Throughput-Oriented Kernel Porting onto FPGAs

Alexandros Papakonstantinou
ECE Department
University of Illinois
Urbana-Champaign, IL, USA
apapako2@illinois.edu

Deming Chen
ECE Department
University of Illinois
Urbana-Champaign, IL, USA
dchen@illinois.edu

Wen-Mei Hwu
ECE Department
University of Illinois
Urbana-Champaign, IL, USA
w-hwu@illinois.edu

Jason Cong
CS Department
University of California
Los Angeles, California, USA
cong@cs.ucla.edu

Yun Liang
EECS School
Peking University
Beijing, China
ericlyun@pku.edu.cn

ABSTRACT

Reconfigurable devices are often employed in heterogeneous systems due to their low power and parallel processing advantages. An important usability requirement is the support of a homogeneous programming interface. Nevertheless, homogeneous programming interfaces do not eliminate the need for code tweaking to enable efficient mapping of the computation across heterogeneous architectures. In this work we propose a code optimization framework which analyzes and restructures CUDA kernels that are optimized for GPU devices in order to facilitate synthesis of high-throughput custom accelerators on FPGAs. The proposed framework enables efficient performance porting without manual code tweaking or annotation by the user. A hierarchical region graph in tandem with code motions and graph coloring of array variables is employed to restructure the kernel for high throughput execution on FPGAs.

1. INTRODUCTION

Tighter integration of latency oriented CPUs with throughput oriented compute architectures with massive parallelism and low power characteristics is becoming common in many compute domains (e.g. mobile, high-performance, compute clusters, etc) [1, 17, 16]. Programming efficiency is a prerequisite for leveraging the benefits of heterogeneous systems. The introduction of parallel programming models and semantics such as CUDA [15], OpenCL [2] and OpenACC [3] addresses the need for programming heterogeneous processors through a homogeneous programming interface. Homogeneous programming models facilitate functionality porting but often necessitate device-specific code tweaking to achieve performance porting.

In this work we propose a throughput oriented performance porting (TOPP) framework that leverages code restructuring techniques to enable automatic performance porting of CUDA kernels onto FPGAs. CUDA offers explicit control over (i) data memory spaces, (ii) computation distribution across cores, and (iii) thread synchronization. Hence, CUDA kernels designed for the GPU architecture may not map efficiently on reconfigurable devices. The TOPP framework proposed in this work, leverages the hierarchical region graph (HRG) representation to efficiently analyse and restructure the kernel code. Restructuring entails a wide range of transformations including code motions, synchronization elimination (through array renaming), data communication elimination (through rematerialization), and idle thread elimination (through control flow fusion and loop interchange). As data handling plays a critical role in the performance of massively parallel CUDA kernels, the proposed flow employs advanced dataflow and symbolic analysis techniques to efficiently manage data. Graph coloring in tandem with throughput estimation techniques is used to optimize kernel data structure allocation and utilization of on-chip memories. Through orchestration of different code transformation and optimization techniques, the TOPP framework generates C code which is fed to high-level synthesis (HLS) to generate high-throughput custom accelerators on the reconfigurable architecture. Our experimental study shows that the proposed flow improves FPGA execution performance by more than 4X without manual code tweaking from the user.

The main contributions of this work are summarized below:

- Introduction of the hierarchical region graph representation of CUDA kernels.
- Implementation of an automated performance porting flow from CUDA to FPGAs.
- Description of efficient throughput metrics for throughput oriented kernel restructuring.
- Experimental evaluation of the performance porting capability of the TOPP framework.

In the next Section we provide further background information on CUDA-to-FPGA flows and introduce the HRG representation. Section 3 offers an overview of the TOPP framework which is complemented by algorithms and other implementation details in the Appendices. Finally, Section 4 contains the experimental evaluation of TOPP followed by conclusion in Section 5.

2. MOTIVATION AND BACKGROUND

CUDA employs a SIMT (single instruction, multiple threads) parallel programming interface which efficiently expresses multiple fine-grained threads executing as groups of cooperative thread arrays (CTA). The GPU architecture comprises high-throughput compute cores grouped in Streaming Multiprocessors (SMs). Computation is distributed across SMs at CTA granularity [15]. A carefully crafted interconnect scheme between SMs and off-chip memory facilitates high-bandwidth data accesses at low latency overhead.

DAC '13, May 29 - June 07 2013, Austin, TX, USA.

Listing 1: CUDA code for DWT kernel

```
1  for(tid=0;tid<bdim;tid++){
2    shr[tid] = id[idata];
3    __syncthreads();
4    data0 = shr[2*tid];
5    __syncthreads();
6    od[tid_global] = data0*SQ2;
7    shr[tid] = data0*SQ2;
8    __syncthreads();
9    numThr = bdim >> 1;
10   int d0 = tid * 2;
11   for(int i=1;i<lev;++i){
12     if( tid < numThr){
13       c0 = id0+(id0>>LNB);
14       od[gpos] = shr[c0]*SQ2;
15       shr[c0] = shr[c0]*SQ2;
16       numThr = numThr>>1;
17       id0 = id0<<1; }
18     __syncthreads();  } }
```

Listing 2: C code for DWT kernel

```
1  for(tid=0;tid<bdim;tid++){
2    shr[tid] = id[idata]; }
3  for(tid=0;tid<bdim;tid++){
4    d0[tid] = shr[2*tid];}
5  for(tid=0;tid<bdim;tid++){
6    od[tid_glob] = d0[tid]*SQ2;
7    shr[tid] = d0[tid]*SQ2;}
8  for(tid=0;tid<bdim;tid++){
9    numThr = bdim >> 1;
10   id0[tid] = tid * 2;}
11 for(int i=1;i<lev;++i){
12   for(tid=0;tid<bdim;tid++){
13     if(tid < numThr){
14       c0 = id0[tid]+(id0[tid]>>LNB);
15       od[gpos] = shr[c0]*SQ2;
16       shr[c0] = shr[c0]*SQ2;
17       numThr = numThr>>1;
18       id0[tid] = id0[tid]<<1; }}}
```

Reconfigurable devices, on the other hand, offer grids of fine-grained compute and storage resources that can be synthesized into parallel processing custom cores (CCs) at different granularities. FPGAs offer the benefit of application-driven compute customization at the cost of area overhead for reconfigurability. HLS flows enhance FPGA design efficiency by facilitating fast and easy design at higher abstraction. Achieving high-throughput implementations on FPGA requires cautious allocation and coordination of the available compute and storage resources. This depends heavily on parallelism expression and organization in the input code of HLS design flows. The TOPP framework combines advanced code transformations to enable high-throughput designs in CUDA-to-RTL HLS flows.

2.1 SIMT-to-C Compilation

Previous works have described SIMT-to-C (S2C) compilation flows porting kernels onto multicore CPUs [23] and FPGAs [20, 22, 19, 13]. A common characteristic of these S2C flows is the expression of threads as loops over the CTA thread ID (tID), hereafter referred to as tID-loop. Thread synchronization is enforced through loop fission (e.g. tID-loop in line 1 of List. 1 is split into 5 loops in List. 2), loop interchange (e.g. loops in lines 11, 12 in List. 2) and variable privatization transformations (e.g. `d0` in line 4 of List. 2). Thread-loop unrolling in tandem with vector loads/stores may be used to exploit the CUDA thread parallelism in the kernel.

