding structure controlled by the *ping-pong* variable. This coding template guides AutoPilot to implement concurrent compute and transfer tasks that alternate the BRAM memories they operate upon in successive iterations of the outer loop (Figure 8(c)).

6. EXPERIMENTAL RESULTS

Our experimental study aims to: (i) evaluate the effect in performance of the different parallelism extraction knobs implemented in FCUDA as user-injected pragma directives and (ii) compare the performance and energy characteristics of the FPGA- and GPU-based kernel executions. The selected CUDA kernels come from the Nvidia SDK [NVIDIA 2012a] and the Illinois Parboil [IMPACT 2012] suites. In the experiments discussed in the following sections we focus on integer computation efficiency. Hence, we use modified versions of the kernels that do not entail any floating-point arithmetic. Moreover, we vary the integer arithmetic precision of the modified kernels to evaluate the performance/energy implications of different datapath bitwidths (32 and 16 bit). Details of the benchmarks and the entailed kernels are presented in Table I. The 1st column lists the names of the application, the kernel, and the benchmark suite. The 2nd column contains information about the data array sizes referenced by each kernel, as well as the aggregate DDR read/write traffic (32-bit datapaths). The 3rd column provides a short description of each application. Evaluation of execution latencies on the FPGA is based on multiplication of the measured clock cycle latencies by the clock period achieved during logic and physical synthesis (Xilinx's ISE synthesis tools). Clock cycle latencies are measured on the ML510 development board [XILINX 2012] which hosts the Virtex-5 FX130T device. The FCUDA-generated RTL for each kernel is instrumented with cycle counters that measure the PE computation and communication clock cycles. Using the PowerPC-440 processors that are embedded on the FX130T device we can read the counter values and also verify the functionality of the configured FPGA (i.e., postsynthesis verification). The cycle latencies can be used to accurately calculate the total execution latency for any Virtex-5 FPGA, by considering the PE count and the clock frequency reported by the ISE synthesis flow. The experimental results reported in the following sections are calculated for the Virtex-5 SX240T device which contains a rich and evenly distributed set of BRAM and DSP modules that serve well the compute- and data-intensive CUDA kernels.

6.1. Parallelism Extraction Impact on Performance

In this part of the experimental evaluation we examine how different FCUDA parallelism extraction transformations affect the final execution latency. Note that in FPGA computing latency depends on the combination of three interdependent factors:

Table II. maxP Scheme Implementation Parameters

Parameter	*mm*		*fwt2*		*fwt1*		*cp*		*dwt*	
	32bit	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*
PE	153	162	126	180	144	171	54	99	126	144
unroll	1	1	1	1	1	1	2	4	1	1
part	1	1	1	1	1	1	1	1	1	1
Freq.(MHz)	128	129	125	117	126	162	137	187	135	171
LUT util. (%)	72	65	71	66	85	75	48	51	78	59
Latency(ms)	1570	1371	0.68	0.4	8.26	5.31	3562	2709	0.072	0.057

Table III. maxPxU Scheme Implementation Parameters

Parameter	*mm*		*fwt2*		*fwt1*		*cp*		*dwt*	
	32bit	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*
PE	45	72	45	45	36	45	36	63	18	54
unroll	8	16	8	16	4	4	4	8	16	8
part	2	1	1	1	1	1	1	2	8	2
Freq.(MHz)	156	151	164	138	170	158	147	181	173	137
LUT util. (%)	54	53	71	88	69	63	43	44	71	69

concurrency (e.g., PE count), cycles (e.g., functional unit allocation), and frequency (e.g., interconnection complexity). The FCUDA user can affect these factors through pragma directives: PE count (*pe* clause), unrolling (*unroll* clause), array partitioning (*part* and *array* clauses), task synchronization scheme (*type* clause), and PE clustering (*cluster* clause). Each of these knobs can affect more than one of the performance determining factors. First we explore the effect of threadblock- (i.e., PE count) and thread-level (i.e., unrolling and array partitioning) parallelism on execution latency. Then we discuss the impact of task synchronization schemes in performance. For the following experiments we target the Virtex-5 SX240T device and we group PEs into 9 clusters. PE clusters facilitate higher-frequency interconnections between PEs and memory controllers at the cost of higher interface logic redundancy (Figure 6(a)).

6.1.1. Threadblock- and Thread-Level Parallelism. To evaluate the effects of threadblock-level and thread-level parallelism in performance we compare three parallelism extraction schemes.

