FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs[*]

Alexandros Papakonstantinou[1], Karthik Gururaj[2], John A. Stratton[1], Deming Chen[1], Jason Cong[2], Wen-Mei W. Hwu[1]

[1]Electrical & Computer Eng. Dept., University of Illinois, Urbana-Champaign, IL, USA
[2]Computer Science Dept., University of California, Los-Angeles, CA, USA
[1]{apapako2, stratton, dchen, w-hwu}@illinois.edu,   [2]{karthikg, cong}@cs.ucla.edu

Abstract— **As growing power dissipation and thermal effects disrupted the rising clock frequency trend and threatened to annul Moore's law, the computing industry has switched its route to higher performance through parallel processing. The rise of multi-core systems in all domains of computing has opened the door to heterogeneous multi-processors, where processors of different compute characteristics can be combined to effectively boost the performance per watt of different application kernels. GPUs and FPGAs are becoming very popular in PC-based heterogeneous systems for speeding up compute intensive kernels of scientific, imaging and simulation applications. GPUs can execute hundreds of concurrent threads, while FPGAs provide customized concurrency for highly parallel kernels. However, exploiting the parallelism available in these applications is currently not a push-button task. Often the programmer has to expose the application's fine and coarse grained parallelism by using special APIs. CUDA is such a parallel-computing API that is driven by the GPU industry and is gaining significant popularity. In this work, we adapt 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 which enables high-abstraction FPGA programming. FCUDA is based on a source-to-source compilation that transforms the SPMD CUDA thread blocks into 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 multi-core 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.**

I. INTRODUCTION

Even though parallel processing has been a major contributor to application speedups achieved by the high performance computing community, its adoption in mainstream computing domains has lagged due to the relative simplicity of enhancing application speed through frequency scaling and transistor shrinking. However, the power wall encountered by traditional single-core processors has forced a global industry shift to the multi-core paradigm. As a consequence of the rapidly growing interest for parallelism in a wider and coarser level than feasible in traditional processors, the potential of GPUs and FPGAs has been realized. GPUs consist of hundreds of processing cores clustered within streaming multiprocessors (SMs) that can handle intensive compute loads with high-degree of data-level parallelism. FPGAs on the other hand, offer efficient application-specific parallelism extraction through the flexibility of their reconfigurable fabric. Besides, heterogeneity in high performance computing (HPC) has been gaining great momentum as can be inferred by the proliferation of heterogeneous multiprocessors ranging from Multi-Processor Systems on Chip (MPSoC) like the IBM Cell [21], to HPC clusters with GPU/FPGA accelerated nodes such as the NCSA AC Cluster [20]. The diverse characteristics of these compute cores/platforms render them optimal for different types of application kernels. Currently, the performance and power advantages of the heterogeneous multi-processors are offset by the difficulty involved in their programming. Moreover, the use of different parallel programming models in these heterogeneous compute systems often complicates the development process. In the case of kernel acceleration on FPGAs, the programming effort is further inflated by the need to interface with hardware at the RTL level.

A significant milestone towards the use of the massively parallel computing power of GPUs in non-graphics applications has been the release of CUDA by NVIDIA. CUDA enables general purpose computing on the GPU (GPGPU) through a C-like API which is gaining considerable popularity. In this work we explore the use of CUDA as the programming interface for a new FPGA programming flow (Fig. 1), which is designed to efficiently map the coarse and fine grained parallelism expressed in CUDA kernels onto the reconfigurable fabric. Our CUDA-to-FPGA flow employs the state of the art high-level synthesis tool, AutoPilot [5], which enables high-abstraction FPGA programming. The flow is enabled by a source-to-source compilation phase, FCUDA, which transforms the SPMD (Single-Program-Multiple-Data) CUDA code into C code for AutoPilot with annotated coarse-grained parallelism. AutoPilot maps the annotated parallelism onto parallel cores ("core" in this context is an application-specific processing engine) and generates a corresponding RTL description which is subsequently synthesized and downloaded onto the FPGA.

The selection of CUDA as the programming interface for our FPGA programming flow offers three main advantages.
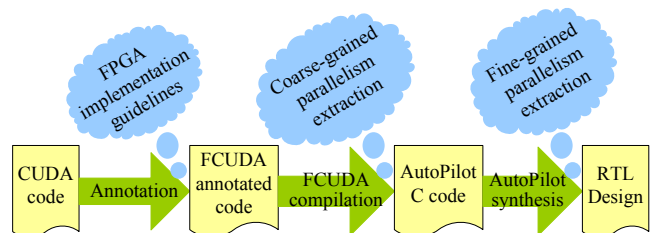
Fig. 1. CUDA-to-FPGA Flow

First, it provides a high-level API for expressing coarse grained parallelism in a concise fashion within application kernels that are going to be executed on a massively parallel acceleration device. Even though CUDA is driven by the GPU computing domain, we show that CUDA kernels can indeed be translated with FCUDA into efficient, customized multi-core compute engines on the FPGA. Second, it bridges the programmability gap between homogeneous and heterogeneous platforms by providing a common programming model for clusters with nodes that include GPUs and FPGAs. This simplifies application development and enables efficient evaluation of alternative kernel mappings onto the heterogeneous acceleration devices without time-consuming kernel code re-writing. Third, the wide adoption of the CUDA programming model and its popularity render a large body of existing applications available to FPGA acceleration.

In the next section we discuss important characteristics of the FPGA and GPU platforms along with previous related work. Section III explains 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 section IV, while section V displays experimental results and shows that our high-level synthesis based flow can efficiently exploit the computational resources of top-tier FPGAs in a customized fashion. Finally, section VI concludes the paper and discusses future work.

