FCUDA-SoC: Platform Integration for Field-Programmable SoC with the CUDA-to-FPGA Compiler

Conference Paper · February 2016

DOI: 10.1145/2847263.2847344

4 authors:

Tan Nguyen
University of PetroVietnam
11 PUBLICATIONS **87** CITATIONS
SEE PROFILE

Swathi T. Gurumani
Advanced Digital Sciences Center
31 PUBLICATIONS **258** CITATIONS
SEE PROFILE

Kyle Rupnow
Inspirit IoT, Inc.
58 PUBLICATIONS **890** CITATIONS
SEE PROFILE

Deming Chen
University of Illinois, Urbana-Champaign
269 PUBLICATIONS **4,899** CITATIONS
SEE PROFILE

Platform Integration of CUDA-to-RTL High Level Synthesis

Tan Nguyen, Swathi Gurumani, Kyle Rupnow, Deming Chen
Advanced Digital Sciences Center
Singapore
{tan.nguyen, swathi.g, k.rupnow}@adsc.com.sg
Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
dchen@illinois.edu

ABSTRACT

Throughput oriented high level synthesis allows efficient design and optimization using parallel input languages. Parallel languages offer the benefit of parallelism extraction at multiple levels of granularity, offering effective design space exploration to select efficient single core implementations, and easy scaling of parallelism through multiple core instantiations. However, study of high level synthesis for parallel languages has concentrated on optimization of core and on-chip communications, while neglecting platform integration, which can have a significant impact on achieved performance. In this paper, we create an automated flow to perform efficient platform integration for an existing CUDA-to-RTL throughput oriented HLS, and we open source the FCUDA tool, platform integration, and benchmark applications. We demonstrate platform integration of 16 benchmarks on two Zynq-based systems in bare-metal and OS mode. We study implementation optimization for platform integration, compare to an embedded GPU (Tegra TK1) and verify designs on a Zedboard Zynq 7020 (bare-metal) and Omnitek Zynq 7045 (OS).

1. INTRODUCTION

High level synthesis (HLS) is increasingly the preferred design method for hardware development due to improved productivity and reduced design effort to effectively explore implementation options. Study of high level synthesis tools initially used serial input languages such as C/C++, C#, SystemC, and Java [5, 7, 22, 25, 28]. HLS of serial languages performs optimization and parallelism extraction with the target of generating a single accelerator core that achieves performance goals. Parallel languages such as CUDA and OpenCL [2,8,12,13,16,17,26] present a throughput-oriented alternative for synthesis; some applications may be 7X or more better with parallel languages than serial languages [12]. Using parallel languages, HLS tools optimize a single accelerator core and increase throughput through multiple instantiations of the core.

FPGA'16, February 21-23, 2016, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/2847263.2847344

Throughput oriented synthesis takes advantage of parallel algorithm representations to allow independent exploration of parallelism within a single accelerator core and parallelism between multiple cores. Thus, in throughput oriented synthesis, platform integration takes even greater importance; accelerator core interconnect and integration with CPUs and memory controllers are critical to producing an overall accelerator system that meets designer goals.

Platform-level integration includes a variety of additional design concerns including HW/SW co-design, core control interfaces, mechanisms for data movement, utilization of internal and external bandwidth, workload distribution, and scalability of core interconnect. The method for CPU to accelerator communication, data transfer, and workload distribution affects both platform design and modeling of incremental benefit of additional accelerator cores. Mechanisms for sharing access to external memory controllers and sharing data among cores affect achievable efficiency of memory bandwidth as well as scaling of bandwidth use. It is important that platform generation is automated; design and interconnection of many-core systems with efficient use of shared resources, efficient mapping for data and workload distribution, and multiple different accelerator core types is a complex and tedious task.

In this paper, we discuss platform integration, optimization, and best-practices for throughput oriented HLS. We develop an automated system for generating optimized platform level designs for a throughput-oriented HLS based on an existing CUDA-to-RTL flow [17], and open-source the CUDA-to-RTL flow and automated platform integration. The automated flow performs integration of the generated HLS cores with the ARM-core of a Xilinx Zynq FPGA platform. We select Zynq platforms for the availability of an on-chip CPU core, and because the existing CUDA-to-RTL flow uses Xilinx Vivado HLS; future Xilinx Ultrascale Zynq products will continue to support ARM-based CPUs.

The automated flow takes in annotated CUDA kernels and host code as inputs and generates an optimized, application-specific platform-level design (bitstream and binary executable) to enable efficient FPGA board-level implementation of CUDA benchmarks. Using the platform integration flow, we map 16 benchmarks to the Zynq platforms, including several simple applications and 12 applications from the Rodinia [6] parallel language benchmarks. In this work, we concentrate on one specific CUDA-to-RTL flow, but this platform integration strategy can be applied to other throughput-oriented languages, such as OpenCL.

This paper contributes to the study of HLS with:

- An open-source academic automated platform generation flow integrated with an HLS tool with complete platform generation, on-chip bus support, and board-level verification.
- Demonstration of platform-level optimization issues in throughput-oriented HLS.
- Discussion of best practices in platform-level design and integration.

The rest of this paper is organized as follows. We discuss related work in interconnection and platform integration in section 2, then introduce the CUDA-to-RTL HLS flow in section 3. In section 4, we discuss features of the Xilinx Zynq platform, and the computation model for throughput-oriented synthesis. In section 5, we discuss platform integration issues, and optimization techniques. Finally, we present results for 16 benchmarks on two Zynq platforms and discuss best-practices in platform integration.

2. RELATED WORK

Platform integration is a common issue in the design and use of FPGA-based accelerators; although design of high-performance cores is common, it remains challenging to integrate those cores with appropriate memory controllers and communications interfaces to make a high performance system. Several recent works have concentrated on this problem by releasing open-source, well-optimized implementations of PCI-Express, DRAM, and Ethernet interfaces [9, 14, 19, 21, 24]. Although these standardized APIs make design and use of FPGA platforms easier, core design, optimization and platform-level optimization are left to the user. System-level linking of hardware and software components [11] has also been explored, but as with the platform interfaces, it concentrates on integration of a small number of cores.

For many core systems, there is extensive prior work on both bus-based [3,4,20] and network-on-chip [1,10] systems. However, these interconnects focus on on-chip communications rather than platform integration. It is critical to consider on-chip interconnects as part of platform integration.