—*maxP*. This represents the designs that expose parallelism primarily at the threadblock level, that is, maximize PE count given a resource constraint. Thread-level parallelism may also be extracted if remaining resource is sufficient.
—*maxPxU*. This represents the designs that maximize the total concurrency, that is, the product of PEs and thread loop unroll degree (*pe* × *unroll*). Array partitioning may also be employed if remaining resource is sufficient.
—*maxPxUxM*. This represents the designs that maximize the concurrency along with the on-chip memory bandwidth, that is, the product of PE count, unroll degree, and array partitioning (*pe* × *unroll* × *part*).

Tables II, III, and IV list the design parameters (i.e., *PE*, *unroll*, *part*) selected for all the kernels under the three different parallelism extraction schemes (2^{nd}–4^{th} rows). As expected, the maxP scheme entails high PE counts with almost no thread loop unrolling or array partitioning. Additionally, the maxPxU scheme instantiates fewer PEs but entails high PE× unroll concurrency with almost no array partitioning. Finally, maxPxUxM results in less PEs than the two previous schemes, but facilitates high unroll and partitioning factors achieving maximum PE×unroll×part products. Table rows 5 and 6 contain frequency and LUT utilization results derived from ISE physical

Table IV. maxPxUxM Scheme Implementation Parameters

Parameter	*Mm*		*fwt2*		*fwt1*		*cp*		*dwt*	
	32bit	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*	*32bit*	*16bit*
PE	27	27	27	27	72	90	9	18	36	18
unroll	8	16	8	8	2	2	8	16	8	16
part	8	16	4	8	1	1	8	16	8	16
Freq.(MHz)	154	127	106	104	174	173	183	155	138	123
LUT util. (%)	48	44	78	69	68	45	29	30	76	55

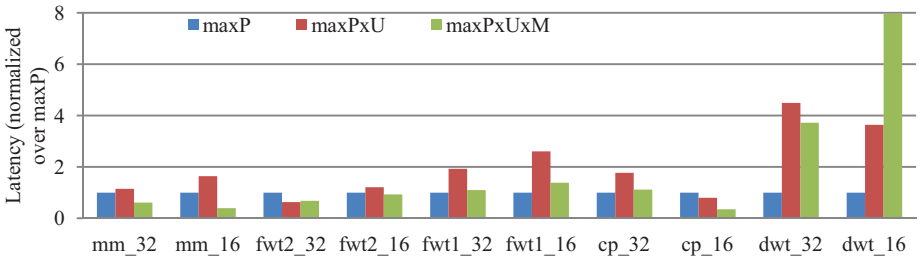


Fig. 9. Performance comparison of parallelism extraction schemes.

synthesis. The LUT utilization can be interpreted as a metric of the logic complexity of each kernel configuration. An upper LUT utilization bound of 85% was applied during configuration selection. Note that LUTs are not the bounding resource in every case (e.g., BRAMs are the parallelism-constraining resource for kernel *cp*). The observed frequency and utilization values show the complex effects of parallelism extraction across different dimensions. FCUDA provides the programmer with a convenient and efficient tool flow for exploring different configurations of kernel parallelism extraction on the FPGA.

Figure 9 depicts the kernel execution latencies for the three schemes, normalized against the latencies of the maxP scheme (Table II row 7). We can observe that no single scheme achieves best performance across all the kernels due to their diverse characteristics. The maxP scheme provides the lowest (=best) latency results for kernels *fwt1* and *dwt*. This can be attributed to the fact that these two kernels contain complicated array access patterns which inhibit array partition for most of their arrays (i.e., partitioning degrees in *dwt* refer to arrays that represent a very small percentage of array accesses). High unrolling degrees in the maxPxU scheme are not efficient without array partitioning (i.e., thread-level parallelism extraction is bottlenecked by the fixed on-chip memory bandwidth). On the other hand, the balance between unrolling and array partitioning achieved in the maxPxUxM scheme offers higher performance for half of the kernels.