Kernel decomposition into computation (COMP) and communication (COMM) tasks has been proposed in [20, 22, 21]. Communication tasks comprise data transfers to/from off-chip memory. Task decomposition is critical in optimizing CTA execution latency on the reconfigurable fabric. Aggregating off-chip memory accesses across CTA threads within COMM tasks facilitates efficient off-chip memory bandwidth through data transfer bursts. Decomposition may also benefit COMP task latency by eliminating data fetch latency through data prefething. The kernel decomposition proposed in [20, 22, 21] is based on user-injected annotations that assist the compiler in identifying COMP and COMM tasks.

This work employs the kernel decomposition philosophy but eliminates the need for user-injected annotations. The proposed flow leverages sophisticated analysis and transformation techniques to identify and re-organize kernel tasks so as to optimize execution throughput on the FPGA architecture.

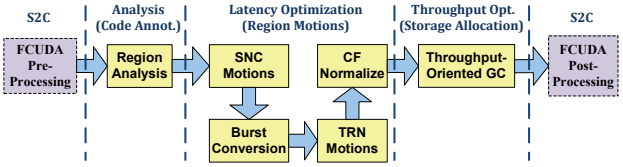


Figure 1: TOPP framework integrated in FCUDA

We will use the Nvidia SDK kernel for discreet wavelet transforms (DWT) as a running example to motivate the importance of throughput-oriented performance porting (TOPP). The DWT kernel (List. 1) contains thread-dependent control flow (line 12), thread synchronization directives (line 3, 5, 8, 18), and intermingled computation and communication regions/statements (lines 6,14) which render manual task annotation cumbersome. Moreover, code restructuring may be required to eliminate kernel fragmentation into fine grained COMP/COMM tasks; e.g. tsk(6), tsk(7), tsk(9,10), etc. in List. 1, where tsk($x[,y]$) denotes the task contained within line(s) x (to y). Fine-grained tasks can negatively impact performance through (i) overhead of implicit thread synchronization across tasks and (ii) increased storage overhead due to variable privatization for variables referenced across tasks (e.g. `data0` is referenced in tsk(4) and tsk(6) and hence it is privatized with respect to `tid`).

The proposed framework leverages rigorous analysis and transformations in tandem with throughput estimation techniques to maximize execution throughput on FPGA. With regard to previous works, the proposed flow elevates the importance of data communication and storage in execution throughput and tries to balance task latency optimization with memory resource allocation for each task in order to maximize kernel execution throughput. Dataflow, value range and symbolic expression analysis in tandem with efficient code motions, and memory allocation optimizations are applied in a phased approach depicted in Figure 1. TOPP is integrated in the FCUDA flow [20, 21] and comprises three major phases: (i) kernel analysis (code analysis and annotation), (ii) task latency optimization (code restructuring) and (iii) throughput optimization (efficient storage allocation). A hierarchical region graph (HRG) representation of the kernel is built during the analysis phase and it is used throughout the subsequent transformation stages.

2.2 Hierarchical Region Graph (HRG)

Hierarchical task graphs (HTGs) have been previously proposed for code representation as a means of extracting parallelism in compilers [10] and HLS flows [12]. In these works the HTG is generated from a sequential low-level 3-address representation of the application and incorporates control and data dependence information along with control-flow hierarchy. The HRG, on the other hand, is generated from high-level SIMT code and summarizes the computation, communication and synchronization characteristics of the application along with data and control flow dependence. The HRG represents the kernel as a tree graph $G_{HRG} = (V, E)$, where each leaf vertex, $v_l \in V$, represents a code region of type t_r and each internal vertex, $v_h \in V$, represents a control-flow (CF) structure of type t_f. Fig. 2 depicts the HRG for the DWT kernel. Region type, t_r, specifies whether the leaf node corresponds to a compute (CMP), communication (TRN), or synchronization (SNC) region. Control-flow type, t_f, identifies the dependence of the CF condition expression from thread ID (tID). Double-rimmed (purple) nodes represent tID-variant (TVAR) CF whereas single-rimmed (green) nodes represent tID-invariant (TiVAR) CF. HRG edges, E, include a set of hierarchy edges, E_H (non-dashed edges connecting nodes into a tree, which denote CF structure), and a set of data dependence edges, E_D (dashed edges). Control flow dependencies

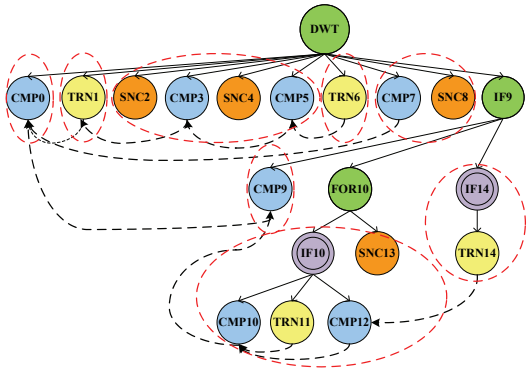
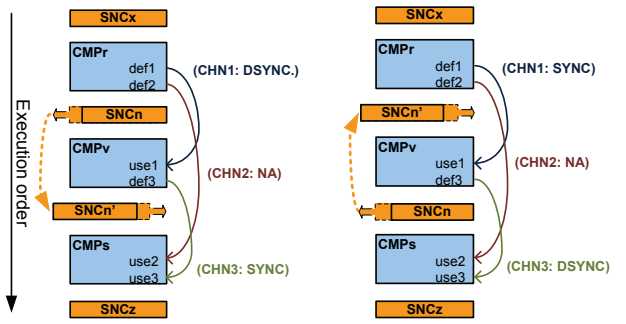


Figure 2: HRG for DWT kernel



(a) Forward SNC shift (b) Backward SNC shift

Figure 3: SNC region motions

can be extracted by a depth-first traversal of the HRG tree (non-dashed edges), i.e. child nodes are ordered in control-flow order.

The HRG summarizes the kernel region organization and enables easy and throughput-driven kernel decomposition into COMP and COMM tasks through depth-first traversals (DFT) of the HRG tree. Moreover, it facilitates efficient feasibility and cost/gain analysis of different code transformations. Hence, transformations may be evaluated on the HRG representation before being applied on the SIMT code. Sequences of transformations implemented on the HRG can be applied to source code by translating the resulting HRG tree to C or CUDA code. The algorithm for CUDA-to-HRG translation is described in Appendix A.

3. TOPP FRAMEWORK OVERVIEW

TOPP has been implemented in the FCUDA flow [20, 21], as an interleaved sequence of analysis and transformation phases (Fig. 1). An overview of the integration of TOPP in FCUDA is provided in Appendix B. TOPP analysis and transformation phases are discussed in the following subsections.

3.1 Analysis Phase

Region analysis, (Fig. 1), identifies the CMP, TRN and SNC regions of the kernel and annotates each statement with region and thread-variance (TVAR) information. The annotated information is used in the generation of the HRG. The analysis process is carried out as a sequence of six steps: (*A1*) Identify global memory accesses, (*A2*) Normalize multi-type statements, (*A3*) Build Def-Use chains [18], (*A4*) Find tID-variant (TVAR) statements, (*A5*) Annotate TVAR statements, and (*A6*) Build the kernel HRG. Initially global memory variables are identified and all global memory references are collected in step *A1*. Global memory variables include CUDA `__constant__` variables and C-pointer parameters of the kernel procedure, as well as all of their alias definitions through pointer arithmetic. During step *A2*, kernel statements are scanned for multi-type statements, i.e. statements entailing both CMP and TRN operations. Each such statement is converted into separate single-typed CMP and TRN statements. Subsequently, dataflow analysis is used to build Def-Use chains (step *A3*) which facilitate tID-variant (TVAR) variable and statement identification during step *A4* and tagging during step *A5* (a statement is tagged as TVAR, if it contains the definition of a TVAR variable). Finally the HRG is constructed in step *A6* (Alg. 1) based on the analysis information annotated on the kernel statements.

