Comparing Performance, Productivity and Scalability of the TILT Overlay Processor to OpenCL HLS

Rafat Rashid, J. Gregory Steffan, Vaughn Betz
Department of Electrical and Computer Engineering
University of Toronto, Ontario, Canada
rafat.rashid@utoronto.ca, {steffan,vaughn}@eecg.toronto.edu

***Abstract*—High-Level-Synthesis (HLS) tools translate a software description of an application into custom FPGA logic, increasing designer productivity vs. Hardware Description Language (HDL) design flows. Overlays seek to further improve productivity by reducing application compile times and raising abstraction by enabling the designer to target a software-programmable substrate instead of the underlying FPGA. We compare the performance, development effort and scalability of two C-to-FPGA approaches: our TILT overlay processor and Altera's OpenCL HLS. Our application-customized TILT implementations of five data-parallel benchmarks have from 41% to 80% of the throughput per unit of layout area achieved by our best OpenCL HLS designs. The time required for initial hardware compilation of these TILT designs and configuration of the target application onto the overlay is roughly comparable to the compile times of the OpenCL HLS designs: 28 and 103 minutes on average respectively. However subsequent reconfigurations due to changes in the application that do not require re-synthesis of the overlay are fast, taking 38 seconds on average. In contrast, OpenCL HLS applications require full recompilation after every code change. TILT also enables smaller, more area-efficient designs than OpenCL HLS when low to moderate throughput is sufficient. For high throughput, the larger spatially pipelined designs of OpenCL HLS are preferable.**

I. Introduction

Current FPGAs offer massive on-chip parallelism with low power compared to other platforms such as CPUs or GPUs [1]–[3] while providing the flexibility to implement any hardware. However, FPGAs are a more difficult design target. First, the compile time of design tools is typically hours for FPGAs vs. minutes for CPUs and GPUs, lengthening design iterations and reducing designer productivity. Second, most FPGA designs are currently specified in Hardware Description Languages (HDLs) such as Verilog and the cycle-accurate description required in such languages is time-consuming to write.

To alleviate these problems, several High-Level-Synthesis (HLS) techniques have been proposed that convert a high level description of an application into custom logic, providing high abstraction FPGA programming. These include FCUDA [4], LegUp [5], Vivado [6] and compilers targeting OpenCL [7], [8]. An alternative method to improve FPGA design productivity is to target a software-programmable substrate, or "overlay", configured into the underlying FPGA. Many overlay processors that execute applications on top of a configurable execution unit have been proposed [9]–[11].

In general, overlay processors increase productivity by eliminating the need to implement application-specific datapaths in HDL while also providing the flexibility to execute multiple similar applications without requiring recompilation of the overlay. Software compilation of an application onto the overlay is fast, usually taking a few seconds compared to hours with direct hardware compilation onto the FPGA. This abstraction is provided at the cost of lower performance and more area. Some overlays provide software tools that can analyze an application and suggest a suitably customized architecture to reduce this overhead [12], [13].

In contrast, HLS tools such as Vivado [6] or OpenCL [7] require full recompilation of the application into hardware after any code change. Small changes in the input code can also lead to large differences in the system area and performance so many design iterations are often necessary to fully optimize a system. Taken together, the combination of many design iterations and long compile times can significantly increase development time compared to using an overlay. However, by generating custom logic, HLS tools generally offer higher performance than overlay processors.

OpenCL is a popular open standard that enables parallel programming across heterogeneous platforms [14]. The programming model separates the application into two parts. The first is the serial *host* program that executes on a CPU and is responsible for managing data and control flow. The host offloads the parallel compute intensive second portion defined within *kernels* onto accelerator(s) such as CPUs, GPUs and recently FPGAs [7].

In this paper, we compare an improved version of the TILT overlay processor proposed in [15] with Altera's OpenCL HLS. TILT takes an algorithm description within a C function and executes it on an application-customized soft processor. Altera's OpenCL HLS takes a similar kernel as input and generates a custom pipelined accelerator on the FPGA that can be executed from a host CPU program [7].

We have chosen Altera's OpenCL HLS as our comparison point because recent studies have shown it can generate FPGA hardware with good performance relative to other platforms. Chen and Singh report their OpenCL FPGA implementation of a Fractal Video Compression algorithm is 3x faster than a high-end GPU while consuming only 12% of the GPU's power [3]. They also demonstrate a huge gain in productivity, with their simplified and error-prone hand-coded FPGA implementation taking a month to complete relative to the few hours it took to develop a working OpenCL version.

We summarize the enhancements we have made to the TILT architecture of [15] and comparison with OpenCL HLS in our list of contributions below:

Memory Fetcher We implement a new configurable and scalable Memory Fetcher unit that prefetches only the required data from off-chip DDR memory through static compiler analysis of the target application and is appropriate for use with TILT and possibly other overlay processors.