II. THE FPGA PLATFORM

With increasing transistor densities, the computational capabilities of commercial FPGAs provided by Xilinx [16] and Altera [17] have greatly increased. Modern FPGAs are technologically in sync with the rest of the IC industry by employing the latest manufacturing process technologies and supporting high-bandwidth IO interfaces such as PCIe, Intel's FSB [6] and AMD's HyperTransport [8]. By embedding fast DSP macros, memory blocks and 32-bit microprocessor cores into the reconfigurable fabric, a complete SoC platform is available for applications which require high-throughput computation at a low power footprint.

The flexibility of the reconfigurable fabric provides a versatile platform for leveraging different types of application-specific parallelism: i) coarse- and fine-grained, ii) data- and task-level and iii) different pipelined configurations. Re-configurability, though, has an impact in the clock frequency achievable on the FPGA platform. Synthesis-generated wire-based communication between parallel modules may limit the throughput of designs with wider parallelism compared to smaller but faster clocked architectures. In our flow we leverage the CUDA programming model to build multi-core acceleration designs with low count of inter-core communication interconnect.

FPGA devices reportedly offer a significant advantage (4X-12X) in power consumption over GPUs. J. Williams et al. [1] showed that the computational density per Watt in FPGAs is much higher than in GPUs. This is even true for 32-bit integer and floating-point arithmetic (6X and 2X respectively), for which the raw computational density of GPUs is higher.

A. Application Domains

FPGAs have been employed in the implementation of different projects for the acceleration of compute intensive applications. Examples range from data parallel kernels [11, 13] to entire applications such as face detection [9]. Although they allow flexible customization of the architecture to the application, the physical constraints of their configurable fabric favor certain kernels over others, in terms of performance. In particular, J. Williams [1] describes 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 in wider bidwidths, such as 32-bit integer and single-precision floating-point operations (0.98X and 0.34X respectively). The performance degradation at large bitwidths comes from the utilization of extra DSP units per operation which results in limited parallelism. Floating-point arithmetic implementation on FPGA is inefficient for the same reason [12]. Often, a careful decision among alternative algorithms is necessary for optimal performance [7].

B. Programmability

Programming FPGAs often requires hardware design expertise, as it involves interfacing with the hardware at the RTL level. However, the advent of several academic and commercial Electronic System Level (ESL) design tools [2-5, 22-23] for High-Level Synthesis (HLS) has raised the level of abstraction in FPGA design. Most of these tools use high-level languages (HLLs) as their programming interface. Some of the earlier HLS tools [2, 3] can only extract fine grained parallelism at the operation level by using data dependence analysis techniques. Extraction of coarse grained parallelism is usually much harder in traditional HLLs that are designed to express sequential execution. To overcome this obstacle, some HLS tools [4, 5, 22] have resorted to employing language extensions for allowing the programmers to explicitly annotate coarse grained parallelism in the form of parallel streams [4], tasks [5] or object-oriented structures [22]. In a different approach, special high-level languages that model parallelism with streaming dataflows have been employed in HLS tools [23]. In this work we use the popular CUDA programming model to concisely express the coarse level parallelism of compute intensive kernels. CUDA kernels are then efficiently translated into AutoPilot input code with annotated coarse grained parallelism, as discussed in the following sections.

III. DETAILS OF PROGRAMMING MODELS

A. CUDA

The CUDA programming model exposes parallelism through a data-parallel SPMD kernel function. Each kernel implicitly describes multiple CUDA threads that are organized in groups called *thread-blocks*. Thread-blocks are further organized into a *grid* structure (Fig 2). Threads within a thread-block are executed by the streaming processors (SPs) of a single GPU streaming multiprocessor (SM) and are allowed to synchronize and share data through the SM *shared memory*. On the other hand, synchronization of thread-blocks is not supported. Thread-block threads are launched in SIMD bundles called *warps*. Warps consisting of threads with highly diverse control flow will result in low performance execution. Thus,
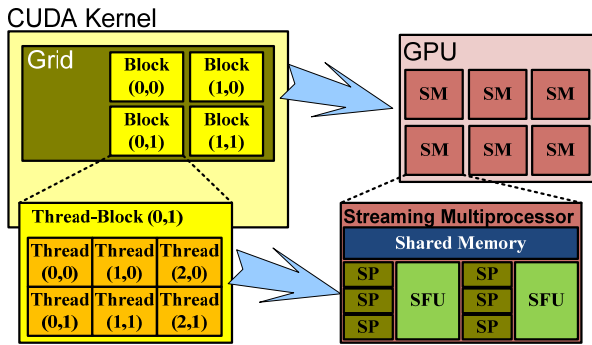
Fig. 2. CUDA Programming Model

Fig. 3. AutoPilot C Programming Model

for successful GPU acceleration it is critical that threads are organized in warps based on their control flow characteristics.

The CUDA memory model leverages separate memory spaces with diverse characteristics. *Shared* memory refers to on-chip SRAM blocks, with each block being accessible by a single SM (Fig. 2). *Global* memory, on the other hand, is the off-chip DRAM that is accessible by all SMs. Shared memory is fast but small, whereas global memory is long-latency but abundant. There are also two read-only off-chip memory spaces, *constant* and *texture*, which are cached and provide special features for kernels executed on the GPU. More details on the CUDA memory spaces are provided in section IV.

B. AutoPilot C

AutoPilot's programming model conforms to a subset of C which may be annotated with pragmas that convey information on different implementation details. Synthesis is performed at the function level producing corresponding RTL descriptions for each function. The RTL description of each function corresponds to an *FPGA core* (Fig. 3) which consists of private datapath and FSM-based control logic. Attached to each core's FSM are *start* and *done signals* that enable cross-function synchronization (including function calls and returns).