3.2 Latency Optimization Phase

The latency optimization phase in TOPP comprises different region motion stages which aim to eliminate the execution latency overhead resulting from excessive (i) CMP and TRN interleaving, and (ii) synchronization directives. Hence, the goal of the transformations applied in this phase is to reduce CTA execution latency through region reorganization so as to enable the creation of coarser COMP and COMM tasks. As an example we can use the organization of regions in the initial DWT HRG (Figure 2) which can be arranged into eight tasks (marked with red dashed circles). Since each task is outlined in a separate task procedure in the FCUDA flow, task boundaries represent implicit synchronization points (ISPs) imposing synchronization overhead and bounding ILP extraction space at the thread level. Moreover, multiple fine-grained tasks result in extra TVAR variables with ISP-crossing lifetimes. This is dealt in FCUDA with variable privatization along the tID dimension, leading to higher BRAM resource usage (e.g. variables `d0` and `id0` in List. 1 are privatized after task decomposition in List. 2). The TOPP framework considers the impact of privatization on BRAM allocation and employs region motions and merging to reduce ISP count.

The HRG in tandem with the annotated Def-Use chain information plays a critical role in region motion feasibility analysis and cost/gain estimation during this optimization phase. Each Def-Use chain that crosses multiple regions is characterized as either *thread shared chain* (TSC) or *thread private chain* (TPC). TSCs refer to chains corresponding to `__shared__` or `global` variables, where explicit synchronization between the definition region and the use region may be required (e.g. chain corresponding to def and use of `__shared__` variable `shr` in lines 2 and 4, respectively of List. 1; represented with dependence edge between TRN1 and CMP3 regions in Figure 2). TPCs, on the other hand, correspond to variables that host values read by the thread that wrote them (i.e. same def and use thread per value) which are not affected by CTA synchronization dependence-wise. Nonetheless, synchronization might affect the storage allocation of TPC variables as discussed earlier. Hence, TSCs affect the feasibility of region motions, whereas TPCs affect the cost/gain estimation analysis of region motions. There are three possible effects that region motions may have on Def-Use chains: (i) Desynchronization (DSYNC), (ii) Synchronization (SYNC), or (iii) Not affected (NA). Desynchronization happens in the case that the explicit or implicit synchronization points between source and sink regions of a chain are removed. For example, chain CHN1 comprised of def1 and use1, in Fig. 3(a), is desynchronized when SNCn region is shifted below CMPv becoming SNCn'. Correspondingly, chain CHN3 between def3 and use3 is synchronized for the same motion of SNCn, whereas CHN2 is not affected by this region motion. Determining which case a region motion corresponds to, is based on the partial ordering enforced by the region identifiers (rIDs) of the involved regions.

The feasibility of a region motion with respect to a TSC is determined by the motion effect on the chain (i.e. DSYNC, SYNC or NA) in combination with the value of its dependence distance vector [5]. Specifically, in case of SYNC or NA motion effects on the TSC, feasibility is positive regardless of the

dependence vector distance (e.g. CHN2 and CHN3 in Figure 3(a)). However, in case of DSYNC motion effect on the TSC, the dependence distance vector needs to be examined in order to determine feasibility. We leverage the work in [11] and extend it by applying dependence distance vectors in determining region motion feasibility. Specifically, the authors of [11] show that it is feasible to remove implicit synchronization points (ISPs) between the source and sink of a Def-Use chain as long as one of the following rules holds with respect to the chain's distance dependence vector v:

- $v[0] == 0$
- $v[0] < 0 \ \wedge \ v[1:(|v|-1)] == 0$
- $v[0] == v[i] : i \in [1:(|v|-1)] \ \wedge \ v[1:i] == 0$

where $v[0]$ corresponds to the index of the tID-loop and $v[0] < 0$ denotes an inter-thread data dependence. For the purpose of determining the feasibility of a region motion we also apply this test to *explicit synchronization points* (ESPs). Distance vectors are evaluated leveraging symbolic analysis ([18]) in combination with range analysis ([6, 9]) and array dependence analysis ([18, 5]). If none of the conditions can be proven, feasibility is not confirmed and the corresponding region motion is rejected. In each of the TRN motions and SNC motions stages, cost-function based evalution is used to quantify the benefit of a motion with regard to the following factors:

- De-synchronized TPCs gain
- Synchronized TPCs cost
- Explicit synchronization point (ESP) elimination gain
- Implicit synchronization point (ISP) overhead cost

Appendix C discusses in further detail the transformation stages in the latency optimization phase of TOPP.

3.3 Througput Optimization Phase

During this phase TOPP leverages throughput estimation techniques along with resource information to guide kernel restructuring. Hence, the optimization goal is shifted toward CTA grid execution throughput (vs. CTA execution latency, previously), taking into account the available resource on the target device.

3.3.1 Throughput Factors and Metrics

Throughput of system configuration C with N custom cores (CCs), TP_C, can be expressed as: $TP_C = \frac{EP_N}{cp}$, where EP_N represents the cumulative CTA execution progress across all CCs completed per clock period, *cp*. For the purpose of throughput-oriented kernel restructuring we leverage the clock period selection feature offered by the HLS engine used in our flow. That is, the generated RTL is pipelined according to the selected clock period, and operation cycle latencies are adjusted accordingly. We have created cycle latency tables (CLT_{cp}) by characterizing operation cycle latencies for different clock periods (*cp*). These tables are used in TOPP to estimate cycle latency and throughput for a chosen clock period. Hence, the CTA execution throughput metric can be expressed in terms of cycle latencies as: $TP_C = N_{CC} \div (CL_{COMP} + CL_{COMM})$, where configuration C has N_{CC} cores with compute and communication task cycle latencies of CL_{COMP} and CL_{COMM}, respectively. The number of cores, N_{CC}, is estimated for the selected FPGA device based on (i) the number of arrays required per CTA by configuration C and (ii) resource allocation feedback provided from the HLS engine. Latencies CL_{COMP} and CL_{COMM} are calculated as the sums of the sequential CMP and TRN region latencies per CTA in configuration C, respectively: $CL_{COMP} = \sum_i CL_{CMP_i}$, and $CL_{COMM} = \sum_j CL_{TRN_j}$. Concurrent tasks are represented by the latency of the longer task (the HLS engine schedules tasks in a bulk synchronous way; tasks may either start concurrently, if not dependent, or sequentially, otherwise.) Cycle latency CL_{CMP_i} of compute region CMP_i, is estimated by determining the task's critical execution path. Def-Use chains are used for identifying the critical execution path, while operation cycle latencies are referenced from CLT_{cp} tables. Cycle latency estimate, CL_{TRN_j}, of data transfer region TRN_j is affected by two main factors: (i) the on-chip memory bandwidth and (ii) the off-chip memory bandwidth. The former is estimated based on the on-chip SRAM memory port bandwidth (BW_S), the execution frequency and the read/write data volume. The latter depends on the off-chip DDR memory system peak bandwidth, (BW_D), provided by the user, the extent of static coalescing achieved by the burst conversion stage in the latency optimization phase and the read/write data volume of the task. The final COMM task latency is calculated as $CL_{TRN_j} = max(CL_{SM_j}, CL_{DM_j})$, where CL_{SM_j} corresponds to the on-chip memory access latency and CL_{DM_j} corresponds to the off-chip memory access latency. As described above, CL_{SM_j} is mainly dependent on the architecture of the chosen configuration, C, while CL_{DM_j} is mainly constrained by the value of BW_D provided by the user.