TILT Enhancements We extend the TILT architecture of [15] to support new application-specific functional units (loop and indirect address) and create a new Predictor tool to quickly choose the best TILT configuration for an application.

Comparison of TILT with OpenCL HLS We quantitatively compare the throughput and area of TILT with OpenCL HLS implementations of five large memory (ie. off-chip DDR required) data-parallel applications. We also compare the compile time and development effort between these two methods. Finally, we assess the ability of TILT and OpenCL HLS to scale up or down to match compute and area requirements.

II. Related Work

A. Soft Overlay Processors on FPGAs

Overlay processors seek to combine the fast compile time of a software-programmable substrate with higher efficiency than a basic soft processor such as Nios or Microblaze can achieve. Several works use vector processors as their overlay including VESPA [16], VEGAS [9] and VENICE [10]. Of these VENICE currently achieves the highest performance; it combines a scalar Nios II/f processor for control with wide vector lanes feeding multi-function ALUs.

VENICE connects to a standard DMA engine that moves data directly between the on-chip scratchpad memory and off-chip DDR. The scratchpad is double-buffered with dedicated ports to the DMA to overlap data movement with computation. Unlike VENICE, the functional units (FUs) in TILT operate on scalar data. TILT's scratchpad is not double-buffered and the off-chip memory transfers are interleaved into the compute schedule using ports that are shared with the FUs.

Soft processors such as VLIW-SCORE execute Very-Long-Instruction-Words (VLIW) stored in on-chip memory [11], like TILT, instead of relying on a scalar processor such as Nios. The instruction can specify a different operation per FU per cycle. An earlier soft processor called CUSTARD features the automatic generation of custom datapaths and instructions intended to accelerate frequently performed computations of the target application alongside standard operations such as add and multiply [12]. TILT uses a weaker form of application customization by varying the mix of pre-configured standard FUs and optionally generating application-dependent custom units to handle predication, loops and indirect addressing.

B. High-Level-Synthesis Tools for FPGAs

LegUp is an academic HLS tool that compiles a standard C program onto a host/accelerator model similar to OpenCL [5]. However, the partitioning of the host and kernel is done automatically by the LegUp compiler. The host executes on an FPGA-based 32-bit MIPS soft processor and communicates with the custom accelerator using an on-chip bus interface. TILT can be easily integrated into LegUp as an accelerator by connecting TILT's external data/address bus to the MIPS host, enabling faster application compilation onto the overlay.

CUDA is a language for expressing parallel applications on Nvidia GPUs that also shares the host/accelerator model [17]. FCUDA transforms CUDA kernels into custom FPGA logic using the AutoPilot HLS tool [18] and demonstrates competitive performance on Virtex 5 FPGAs, outperforming Nvidia's G80 GPU in some cases [4]. Studies including [19] demonstrate CUDA's slightly higher performance when compared to OpenCL. However, they also show it is relatively easy to translate CUDA programs to OpenCL and that OpenCL is portable, achieving good performance on other platforms with only minor code modifications. This makes OpenCL a compelling standard to compare against C-to-FPGA approaches such as overlays.

Stitt and Coole compile OpenCL applications to a spatial pipeline on their pre-compiled intermediate fabrics (IFs) composed of fixed coarse computational resources and configurable interconnect instead of directly targeting the underlying FPGA [13]. This approach is similar to TILT since the IF is customized to the requirements of the kernel and allows rapid kernel compilation and reconfiguration (seconds vs. hours) while incurring a performance penalty and area overhead compared to direct OpenCL synthesis. However, as we show in Section VI-E, TILT also enables smaller implementations than OpenCL HLS when a lower throughput is adequate, allowing a larger range of the design space to be explored.

III. TILT-System Architecture

A. TILT Overlay Processor

TILT is a highly configurable overlay compute engine for FPGAs with multiple, varied and deeply pipelined floating-point FUs [15]. TILT is capable of supporting multiple independent thread contexts, each of which can issue multiple operations every cycle. As illustrated in Figure 1, TILT has read and write side crossbars that connect the array of FUs to an explicitly managed banked multi-ported data memory built using on-chip BRAMs [20]. TILT relies on static compiler instruction scheduling to obtain high utilization of its FUs and to reduce hardware complexity [21] and does not require forwarding logic or dynamic data hazard detection.

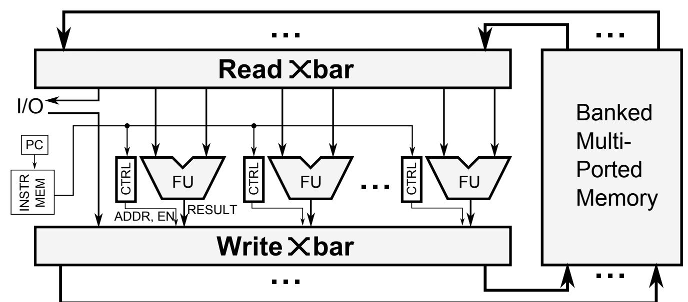


Fig. 1: TILT architecture [15].