The front-end engine of AutoPilot (based on the LLVM compiler [18]) uses dependence analysis techniques to extract instruction-level parallelism (ILP) within basic blocks. Coarser parallelism, such as loop iteration parallelism, can also be exploited by injecting *AUTOPILOT UNROLL* pragmas in the code (assuming there are no loop-carried dependencies). Note that unrolling and executing loop iterations in parallel, impacts FPGA resource allocation proportionally to the unroll factor. Concurrency at the function level is specified by the *AUTOPILOT PARALLEL* pragma within a code region (Fig 3). The affected functions are launched concurrently by the parent function, which stalls executing until every child function has returned. Thus it is possible to implement an MPMD (Multi Program Multi Data) execution model with a configuration of heterogeneous FPGA cores (i.e. parallel cores corresponding to different functions). Note that AutoPilot will schedule two functions (cores) to execute in parallel only when they cause no hazards. A hazard arises when two functions access the same memory block (resource hazard) or pass data from one function to another (data hazard).

With regards to memory spaces, AutoPilot may map variables onto local (on-chip) or external (off-chip) memories. By default all arrays get mapped onto local BRAMs while scalar variables are mapped on configurable fabric logic. C
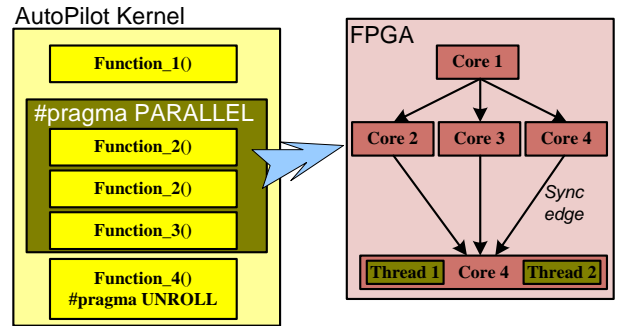
pointers may also be used (with some limitations) in the input code and combined with the *AUTOPILOT INTERFACE* pragma they can infer off-chip memory accesses.

C. CUDA-to-FPGA Flow Advantages

The advantages offered by the CUDA programming model in our FPGA design flow are multifold. First, both CUDA and AutoPilot's programming model are based on the C language. CUDA extends the language with some GPU specific constructs while AutoPilot uses a subset of C, augmented with synthesis pragma annotations (ignored during gcc compilation). Thus, FCUDA source-to-source compilation does not require translation between fundamentally different languages. Second, even though CUDA incorporates more memory spaces than AutoPilot, they both distinguish between on-chip and off-chip memory spaces, and leverage programmer-specified data transfers between off- and on-chip memory storage.

Coarse-grained parallelism in CUDA is expressed in the form of thread-blocks that execute independently on the SMs. Moreover, the number of thread-blocks in CUDA kernels is typically in the order of hundreds or thousands. Thus, thread-blocks constitute an excellent candidate in terms of lack of synchronization requirements and workload granularity for FPGA core implementation. Mapping thread-blocks onto parallel cores on the FPGA minimizes inter-core communication without limiting parallelism extraction. Low inter-core communication helps achieve higher execution frequencies and eliminate synchronization overhead. As a final point, CUDA provides a very concise programming model for expressing coarse-grained parallelism through the single-thread kernel model. AutoPilot (as most existing HLS tools), on the other hand, employs a programming model that expresses coarse grained parallelism explicitly in the form of multiple function calls annotated with appropriate pragmas (Fig. 3). FCUDA automates the extraction of the inferred parallelism in CUDA code into explicit parallelism in AutoPilot input code while handling data partitioning and FPGA core synchronization. Thus, it eliminates the tedious and error-prone task of directly expressing the coarse grained parallelism in C for AutoPilot. Our FPGA design flow allows the programmer to describe the parallelism in a more compact and efficient way through the CUDA programming model regardless of the implemented number of FPGA cores.

IV. FCUDA: CUDA-TO-FPGA FLOW

Our CUDA-to-FPGA flow (Fig. 1) is based on a code transformation process, FCUDA (currently targeting the AutoPilot HLS tool), which is guided by preprocessor

directives (FCUDA pragmas) inserted by the FPGA programmer into the CUDA kernel. These directives control the FCUDA translation of the expressed parallelism in CUDA code into explicitly-expressed coarse-grained parallelism in the generated AutoPilot code. The FCUDA pragmas describe various FPGA implementation dimensions which include the number, type and granularity of tasks, the type of task synchronization and scheduling, and the data storage within on- and off-chip memories. AutoPilot subsequently maps the FCUDA specified tasks onto concurrent cores and generates the corresponding RTL description. Moreover, AutoPilot uses LLVM's [18] dependence analysis techniques and its own SDC-based scheduling engine [5] to extract fine grained instruction-level parallelism within each task. Finally Xilinx FPGA synthesis tools are leveraged to map the generated RTL onto reconfigurable fabric. We demonstrate that the FPGA accelerators generated by our FPGA design flow can efficiently exploit the computational resources of top-tier FPGAs in a customized fashion and provide better performance compared to the GPU implementation for a range of applications.

A. FCUDA Philosophy

Concurrency in CUDA is inferred through a single-thread kernel with built-in variables that are used to distinguish the tasks of each thread. Application parallelism is expressed in the form of fine-granularity threads that are further bunched into coarse-granularity thread-blocks (Fig. 2). Even though thread-level parallelism can improve performance, thread-blocks offer higher potential for an efficient multi-core implementation on FPGA. As discussed previously, CUDA thread-blocks comprise autonomous tasks that operate on independent data sets and do not need synchronization. Conversely, CUDA threads within a thread-block usually reference shared data which often results in synchronization overhead and/or shared memory access conflicts.