3.3.2 Throughput-Driven Graph Coloring

Graph coloring is often used in compilers for the allocation of registers to program variables and temporary values [8, 7], due to its ability to lead to efficient solutions. Registers represent the most scarce but efficient storage resource at the topmost level of memory hierarchy and thus good register allocation is critical to performance. The SIMT programming model used in FCUDA offers visibility of different memory address spaces with different memory attributes. The goal of the throughput-driven graph coloring (TDGC) transformation in TOPP is to enhance the allocation of kernel arrays onto FPGA Block-RAM (BRAM) memories, considering both kernel characteristics and resource availability. The proposed TDGC algorithm leverages the throughput metrics described in Section 3.3.1 to optimize performance through efficient (i) allocation of arrays onto BRAMs and (ii) off-chip data transfer scheduling.

TDGC entails three main steps: (*GC1*) Array coloring, (*GC2*) Throughput estimation, and (*GC3*) Data communication task (COMM) rescheduling. The three steps may be iterated until no more throughput improving rescheduling alternatives are available. In most cases, the number of iterations is small (not exceeding 3). Initially, candidate arrays for allocation are identified (step *GC1*) and an interference graph, G_I, is generated (Fig. 12(a)). Vertices in G_I correspond to array lifetimes, whereas edges represent overlapping array lifetimes in the kernel. The interference graph, G_I, is colored using a modified R-coloring [18] algorithm (R represents the number of BRAMs per CTA). Coloring determines a BRAM allocation configuration which is used in step *GC2* to estimate throughput using the metric discussed in 3.3.1. The number of instantiated CCs, N_{CC}, is determined based on the BRAM allocation selected in step *GC1* and resource estimation feedback from the HLS with respect to other type of resources. If BRAM turns out to be the throughput limiting resource (i.e. it constrains N_{CC}), we employ COMM task rescheduling in step *GC3* as a means to reduce BRAM requirements. Specifically, G_I nodes are characterized based on their interference degree, L_{ID}, and their *idle lifetime intervals* (ILI), L_{II}; we define as idle the intervals of an array lifetime that correspond to HRG regions where the array is not accessed. Subsequently, nodes are sorted with respect to *lifetime scatter*: $L_S = L_{ID} * \frac{L_{II}}{L_T}$, where L_T represents the total lifetime interval. Nodes are examined in decreasing L_S order with regard to the feasibility of reducing their ILI (and subsequently their interference degree) through TRN region

motions and the benefit of such motions in the interference degree of the G_I graph. If a node fulfilling these requirements is found, the HRG is modified and the TDGC steps reiterated until no further candidate nodes are available. At each iteration of the TDGC steps, the TP_C of the new configuration is estimated (step *GC2*) and the TRN region motion is committed only for configurations with higher TP_C (See Appendix D for further details in TDGC and R-coloring).

4. EXPERIMENTAL EVALUATION

TOPP framework is implemented within the FCUDA flow and its analysis phase essentially replaces the (manual) annotation task (Fig. 7). Moreover, the latency and throughput optimization phases of TOPP apply performance oriented code restructuring prior to compiling the SIMT code into explicitly parallel C code for the HLS engine. The HLS engine integrated in the flow is Vivado-HLS [4], which is the successor of AutoPilot [24] used in [20, 22]. The CUDA kernels used in our experiments have been selected from the Nvidia SDK [15] suite. Our experimental evaluation is centered around exposing the effect of the employed TOPP transformations on performance. Specifically, in the next section we measure the performance impact from the individual latency oriented transformations on execution. Subsequently, the effectivenes of the metric used to guide throughput optimization is tested in Section 4.2. Finally, we evaluate the total kernel execution speedup achieved by integrating TOPP into FCUDA, in Section 4.3.

4.1 Latency Optimization Evaluation

First we evaluate the effect of the transformations applied during the latency optimization phase (Figure 1) on CTA compute task latency. Specifically, we measure the effect of each individual transformation on latency by disabling the TDGC transformation in the throughput optimization phase of TOPP and enabling only the desired latency transformations in the latency optimization phase. Initially we enable only the SNC motions (SM) transfomation and measure the relative speedup of the compute latency over FCUDA. We gradually enable the other transformations of the latency optimization phase in the order executed within TOPP (1) and compare the cummulative speedups achieved over the original FCUDA flow in [20] (Fig. 4). The speedup achieved by each set of enabled latency transformations depends on the kernel and code structure characteristics. Kernels that either contain long dataflow paths (e.g. FWT2) or more convoluted control flow paths (e.g DWT) offer more opportunities for optimization. We observe that TRN motions (TM) can have significant impact in the compute latency (e.g. FWT2 and DWT). This is due to enabling the generation of coarser COMP tasks by shifting interleaved TRN regions. It is interesting to observe that burst conversion (BC) results in good speedups for some kernels (e.g. FWT1 and FWT2), even though COMM task latency is not considered in Fig. 4. This is mainly due to the address calculation simplification from consolidating the

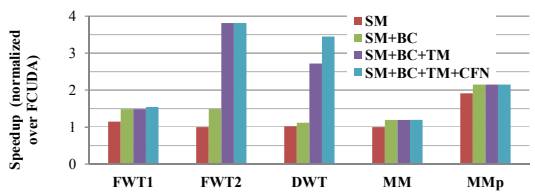


Figure 4: Execution speedup over FCUDA from cummulative application of latency transformations: (i) SNC region motions (SM), (ii) burst conversion (BC), (iii) TRN region motions (TM), and (iv) control flow normalization (CFN).

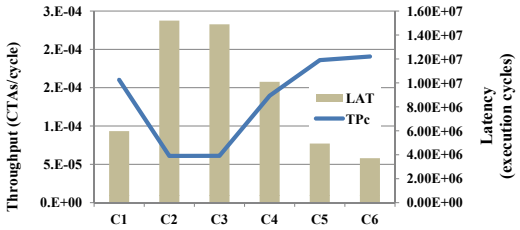


Figure 5: Effectiveness of TP_c metric (Left axis shows TPc value, right axis shows execution latency).

memory address computation from all the threads into burst address computations at the CTA level. On the other hand, SNC region motions do not seem to affect compute latency in a considerable way. However, they enable elimination of excessive variable privatization during FCUDA postprocessing, optimizing BRAM resource per CC, and hence throughput. Finally, note that the MMp kernel is an optimized version of the MM kernel derived through loop pipelining during HLS. The MMp speedup values are normalized over FCUDA latency of MM kernel. This shows that TOPP and HLS optimizations can be applied cummulatively.

4.2 Throughput Metric Evaluation

Here we measure the correlation of the throughput estimation metric to the actual execution latency. For this purpose we use the DWT kernel that has served as a running example throughout the previous sections. Specifically, intermediate configurations Ci, of DWT during compilation through the TOPP stages are extracted and fed to FCUDA postprocessing stage to collect execution results. TP_C is calculated for each configuration and it is depicted with execution latency results in Fig. 5. The gray bars correspond to execution latency, whereas the blue line corresponds to calculated TP_C values. We can observe the inverse correlation between the two performance metrics. This shows the effectiveness of the throughput metric in guiding the selection of high-performance configurations during the throughput latency phase.

4.3 TOPP vs. FCUDA

This evaluation measures the total kernel execution speedup achieved with TOPP over FCUDA [20]. Here, all transformations are enabled in both latency and throughput optimization phases of TOPP. In order to evaluate the performance effect of the code transformations in TOPP, we target the same execution frequency for all the kernels. Thus, we eliminate the fuzziness induced by the effect of synthesis and place-and-route optimizations on different RTL structures. Instead, we synthesize all the kernels at 200MHz, but run them at 100MHz to ensure that routing will not affect our evaluation (note that overconstraining the clock period during synthesis is a common practice in industry, in order to absorb the frequency hit from routing delays). In terms of memory interface and bandwidth we model in our evaluation a similar memory interface as the one used in the [14] hybrid computer, where the compute-acceleration FPGA leverages the high-speed serial tranceivers to transfer data to off-chip memory controllers that support high-banwidth DDR memory accesses.