TILT seeks to achieve a high computational throughput per unit area on data parallel applications. We accomplish this by customizing TILT's architectural parameters (such as the FU mix, organization of banked data memory, number of threads and number of operations that can issue or complete in parallel) to closely match the memory and compute requirements of the target application. If the throughput is insufficient, TILT can

be scaled by instantiating multiple copies of the TILT core, composed of the data memory, crossbars and FUs. All cores share a single instance of the instruction memory and execute in parallel in SIMD (single-instruction-multiple-data) fashion. We call this architecture TILT-SIMD.

B. External Memory Fetcher Unit

The proposal of TILT in [15] did not include a mechanism to communicate with off-chip DDR memory. The authors assumed all data was present within TILT's data memory prior to computation and only the compute portion of the benchmarks was scheduled and evaluated. To evaluate a more compelling complete system on realistic, large memory applications, we have designed a separate Memory Fetcher unit.

The Fetcher is responsible for efficiently moving data between multiple TILT cores via their external IO ports and off-chip memory in between or during computation. The Fetcher is aware of the computation's off-chip data movement behaviour through static compiler analysis of its accesses. The Fetcher performs clock domain crossing between TILT-SIMD and the DDR controller and provides deterministic external memory latency guarantee to the processor by buffering data within intermediate FIFOs. TILT-SIMD is connected to the Fetcher to create the TILT-System as illustrated in Figure 2. By separating the data movement and computation, we are able to optimize the Fetcher and TILT-SIMD for their respective tasks.

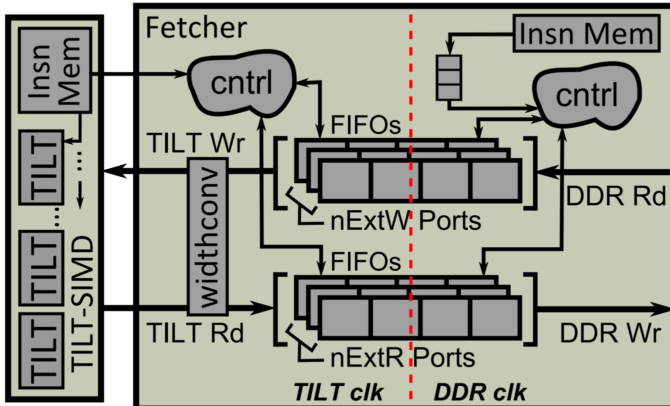


Fig. 2: TILT-System - TILT-SIMD connected to off-chip DDR via the Memory Fetcher.

Each TILT core computes on 32-bit words. The Fetcher operates on 256-bit words, the same width as our interface to DDR. This means 8 TILT cores can communicate with the Fetcher in parallel in the same cycle. The *widthconv* module converts the TILT-SIMD word to a multiple of 256-bits or vice-versa. So for TILT-SIMD with 12 cores, data from the first 8 will be sent to the Fetcher on the first cycle and data from the remaining 4 cores will be sent on the next cycle.

We add to TILT the ability to halt the processor (temporarily prevent execution of future instructions) if the Fetcher gets too far behind or to halt the Fetcher unit if it gets too far ahead. The synchronization logic is implemented within the Fetcher. Otherwise the Fetcher and TILT-SIMD execute their own respective instruction schedules, generated statically by the compiler, to provide independent operation of computation and data movement on the same memory address space.

The new instructions that move data between TILT-SIMD and off-chip memory are decoupled into two sets of instructions. The first is the Fetcher-DDR instructions that facilitate communication between DDR and the data FIFOs inside the Fetcher. The second is scheduled as part of TILT-SIMD's instructions and are responsible for moving data between the FIFOs and data memory of the TILT cores on cycles when the memory's read or write ports are not being used by compute operations. The two sets of decoupled instructions execute in the same order so the data does not need to be tagged with where it is going when inserted into the FIFOs.

Decoupling the instructions allows the Fetcher to behave as another FU to the TILT cores, providing deterministic latency to TILT-SIMD while communication with off-chip memory can remain non-deterministic. Further, several DDR read and write bursts can be enqueued together to fetch data needed by future computations into the FIFOs and flush results already computed to external memory. The depth of the FIFOs and burst sizes can be varied to improve DDR bandwidth and mask latency, providing optimized communication with off-chip memory while minimizing processor stalls.

Our approach is inspired by the Decoupled Access/Execute processor proposed by James E. Smith [22] and Outrider which splits a thread's instruction context into memory-accessing and memory consuming streams that execute in separate hardware contexts [23]. The memory-accessing stream fetches data non-speculatively substantially ahead of the memory-consuming stream to tolerate memory latency.