There are also several industrial throughput HLS tools such as Altera's OpenCL [2] or Xilinx's SDAccel [26] which focus on OpenCL-to-RTL flow for CPU-FPGA systems in data center applications. Similarly, Xilinx's SDSoC [27] builds SoC applications, but does not use throughput oriented languages.

In this paper, we use an existing throughput-oriented CUDA-to-RTL HLS toolflow [16, 17] to design computation cores, and then explore the platform design and optimization using two Xilinx Zynq platforms. On each platform, we explore platform integration optimizations through a fully automated platform integration flow, and present best practices in platform integration for many core, throughput-oriented designs.

3. THROUGHPUT-ORIENTED HLS ON ZYNQ

The Xilinx Zynq platform combines Dual ARM Cortex A9 CPU cores in a processing system (PS) tightly integrated with programmable logic (PL) that can be configured, and controlled by the CPU. We use two platforms: the Zedboard platform contains a Zynq 7020 FPGA with 512MB of DDR2, and the Omnitek platform contains a Zynq 7045 FPGA with 1GB of DDR3. The Zynq 7045 has an identical ARM processor, but substantially more reconfigurable logic: about four times more LUTs, BRAMs and DSPs. The larger platform allows us to specifically concentrate on the scalability of the platform with significant numbers of accelerator cores. On both platforms, the CPU controls the accelerator cores to start computation and monitor completion. We will now briefly discuss the existing CUDA-to-RTL flow that we use in this work, and the on-chip integration.

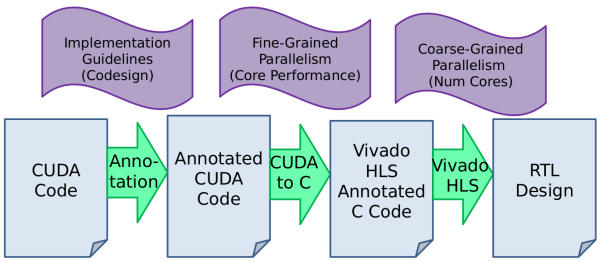

Figure 1: CUDA-to-RTL Flow

3.1 CUDA-to-RTL Synthesis Flow

The CUDA programming model is a single program multiple data (SPMD) computation model, where kernels are programs with many threads that operate on independent data with minimal inter-thread communication. In CUDA, each thread follows its own data-dependent control path. The CUDA-to-RTL flow used in this work (Figure 1) translates this SPMD-style code into a core that performs the computation of one or more CUDA threads (typically at least 32 threads). The entire kernel computation is performed by repeated use of the computation core, workload distribution among multiple instantiations of the core, or both.

Starting with CUDA code, the programmer may modify the code to group together data transfers to overlap communication and computation as a performance optimization. The user uses pragmas to identify compute and data transfer code regions as in Figure 2; a transfer pragma translates an assignment into a memcpy(), which Vivado HLS will translate into a burst-mode memory transfer to copy contiguous data with a specified burst length, similar to the memory coalescing access in the CUDA programming model. Transfer pragmas specify burst mode, which variable is the external pointer, the size and direction (input (0) or output (1)) of a data transfer. The compute pragma simply creates a for loop to iterate over the entire computation once for each thread ID. In general, the programmer may specify unrolling, parallelism and memory partitioning in the compute pragma. Note that the CUDA to C translation performs unrolling, parallelisation, and partitioning and does not use the related Vivado HLS-specific pragmas. In this work, we concentrate on scalability issues in the platform integration of cores rather than the performance effects of alternate core designs. Thus, we do not use design space exploration [18], which would examine alternate settings for unrolling, parallelism and data partitioning. We instead concentrate on integrating as many instantiations of a default core design as possible in a single platform.

After inserting pragmas into the CUDA code, the annotated CUDA is passed into a source-to-source compiler based on MCUDA [23] and the Cetus framework to produce C-code annotated with I/O interface pragmas (Figure 3) that will

```
#pragma FCUDA TRANSFER begin type=burst
    pointer=[A] size=[16] dir=[0]
  //Input Data Reading Statement
  As[tIDy][tIDx] = A[a + wA*tIDy + tIDx];
#pragma FCUDA TRANSFER end

#pragma FCUDA COMPUTE begin unroll=4 mpart
    =2 array_split=[As]
  //Kernel code performing computation
#pragma FCUDA COMPUTE end

#pragma FCUDA TRANSFER begin type=burst
    pointer=[C] size=[16] dir=[1]
  //Output Data Writing Statement
  C[c + wB*tIDy + tIDx]=Csub[tIDy][tIDx];
#pragma FCUDA TRANSFER end
```

Figure 2: Data Transfer Pragmas for FCUDA

```
// Control port for parameter wA
#pragma HLS INTERFACE ap_none register
    port=wA
#pragma HLS RESOURCE core=AXI4LiteS
    variable=wA

// Control port of the whole core
#pragma HLS RESOURCE core=AXI4LiteS
    variable=return

// Data port for parameter A
#pragma HLS INTERFACE ap_bus port=A
#pragma HLS RESOURCE variable=A core=
    AXI4M
```

Figure 3: Communications Pragmas for Vivado HLS

be processed by Xilinx Vivado HLS to produce RTL. These pragmas specify which variables are in the I/O interface, and the interface type. We will discuss the partitioning of I/O ports and selection of port types in more detail in section 4.

3.2 On-Chip Integration

In a particular application, there may be multiple CUDA kernels; we also support mapping of multiple independent hardware accelerators, each of which may have one or more parallel cores. Our automated flow can instantiate any number of cores. Thus, in applications with multiple CUDA kernels, we simply distribute total FPGA resources evenly between kernels (e.g. two kernels are allowed roughly half of the reconfigurable resources each), unless data-dependence or the number of thread-blocks of the CUDA kernel is the limiting factor (rather than available FPGA area) in number of instantiable cores. We do not explore options for uneven distributions among kernels, but always attempt to find the maximum FPGA utilization that remains feasible for implementation. In section 4.3, we explain the analytical model for determining the number of cores for each kernel.

The cores are integrated together using the ARM-controlled AXI bus. The ARM core signals the cores to begin computation, and the cores will individually perform memory accesses (burst accesses due to the TRANSFER pragma) to read data into each core's local memory. The ARM core monitors the cores for completion: once all cores are complete, the ARM can continue with other computation or start the cores again on new input data. In the following section, we will discuss optimizations of memory use, data-sharing mechanisms, workload distribution techniques, and core interconnect techniques.