Figure 6 depicts the speedup of the TOPP-compiled kernels against the FCUDA-compiled ones. The FP-SX50 and FP-SX95 bars use floating point kernels and target SX50T and SX95T Virtex-5 devices, respectively. The third bar (INT-SX50) uses integer kernels and targets device SX50T. Each bar is normalized against the execution latency of FCUDA for the same device and kernel. We can observe that the speedup achieved on the bigger SX95T device is slightly lower than the SX50T (even though in absolute terms latency on SX95T is lower from latency on SX50T). The main reason for this trend

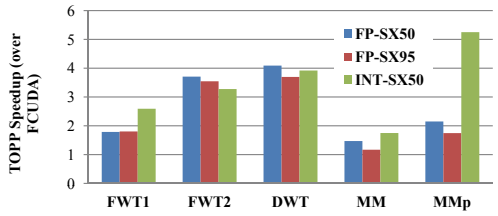


Figure 6: TOPP execution speedup over FCUDA

is due to the fact that the compute/memory resource capacity of SX95T is 1.8X higher then SX50T resource capacity, but the off-chip bandwidth of SX95T is only 50% higher than the off-chip bandwidth in SX50T thus limiting the speedup that can be achieved by the TOPP transformations. With regard to speedup of the integer kernels, this is similar to speedup for floating point kernels in most cases. FWT1 and MMp stand out for different reasons; FWT1 optimizes away integer multipliers for powers of two, while MMp expoits loop pipelining more efficiently with integer operations (Note: the MMp kernel speedup is here, also, normalized against the FCUDA-compiled latency of MM kernel.)

Finally, comparing the speedup corresponding to bars FP-SX50 with the compute latency results in Section 4.1, we can observe that the performance advantage of the TOPP flow is further improved. This is partially due to the better allocation of BRAMs achieved by the TDGC stage and partially due to more efficient exploitation of the off-chip memory bandwidth (i.e. transfers can be more efficiently disentangled from compute and converted to bursts).

5. CONCLUSIONS

In this paper we present the throughput-oriented parallelism synthesis (TOPP) framework which aims to provide throughput-oriented performance porting of CUDA kernels onto FPGAs. The techniques applied in this work could potentially be employed in other application programming interfaces with similar SIMT programming semantics that target heterogeneous compute systems (e.g. OpenCL [2]). Our experimental evaluation demonstrates the effectiveness of performance porting achieved through orchestration of advanced analysis with latency and throughput optimizations in the TOPP framework.

As computing is moving toward massively parallel processing for *big data* applications, it is critical to increase the abstraction level of optimization and transformation techniques. Representing and leveraging application algorithms at a higher level is crucial for delivering high throughput and high performance in massively-parallel compute domains. In this work, we have dealt with the issue of raising the abstraction level in the field of high-level synthesis of parallel custom processing cores. We have developed efficient throughput estimation and optimization techniques that improve performance beyond thread latency by dealing with conflicting performance factors at the CTA level and managing the compute and storage resources accordingly.

6. ACKNOWLEDGMENTS

This work is partially supported by the Gigascale Systems Research Center (GSRC) and Intel Corporation. We also thank Steven Burns, Mustafa Ozdal, Kanupriya Gulati and Taemin Kim of Intel and Kyle Rupnow of ADSC (Illinois Center in Singapore) for their helpful comments and discussions.

7. REFERENCES

[1] AMD Fusion family of APUs: Enabling a superior, immersive PC experience. White Paper. `http://sites.amd.com/us/Documents/48423B\_fusion\_whitepaper\_WEB.pdf`, Mar. 2010.

[2] The OpenCL specification. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, Sept. 2010.

[3] The OpenACC application programming interface. `http://www.openacc.org/sites/default/files/OpenACC.1.0\_0.pdf`, Nov. 2011.

[4] Vivado design suite user guide: High-level synthesis. UG902(v2012.2). `http://www.xilinx.com/support/documentation/sw\_manuals/xilinx2012\_2/ug902-vivado-high-level-synthesis.pdf`, July 2012.

[5] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, first edition, 2002.

[6] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expression. In *Proc. ACM/IEEE Conf. on Supercomputing (SC'94)*, Nov. 1994.

[7] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Prog. Languages and Systems*, 16(3):428–455, May 1994.

[8] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices - Best of PLDI 1979-1999*, 39(4):66–74, Apr. 2004.

[9] C. Dave, H. Bae, S. J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42(12):36–42, Dec. 2009.

[10] M. Girkar and C. Polychronopoulos. Extracting task-level parallelism. *ACM Transactions on Prog. Languages and Systems*, 17(4):600–634, 1995.

[11] Z. Guo, E. Z. Zhang, and X. Shen. Correctly treating synchronizations in compiling fine-grained spmd-threaded programs for cpu. In *Proc. ACM Int'l Conference on Parallel Architectures and Compilation Techniques (PACT'11)*, Sept. 2011.

[12] S. Gupta, R. Gupta, and N. Dutt. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):441–470, 2004.

[13] S. Gurumani, K. Rupnow, Y. Liang, H. Cholakkail, and D. Chen. High level synthesis of multiple dependent CUDA kernels for FPGAs. In *Proc. IEEE/ACM Asia and South Pacific Design Automation Conference*, Jan. 2013.

[14] The Convey HC-1: The world's first hybrid core computer. Datasheet. `http://www.conveycomputer.com/Resources/HC-1\%20Data\%20Sheet.pdf`, 2009.

[15] CUDA: Parallel programming and computing platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[16] Zynq-7000 all programmable SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm`, 2012.

[17] Tegra super processors. `http://www.nvidia.com/object/tegra-4-processor.html`, 2013.

[18] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, first edition, 1997.

[19] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos. Synthesis of platform architectures from opencl programs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*, May 2011.

[20] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W. Hwu. FCUDA: enabling efficient compilation of cuda kernels onto FPGAs. In *Proc. IEEE Symposium on Application Specific Processors*, June 2009.

[21] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W. Hwu. Efficient compilation of CUDA kernels for high-performance computing on FPGAs. *ACM Transactions in Embedded Computing Systems*, Vol. 13, 2014.

[22] A. Papakonstantinou, Y. Liang, J. Stratton, K. Gururaj, D. Chen, W. Hwu, and J. Cong. Multilevel granularity parallelism synthesis on FPGAs. In *Proc. IEEE Int'l Symposium on Field-Programmable Custom Computing Machines*, May 2011.

[23] J. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore cpus. In *Proc. ACM Int'l Symposium on Code Generation and Optimization (CGO'10)*, Feb. 2010.

[24] Z. Y. Zhang, F. W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based ESL synthesis system. In P. Coussy and A. Moraviec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter 6, pages 99–112. Springer, 2008.