6.1.2. Compute and Data-Transfer Task Parallelism. To evaluate the effect of the ping-pong task synchronization scheme on performance we use the *MM* kernel, which entails a loop comprising a data-transfer task and a compute task. Through ping-pong task synchronization it is possible to overlap compute and data-transfer tasks. However, the efficiency of task concurrency comes at the cost of higher resource utilization and lower execution frequencies, due to more complex interconnection between compute logic and BRAM buffers (ping-pong synchronization leads to 27% clock frequency degradation compared to sequential). Figure 10 compares the execution latency between sequential (*seq*) and ping-pong (*pp*) task synchronization for three different off-chip memory bandwidth scenarios: (i) BW1, which corresponds to a low off-chip bandwidth and makes the pp-based execution bound by data-transfer latency, (ii) BW2, which is close to the bandwidth required to achieve an equilibrium between compute and data-transfer

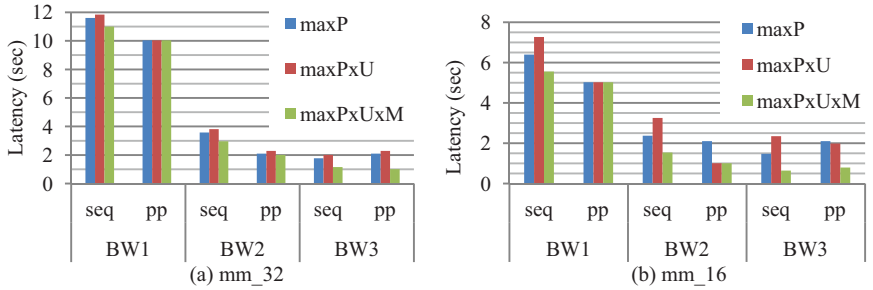


Fig. 10. Task synchronization evaluation.

latencies, and (iii) BW3, which provides 10X higher bandwidth than BW2 and facilitates smaller data-transfer latencies compared to compute latencies. We can observe that for both versions of the MM kernel (32 and 16 bit), pp synchronization provides better execution latency for lower bandwidth values, BW1 and BW2. However, for higher bandwidths (BW3) the sequential synchronization scheme achieves faster execution. In conclusion, pp synchronization is useful for kernels that are data communication bound whereas compute bound kernels will most likely gain from the sequential synchronization scheme.

6.2. FPGA vs. GPU

In this set of experiments we evaluate the performance of the FCUDA-generated FPGA accelerator against the GPU execution. For a fair comparison the devices we use are fabricated with the same process technology (65nm) and provide comparable transistor capacities: Nvidia G92 (128 Stream Processors (SPs) running at 1500 MHz and 64GB/sec peak off-chip memory bandwidth) and Xilinx Virtex-5-SX240T (1056 DSPs, 1032 BRAMs with aggregate capacity of roughly 2MB). Note that the BRAM memory on the FPGA is used for storing: (i) input data read from off-chip memory, (ii) intermediate data across synchronization points, and (iii) output data written back to off-chip memory. The total required on-chip data storage depends on the array dimensions of the input, output, and intermediate data as well as the tiling of the input/output data containers across threadblocks. For some kernels, such as matrix-multiply (*mm*), the same input data tile may be referenced by multiple threadblocks; hence it may be stored on different BRAMs. Currently, it is the programmer's responsibility to choose appropriate threadblock sizes and partitioning degrees to enable efficient BRAM utilization.

One of the most critical performance factors is data communication bandwidth, especially for off-chip memory transfers. In our experimental study we leverage the scalable data communication scheme depicted in Figure 6(b). The Memory Controller (MC) and the PEs run on different clock domains connected through dual-clock FIFO buffers. Leveraging the parallelism of PEs and clusters we can control transfer bandwidths by scaling the number of DDR2 channels/MCs (similar to the architecture in Convey [2011], which achieves 80GB/sec with 16 channels). We compare execution latencies for two different off-chip transfer bandwidths: 16 and 64GB/sec. The lower value represents a realistic off-chip bandwidth value for single-FPGA systems (e.g., NALLATECH [2012]), whereas the higher value may be realized on multi-FPGA systems (e.g., Convey [2011]) and offers the opportunity to compare the compute efficiency of the two devices for equal off-chip memory bandwidths. Consideration of communication latency in the overall execution latency is achieved by scaling the off-chip communication latency measured on the development board [XILINX 2012] according to the ratio of the target system's bandwidth over the ML510 board