Parallelism in C code for FPGA synthesis by AutoPilot is explicitly expressed through parallel function calls (Fig. 3). A single callee function with a different set of arguments in each call may be used to infer a homogeneous multi-core configuration similar to the GPU organization, whereas different callee functions may model a heterogeneous multi-core configuration on FPGA. Therefore, the core task of the FCUDA source-to-source translation can be simply described as converting thread-blocks into C functions and invoking parallel calls of the generated functions with appropriate argument sets. Having extracted the coarse-granularity parallelism at the thread-block level, fine-granularity parallelism at the thread level may also be extracted, provided that non-allocated resources exist on the FPGA. This disparity in the thread parallelism extraction scheme between GPU and FCUDA may lead to different combinations of concurrently executing threads in the two devices. Nevertheless, the degree of parallelism will not differ in typical CUDA kernels that comprise hundreds of threads per thread-block and thousands thread-blocks per grid.

Another important feature of the FCUDA philosophy consists of decoupling off-chip data transfers from the rest of the thread-block operations. The main goal is to prevent long latency references from impacting the efficiency of the multi-core execution. This is particularly important in the absence of GPU-like fine-grained multi-threading support in FPGAs.
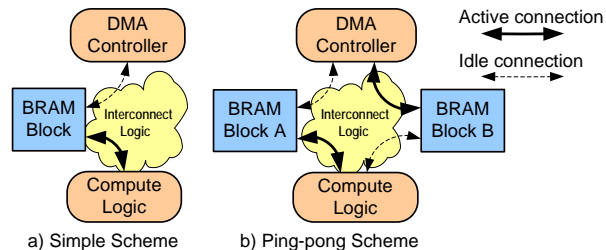

Fig. 4. Scheduling Schemes

Moreover, by aggregating all of the off-chip accesses into DMA burst transfers from/to on-chip BRAMs, the off-chip memory bandwidth can be utilized more efficiently.

FCUDA also leverages synchronization of data transfer and computation tasks based on the FCUDA annotation injected by the FPGA programmer. The selection of the synchronization scheme often incurs a tradeoff between performance and resource requirements. The FPGA programmer needs to consider the characteristics of the accelerated kernel in order to make an educated decision. A simple and resource efficient scheme is the *simple DMA* synchronization (Fig. 4a) which serializes data communication and computation tasks. This scheme is memory overhead free and it can also be a good fit for kernels that are compute intensive and incur low data communication traffic. At the opposite end, the *ping-pong* synchronization scheme overlaps data communication with computation by doubling the number of BRAM blocks (Fig. 4b). The interconnection logic interchangeably connects each BRAM block to the compute logic and the DMA controller, ensuring that each BRAM block is actively connected to only one of the two modules in each cycle. However, this scheme may result in BRAM utilization overhead, impacting the number of cores that can be instantiated on the FPGA.

B. FCUDA Pragma Directives

Fig. 5a depicts the FCUDA pragma annotation of the *coulombic potential* (*CP*) kernel. The kernel function is wrapped within *GRID* pragmas that define the sub-array of thread-blocks that can be computed by the available FPGA cores within one iteration (in Fig. 5a two thread-blocks with sequential *x* coordinates and same *y* coordinates). The *BLOCK* pragma determines the sub-grid of all thread-blocks that this kernel is assigned to compute. By splitting the original CUDA grid of thread-blocks into sub-grids, FPGA cores can be split into clusters with each cluster being assigned a sub-grid of thread-blocks. This can help further eliminate long wire interconnections between compute and synchronization cores and could enable asynchronous operation of different clusters. The *SYNC* pragma sets the type of synchronization scheme (currently a choice between *simple* or *pingpong*) to be implemented by the cluster synchronization core. *COMPUTE* and *TRANSFER* pragmas are used to wrap the computation and the data communication tasks of the kernel, respectively. In Fig. 5a, two TRANSFER sections are used: one for fetching off-chip data into the *atominfo* array and one for storing results to the *energygrid* off-chip storage. The following section describes how FCUDA leverages the translation of the CUDA code into properly crafted C code for AutoPilot. FCUDA compilation is based on the Cetus source-to-source compiler framework [19] and it consists of two major stages.
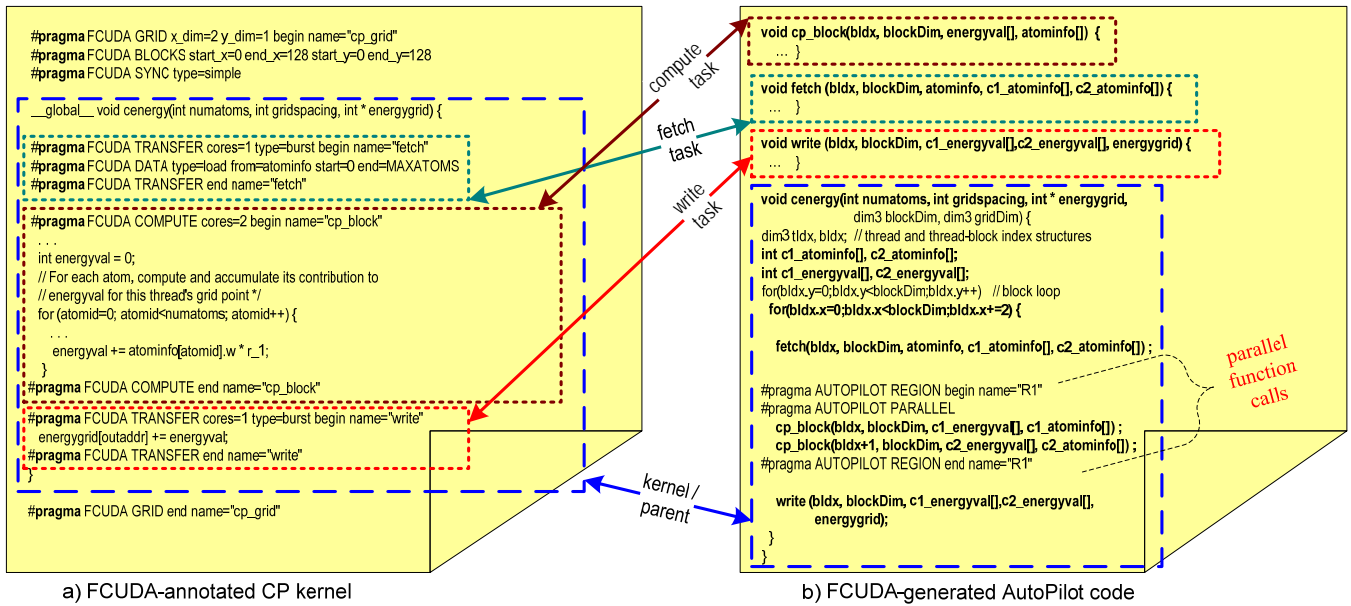
Fig. 5. Coulombic Potential (CP) kernel transformation through FCUDA