APPENDIX

A. HRG GENERATION

As described in Section 3.1 the HRG is generated during step *A6* in the analysis phase. Having identified the global memory accesses in step *A1* and split the multitype statements into single-type ones in step *A2*, a depth-first traversal (DFT) is carried out on the kernel AST (abstract syntax tree) representation used in the FCUDA compiler [9]. DFT is implemented by the recursive `hrgGen()` procedure (Alg. 1) which classifies each kernel statement as TRN, CMP, SNC or CF. Non CF statements are collected into a list (*sLst*) which is used to build HRG leaf nodes from statements of same type (`sTyp()`) and same AST level (*lvl*). Each region is assigned a region ID (rID) which helps maintain partial ordering of the HRG nodes. CF statements form HRG internal nodes by themselves and get assigned the smallest rID of their child leaf nodes (Alg. 1). Region IDs infer execution ordering and facilitate region motion feasibility checks during latency and throughput optimization phases of TOPP. The generated HRG is also structured as an AST and each node contains pointers to the code statements summarized by the HRG node. Hence, it is easy to reconstruct a new kernel AST from an optimized HRG. Finally, a similar traversal of the HRG tree is used to group HRG nodes into tasks that satisfy two rules, (i) a task may contain control-flow (CF) nodes as long as every child node of a contained CF node is also included in the task, and (ii) a task may contain nodes across different control-flow hierarchy levels as long as the corresponding CF nodes are also included in the task (e.g. grouping nodes CMP12 and SNC13 in Fig. 2 within the same task is only allowed if nodes IF10, CMP10 and TRN11 are also included in the task). Note that HRG leaf nodes are grouped into tasks based on their type; COMM tasks comprise only TRN nodes whereas COMP tasks may include CMP and SNC nodes.

Algorithm 1: HRG generation

```
/* hrgGen(hAST,level,cID,sLst):  Generate HRG
through code depth first traversal (DFT)    */
Input: Abstract syntax tree of code hierarchy hAST
Input: Level in kernel lvl, region ID pID
Output: region ID rID
Output: region statement list: sLst
1  cID ← pID                              // Update current ID
2  cTyp ← -1                          // Invalidate current type
3  while S ← next(hAST) do     // Get next statement
4      switch typ ← sTyp(S) do          // get type of S
5          case TRN                          // S type is TRN
6              if typ ≠ ctyp then      // Different region
7                  n ← hrgNod(sLst)       // New HRG node
8                  pn ← getNod(hAST)      // parent node
9                  addChld(n, pn)         // Link n to pn
10                 empty(sLst)              // Clean sLst
11                 cID ← cID + 1      // Update region ID
12             annot(S, TRN)                 // Annotate S
13             annot(S, cID)                 // Annotate S
14             annot(S, lvl)                 // Annotate S
15             push(sLst,S)      // Push S into sLst list
16             cTyp ← typ                    // Update type
17         case CMP                        // S type is CMP
18             ...                  // Similar to TRN case
19         case SNC                        // S type is SNC
20             ...                  // Similar to TRN case
21         case CF                           // Control flow
22             n ← hrgNod(sLst)              // New HRG node
23             pn ← getNod(hAST)    // Get parent node
24             addChld(n, pn)                // Link n to pn
25             empty(sLst)                     // Clean sLst
26             cAST ← getAST(S)              // get AST of S
27             annot(S, cID)      // Annotate S with cID
               // Recurse for next AST level
28             cID ← hrgGen(cAST,lvl+1,cID,sLst)
29             n ← hrgNod(S)                 // New HRG node
30             addChld(n, pn)                // Link n to pn
31             return cID
```

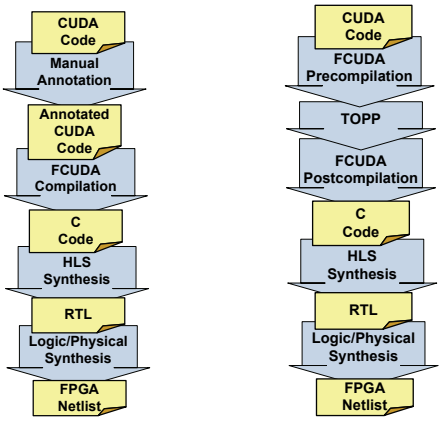
B. FCUDA FLOW DETAILS

FCUDA (Fig. 7(a)) provides the underlying basis flow on which TOPP is built to provide throughput-oriented kernel restructuring. The proposed flow (Fig. 7(b)) leverages FCUDA utilities during preprocessing and postprocessing stages for conditioning the input CUDA code and translating the TOPP output into parallel C code, respectively (Fig. 1). Moreover, integration of TOPP in FCUDA removes the burden of manual annotation from the user. FCUDA annotations consist of lightweight pragma directives that provide user guidelines for the transformations and optimizations applied in the flow. The main types of annotations are COMPUTE and TRANSFER directives which guide decomposition of the kernel into COMP and COMM tasks. Other types of annotation include SYNC and BLOCK directives which guide the synchronization of tasks and the logical layout of threads in CTAs. The analysis phase in TOPP in combination with the generated HRG representation and the throughput estimation metric eliminate the need for user annotations and help increase the exploited optimization opprotunities.

Preprocessing utilities entail kernel procedure identification and code conditioning through a sequence of transformations: (i) declaration normalization (i.e. hoist declarations out of executable kernel regions), (ii) return normalization (i.e. convert multiple return points into a single return point), (iii) procedures inlining (i.e. inline non-library procedure calls within the kernel procedure) and (iv) unsupported code identification and assertions (i.e. check and flag unsupported code structures). Note that callee inlining in the current implementation facilitates easier kernel restructuring in subsequent processing stages, but is not required for most transformations. Unsupported code structures include unstructured control flow (e.g. goto statements) and other CUDA features not currently supported by the compiler (e.g. texture memory variables). Postprocessing entails (i) variable privatization (i.e. variables referenced across different HRG regions with thread-variant data), (ii) kernel task outlining (i.e. extracting the TOPP-generated COMP and COMM tasks into separate procedures), (iii) intra-CTA parallelism extraction (i.e. tID-loop unrolling and on-chip memory banking), and (iv) inter-CTA parallelism extraction (i.e. replication of task calls, through CTA-loop unrolling). Preprocessing and postprocessing transformations, as well as annotation directives are discussed in further detail in [20, 23, 21].

C. LATENCY PHASE TRANSFORMATIONS

The transformations applied in the latency phase of TOPP aim to facilitate the generation of coarser tasks through region motions in the kernel HRG. Each region is assigned a region ID, rID, which is updated after every region motion to ensure that partial ordering with respect to rID reflects execution ordering. Reflection of execution ordering in the rID value in tandem with Def-Use chains facilitates easy motion feasibility testing. HRG nodes corresponding to CF structures are initially assigned the same rID as their first (in DFT order) leaf node. The space of rIDs in the HRG may be sparse due to transformations that result in region merging or elimination. A region motion can be encoded with respect to the initial rID, i and the final rID, j, as mot(i,j). Reflecting the execution ordering in the rID field during region motion mot(i,j) may require updates in the rID of regions with rID = k, where $i < k \leq j$, if $i < j$ or $j \leq k < i$, if $i > j$.



(a) Original FCUDA (b) FCUDA with TOPP

Figure 7: Integration of TOPP in FCUDA

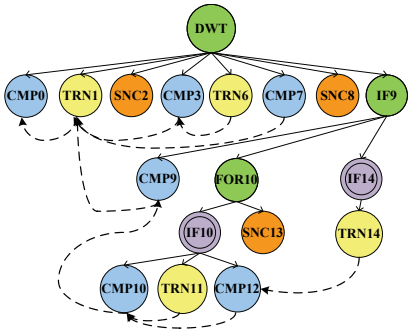


Figure 8: DWT HRG after SNC motions (SM)

C.1 SNC Region Motions (SM)

This stage identifies feasible SNC motions that facilitate region merging. The only type of SNC region motions considered are those with destinations within the same HRG level. SM stage involves four main steps: (*SM1*) Collect all SNC regions, (*SM2*) Get feasible destinations, (*SM3*) Estimate motion cost/gain, and (*SM4*) Perform motion. Initially SNC regions are collected (step *SM1*) and ordered with respect to their region identifier, rID. For each synchronization region, SNCn (with rID(SNCn) $= n$), feasible destination candidates are identified in step *SM2*. Candidate destination locations are explored in two sweeps of the corresponding HRG level: a forward and a backward sweep (Fig. 3) starting from the initial location of SNCn in the HRG. During a sweep, the candidate destinations are sequentially evaluated until (i) another SNC region is encountered, (ii) a destination is assesed as non-feasible or (iii) no candidates are left. Feasibility is tested as described in Section 3.2. TSCs with source rID, $r : r < n$ and sink rID, $v : v > n$ do not break feasibility if the destination rID, n', satisfies the following condition: $n' > v$, for forward sweeps (Fig. 3(a)), or $n' < v$, for backward sweeps (Fig. 3(b)). A destination location is selected based on the evaluation of all feasible destinations. Application of SM on the DWT HRG (Fig. 2) shifts SNC4 immediately after SNC2 region, effectively resulting in its elimination (Figure 8).