C. TILT-System Simulation and Predictor Model

Due to the wide range of parameters that can be varied, it can be difficult to determine the best TILT-System configuration for a given application. It is infeasible to explore the entire design space of thousands of configurations using Altera's hardware synthesis tools as each compilation can take hours. Tili in [21] applied regression models on a varied suite of compiled TILT designs to quickly estimate the area cost of adding an additional thread, memory bank, read/write port or an FU without the need to synthesize each new design on Stratix IV FPGAs. TILT's compiler can also quickly calculate FU utilization, throughput, scheduling conflicts and other metrics by statically scheduling the application. We extend this work to include the TILT-System components and develop a Predictor model targeting Stratix V chips that can recommend the best TILT-System configuration for an application using a heuristic based exploration of the parameter space. The Predictor can analyze in minutes thousands of potential TILT and Fetcher configurations that would take weeks to compile in Quartus.

IV. Implementation

A. Benchmarks

We compare the efficiency of the TILT-System with Altera's OpenCL HLS [7] on the data parallel benchmarks presented in Table I. TILT threads and OpenCL work-items execute independently and perform the same computation. Optimizations specific to TILT are discussed in Section IV-B and IV-C below. Table I presents the FU operation usage numbers taken from TILT's compute schedule for each thread of a benchmark, providing an estimate of their compute size.

Benchmark	Functional Units								Total
	AddSub	Mult	Div	Sqrt	Exp	Cmp	Log	Abs	
BSc	23	30	9	4	4	4	1	2	77
HDR	6	4	3	1	0	0	0	0	14
Mandelbrot	177	142	0	0	0	210	0	0	529
HH	36	24	28	0	15	12	0	0	115
FIR 64-tap	64	64	0	0	0	0	0	0	128

TABLE I: TILT FU operation usages per thread.

Black-Scholes Option Pricing (BSc). The BSc model is based on a partial differential equation that approximates the prices of European call and put options given inputs such as stock price, interest rate and volatility [24]. Each BSc thread computes the call and put options for a single set of inputs.

High Dynamic Range (HDR). This benchmark takes 3 input images of the same scene captured by standard cameras at 3 different exposures and produces a single output image with a greater range of luminance than the input images [25]. Each HDR thread computes a single pixel component.

Mandelbrot Fractal Rendering. We use Altera's Mandelbrot implementation where each thread computes a single pixel of a 800x640 image frame [26].

Hodgkin-Huxley (HH). This neuron simulation benchmark describes the electrical activity across a patch of a neuron membrane [27]. The computation involves solving four first-order differential equations using Euler's method to iteratively compute simple finite differences.

FIR Filter. We use the fully pipelined 64-tap Time-Domain Finite Impulse Response filter benchmark from the HPEC Challenge Benchmark suite [28]. The OpenCL HLS implementation is provided by Altera [26].

B. Implementation on TILT

TILT's schedule is configured to be cyclic for all benchmarks, wrapping around to the start to execute the same instructions on different data. Table II presents benchmark memory requirements per thread on TILT. External inputs and outputs define the number of data words that must be read or written to off-chip memory per thread. One notable difference in the Fetcher implementation for the FIR filter is that TILT-SIMD is halted while loading the filter coefficients into the data memory of the cores. The TILT-System then resumes normal concurrent operation of the Fetcher and TILT-SIMD.

Benchmark	Data Words	Ext Inputs	Ext Outputs
BSc	38	5	2
HDR	18	6	1
Mandelbrot	25	6	1
HH	86	5	4
FIR 64-tap	192	1*	1

TABLE II: TILT data memory requirements per thread (in 32-bit words). *Filter coefficients are loaded initially.

Some of the FUs share both their operation field in TILT's instruction and their read and write ports with another FU (usually the least utilized) to generate a more computationally dense schedule and to reduce the size of the crossbars. In this case, only one of the two FUs can issue and/or complete an operation per cycle. TILT's software compiler ensures there are no scheduling conflicts. This is summarized in Table III. As an example, the Mult and Cmp units for the Mandelbrot benchmark share the same operation field and ports since they have similar FU latencies and because all multiplies precede compare operations for a given thread. The latencies of the TILT FUs vary between 1 cycle for Abs to 28 cycles for Sqrt.

Benchmark	FUs - Operation Field / Port Sharing
BSc	Exp+Cmp and Log+Abs
Mandelbrot	Mult+Cmp
HH	Exp+Cmp

TABLE III: TILT FUs that share their operation field and ports.

C. TILT Efficiency Enhancements

The TILT-System architectural parameters presented thus far provide a high degree of flexibility to closely suit the needs of different applications. We further extend TILT with looping support for Mandelbrot and shift register addressing mode for the FIR filter to obtain a more area-efficient design than can be generated with the TILT architecture of [15]. Note that as TILT is an application customized "family" of compute cores, these enhancements are optionally configured and generated for the benchmarks that benefit from them.