3.3 System Integration

In this paper, we use the Zynq platforms both as a bare-metal platform and as a platform running a Linux-based operating system. At the circuit level, both platform styles are nearly identical. However, there are several specific differences between the integration styles. Most importantly, in the bare-metal platform, the CPU does not use virtual memory, and thus both the CPU and kernel cores can directly use physical memory addresses. The OS-platform uses virtual memory; however, in order to perform parallel accesses without using the ARM core's MMU, the kernel cores must directly access physical memory addresses, and userspace applications are explicitly forbidden from knowing the virtual-to-physical translation. We use a mechanism to maintain our own translation by memory-mapping one or more pages of virtual memory to specific physical addresses, and thus we can always provide the kernel cores with appropriate physical addresses that correspond to the data structures in the userspace application.

In addition, there are a few minor differences in the supporting configuration files. In bare-metal mode, the user must supply a Board Support Package with drivers for the accelerator core and any required peripherals. In OS mode, all peripheral drivers are already part of the OS package, so only an accelerator driver is necessary. The OS features simplify development of platform I/O compared to needing to design custom drivers as in a bare-metal platform.

4. PLATFORM INTEGRATION

In this paper, we perform platform integration of HLS-generated cores on a Zedboard with a Zynq 7020 and an Omnitek Zynq 7045 platform. Before we discuss the details of our integration and optimization, we first introduce the features of the Xilinx Zynq chips, particularly, the communications and control for our platform integration. Both platforms contain Xilinx Zynq 7000 series chips with ARM CPUs (667MHz and 800MHz, respectively) and speed grade -1. External memory bandwidth is thus higher on the Omnitek platform, and the Zedboard has less memory (512MB vs. 1GB) The Zynq 7045 also has more reconfigurable resources than the Zynq 7020.

4.1 Xilinx Zynq

The Xilinx Zynq SoC platform consists of a region with a dual-core ARM Cortex A9 processor, together with hard-core implementations of communications interfaces such as USB, Gigabit Ethernet, DDR controllers, and general purpose IOs, and a reconfigurable region. The processor region can be connected to the reconfigurable region via AXI interconnect. There are three types of AXI functional interfaces: a cache-coherent interface (ACP), four high-performance and bandwidth interfaces (HP), and four general purpose interfaces (GP). Although the ACP port provides high-bandwidth and direct connection to the CPU's L1/L2 cache, it only supports up to 8 master devices, limiting the scalability of an approach using this port. Furthermore, the CUDA programming model is explicitly not cache-coherent. Thus, we

use the high-performance interfaces for core communications and a general purpose interface for control.

4.2 Computation Model

In our computation model, the accelerator cores operate similar to the CUDA programming model. The ARM cores allocate memory resources for input and output buffers, control the accelerator cores by sending run-time parameters, start computation, and wait for cores to signal completion. However, unlike the CUDA computation model, the CPU is not responsible for moving data from CPU memory into core's local memories: each core will generate streams of memory requests that will fill their local memories and then copy final results into global memory. Due to the TRANSFER pragmas, each memory request is a burst transfer of multiple data items for efficient bandwidth usage.

4.3 Synthesis and Platform Design Flow

An overview of our automated platform integration flow, depicting the inputs, automated steps and output is shown in Figure 5. In our automated flow, the designer provides the annotated CUDA kernel code, host code, the number of CUDA thread blocks of the kernels, and FPGA device information as input. Through a series of steps, the automated flow generates an optimized platform-level design including the bitstream for the reconfigurable logic, and the binary for software components. We now explain the detailed sequence of steps in our automated flow.

As a first step, our automation script invokes the prior CUDA-to-RTL compiler tool to translate the annotated input CUDA kernels into Vivado-synthesizable C-code (1). This translation step also inserts Vivado communication pragmas as in Figure 3. The annotated C-code is then synthesized with Vivado HLS to create an RTL IP for the kernel. For computational core generation, the core interface is a significant issue. The original CUDA function prototype contains input and output data pointers, and information about the number of threads, thread blocks and their organization is handled by the GPU device driver. Furthermore, the CUDA prototype assumes that function arguments are pointers to data accessible by the GPU (i.e. data that is already copied to GPU memory)[1]

For the FPGA interface, we must both pass references to input and output data buffers ,and call-specific parameters specifying the number of threads, thread-blocks, their organization, memory addresses, and workload distribution among cores. An example showing the difference between the original CUDA prototype and alternate CUDA-to-RTL prototypes is shown in Figure 4. The corresponding set of run-time configurable parameters is shown in Table 1. We will discuss optimizations selecting how to partition and/or merge interface ports in Section 5.

Next, our flow performs system integration using Vivado IP integrator to instantiate the IP core with the ARM CPU, DDR interface, and AXI interconnect (2). This allows the ARM CPU (Zynq processing system (PS)) to control the IP core, and allows the IP core to read and write data to DDR memory. This initial system is only a single core system, but is used to gather post-synthesis area information about the core so that we can automatically estimate the maximal number of core instantiations given an allowed area budget.

[1]CUDA 6 introduced unified memory, which, similar to our model, allows the GPU to read directly from host-memory

Table 1: Run-time Configurable Parameters

num_cores	Number of cores used for this kernel call
core_id	Per-Core identifier for the kernel call
wX	Set the scalar value of input scalar wX
gridDim.x	x-dimension value of CUDA grid
gridDim.y	y-dimension value of CUDA grid
gridDim.z	z-dimension value of CUDA grid
blockDim.x	x-dimension value of CUDA thread block
blockDim.y	y-dimension value of CUDA thread block
blockDim.z	z-dimension value of CUDA thread block
X_addr	Set the address of input or output buffer X

```
//Original CUDA function prototype
__global__ void matrixMul(float *A, float
    *B, float *C, int wA, int wB)

//CUDA-to-RTL merged function prototype
void matrixMul(float *memport)

//CUDA-to-RTL parallel function prototype
void matrixMul(float *A, float *B, float *
    C, int wA, int wB, dim3 gridDim, dim3
    blockDim, int A_addr, int B_addr, int
    C_addr, int num_cores, int core_id)
```

Figure 4: Comparison of CUDA vs. CUDA-to-RTL Function Prototypes

The tool flow then synthesizes the design to get an initial estimated resource report (3). In applications with multiple independent kernels, this synthesis is performed once for each kernel. Then, based on the available resources of the target FPGA device (or a proportion thereof), and the number of CUDA threadblocks for the kernel, our flow determines the maximal number of instantiable cores (4). In smaller FPGAs such as the Zynq 7020, the LUT, FF, BRAM or DSP resources are always the limiting factor; however, in larger FPGAs, a large number of cores may not be routable despite small area utilization. We use an analytical model to determine whether a target number of cores (and total I/O ports) is routable [15].