C. FCUDA Front-End Transformation

The front-end engine of FCUDA aims to transform the single-thread kernel into semantically equivalent C code which explicitly expresses the execution of all the kernel threads in a serialized fashion. This is achieved by converting the CUDA built-in variables that hold the thread (and thread-block) IDs into regular C variables which are used as induction variables in *thread-loops* (and *block-loops*). Fig. 6, illustrates the transformation of the CUDA kernel threads into thread-loops by the FCUDA front-end engine (Serialization of thread-blocks, though not depicted for space and clarity reasons, also takes place during this FCUDA phase).

As shown in Fig. 6, synchronization directives within the CUDA kernel need to be considered during the front-end transformation phase of FCUDA in order to maintain the ordering semantics of thread execution within the serialized thread-blocks. Synchronization points are indicated by CUDA sync directives, FCUDA *COMPUTE* and *TRANSFER* pragmas and irregular control flow statements (i.e. break, continue and return). A loop-fission technique proposed in [10] is used to break the initially generated kernel-wide thread-loop into localized thread-loops which do not cross any of the synchronization directives encountered in the code. Fig. 6 depicts the result of loop fission in a kernel with a single synchronization point. The initial kernel thread loop is split into two thread-loops: the first thread-loop implements the thread operations preceding the sync point, while the second thread-
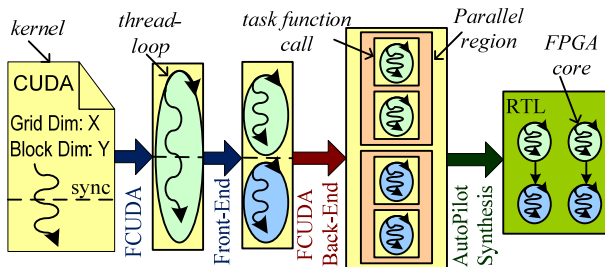
loop implements the thread operations following the sync point. This way serialized execution of threads maintains the thread-block synchronization semantics. FCUDA extends the MCUDA [10] implementation of loop-fission by adding *COMPUTE* and *TRANSFER* pragmas to the list of synchronization directives. COMPUTE and TRANSFER pragmas are used by the FPGA programmer to annotate computation and off-chip data communication tasks. Thus, synchronization of threads between tasks is required. Synchronization primitives can be removed after loop-fission, except for FCUDA pragmas which carry implementation information used by the back-end engine of FCUDA.

Thread serialization creates the opportunity for variable sharing among threads. However, there are cases in which each thread must have its private copy of a kernel variable. This happens usually for variables that are accessed across synchronization points (e.g. *energyval* in Fig. 5a). Scalar variable expansion [10] is applied for such variables in order to create thread-private copies of the variable. Fig. 7 depicts how the computation task of CP (Fig. 5a) is transformed after the front-end processing of FCUDA. Loop-fission forms a thread-loop within COMPUTE pragmas whereas selective scalar expansion results in vectorization of *energyval* which is referenced across thread loops.

D. FCUDA Back-End Transformation

The back-end engine of FCUDA leverages the implementation information annotated in the FCUDA pragma

Fig. 6. Extracting the CUDA coarse-grained parallelism in FCUDA

Fig. 7. FCUDA front-end processed CP compute task

directives to guide the translation of the kernel coarse grained parallelism into the function-level type of parallelism supported by AutoPilot (Fig. 6). Tasks annotated through FCUDA COMPUTE and TRANSFER pragmas are transformed into newly generated *task functions* which are called from the original kernel function, referred hereafter as *parent function*. Multiple calls of the task functions wrapped within *AUTOPILOT REGION* and *PARALLEL* directives in the parent function (Fig. 5b) drive the synthesis tool to instantiate parallel processing cores on the configurable fabric. The degree of parallelism is specified by the parameter information included in the COMPUTE and TRANSFER pragmas (Fig. 5a) and it is used to adjust the stride length of the *block-loop* in the parent function (Fig. 5b). One of the critical tasks of this transformation is the facilitation of data communication between the different task functions and the parent function. Variable analysis is performed to determine which variables are private to the task and which ones need to be communicated to/from the task function. Communication of both scalar and array variables is implemented through the task function parameters. Apart from the type and number of cores, the FPGA programmer can also extract thread parallelism (provided available resources exist) by injecting *AUTOPILOT UNROLL* and *PIPELINE* pragmas within the FCUDA COMPUTE annotated tasks, to specify thread-loop unrolling and pipelining, respectively.