C.2 Burst Conversion (BC)

This latency optimization stage analyzes the address computation patterns of global memory accesses and determines the feasibility for coalescing memory accesses across threads into burst transfers. Moreover, this transformation offers two additional benefits: (i) reduces address computation overhead (base address calculation shared by all threads) and (ii) facilitates region consolidation in the HRG (address computation combined with data transfer in one region) which may reveal

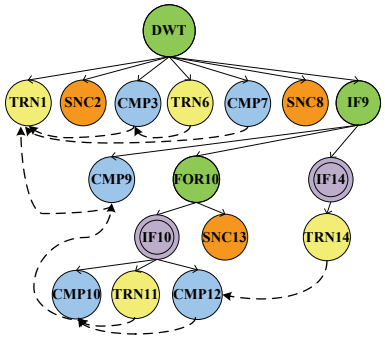


Figure 9: DWT HRG after burst conversion (BC)

new optimization opportunities. Region consolidation from burst conversion results in the elimination of CMP0 region in our DWT running example (Fig. 9).

Burst conversion involves four main steps: (*BC1*) Identify all CMP regions involved in address calculation, (*BC2*) Analyze coalesced accesses with respect to tID, (*BC3*) Analyze address coalescing with respect to TiVAR loop indexes in the kernel, and (*BC4*) Perform HRG restructuring. Initially, the Def-Use chains computed earlier in the flow are leveraged to identify statements containing address computation (step *BC1*). Subsequently, symbolic analysis and value range analysis is used to determine whether the range of computed addresses per CTA is coalesced with respect to tID. In particular, forward substitution is used to derive the address calculation expression, E_A. Then, the tID variant (TVAR) analysis performed during region analysis stage is used to decompose the expression into a TVAR part, E_{TVAR}, and a tID invariant part, E_{TiVAR}: $E_A = E_{TVAR} + E_{TiVAR}$. Symbolic and range analyses are used to examine the E_{TVAR} expression and determine whether memory accesses are coalesced in piecewise ranges, (s_i, e_i), of the tID domain. If such piecewise domain ranges can be identified, their maximum range value is returned; otherwise, a negative value is returned. Subsequently, a similar analysis of the address calculation expressions is carried out to identify coalescing opportunities across piecewise ranges of non tID-loops (step *BC3*). Any additional piecewise ranges found are used to extend the tID piecewise ranges identified previously (step *BC2*). Finally, during the last step of this stage, the address computation analysis results are utilized to perform any required HRG modifications. In the case of statically identified coalesced address ranges, individual thread accesses are converted into `memcpy` calls where E_{TiVAR} serves as the source/destination address and the size of the piecewise address range, (s_i, e_i), as the transfer length. `memcpy` calls are subsequently transformed into DMA-based bursts by the HLS engine. In the case that no address ranges are returned by static analysis, address computations are kept within CMP regions and computed addresses are stored for use by the corresponding TRN regions.

C.3 TRN Region Motions (TM)

This transformation stage shifts TRN regions to more profitable locations within their current HRG level. In particular, *TRN-Read* (TRN-R) regions (i.e. off-chip reads) are shifted toward the beginning of the HRG level, whereas *TRN-Write* (TRN-W) regions (i.e. off-chip writes) are shifted toward the end of the HRG level. This transformation aims to enable coarsening of CMP regions into bigger regions with more opportunity for ILP extraction and resource sharing. For example, Fig. 10 depicts the DWT HRG after TM transformation, where TRN6 is shifted to the right of the level and CMP3 and CMP7 are merged into CMP3. TRN motions involve four main steps: (*TM1*) Collect all TRN regions in two lists representing TRN-R and TRN-W regions, respectively, (*TM2*) Get feasible shift destinations, (*TM3*) Estimate motion cost/gain,

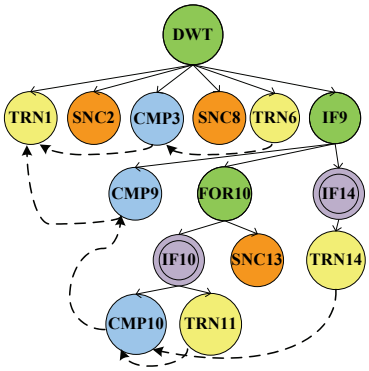


Figure 10: DWT HRG after TRN motion (TM)

Listing 3: Unnormalized CF with CMP and TRN

```
1  // tID := threadIdx.x
2  tidx=(blockIdx.x*blockDim.x);
3  for(tID=0; tID<blockDim.x; tID++) // tID-loop
4    for (pos=tidx+tID; pos<N; pos+=numThreads){// TVAR
5      locA1 = (locA0 * (locB * rcpN)); // CMP
6      d_A[pos] = loc1A;  }             // TRN
```

and (*TM4*) Perform TRN motion.

Initially, TRN regions are sorted with regard to their region ID into two lists corresponding to TRN-R and TRN-W transfers (step *TM1*). Regions in the TRN-R (TRN-W) list are processed in increasing (decreasing) rID order to find candidate destination locations (step *TM2*) within their current HRG level. In the case of TRN-R (TRN-W) regions with rID = r, candidate destinations include earlier (later) points in the HRG level with rID = z. The TRN motion, mot(r,z), is feasible unless there is a region with rID = q that bears true dependence to the TRN-R (TRN-W) region and its rID satisfies the expression $z \leq q < r$ ($r < q \leq z$). The candidate destination locations are examined for feasibility in increasing order of: $|r - z|$; if a nonfeasible destination is identified, any remaining candidates are dumped from the candidate list. Finally, a destination location for each considered TRN region is selected based on the candidate destination evaluation (step *TM3*) and the motion is applied (step *TM4*).