1) Looping Support: The TILT compiler is capable of fully unrolling loops prior to scheduling. However this results in a very long schedule and a large instruction memory in hardware for loops that iterate many times. As an alternative, we have implemented the *LoopUnit* FU which contains an iteration counter. Loop start and end operations are inserted into TILT's compute schedule to mark the loop boundaries and update the counter. The *LoopUnit* operations are shared with another FU as they show up only twice for each loop in the computation.

Due to the static nature of the schedule, only loops with known bounds at compile time are supported and the scheduled instructions must not cross loop boundaries, resulting in less dense schedules than a fully unrolled schedule. However, with this approach, the loop body needs to be scheduled only once, resulting in fewer instructions and shorter schedules. Control logic inside the *LoopUnit* updates TILT's program counter (PC) to either jump to the start of the loop or continue executing the next instruction after the loop body. The *LoopUnit* is used to implement the Mandelbrot application on TILT as it features a loop that iterates a maximum of 1000 times.

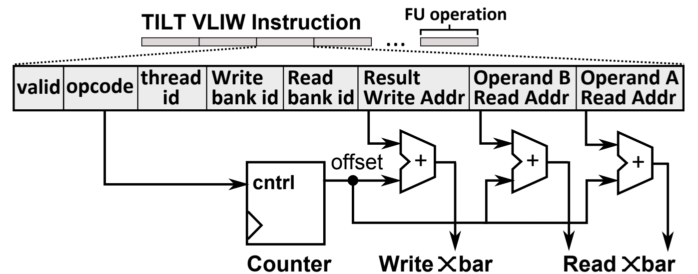


Fig. 3: Shift register indirect addressing mode.

2) Shift Register Addressing Mode: The FIR filter is most effectively implemented with input and output data placed inside shift registers. It is possible to model this behaviour on the TILT architecture of [15] but it will be extremely inefficient as separate operations must be scheduled to read data from

the TILT data memory and move them to adjacent locations between the computation of each output. As illustrated in Figure 3, by adding an offset to the static memory addresses using a counter controlled by the operation's opcode, we can obviate the need for these data movement operations. Figure 3 shows all 3 operand addresses being shifted. However, the FIR filter is configured to shift only the result.

V. Methodology

Platform. TILT and OpenCL HLS designs were generated using Altera's Quartus 13.1 and OpenCL SDK targeting the Stratix V *5SGSMD5H2F35C2* FPGA with 2 banks of 4 GB DDR3 memory on the Nallatech 385 D5 board.

Metrics. We measure computational throughput in millions of work-items per second (M wips). An work-item in OpenCL is analogous to a thread in TILT. We report area in equivalent ALMs (eALMs) which accounts for the total layout area of the FPGA resources consumed by our designs. An M20K BRAM on Stratix V costs 40 eALMs as its layout area is 40x that of an ALM [29]. Similarly, a DSP block has 30x the layout area of an ALM so it costs 30 eALMs [29]. We compare different designs using a ratio of throughput per unit of area (M wips / 10k eALMs) which we define as compute density.

A. TILT-System Evaluation

For each benchmark, the Predictor model of Section III-C is used to predict the TILT-System configurations with the highest compute densities. The Predictor selects the Fetcher configuration with the smallest area that is able to meet the external input and output bandwidth requirements of the selected TILT design. The top 10 TILT-System configurations ranked by compute density are compiled in Quartus to obtain their actual area. Throughput is obtained through cycle accurate simulation in ModelSim 10.1d assuming all input data is initially available in DDR3. The configuration with the highest compute density is then selected as the TILT result.

B. OpenCL HLS Evaluation

Quartus designs are generated using Altera's OpenCL HLS. The OpenCL host program is developed using Visual Studio 2013 and compiled into an executable with default Release configuration. To obtain throughput numbers, the input data of the entire benchmark workload is flushed to DDR3 prior to the execution of the kernel across the entire workload. This is to ensure a fair comparison with TILT by excluding the host-accelerator transfer time. Elapsed time is obtained by capturing the wall clock time before and after kernel execution using Windows high resolution timers with a precision of <1 µs. The number of work-items is increased until throughput saturates.

VI. Evaluation

A. Top TILT-System Configurations

Table IV provides the top (most computationally dense) TILT configurations for the benchmarks of Table I. The Fetcher configurations used with these TILT designs are provided in Table V. As shown in Figure 2, external write and read ports determine the number of 256-bit words that can be written to or read from the TILT cores per cycle. Similarly, the depths in Table V correspond to the incoming and outgoing data FIFOs respectively. Appropriate FIFO depths are a function of the bandwidth requirements of TILT-SIMD which can be accurately predicted when the number of TILT cores, threads, port dimensions and the number of words each thread reads or writes to external memory (Table II) are known.