With the updated estimate for the maximal design in number of cores, our flow generates a new C-code wrapper for the required CPU-core control communications. The flow also iterates with Vivado HLS to verify that the integrated system is actually feasibly synthesized as shown in Figure 5 (4). On occasion, the first system design is not actually synthesizable; our analytical model may be aggressive, as it does not attempt to model scaling effects in multiple instantiations other than the loose upper bound on routability. In this case, we simply iteratively remove one core and retry until a design is verified as synthesizable.

Once we have a verified design with the maximum number of cores, the automated flow performs a binary search on the target frequency to find the maximal achievable frequency of the whole design (5). In this work, we do not explore the performance difference of designs with fewer cores but higher achievable frequency; for our purposes, we simply want to demonstrate that our automated flow can find and implement designs with many cores – we do not argue that the maximal number of cores is necessarily the best performing

configuration. Similarly, as noted before, we do not compare different core designs; our data simply demonstrates that for any particular core design, we can create an efficient platform-level implementation with many instantiations.

After determining the maximum achievable frequency, the bitstream is generated to configure the Zedboard or Omnitek platform (6), and the corresponding C-code is compiled to an executable for the host ARM CPU. We generate drivers for the ARM cores to interface with the IP, and compile those drivers and the host code to produce a binary (7).

For bare-metal platform implementations, we compile a Board Support Package (BSP) comprising of necessary drivers. The bitstream is downloaded to the board via a JTAG cable using the Xilinx SDK or Xilinx CPU Debugger. For OS mode platform implementations, we do not require a BSP; instead, the FPGA is configured by simply copying the bitstream to the FPGA's device file (via the *cat* command). However, we must also generate a device tree file and include it in the boot image to list all system hardware components and their respective system addresses. When the board boots the OS, it will initialize the hardware components, and the accelerator cores will be available as userspace I/O (uio) devices that will be configured by the core driver.

As discussed earlier, the OS platform also requires additional userspace code in the application to maintain a physical to virtual address mapping, such that we can provide the kernel cores with correct physical addresses. For this, we simply memory map a page-aligned pointer to a physical address and use this memory region for data that needs to be accessible by the kernel cores (for reading or writing). Then, when we need to pass pointers to the kernel cores, we can simply translate a specified physical address to the mapped virtual address.

For functional verification, we generate several related versions of the host code. First, we configure the host code to independently perform the kernel computation and compare CPU-computed and core-computed results. For performance evaluations we generate a version that does not use the core for a CPU-only performance. Then, we also gather the performance using the accelerator cores. In both cases, we measure full application performance including data allocation and transfers. However, we do not measure FPGA configuration time; we are evaluating these platforms as stand-alone platforms, not run-time reconfigurable accelerator boards. If a designer wishes to reconfigure the Zynq platform at runtime, they are responsible for ensuring that each configuration is used sufficiently long to amortize the reconfiguration overhead.

5. PLATFORM OPTIMIZATION

The automated design flow as described produces functionally correct accelerated applications. However, there are many potential design options within this general flow, and performance optimization of the platform characteristics is critical. In this section, we describe performance optimizations in order to generate efficient implementations.

5.1 Core Interfaces

For all core I/Os, we can merge ports together or leave I/Os as multiple parallel ports. Merged interfaces require fewer ports for interconnect at the system level and thus simplify platform integration and the area of system-level interconnects. However, parallel interfaces allow higher performance, and flexible separation of control, input and output data. Although all platform methods must use a unified memory space, both methods can use a unfied address space.

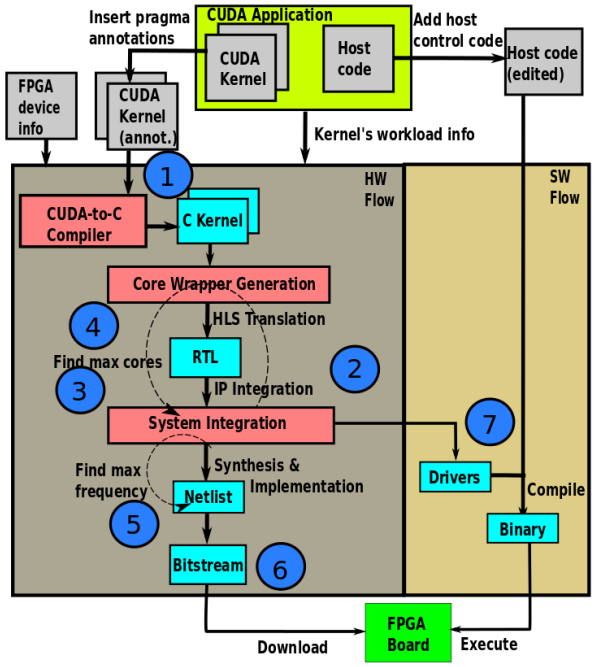

Figure 5: Automated Platform Integration Flow

In certain cases, it may be important to partition input and output ports; when all control, input and output data are merged into a single port, we cannot employ ping-pong buffers to pre-fetch the next set of input data because input and output buffers share the same port. Furthermore, merged ports would prevent system-level pipelining between multiple communicating kernels; although we do not implement system-level pipelining in this work, keeping input and output ports separate enables future designs where the output of one kernel fills the input buffer to the next. To prevent large platform-level crossbars, we generally merge multiple ports in the core design using a single memory port with per-array offsets. Although this reduces core parallelism, it also reduces the size of interconnect. When ports have different datatypes, they are left as parallel interfaces to minimize kernel modifications.

For core control signals, we also have the choice of merging ports, but in addition to allowing run-time configurability, we can also assign fixed signals. A fixed num_cores value increases complexity to reuse the cores with multiple input data sizes. Although the number of cores may be fixed, the total amount of work, location of input/output data buffers, and workload distribution may vary at runtime. Thus, we allow run-time control of parameter values for core identification, grid and block dimensions, total number of cores used, and memory offsets for input data.