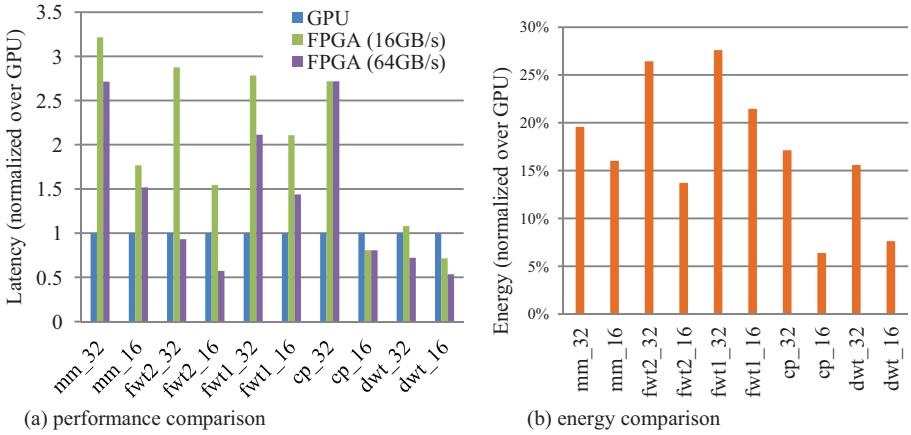


Fig. 11. FPGA vs. GPU.

bandwidth (since off-chip transfers are organized into threadblock burst transfers, it is easy to calculate total execution latency by combining compute and transfer latencies).

Figure 11(a) depicts the execution latencies of the FCUDA-generated accelerators, normalized over the GPU execution latencies. Note that GPU execution latencies are based on 32-bit kernels. This ensures best performance on the 32-bit GPU architecture (e.g., GPU performance is 8.7X and 3.5X lower for 16-bit fwt2 and fwt1 kernels, compared to 32-bit kernels) and also provides a common reference for comparing the 32-bit and 16-bit FPGA implementations. The FPGA execution latencies compared in Figure 11(a) are based on the best-performing parallelism extraction scheme (maxP, maxPxU, or maxPxUxM) for each kernel. Note that higher FPGA performance may be possible through further tuning of the FCUDA parallelism extraction knobs based on frequency and cycle information feedback from synthesis.

From the chart in Figure 11(a) we observe that 16-bit kernel versions have lower latency than 32-bit ones. This is expected and it is due to higher concurrency (fewer resources per operation) and lower cycle counts (shorter critical paths and fewer cycles per operation) feasible with narrower datapaths. With regard to off-chip bandwidth (BW), comparison of the 2nd and 3rd bars in Figure 11(a) shows that high off-chip BW is extremely important for these massively data-parallel kernels. Interestingly the *cp* kernel seems to be insensitive to off-chip BW. The main reason for this is that its compute-to-communication ratio is much higher than the other kernels. Useful conclusions can be drawn by comparing the 1st and 3rd bars in Figure 11(a). In particular we can observe that the FPGA integer compute latency is highly competitive to the compute latency of the GPU, especially considering the big difference in clock speed (GPU SPs run at 1500 MHz, i.e., 10X faster than the FPGA, while the rest of the GPU runs at 600 MHz, i.e., 4X faster than the FPGA). In addition, the compute characteristics of the kernels affect the relative performance of the FPGA implementation (bar 3) with respect to GPU (bar 1). For example *mm_32* and *cp_32* are multiply intensive with each 32-bit multiply based on 3 interconnected DSP multipliers. This results in resource pressure for DSPs, while rendering the multiplier datapaths susceptible to routing congestion. On the other hand, kernels like fwt2 that do not need DSPs for wide bitwidth operations (fwt2 contains only 32-bit addition which is implemented with LUTs) and contain control flow that facilitates high ILP extraction during HLS offer superior performance on the FPGA. Finally, high compute-to-communication ratio is extremely important in FPGA-based computing for two reasons: (i) higher thread

ILP extraction within each PE offsets lower clock frequency, (ii) static scheduling of threadblock execution does not allow off-chip communication latency hiding (as enabled by quick context switching in GPUs). This can be confirmed by the *cp_16* kernel which is highly compute intensive and, unlike *cp-32*, does not contain 32-bit multiplies (Virtex-5 DSPs do not provide efficient 32-bit multiplication).

Figure 11(b) compares energy consumption between the same pair of FPGA and GPU devices. We use the Xilinx Power estimator to estimate energy consumption on the FPGA device, whereas GPU energy values are based on reported average power dissipation (i.e., ∼100W). As depicted in Figure 11(b), execution on the FPGA is significantly more energy efficient, ranging between 6% and 27% of the corresponding GPU execution energy for the chosen set of kernels.