Benchmark	FU Mix	Threads	Mem Banks	Bank Depth	W/R Ports / Bank	Insn Mem WidthxDepth
BSc	2-3-1-1-1-1	64	4	1024	2-4	476 x 933
HDR	2-2-1-1-0-0	64	4	512	2-4	318 x 304
Mandelbrot	1-2*	8	1	256	3-6	140 x 103
HH	3-2-2-2-0-0	32	4	1024	2-4	467 x 611
FIR 64-tap	1-1-0-0-0-0	2	1	512	2-4	76 x 144

TABLE IV: Top TILT configurations with highest compute density for each benchmark. FU Mix: AddSub/Mult/Div/Sqrt/ExpCmp/LogAbs *AddSubLoopUnit/MultCmp.

Benchmark	ExtW/R Ports	W/R FIFO Depths 256-bit words	Area eALMs	Fmax MHz
BSc	1-1	2048 / 1024	1,995	354
HDR	2-1	1024 / 512	1,816	343
Mandelbrot	1-1	256 / 128	934	385
HH	1-1	512 / 512	1,413	386
FIR 64-tap	2-1	128 / 128	1,321	357

TABLE V: Fetcher designs (for the top TILT configurations).

The external bandwidth of the FIR benchmark after the filter coefficients are initially loaded is low: 2 input and output 256-bit words every 144 cycles. However, the relatively deep FIFO depth of 128 is selected to prefetch the coefficients ahead of time and communicate with DDR in bursts. As the M20K BRAMs used to build the FIFOs support a minimum depth of 512, using a smaller depth does not save area.

B. TILT Core and System Customization

We seek to maximize throughput per unit area by customizing the TILT core and system architecture to the compute and bandwidth requirements of the application. Table VI presents the improvement in performance and area achieved with the tuned TILT configurations of Table IV relative to the baseline which is composed of 1 of each required FU with 1 thread and 1 data memory bank with 2 read and 1 write ports.

TILT-System with 1 TILT core			
Benchmark	**Throughput** M wips	**Area** eALMs	**Compute Density** Mwips/10keALMs
	Top / Base	**Top / Base**	**Top / Base**
BSc	16.3 / 0.48	13,930 / 5,567	11.7 / 0.87
HDR	46.9 / 1.54	8,231 / 3,128	57 / 4.91
Mandelbrot	0.76 / 0.14	3,320 / 1,987	2.3 / 0.69
HH	11.8 / 0.61	13,718 / 3,590	8.6 / 1.71
FIR 64-tap	3.8 / 2.06*	2,674 / 1,909	14 / 10.8*

TABLE VI: Comparison of top TILT-System designs with minimal area baseline. *FIR throughput is in M inputs/sec.

As an example, the minimal BSc TILT-System with 1 TILT core computes 0.48 M wips at the cost of 5,567 eALMs. An additional 63 threads and 3 data memory banks with 2 extra read and 1 extra write ports improves throughput by 17x by executing many threads in parallel but requires 2x more area. An extra AddSub and 2 more Mult FUs improves throughput further by 2x, at a cost of 1.1x more area, resulting in an overall 13.5x improvement in compute density.

We can improve the throughput of the TILT core further with diminishing returns by adding more FUs, threads, data memory banks and/or read/write ports to allow more operations to issue and complete every cycle but at an increasingly higher area cost, causing the compute density to decrease. As was shown in [15], the area of large TILT cores is dominated by the quadratic increase in the crossbar area, making large number of FUs or memory ports unwise.

Beyond customizing the TILT core's mix of standard FUs, we can now add application targeted FUs to improve compute density further. For example, the addition of the new *LoopUnit* for Mandelbrot significantly reduces the size of the instruction memory, requiring 4 BRAMs for the top Mandelbrot design in Table VI instead of the 527 that would be needed if the loop was fully unrolled. The required bank depth is also reduced to 256 words from 512. However, the compute schedule becomes 35% less dense, contributing to a throughput drop of 20%. Overall, the *LoopUnit* improves compute density by 6.1x, while consuming only 9.8 ALMs.

Similarly for the top FIR design in Table IV, a conventional TILT core without an indirect addressing mode will require 64 reads and writes per thread between the computation of each output, resulting in a 415 cycle instruction schedule. The addition of the mode costs only 54 eALMs but shortens the schedule to 144 cycles, improving compute density by 2.9x.

As illustrated by the tuning of the BSc TILT core to maximize compute density and the optionally generated application-dependent custom units to more efficiently handle loops (Mandelbrot) and indirect addressing (FIR), the TILT core presents multiple degrees of freedom to the designer to reduce area or to increase throughput and compute density. The significant performance improvement that can be achieved demonstrates the value of our Predictor tool. The minimal TILT-Systems in Table VI are small, consuming between 1.9k and 5.6k eALMs.