5.2 Communication Interfaces

The Zynq platform has cache-coherent (ACP), high performance (HP) and general purpose (GP) AXI interfaces. A cache coherent interface is inappropriate – the CUDA programming model explicitly specifies that memory writes by

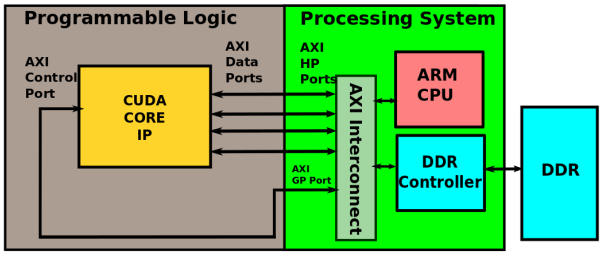

Figure 6: System Integration

CUDA kernels are not coherent, and the interface is limited to 8 cores on the bus due to the 3-bit identifier field. For data ports, the high-performance interface allows high-bandwidth memory accesses. For parameters, a general purpose interface is sufficient; furthermore, partitioning traffic into data and control networks prevents control transactions from affecting efficiency of the high-bandwidth interface. We show an overview of how the programmable logic is connected to the processing system using different ports of the AXI interconnect in Figure 6. The hardware and the processor system communicate with the DDR memory through the DDR controller that is also connected to the AXI interconnect.

To use the high-performance interfaces, we partition cores into four equal groups and, through an AXI interconnect, connect to one of the four HP AXI interfaces. Each AXI interconnect may have up to 16 ports, thus we can support up to 64 cores in a single level of hierarchy. If more cores will be simultaneously active, we can build a hierarchical interconnect to support additional cores. However, in our designs, we did not require more than 64 simultaneously active cores. In platforms with more than 64 total cores, we are able to share ports as we will describe in section 5.6

As shown in Section 3, the existing source-to-source translation generates Xilinx Vivado-HLS annotations for communications interfaces of the I/O ports. The CUDA to C translation directly performs any unrolling, pipelining, parallelism or partitioning of memory arrays instead of using Vivado HLS pragmas. However, as noted earlier, in this paper we do not perform unrolling, parallelism or partitioning. Depending on the type of interfaces, we specify different annotations; data ports become AXI4 memory-mapped interfaces integrated with a high-performance AXI port for higher bandwidth, and control ports become registers integrated with an AXI4LiteS interface for better scalability and independence from the high-performance ports. An example of the necessary pragmas was shown in Figure 3.

5.3 Workload Distribution

For CUDA kernels, the kernel code is designed so that the number of computation threads and the dimensions of the grid of thread blocks is flexible but directly related to the amount of data to process. During execution, the CUDA driver is responsible for distributing blocks of computation work to the GPU's execution units, assigning a thread-block of work to a single streaming multiprocessor (SM).

In our computation model, the CPU is responsible for setting initial parameters for the core computation, but it is not desirable for the CPU to explicitly handle workload distribution. The extra overhead of the CPU explicitly managing a queue of tasks would eliminate some performance and energy benefits of using FPGA-based acceleration. Thus, it is desirable for the cores to automatically compute which thread blocks of the overall computation will be executed based on the cores' identifiers and the total number of cores in the designed platform. Figure 7 shows how a core selects a set of thread blocks for computation. Although this is a static workload distribution, it reduces management overhead, and is a reasonable choice when there is little data-dependent variance in core computation latency.

```
for (b_index = 0; b_index < gridDim.x;
    b_index += num_cores) {
    bIdx.x = b_index + core_id;
    // Perform computation
}
```

Figure 7: Core computation to determine Workload distribution

5.4 Shared Control

When instantiating many identical cores, we can assign unique AXI core identifiers in order to make each core individually controllable. However, individual controllability also means that each core must be started individually rather than starting all cores simultaneously. Furthermore, this also requires wiring each AXI4LiteS port through an AXI interconnect block, which consumes additional area. In the CUDA programming model, it is the normal case that we wish to use all cores simultaneously. Thus, we only use one port of the AXI interconnect by wiring all of the control ports together and setting individual identifiers as fixed parameters used only for workload distribution.

5.5 Shared Data

Many CUDA kernels use CUDA constant memory as shared input data; in our processing model, each core would independently generate a stream of external memory requests to copy this identical data into cores' internal BRAMs. To improve the performance of this behavior and reduce external bandwidth demand, we can instead create an AXI stream interface for each core's data port and instantiate AXI DMA engine (configured in simple mode) for each core to stream the constant data to the data port of the cores. However, in a situation where the size of the constant memory is too large to duplicate for all the cores, it is more logical to perform a single memory copy to bring data on-chip by using an AXI central DMA (CDMA); and then, each core can fetch the data from on-chip buffers for their computation instead of performing an expensive off-chip access. The AXI DMA can also be configured in scatter-gather mode to perform a single transaction that distributes data streams to all cores with the multi-channel feature. We show the performance difference between CDMA, multi-DMA versus non-DMA implementation for benchmark coulombic potential (cp) in Figure 8. Each core's memory transactions already use burst-mode requests, so there is little performance benefit with DMA techniques for fewer cores, the area efficiency of the DMAs do not prevent instantiation of additional cores and hence we choose to have CDMA in our design.

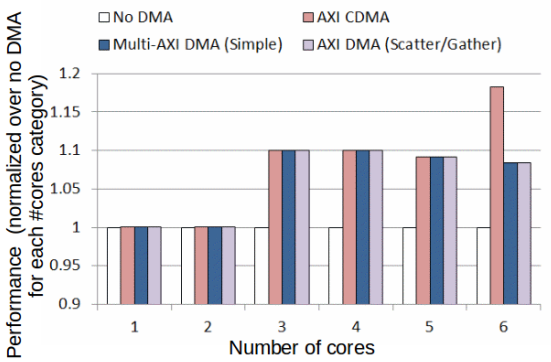

Figure 8: Performance Difference by DMA type of Benchmark cp

5.6 Multiple Core Types

Applications may have multiple computation kernels, each with independent core accelerators. In our system, this is natively handled: each core has independent resource identifiers, so cores can be independently controlled and data ports can inherently share access to external memory over the high-performance AXI interconnect. In general, a user should explore the number of cores of each type to instantiate in order to maximize performance; in this work, we distribute resources proportionally based on the CUDA kernels' total workload.

In order to facilitate better organization of kernel instantiations, we create a hierarchy of instantiations. For each kernel, we create a wrapper module that instantiates all copies of the kernel core; then at the system level, we instantiate just the wrapper modules. This has little effect on the area or frequency, but better organizes the instantiations.

However, in applications where the kernels' executions are serialized, we can explicitly share AXI ports when it is statically guaranteed that the two respective cores will never be active at the same time. This technique can substantially reduce area consumption due to the rapid scaling of AXI interconnects. Figure 9 demonstrates a system-level design where kernel 1 and kernel 2 are serialized and can thus share two out of three AXI ports instead of requiring 5 ports.