C.4 Control Flow Normalization (CFN)

This stage handles control-flow (CF) structures that use TVAR expressions as conditions. TVAR CF structures containing CMP and TRN regions need special handling in order to expose the implicit synchronization points (ISPs) between compute and communication tasks. Exposing the ISPs is critical in exploiting data transfer coalescing across neighboring threads in TRN regions as well as exposing the data-level compute parallelism in CMP regions. Exposing the ISPs requires interchanging the TVAR CF with the tID-loop which expresses the CTA threads. List. 3 depicts a TVAR loop (line 4) within the tID-loop (line 3) which contains a CMP (line 5) and a TRN statement (line 6). The CF normalization converts the TVAR loop into a TiVAR loop (line 6 in List. 4) preceded by initialization of the induction variable (lines 3-4) of the original TVAR loop. Thus, the ISPs between regions are exposed through tID-loops wrapped around each region (lines 7, 10). Note that variables in List. 4 are in SIMT notation. Postprocessing stage in FCUDA determines whether they should be privatized (storage redundancy) or reimplemented (compute redundancy). Variable `pos`, for example, would become an array of size `blockDim.x` in the case of privatization, whereas reimplementation would result in the code shown in List. 5. Figure 11 depicts the resulting HRG representation of the DWT after CF normalization: *IF10* node is split into *IF10* and *IF11* nodes

Listing 4: CF normalization using privatization

```
1  // tID := threadIdx.x;
2  tid=(blockIdx.x*blockDim.x);
3  for (tID=0; tID<blockDim.x;tID++)
4    pos = tidx+tID;
5  cfCond=true;
6  while(cfCond){
7    for(tID=0;tID<blockDim.x;tID++)//tID-loop
8      if (pos<N)
9        locA1 = (locA0 * (locB * rcpN));
10   for(tID=0; tID<blockDim.x;tID++)//tID-loop
11     if (pos<N)
12       d_A[pos] = loc1A;
13   cfCond = false;
14   for(tID=0;tID<blockDim.x;tID++)//tID-loop
15     if (pos<N) {
16       pos += numThreads;
17       cfCond |= (pos<N);    }}
```

Listing 5: CF normalization using reimplementation

```
1  // tID := threadIdx.x;
2  tidx=(blockIdx.x*blockDim.x);
3  pos = tidx;
4  cfCond=true;
5  while(cfCond) {
6    for(tID=0;tID<blockDim.x;tID++)//tIDX-loop
7      if ((pos+tID)<N)
8        locA1 = (locA0 * (locB * rcpN));
9    for(tID=0;tID<blockDim.x;tID++)//tIDX-loop
10     if ((pos+tID)<N)
11       d_A[pos] = loc1A;
12   cfCond = false;
13   pos += numThreads;
14   cfCond |= (pos<N); }
```

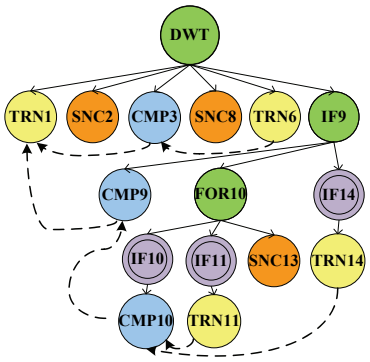


Figure 11: DWT HRG after CF normalization (CFN)

D. TDGC ALGORITHM DETAILS

Alg. 2 provides an overview of the TDGC flow described in Section 3.3.2. The three steps of the TDGC flow (*GC1, GC2, GC3*) are distinguished in the algorithm comments. During step *GC3* a recursive call to `tdgc` procedure with the rescheduled HRG is made (if node n can be shifted.) If the rescheduled HRG does not provide a higher throughput (TPc), the rescheduled HRG is discarded and the next candidate node n is evaluated for rescheduling. R-coloring is discussed in the next section.

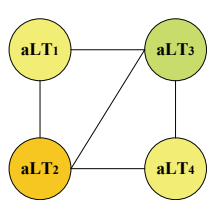
D.1 R-Coloring Algorithm

As mentioned in Section 3.3.2 an interference graph, G_I, is generated based on the analysis of the array lifetimes with respect to the regions in the HRG. The interference graph is colored using a modified R-coloring [18] algorithm which dynamically determines the value of R, i.e. the number of allocated BRAMs. Note that the interference graph represents the lifetime interferences of arrays per CTA; these interferences affect the BRAM resource requirement per CC. For a total BRAM count of N_B at the system level, there is a tradeoff between the number of instantiated CCs, N_{CC}, and the number of BRAMs, R, allocated per CC: $R = \lfloor \frac{N_B}{N_{CC}} \rfloor$. Unlike traditional graph coloring implementations where the number of colors (resource units), R, is a fixed constraint, the number of allocated BRAMs per CC, in TDGC, can range across a set of values that fulfill the previous constraint on R. In other words, by modifying the CTA region schedule we can generate different HRG configurations that have different

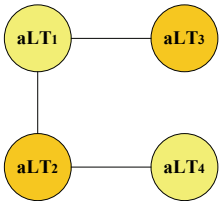
Algorithm 2: TDGC Flow overview

Input: HRG: *Gin*, Throughput: TPci
Output: New HRG: *Gout*

```
1  arrs ← getArr(Gin)                    // Collect array variables
   // GC1 step
2  Gi ← bldInterf(arrs,Gin)              // Build interference graph
3  rColor(Gi)                            // do R-coloring
   // GC2 step
4  TPco ← getTput(Gin)                   // Estimate throughput
5  if TPco < TPci then                   // Previous TPc is better
6      return 0
7  if tpcLim(TPco) == BRAM then          // Is BRAM throughput limiter?
       // GC3 step
8      calcILI(Gi)                       // Calculate ILI in Gi
9      nods ← calcLs(Gi)                 // Calculate scatter, (Ls) in Gi and
                                         // sort nodes wrt Ls in nods
10     foreach n ∈ nods do
           // Check dependency constraints for move
11         if canMov(n,Gin) then
12             Gout ← movNod(n,Gin)      // Build new HRG
13             Gout ← tdgc(Gout,TPco)    // Try TDGC on Gout
14             if Gout ≠ 0 then          // if success
15                 break                 // Do not check more nodes
16         else
17             Gout ← Gin                // Move failed
18 return Gout
```



(a) Interference graph before TDGC



(b) Interference graph after TDGC

Figure 12: DWT interference graphs

resource requirements and latencies, hence different N_{CC} and throughput. The goal of TDGC is to identify the value of R (and the corresponding feasible HRG configuration) that maximizes system execution throughput under a given resource constraint. Hence, our graph coloring initially sets R to one and dynamically adjusts its value during the first phase of graph coloring.

Algorithm 3: Graph coloring of the interference graph

```
/* tdgc(Gin): Throughput Driven Graph Coloring */
```

Input: Uncolored interference graph G_I
Output: Colored interference graph G'_I

```
1  nodes ← nods(G_I)
2  R ← 1                                 // initialize max R
3  while nodes ≠ ∅ do                    // Node pushing
4      sort(nodes)  // Sort nodes wrt interf. degree
5      n ← getNod(nodes)                 // Get first node
6      d ← minDegree(n)                  // Get interference degree
7      R ← max(R, (d+1))                 // Update degree
8      push(n, stack)  // Push to stack and prune graph
9  while stack ≠ ∅ do                    // Node poping
10     n ← pop(stack)                    // Pop node from stack and
                                         // add back to graph
11     getMinColor(n, R)                 // Allocate min color id≤R
                                         // not used by neighbors of n
```

The coloring process comprises an initial *node pushing* phase, during which, nodes are pruned in increasing order of interference degree from G_I and pushed into a stack. (This resembles traditional R-coloring with fixed R, where nodes with degree less than R are pruned first based on the observation that a graph with a node of degree less than R is R-colorable if and only if the graph without that node is R-colorable.) During each node pruning, R is adjusted as depicted in line 7 of Alg. 3 and the interference degrees of its neighboring nodes are decremented. Once all of the nodes are pushed into the stack, they are popped back into the graph in reverse order and assigned a color (Alg. 3). The assigned color for each popped node is the minimum color number that has not been assigned to any of the previously popped neighboring nodes. At the end of *node popping* all the graph nodes are going to be colored with at most R_m colors, where R_m is the maximum value of R used during the node pushing phase of coloring.

Figure 12(b) depicts the updated interference graph for DWT kernel after the throughput optimization phase. The new interference graph entails lower BRAM pressure and coloring results in the allocation of two BRAMs (compared to three BRAMs for the initial interference graph in Figure 12(a)).