Beyond customizing the TILT core, we can improve the throughput and compute density further by connecting multiple area-efficient TILT cores to operate in SIMD. This is preferable to increasing the throughput by making the TILT core larger which will result in a less computationally dense design. The improvement can be observed for the TILT-Systems in Table VII where the TILT core of Table VI is replicated 7 more times, allowing us to achieve near-linear growth in throughput. The area cost of the TILT instruction memory (provided in Table VIII) and the Fetcher (Table V) becomes amortized, causing the overall increase in compute density.

C. TILT-System and OpenCL HLS Performance Comparison

Table VII compares the performance of the 8 core TILT-Systems using the top TILT and Fetcher configurations of Table IV and V with our best OpenCL HLS designs. The compute density of these TILT-Systems is 41% (HH) to 80% (HDR) of the density of the OpenCL designs. The reported Fmaxes are those of TILT-SIMD and the OpenCL kernel system. Both achieve similar Fmax values of over 200 MHz. The DMA area of our OpenCL designs is 4,367 eALMs on average, roughly 3x larger than the relatively small Fetcher which consumes between 1k to 2k eALMs (Table V). Both the DMA and Fetcher serve the same purpose: moving data between off-chip DDR and on-chip memory.

For the top TILT-Systems of Table VII, the area breakdown of the TILT cores is presented in Table VIII. The average area of the core varies widely between 1,358 (FIR) and 11,713 (HH) eALMs, showing the different benchmarks prefer quite different TILT cores. This further highlights the utility of our Predictor as determining the top TILT-System design is non-trivial. To achieve the most area-efficient design, our objective is to minimize the non-FU area, comprising the crossbars, instruction and data memory. The purpose of these components is to keep the FUs busy with minimal area. The average FU area for our benchmarks is 41% of the total.

Benchmark	**Fmax** MHz	**Tput** M wips	**Area** eALMs	**Compute Density** Mwips/10keALMs
TILT-System with 8 TILT cores				
BSc	220	121	95,893	12.6
HDR	223	359	51,163	70.1
Mandelbrot	246	6.1	19,051	3.2
HH	215	90	96,073	9.4
FIR 64-tap	270	30*	12,187	24.6*
OpenCL HLS - Kernel System with DMA				
BSc	221	153	51,982	29.5
HDR	234	231	26,246	88.1
Mandelbrot	268	23	46,204	5.0
HH	236	116	50,571	23.0
FIR 64-tap	274	239*	51,577	46.4*

TABLE VII: Top TILT-System and OpenCL HLS performance numbers. *FIR throughput is in M inputs/sec.

Benchmark	**Insn Mem**	**Average Area (eALMs) per TILT Core**				**Total** eALMs	**FUs/ Total**
		Mem	**FUs**	**Rd Xbar**	**Wr Xbar**		
BSc	961	2,560	5,016	3,070	971	12,578	40%
HDR	321	1,280	2,489	1,738	621	6,449	39%
Mandelbrot	161	720	1,049	396	80	2,406	44%
HH	961	2,560	4,777	3,391	985	12,674	38%
FIR 64-tap	81	320	624	296	108	1,429	44%

TABLE VIII: Area breakdown of the TILT cores for the TILT-Systems in Table VII.

D. Designer Productivity

Benchmark	**TILT-System with 8 cores**				**OpenCL Kernel Compile** mins
	Initial Setup			**Kernel Update**	
	Kernel secs	**Predictor** mins	**Overlay** mins	secs	
BSc	0.66	4.7	53	39	108
HDR	0.15	0.4	31	38	86
Mandelbrot	0.10	0.5	16	38	111
HH	0.52	2.8	52	39	106
FIR 64-tap	0.10	0.2	13	38	107
Geomean	**0.22**	**0.9**	**28**	**38**	**103**

TABLE IX: Runtime of TILT-System and OpenCL HLS tools.