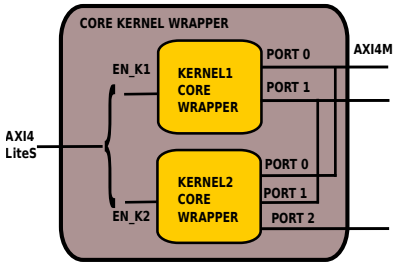

Figure 9: Serialized Kernels Sharing AXI Ports

5.7 Host Control Code

The host control code running on the ARM is a simple set of loops: one to setup core parameters, one to initiate computation of the core, and a third to wait until the core has signaled completion. An example host code containing the three loops is shown in Figure 10. As discussed earlier, in the OS version of the host-control code, the host code must also perform a translation so that it can provide the cores with appropriate physical addresses to access.

```
// setup core parameters
//One for each input scalar port
XCUDAKern_SetWX(&xcore, wX);

//CUDA Kernel x-Dimension (similar to y, z)
XCUDAKern_SetGriddim_x(&xcore, gridDim.x);
XCUDAKern_SetBlockdim_x(&xcore, blkDim.x);

//One for each input & output buffer port
XCUDAKern_SetX_addr(&xcore,
    (u32)X / sizeof(float));

//Turn on Kernel 1 for execution
XCUDAKern_EnK1(&xcore, 1);

// initiate computation of the core
XCUDAKern_Start(&xcore);

// wait until the core signal completion
while(!XCUDAKern_IsDone(&xcore));
```

Figure 10: Host Control Code

In addition, a designer may wish to integrate features to dynamically decide whether to use the kernel cores. For example, if the overhead of kernel calls sets a limitation on the minimum data size to achieve speedup, the user may wish to use the CPU for smaller data sizes. Similarly, the host code may overlap kernel-core execution with using the CPU for some of the computation. In this work, we concentrate on the hardware platform issues, and do not explore implementation of additional features in the host code.

6. RESULTS AND ANALYSIS

We now present the results of 16 benchmarks running through our automated process, including matrix multiplication (matmul), coulombic potential(cp), discrete wavelet transform (dwt), Fast-Walsh transform (fwt) and 12 benchmarks from the Rodinia benchmark suite [6]. One benchmark from Rodinia is unsupported due to use of computation functions not available from Xilinx Vivado HLS (e.g. floating-point power function). The remaining benchmarks are supported using our flow when provided with a correctly annotated CUDA kernel as input.

The applications are selected to provide a variety of complexity for both the CUDA-to-RTL flow as well as the platform integration demands. As noted earlier, we do not perform unrolling, parallelism, or partitioning. Furthermore, as most of the applications are floating-point, we do not perform datatype bitwidth minimization.

For each benchmark, we use the automated flow to implement the core, select the number of cores to implement, and generate the implementations for both the Zedboard and Omnitek platforms. We present speedup of the maximal number of cores compared to the CPU-only version measured on the respective platform's ARM CPU. In addition, we measure the CPU-only and GPU-accelerated speedup on the NVidia Jetson TK1 platform that contains a Tegra K1 SoC with quad-core ARM Cortex A15 and Kepler GPU with 192 cores. The Jetson TK1 has double the host-accelerator bandwidth of the Zynq platforms. We report the applica-

tion speedup on the TK1 and will also compare performance per watt on the TK1 with the Zynq platforms. In the case of applications with multiple CUDA kernels, we distribute FPGA resources evenly among the kernels, and then instantiate the maximal number of cores given area, routability, and kernel structure constraints as described in section 4.

Table 2 shows the number of instantiated cores, measured application speedup for each benchmark, and the achieved core execution frequency. When an application has multiple core types, the number of instances is a comma delimited list. We first observe that on the Zedboard platform we have between 2 core instantiations (lavaMD) and 28 cores (bfs), with achievable frequency as high as 100MHz. However, 5 of the 16 benchmarks do not achieve speedup over the ARM CPU. Due to the low total available reconfigurable logic, there is insufficient space to instantiate enough cores to overcome overheads and achieve significant speedup.

This is illustrated by the Omnitek platform results: with 4× more reconfigurable logic, we instantiate roughly 4× as many cores, and achieve sometimes substantially improved speedup. On the Omnitek platform, only 3 benchmarks do not achieve application speedup, including one that does not achieve speedup on the TK1 platform either. These three benchmarks include complex data traversals that may not be inherently well-suited for FPGA acceleration. However, the additional area allows substantially improved performance: sometimes better speedup than the Jetson TK1 (fwt, matmul, dwt, backprop, hotspot, lavaMD, and nw)

It is important to note however, that the Zedboard and Omnitek results are not directly comparable; the ARM core on the Zedboard runs at 667MHz compared to 800MHz on the Omnitek, and we use bare-metal platform integration on the Zedboard vs. OS-based integration on the Omnitek. Speedup results are quoted with respect to each platform's CPU, and are not intended to be directly comparable. Rather, these results simply demonstrate that our automated platform integration flow supports complex applications and many core instantiations, and with sufficient available area, can obtain overall application speedup.

Among the applications that do not achieve speedup on either Zynq platform, the bfs benchmark depends on GPU caching instead of using local storage within the GPU kernel, and thus it is unable to achieve overall speedup. Similarly, particlefilter has complex memory traversals that significantly affect performance without GPU caching. In addition, several benchmarks achieve lower speedup on the Omnitek platform than the Zedboard despite the additional resources. It is important to reiterate that the speedup numbers are relative to a higher-performing CPU, and that the OS-based implementation incurs additional overhead, particularly to maintain the physical-to-virtual address translation necessary in OS-mode.

In all applications, a developer can redesign at the CUDA level to improve local data use, access patterns, or required resources. Furthermore, unrolling, parallelism, and data partitioning can further improve the performance of individual core designs, which would correspond to further improvement at the platform level. This work performs efficient, automated platform integration, and prior work has suggested that design space exploration may improve speedup and area per core by over an order of magnitude [18].

In addition to performance, we also estimated the power consumption on both platforms: we use the nominal power reported for Jetson TK1 and the nominal power consumption of the Zynq reported by Vivado Power Estimator. Using this power consumption, we compute the performance per Watt of each platform. Normalizing to the Jetson TK1, we see that the smaller Zedboard platform can have gains of over 2× with several benchmarks that did not achieve significant speedup on the Tegra K1, but both the Zedboard and Omnitek platforms generally have worse performance per Watt. However, this result is expected: most of the applications are floating-point applications with no particular bit-width optimization, which is where FPGA implementations have particular advantages over GPUs. Nonetheless, performance per Watt demonstrates that our platform integration produces designs that are sometimes more efficient than an embedded GPU and typically within 3× of the GPU efficiency in perf/W despite no particular optimization emphasis.