As discussed previously, FCUDA TRANSFER pragmas are used to annotate data communication tasks to off-chip addresses. According to the FCUDA philosophy, off-chip data communication usually infers DMA burst transfers of data between off-chip memory storage and on-chip BRAM arrays. The FCUDA back-end engine is also responsible for instantiating array variables which will infer BRAM block allocation during synthesis by AutoPilot. BRAM associated arrays are instantiated at the parent function and their number is determined by the degree of parallelism annotated in the compute tasks that reference them (Fig. 5b). BRAM associated arrays may be passed as arguments to compute and transfer functions similarly to the rest of the variables. A challenging task of the BRAM array instantiation is the determination of their dimensions. There are two ways that this is accomplished: i) through variable access analysis and consideration of the containing thread-loop induction variable range space ("write" TRANSFER in Fig. 5a) ii) through FCUDA DATA parameter information ("fetch" TRANSFER in Fig. 5a). More details on the leveraging of different CUDA memory spaces within FCUDA are discussed in the following section.

Finally, synchronization of the FCUDA annotated tasks is performed by the back-end engine according to the *FCUDA SYNC* parameter information (Fig. 5a). In the current implementation the available SYNC options are *simple* and *pingpong*. The former corresponds to the simple-DMA scheme and does not require any further code massaging before feeding it to AutoPilot. It essentially results in the serialization of all the COMPUTE and TRANSFER tasks of the kernel. The pingpong option selects the ping-pong DMA scheme in which two copies of each local array are declared, doubling the amount of inferred BRAM blocks on the FPGA. Moreover, the parent function is altered based on a double-buffering coding template. Fig. 8 displays how the CP parent function is transformed for a ping-pong synchronization scheme. An *if-else* structure is used to implement the switching of the accessed BRAM block in each iteration of the block-loop.

E. CUDA Memory Spaces

The different memory spaces leveraged in the CUDA programming model have to ultimately be mapped onto local BRAM memories, as described in the previous sections. The most simple memory space to handle is *shared memory* due to its common semantics with BRAM memories. Both of them refer to local memory blocks that are private to threads within a thread-block. Thus direct mapping of shared memory arrays into BRAM memory is feasible. A distinguishing characteristic of shared memory is its 16-bank organization which allows 16 concurrent accesses by equal threads. BRAM memories, on the other hand, only support dual access concurrency. However, the serialization of block threads in the FCUDA flow eliminates the potential latency overhead of increased BRAM access conflicts in kernels that are engineered to take advantage of the multi-bank organization of shared memory. Besides, FPGA configurability offers the flexibility of organizing BRAM blocks in a multi-bank scheme if necessary (though this is not supported by our current implementation). Moreover, the BRAM block size customizability enables flexible tuning of kernels without the constraining restrictions imposed by the small size of shared memory on GPUs.

The *constant memory* space is shared by all the thread-blocks running on the GPU, but it is read-only and it is used for references that exhibit locality, since it is cached. These attributes make it a good match for the different DMA bursts schemes described earlier. A portion of the off-chip DRAM will serve as constant memory and the BRAMs will be used as read-only buffers that will be filled with the corresponding block of data before the thread-block execution. It may be possible to share BRAM blocks that contain constant memory data among a few compute cores on the FPGA, to reduce BRAM resource requirements per core, if it does not severely impact execution frequency.

Global memory corresponds to the off-chip memory of the GPU which is globally accessible at a high latency, but with

```
void cenergy(int numatoms, int gridspacing, int * energygrid,
             dim3 blockDim, dim3 gridDim) {
 . . .
  int c11_atominfo[],c12_atominfo[],c21_atominfo[], c22_atominfo[];
  int c11_energyval[],c12_energyval[],c21_energyval[], c22_energyval[];
  int pingpong=0;
  for(bldx.y=0;bldx.y<blockDim;bldx.y++)   // block loop
    for(bldx.x=0;bldx.x<blockDim;bldx.x+=2) {
      if(!pingpong) {
#pragma AUTOPILOT REGION begin name="R1"
#pragma AUTOPILOT PARALLEL
        fetch(bldx, blockDim, atominfo, c11_atominfo[], c21_atominfo[]) ;
        cp_block(blockIdx, blockDim, c12_energyval[], c12_atominfo[]) ;
        cp_block(blockIdx+1, blockDim, c22_energyval[], c22_atominfo[]) ;
        write (blockIdx, blockDim, c11_energyval[],c21_energyval[], energygrid);
        pingpong = 1;
#pragma AUTOPILOT REGION end name="R1"

      } else {
#pragma AUTOPILOT REGION begin name="R2"
#pragma AUTOPILOT PARALLEL
        fetch(bldx, blockDim, atominfo, c12_atominfo[], c22_atominfo[]) ;
        cp_block(blockIdx, blockDim, c11_energyval[], c11_atominfo[]) ;
        cp_block(blockIdx+1, blockDim, c21_energyval[], c21_atominfo[]) ;
        write (blockIdx, blockDim, c12_energyval[],c22_energyval[], energygrid);
        pingpong = 0;
#pragma AUTOPILOT REGION end name="R2"
} }
```

Fig. 8. CP kernel function with ping-pong scheduling

high bandwidth. Taking advantage of the high bandwidth requires access coalescing, which infers spatial locality among threads. Thus, data in global memory space can potentially be handled similarly with constant memory, by pre-fetching blocks of off-chip memory data through DMA bursts onto local BRAMS. The data block address range may be explicitly specified in the code in the case of global to shared-memory transfers, or it may be necessary to specify it through insertion of FCUDA DATA pragma directives (Fig. 5a).