7. CONCLUSION AND FUTURE DIRECTIONS

In this article we present FCUDA, a new FPGA programming tool which offers an efficient flow for accelerating massively parallel CUDA kernels on reconfigurable hardware. FCUDA combines source-code transformations and optimizations with High-Level Synthesis (HLS) to enable FPGA programming with a programming model that offers high abstraction and concise parallelism expression. Moreover, FCUDA eliminates significant coding effort during application porting between heterogeneous acceleration devices, such as GPUs and FPGAs. The programmer can inject simple pragma annotations into the code to efficiently specify how coarser-grained parallelism across different granularities may be exposed in the generated RTL. A Source-to-Source Transformation and Optimization (SSTO) engine then transforms the code according to the programmer's annotations while leveraging compute and data-transfer task decoupling for high-performance application-specific acceleration. Our experimental study has helped demonstrate the performance trade-offs that the programmer needs to consider when deciding how to extract parallelism at different granularities within inherently parallel kernels. Moreover, we demonstrated that FPGA execution of integer kernels processed through the FCUDA flow is highly competitive with execution on the GPU. FCUDA provides a useful tool for exploring the impact of different parallelism extraction decisions on the reconfigurable hardware performance. We believe that FCUDA is an important step toward a new set of tools that enable efficient acceleration on configurable hardware by combining platform-specific code transformations and optimizations in tandem with HLS-based translation of high-level language code to RTL descriptions. Moreover, tools like FCUDA are essential for democratizing the performance, power, and cost benefits of heterogeneous compute systems.

As we discussed in previous sections, there are several advantages of programming FPGAs at the abstraction of the CUDA programming model. Nevertheless, as CUDA is designed to target GPU architecture, some of its features may not map optimally on the reconfigurable fabric in terms of performance. In this work we have used code transformations in tandem with user annotations to deal with several of them (e.g., static memory access coalescing, customization of multigranularity parallelism exposure onto FPGA resources, array partitioning to deal with BRAM access bandwidth bottlenecks, etc.). One of the inherent characteristics of the GPU architecture reflected in the CUDA model is the data and synchronization independence between threadblocks. Hence, data shared between concurrently executing threadblocks needs to be separately transferred for each threadblock. This has a significant impact on the FPGA execution latency. Furthermore, inter-block synchronization can only be achieved via kernel invocation (i.e., splitting computation into multiple kernels) requiring redundant storing/loading of data to off-chip memory across kernels. The architecture flexibility offered by FPGAs can be better exploited to organize computation so as to avoid

off-chip transfers via on-chip data sharing and inter-PE data streaming. Our future work includes integration of new and more aggressive code transformations to facilitate: (i) higher data sharing between PEs of a single kernel and (ii) data streaming between PEs of data-dependent kernels.

Even though our experimental study did not include kernels with Floating-Point (FP) arithmetic, the FCUDA flow supports FP computation through a library of Xilinx floating-point IP cores available within AutoPilot. Nevertheless, the efficiency of FP arithmetic on the FPGA is relatively low compared to integer arithmetic. For example, each PE of the floating-point *mm* version takes 2.9X more LUTs and 1.67X more DSPs than the corresponding integer version. Additionally, the cycle latency of the FP version is 2.66X higher than the integer version. Recent reconfigurable computing advances [Parker 2011] show promise for efficient FP arithmetic on FPGAs. We plan to explore possible ways of leveraging such advanced FP compilation techniques in FCUDA.

Supporting OpenCL is also included in our future work plans. Our current CUDA-based flow can be relatively easily extended to support the programming extensions introduced in OpenCL. As the core OpenCL parallelism organization resembles CUDA's SIMT model, we will extend our current framework to achieve high-performance OpenCL kernel execution on FPGAs (note that previous efforts in OpenCL-to-FPGA flows do not address high performance). In addition, supporting OpenCL onto FPGAs requires implementation of the host runtime API for queuing and executing kernels on FPGA device coprocessors (only partially implemented in previous efforts).

REFERENCES

AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers, Principles, Techniques and Tools,* 2nd ed. Addison-Wesley.

ALLEN, R. AND KENNEDY, K. 2002. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, Academic Press.

AMD. 2012. Accelerated processing units. http://www.amd.com/us/products/technologies/fusion/Pages/fusion.aspx.

BDTI. 2010. An independent evaluation of: The autoesl autopilot high-level synthesis tool. http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf.

CHE, S., LI, J., SHEAFFER, J. W., SKADRON, K., AND LACH, J. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the 6th Symposium on Application Specific Processors*. IEEE, 101–107.

CHEN, D., CONG, J., FAN, Y., HAN, G., JIANG, W., AND ZHANG Z. 2005. XPilot: A platform-based behavioral synthesis system. In *Proceedings of the TechCon Conference*.

CHO, J., MIRZAEI, S., OBERG, J., AND KASTNER, R. 2009. Fpga-based face detection system using haar classifiers. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. ACM Press, New York, 103–112.