To demonstrate that customization can further help FPGA results, we also use an integer version of the matrix multiplication benchmark (not presented in Table 2), and compare the performance per Watt of both a 32-bit integer and 16-bit integer version on the Omnitek platform and Jetson TK1. For the 16-bit version, we do not perform design space exploration – we simply pad the integer values to the 32-bit bus width, and keep the same total number of cores in the 16- and 32-bit versions. As expected, performance slightly improves due to efficiency improvements in the 16-bit datapath, and thus performance per Watt gains compared to the Jetson TK1 improve from 1.5× for 32-bit integer to 2.4× for 16-bit integer. In practice, a designer should perform more detailed optimization of the memory bandwidth, and core datapath, as well as perform design space exploration, but this simple experiment demonstrates expected optimization opportunity when using datapath customization.

We also present the area of each integrated platform design in Table 3. As discussed above, we observe that DSP utilization is higher due to the floating-point nature of the benchmarks. We now discuss some best practices in throughput oriented core design and platform integration.

6.1 Automation Flow Performance

Our automation flow includes several steps, and potentially iteratively performing expensive synthesis, place and route. However, these synthesis steps are similar to the iterative optimization and search for achieveable frequency of a typical manual design process. In this article, we use this automation flow with a single, default core design; in deployment, users should either use the flow with a core design previously selected through design space exploration, or, if many full place and route syntheses are acceptable, as part of a design space exploration between multiple alternative core designs.

6.2 Best Practices in Throughput Platforms

Platform integration in throughput oriented designs requires design tradeoffs for scalability throughout the design process. From the selection of core interface, techniques for controlling the cores, the interconnection network, to workload distribution, data sharing, and supporting multiple simultaneous kernels, it is critically important to consider how these decisions will scale to a system with dozens of cores.

The interconnection network is of particular importance, because our decisions and techniques for optimizing the in-

Table 2: Platform Performance Results

	Tegra K1	Zedboard Zynq 7020				Omnitek Zynq 7045			
Benchmark	Application Speedup over ARM	Num Cores	Application Speedup over ARM	Execution Frequency (MHz)	Perf./Watt over TK1's	Num Cores	Application Speedup over ARM	Execution Frequency (MHz)	Perf./Watt over TK1's
Simple CUDA Kernels									
cp	186x	6	3.4x	90	0.02x	20	16.6x	107	0.03x
fwt	6.7x	8,8	9.4x	100	1.05x	36,36	7.7x	97	0.28x
matmul	3.6x	16	4.9x	77	1.74x	64	7.3x	97	1.1x
dwt	2x	14	2.9x	83	2.24x	52	4.4x	87	1.1x
Rodinia CUDA Benchmarks									
backprop	0.4x	6,6	0.5x	90	0.41x	16,16	1.33x	71	0.7x
bfs	1.3x	14,14	0.4x	56	0.33x	48,48	0.96x	71	0.4x
gaussian	6.5x	2,14	1.8x	83	0.26x	2,44	3.8x	71	0.3x
hotspot	6.7x	4	1.3x	77	0.29x	16	7.3x	87	0.47x
lavaMD	17.5	2	1.6x	63	0.16x	8	26.8x	71	0.90x
lud	45.2x	2,3,3	2.3x	77	0.07x	1,14,14	20x	71	0.28x
nn	1x	14	1.4x	59	2.48x	32	0.74x	82	0.54
nw	3.8x	10,10	2.9x	83	1.40x	40,40	4.59x	77	1.01x
particlefilter	0.8x	27	0.6x	56	1.17x	27	0.55	71	1.17x
pathfinder	1.6x	18	1.5x	67	2.75x	32	1.40x	77	1.0x
srad	1.8x	3,3	0.4x	83	0.36x	9,9	1.67x	77	0.62x
streamcluster	2.0x	10	0.6x	67	0.20x	16	1.11x	89	0.14x

Table 3: Platform Area Results

	Zedboard Zynq 7020				Omnitek Zynq 7045			
Benchmark	FF (%)	LUT (%)	BRAM (%)	DSP (%)	FF (%)	LUT (%)	BRAM (%)	DSP (%)
Simple CUDA Kernels								
cp	28.7	57.4	72.9	100.0	21.2	40.2	62.4	62.2
fwt	30.0	77.0	40.0	60.0	36.4	90.3	52.8	76
matmul	46.6	86.4	17.0	80.0	41.2	69.0	17.6	78.2
dwt	38.1	82.3	30	97.3	34.0	66.9	28.6	92.4
Rodinia CUDA Benchmarks								
backprop	35.6	73.5	6.4	100	22.0	35.2	4.4	80.0
bfs	41.3	90.9	0.0	38.1	33.2	73.6	0.0	32.0
gaussian	35.4	85.2	0.0	98.2	23.7	49.5	0.0	85.1
hotspot	33.4	61.7	5.7	100.0	23.4	43.4	5.9	97.8
lavaMD	14.1	37.1	10.0	69.9	13.2	36.4	10.3	67.6
lud	39.5	68.7	13.6	93.2	29.9	57.6	13.3	87.6
nn	37.3	87.6	0.0	89.1	19.5	46.7	0.0	49.8
nw	39.7	82.8	21.4	27.3	37.4	75.9	22.0	26.7
particlefilter	42.3	86.2	0.0	36.8	10.5	21.6	0.0	9.0
pathfinder	29.7	70.0	19.3	49.1	22.0	42.4	22.4	21.3
srad	42.0	88.6	29.3	100.0	29.0	60.8	24.8	97
streamcluster	42.4	87.4	10.7	100.0	16.1	33.1	4.4	39.1

terconnect influence our abilities to optimize all of the other features. Although AXI interconnect IPs are quite efficient, it is still important to apply techniques to share ports and minimize the total amount of interconnect IP blocks needed. Using static information such as cores that cannot execute simultaneously can allow port sharing that may reduce the number of IP cores by 2× or more.

In embedded platforms, it is common to have multiple communications interfaces such as the high-performance and general-purpose AXI interfaces. It is important to partition signals between interfaces so that low-bandwidth signals such as control operations do not compete for bandwidth with high-bandwidth data signals. Using high-performance ports efficiently requires that only high-priority transactions use high-bandwidth. Effective use of available interfaces requires consideration of the frequency of operations on the interface and the relative bandwidth requirements.