Finally, textured memory is a cached read-only memory with the addition of filtering and special addressing capabilities. It could potentially be handled similarly to constant memory but it is not supported currently by our CUDA-to-FPGA flow.

V. EXPERIMENTAL RESULTS

As mentioned earlier, the CUDA programming model fits well in our FPGA design flow and offers the advantage of high-abstraction. Nevertheless achieving optimal performance requires some knob tuning from the programmer as well. One of the critical issues in achieving good application acceleration is the memory organization and the data access patterns to/from memory. The next subsection discusses the impact on memory from different implementation options. Subsequently we evaluate our FPGA programming flow by comparing the performance of CUDA kernels when executed on the GPU and FPGA (using FCUDA) platforms.

A. Memory Design Decisions

Table 1 lists BRAM experimental results from different multi-core implementations of the matrix-multiplication (matmul) kernel on the Xilinx FPGA. The first column lists the BRAM size, dimensions and element type (4, 2 or 1byte). Although Block RAMs can be flexibly combined into different size configurations and accessed through independent ports they have to obey the granularity of their architecture: 18Kbits for Xilinx devices. Thus, using them to store arrays of arbitrary sizes may result in waste of local memory space (column 2 in Table I) as the remaining space can not be used by other variables. This can be detrimental to parallelism extraction. Moreover, small sparsely populated BRAM blocks may affect the off-chip data transfers bandwidth.

TABLE I. BRAM DESIGN CHARACTERISTICS

BRAM (depth x bytes) elem type	BRAM waste	Burst Bandwidth	Total Bandwidth (#cores)
8Kb (16x64) 4-byte elem	55.5%	91.55MB/s	3.5GB/s (128)
4Kb (16x32) 2-byte elem	58.3%	52.45 MB/s	3.07GB/s (172)
4Kb (16x16) 1-byte elem.	59.7%	28.61MB/s	1.54GB/s (172)
16Kb (32x64) 2-byte elem.	11.1%	80.1MB/s	1.6GB/s (172)
8Kb (32x32) 1-byte elem.	55.5%	42.9MB/s	0.8GB/s (172)

The third column of Table I displays the achievable bandwidth with regards to the BRAM dimensions and the access type. Since BRAMs are often used for storing tiles of larger arrays (like in matmul) bursts are done in BRAM row granularity. The bandwidth of DMA bursts is therefore affected by the horizontal dimension of the BRAM memories (e.g row 3 vs row 5). The DMA burst efficiency depends also on the type of the array elements. Smaller elements may result in low bus utilization and more BRAM accesses. Both of these implementation details can impact the burst speed. Finally, the size of BRAM blocks affects the amount of computation involved in between DMA transfers and consequently the required off-chip bandwidth in a ping-pong synchronization scheme, where full overlap of data communication and computation is desired (4th column of Table I). For example, a configuration of 172 cores with 8Kbit arrays per core (row 6 in Table I) requires almost half the off-chip bandwidth compared to a similar configuration that only allocates 4Kbits of BRAM to each core (row4). This is due to the fact that in matrix multiplication the amount of computing is quadrupled for every doubling of the array data size. Thus, the window of time for transferring two times the data is increased by four, cutting the required bandwidth by half.

B. FPGA – GPU Comparison

For the evaluation of our FPGA design flow we targeted Xilinx Virtex5 FPGA devices. Virtex5 FPGAs are fabricated in 65nm CMOS technology and can be clocked at frequencies of up to 550MHz. These features render them good candidates for making meaningful comparisons with most of the currently used GPU devices. Moreover, the Virtex5 family includes some of the biggest and most advanced FPGAs available today that have the capacity to efficiently host high-concurrency multi-core accelerators. For our comparison experiments we chose the XC5VFX200T Virtex5 device, which has more than 100K LUTs, 16Mbits of on-chip BlockRAM memory and 384 DSP units. The GPU device used for the comparisons was Nvidia's G80 (90nm fabrication technology) with 16 SM units and 128 cores [14].

TABLE II. CUDA KERNELS

Kernel	Configuration	Description
Matrix Multiply (matmul)	1024x1024	Common kernel in many imaging, simulation, and scientific application
Coulombic Potential (cp)	4000 atoms, 512x512 grid	Computation of electric potential in a volume containing charged atoms
RSA Encryption (rc5-72)	4 Billion Keys	Brute force encryption key generation and matching

The CUDA kernels we used in these experiments are described in Table II. Two of them (matmul and cp) were based on GPU-optimized versions that were tailored into different integer bitwidth versions. The third kernel was implemented without any device-specific optimizations. In these experiments we focused on integer performance. Fig. 9 compares the FPGA and GPU performance for all versions of the 3 kernels. The rc5-72 kernel is intrinsically based on modulo-shift operations within 32-bit integers, and thus it was not transformed into smaller integer bitwidth implementation. The FPGA performance results are based on the assumption that the off-chip transfers are implemented by means of a high-bandwidth bus, such as the FSB (8.5GB/s) [19]. The computation task latencies are measured on the FPGA. Ping-pong synchronization is used between compute and data communication tasks and the latency of a single invocation is

measured for all kernels. The GPU latencies do not include the data communication from/to the CPU.

Table III lists the implementation details of the multi-core accelerators for all the benchmark versions. The implemented number of cores and the required off-chip memory bandwidth are shown in columns 2 and 3 respectively. Column 4 lists the resource type that constrained the number of implemented cores.