CHREC. 2012.NSF center for high performance reconfigurable computing. http://www.chrec.org/facilities.html.

CONG, J., LIU, B., NEUENDORFFER, S., NOGUERA, J., VISSERS, K., AND ZHANG, Z. 2011. High-level synthesis for FPGA: From prototyping to deployment. *Comput. Aid. Des. Integr. Circ. Syst. 30*, 4, 473–491.

CONG, J. AND ZOU, Y. 2008. Lithographic aerial image simulation with FPGA-based hardware acceleration. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*. ACM Press, New York.

CONVEY COMPUTER. 2011. http://www.conveycomputer.com.

DINIZ, P., HALL, M., PARK, J., SO, B., AND ZIEGLER, H. 2005. Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis system. *Microprocess. Microsyst. 29,* 2–3, 51–62.

GAJSKI, D. 2003. NISC: The ultimate reconfigurable component.Tech. rep. 03-28. Center for Embedded Computer Systems, UCI. http://www.cecs.uci.edu/technical_report/TR03-28.pdf.

GUPTA, S., GUPTA, R. K., DUTT, N. D., AND NICOLAU, A. 2004. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst. 9*, 4, 441–470.

He, C., Papakonstantinou, A., and Chen, D. 2009. A novel soc architecture on fpga for ultra fast face detection. In *Proceedings of the 27th International Conference on Computer Design*. IEEE, 412–418.

Huang, S. S., Hormati, A., Bacon, D. F., and Rabbah, R. 2008. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*. Springer, 76–103.

IBM. 2006. The cell architecture. http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html.

Impact. 2012. Parboil benchmarks. http://impact.crhc.illinois.edu/parboil.aspx.

Impulse. 2003. Impulse accelerated technologies inc. http://www.impulseaccelerated.com.

Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., and Rabbah, R. 2008. Optimus: Efficient realization of streaming applications on FPGAs. In *Proceedings of Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM Press, New York, 41–50.

Khronos. 2011. OpenCL specification, version 1.1. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

Lee, S., Johnson, T. A., and Eigenmann, R. 2003. Cetus - An extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*. Springer, 539–553.

Lin, M., Lebedev, I., and Wawrzynek, J. 2010. OpenRCL: Low-power high-performance computing with reconfigurable devices. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 458–463.

Ling, L., Oliver, N., Bhushan, C., Qigang, W., Chen, A., et al. 2009. High-performance, energy-efficient platforms using in-socket fpga accelerators. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. ACM Press, New York, 61–264.

LLVM. 2007. The LLVM compiler infrastructure. http://www.llvm.org.

Mentor Graphics. 2012. Catapult C synthesis overview. http://www.mentor.com/esl/catapult/overview/.

Nallatech. 2012. DATA v5. http://www.nallatech.com/Modules/data-v5-xilinx-virtex-5-fpga-ddr2-sdramqdr-ii-sram-and-io-module.html.

Nvidia. 2012a. CUDA developer zone. http://developer.nvidia.com/category/zone/cuda-zone.

Nvidia. 2012b. GeForce 8 series. http://www.nvidia.com/page/geforce8.html.

Owaida, M., Bellas, N., Daloukas, K., and Antonopoulos, C. 2011. Synthesis of platform architectures from opencl programs. In *Proceedings of the 19th Symposium on Field-Programmable Custom Computing Machines*. IEEE, 178–185.

Parker, M. 2011. Hardware-based floating-point design flow. In *Proceedings of the DesignCon Conference*.

Showerman, M., Enos, J., Kidratenko, C., Steffer, C., Pennington, R., and Hwu, W. W. 2009. QP: A heterogeneous multi-accelerator cluster. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*.

Stratton, J. A., Stone, S. S., and Hwu, W. W. 2008. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 21st International Conference on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 5335, Springer, 16–30.

Thomas, D. B., Howes, L., and Luk, W. 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. ACM Press, New York, 63–72.

Tilera. 2012. Tilera corporation. http://www.tilera.com.

Williams, J., Richardson, J., Gosrani, K., and Suresh, S. 2008. Computational density of fixed and reconfigurable multi-core devices for application acceleration. In *Proceedings of the 4th Annual Reconfigurable Systems Summer Institute*.

Xilinx. 2012. Virtex-5 FXT ML510 embedded development platform. http://www.xilinx.com/products/boards-and-kits/HW-V5-ML510-G.htm.

Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., and Cong, J. 2008. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds., Springer, 99–112.

Received March 2011; revised February 2012; accepted August 2012