Given an efficient interconnect, it remains critical to ensure that cores are using the interconnect efficiently. If multiple cores will use the same input data, using DMAs and local buffering to make data movement more efficient can substantially improve performance over identical networks that do not attempt to localize communications and minimize duplication of external memory requests. The interconnect presents an expensive component of the platform, especially if a core has multiple data ports, the resource consumption will escalate quickly as the design scales with many cores. Therefore, merging data ports reduces the number of interconnect which leaves more area for instantiating more cores in compensation of parallel data communication of a core.

It is also important to consider how the partitioning and interconnect technique affects CPU control. Using shared control signals can effectively start many cores at the same time, significantly reducing management overhead at a cost of lower ability to individually control cores. This tradeoff is typically reasonable for throughput-oriented designs where the common case is to use all cores simultaneously to perform portions of a larger task in parallel.

The input CUDA program must also adopt good programming practices for GPU programming in order to achieve better performance on FPGA. For example, ensuring memory coalescing access is important as it is equivalent to memory bursting which efficiently copies a chunk of data. Using

conditional or branching instructions inherently creates dependency which might prevent further parallelism such as unrolling. Exploiting local (shared) memory to cache data helps to reduce memory access latency. In addition, techniques to insert FCUDA pragmas to divide a kernel into multiple sub-tasks which overlaps the execution of those subtasks can potentially improve the core performance.

Having more cores does not always correlate to higher performance and performance/Watt. In reality, when an additional core is instantiated into the design, an amount of programmable fabric is spent on implementing interconnect to that core. The cost of this extra logic on area as well as power could outweigh the benefit of adding the core if it provides little benefit to performance. While this work only focuses on building a platform with as many cores as the FPGA can implement (depending on input workload) and the interconnect network between them, to attain better performance, it is recommendable that single core performance optimizations such as unrolling, array partitioning, pipelining via ping-pong buffers, bit-width optimization should also be explored.

7. CONCLUSION

We presented the first academic fully-automated platform generation flow for throughput oriented HLS. Our automated system uses a CUDA-to-RTL flow to generate initial platform designs, estimate maximal core instantiations, and generates optimized platforms for either bare-metal or OS-based platform integration. We demonstrated our automated flow on a Zedboard (Zynq 7020) and Omnitek (Zynq 7045) platform with 16 benchmarks including 4 simple benchmarks and 12 benchmarks from the Rodinia suite. We additionally compared performance and performance-per-Watt to a Jetson TK1 platform. Compared to the ARM CPU, our efficient platform integration achieves speedup in nearly all cases: with sometimes superior performance and performance-per-Watt than the Jetson TK1 despite no particular core optimization effort. Our open-source tool flow can be found at http://dchen.ece.illinois.edu/tools.html.

8. ACKNOWLEDGEMENT

This study is supported in part by the research grant for the Human-Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR).

9. REFERENCES

[1] M. Abdelfattah and V. Betz. The power of communication: Energy-efficient NOCS for FPGAS. In *FPL*, pages 1–8, Sept 2013.

[2] ALTERA. Altera SDK for OpenCL. http://www.altera.com/products/software/opencl/opencl-index.html.

[3] Altera. Avalon Bus Specification Reference Manual. 2003.

[4] ARM. AMBA AXI and ACE Protocol Specification. http://www.arm.com.

[5] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *OOPSLA*, pages 89–108, 2010.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Oct 2009.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE TCAD*, 30(4):473–491, April 2011.

[8] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From OpenCL to High-performance Hardware on FPGAS. In *FPL*, pages 531–534, Aug 2012.

[9] K. Eguro. SIRC: An Extensible Reconfigurable Computing Communication API. In *FCCM*, pages 135–138, May 2010.

[10] A. Ehliar and D. Liu. An FPGA Based Open Source Network-on-Chip Architecture. In *FPL*, pages 800–803, 2007.

[11] S. Fleming, D. Thomas, G. Constantinides, and D. Ghica. System-level Linking of Synthesised Hardware and Compiled Software Using a Higher-order Type System. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 214–217, New York, NY, USA, 2015. ACM.

[12] S. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen. High-level synthesis of multiple dependent CUDA kernels on FPGA. In *ASP-DAC*, pages 305–312, Jan 2013.

[13] S. T. Gurumani, J. Tolar, Y. Chen, Y. Liang, K. Rupnow, and D. Chen. Integrated CUDA-to-FPGA Synthesis with Network-on-Chip. In *FCCM*, pages 21–24, May 2014.

[14] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *FPL*, pages 1–8, Sept 2013.

[15] A. H. Lam. An Analytical Model of Logic Resource Utilization for FPGA Architecture Development. Master's thesis, University of British Columbia, 2010.

[16] A. Papakonstantinou, D. Chen, W.-M. Hwu, J. Cong, and Y. Liang. Throughput-oriented kernel porting onto FPGAs. In *DAC*, pages 1–10, May 2013.

[17] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *SASP*, pages 35–42, July 2009.

[18] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W.-M. W. Hwu, and J. Cong. Multilevel Granularity Parallelism Synthesis on FPGAs. In *FCCM*, pages 178–185. IEEE Computer Society, 2011.

[19] A. Parashar, M. Adler, K. Fleming, M. Pellauer, and J. Emer. LEAP: A virtual platform architecture for FPGAs. In *CARL*, 2010.

[20] M. Sharma and D. Kumar. Wishbone bus Architecture-A Survey and Comparison. *arXiv preprint arXiv:1205.1860*, 2012.

[21] J. Siegel, S; Kulp. OpenCPI HDL Infrastructure Specification. Tech Rep., 2010.

[22] S. Singh and D. J. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *FCCM*, pages 3–12. IEEE Computer Society, 2008.

[23] J. Stratton, S. Stone, and W.-m. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Languages and Compilers for Parallel Computing*, volume 5335 of *LNCS*, pages 16–30. 2008.

[24] K. Vipin, S. Shreejith, D. Gunasekera, S. Fahmy, and N. Kapre. System-level FPGA device driver with high-level synthesis support. In *FPT*, pages 128–135, Dec 2013.

[25] Xilinx. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[26] Xilinx. Xilinx SDAccel. http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[27] Xilinx. Xilinx SDSoC. http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html.

[28] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen. Fast and Effective Placement and Routing Directed High-level Synthesis for FPGAs. In *FPGA*, pages 1–10, 2014.