TABLE III.   KERNEL IMPLEMENTATION CHARACTERISTICS

Benchmark	Core #	DRAM Bandwidth	Limiting Resource
matmul 32bit	128	3,5GB/s	DSP
matmul 16bit	172	1.6GB/s	BRAM
matmul 8bit	172	0.8GB/s	BRAM
cp 32bit	25	0.128GB/s	DSP
cp 16bit	96	0.19GB/sec	DSP
cp 8bit	96	0.1GB/sec	DSP
rc5-72 32bit	80	≈ 0GB/sec	LUT

As can be seen, the generated multi-core accelerators can outperform the GPU, especially in the case of smaller bitwidths, where application specific customization can adapt the datapath of the cores and use the freed resources for instantiating more cores. Moreover, the narrower datapaths allow faster operation execution. For example the FSMs of the 16bit matmul has fewer states and also uses less DSP resources than the 32-bit one. In the case of 32bit CP, the compute intensive nature of the kernel results in high DSP utilization per core and low number of cores.

As mentioned above, the number of implemented cores was determined by the resource (LUTs, BRAMs and DSPs) that was most restrictive. However, in the case that BRAM or DSP blocks are the limiting resource it may be possible to extract more parallelism without increasing the number of cores. For example, in the case of 16bit and 8bit matmul kernels where BRAM constrains the number of cores, a 2X performance increase was achieved by exploiting thread-level parallelism within threadblocks. This is enabled by using *AUTOPILOT UNROLL* pragmas in the generated thread-loops.
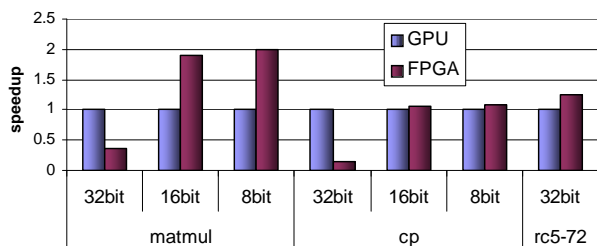

Fig. 9. GPU – FPGA Performance comparison

VI. CONCLUSIONS

In this paper, we present a new FPGA design flow that takes annotated CUDA code as input and generates C code for AutoPilot with task level parallelism that is synthesized into customized multi-core accelerators on FPGA. We demonstrate that the user can indeed use a single starting point for an efficient acceleration, irrespective of whether the target platform is GPU or FPGA. CUDA allows the user to express

application parallelism which our compiler can transform into explicitly parallel C suitable for advanced High Level Synthesis (HLS) tools such as AutoPilot. To our knowledge, this is the first such flow from CUDA to C code that leverages the potential of HLS tools for efficient FPGA design with high-abstraction. Initial results are very promising – even with limited optimization in our translation process, we have observed competitive speedup for 32-bit versions of the kernel and superior speedups for smaller bitwidth versions of the kernel. We expect to see even better performance from the FPGA accelerators as we optimize the FCUDA flow to make more efficient use of the memory resources and to better exploit the benefits that reconfigurable devices provide such as custom data paths, bitwidth optimized functional units, pipelining and multi-rate designs, when compared to GPUs. Future work also includes extending the interaction between compiler and synthesis techniques involved in FCUDA to new domain-independent parallel programming models, such as OpenCL.

REFERENCES

[1] J. Williams et. al. "Computational density of fixed and reconfigurable multi-core devices for application acceleration", *RSSI*, 2008.

[2] D. Gajski, "NISC: The Ultimate Reconfigurable Component", Center for Embedded Computer Systems, UCI, TR 03-28, 2003.

[3] D. Chen et. al. "xPilot: A Platform-Based Behavioral Synthesis System". *SRC TechCon'05*, 2005.

[4] http://www.impulsec.com/

[5] Z. Zhang et al. "AutoPilot: A Platform-Based ESL Synthesis System", High-Level Synthesis, Springer Netherlands, 2008, www.autoesl.com.

[6] L. Ling et al. "High-performance, energy-efficient platforms using in-socket FPGA accelerators", *Int. Symposium on FPGAs*, 2009.

[7] D. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation", *Int. Symposium on FPGAs*, 2009.

[8] XtremeData Inc., http://www.xtremedata.com

[9] J. Cho, S. Mirzaei, J. Oberg and R. Kastner, "FPGA-based face detection system using haar classifiers", *Int. Symposium on FPGAs*, 2009.

[10] J. Stratton et. al. "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs". *Int. Workshop on Languages and Compilers for Parallel Computing*, 2008.

[11] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Symposium on Application Specific Processors*, 2008.

[12] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert, "Embedded floating-point units in FPGAs," *Int. Symposium on FPGAs*, 2006.

[13] J. Cong and Y. Zou, "Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration", Int. Symposium on FPGAs, 2008.

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

[15] EJ Kelmelis, J. Durbano, J. Humphrey, F. Ortiz, "Modeling and simulation of nanoscale devices with a desktop supercomputer," *Int. Society for Optical Engineering*, 2006.

[16] Xilinx Inc., http://www.xilinx.com

[17] Altera Inc., http://www.altera.com

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

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

[20] M. Showerman, W.W. Hwu, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, "QP: A Heterogeneous Multi-Accelerator Cluster," *Int. Conf. on High-Performance Cluster Computing*, 2009.

[21] IBM Cell Processor, http://www.research.ibm.com/cell/

[22] S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah, "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary", *ECOOP*, 2008.

[23] A. Hormati, M. Kudlur, S. A. Mahlke, D. F. Bacon, and R. M. Rabbah, "Optimus: efficient realization of streaming applications on FPGAs", *CASES*, 2008.