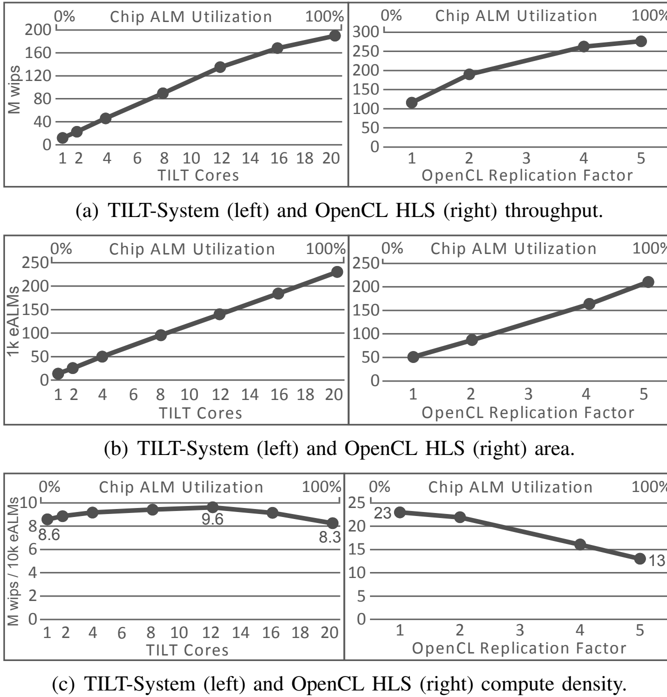
The runtime of the TILT and OpenCL HLS tools for the designs in Table VII are summarized in Table IX. For the initial setup of the TILT-System, the TILT and Fetcher instruction schedules are first generated from the C kernel using our compiler. Then the top TILT-System configuration recommended by our Predictor is synthesized into hardware using Quartus, with the schedules loaded into the instruction memories during compilation. The initial setup is dominated by the compilation of the overlay, with an average runtime of 28 mins. After a kernel code change that does not require the overlay to be recompiled, determined by running the Predictor on the modified kernel, the instruction memories of TILT and the

Fetcher are updated with the regenerated schedules, taking only 38 secs on average. By comparison, any change made to an OpenCL kernel requires full recompilation, taking an average of 103 mins, a 163x increase. Moreover, the fast runtime of our Predictor enables us to obtain a suitably customized, high performance design for an application significantly faster than would be possible through an exhaustive search using Quartus.

E. TILT-System and OpenCL HLS Scalability

Many applications combine several kernels with different throughput requirements and need to fit them all on a chip with a finite area budget. In this section, we scale the HH and FIR benchmarks to show how efficiently the TILT-System and OpenCL HLS scale up or down to meet such requirements.

We scale up the TILT-System of HH up to chip capacity in Figure 4 and compare with OpenCL HLS where the computation is replicated to execute in parallel. The maximum size of our designs is limited by the ALMs available on our FPGA. Each TILT core requires 19 DSPs for FUs, with each spatially pipelined instance in OpenCL HLS requiring 143 DSPs.



(a) TILT-System (left) and OpenCL HLS (right) throughput.

(b) TILT-System (left) and OpenCL HLS (right) area.

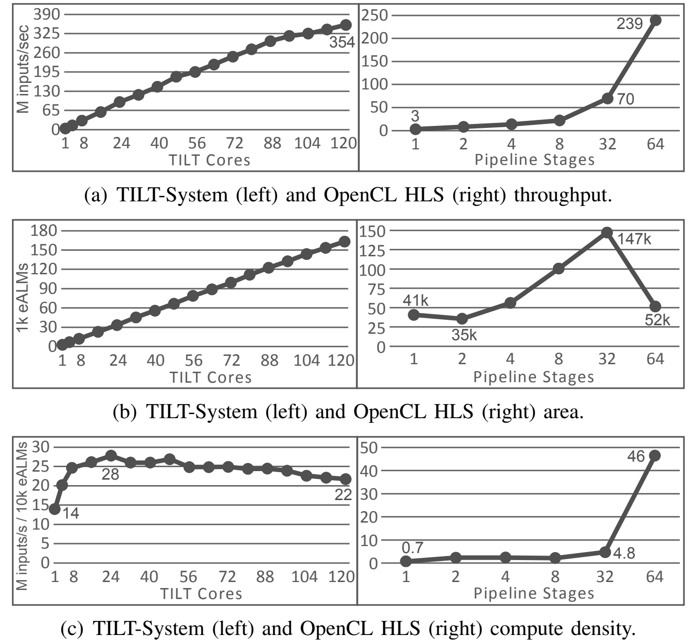
(c) TILT-System (left) and OpenCL HLS (right) compute density.

Fig. 4: Scaling HH on TILT-System and OpenCL HLS.

In Figure 4(a), we observe near-linear scaling in the TILT-System throughput, with the small drop from 16 to 20 cores caused by a drop in Fmax from 201 to 181 MHz. In Figure 4(b), the OpenCL HLS design cannot scale below 51k eALMs while the TILT-System is able to scale down to 14k eALMs. In Figure 4(c), the compute density of the TILT-System remains fairly constant (from 9.6 at 12 cores to 8.3 at 20) but is lower than the OpenCL HLS designs.

In Figure 5, we study the OpenCL HLS compiler's ability to scale down the spatially pipelined 64-tap FIR design. With 1 pipeline stage, a single multiply-add (MA) unit is shared by all 64 filter coefficients to produce an output every 64 cycles. The availability of two parallel units with 2 stages allows an output to be computed every 32 cycles. The first MA takes 32 cycles to apply the first 32 coefficients before forwarding the sum to the second MA and receiving a new input to compute.

OpenCL's fully unrolled FIR design with 64 stages in Figure 5 provides the best throughput with a comparatively low area as the overhead of forwarding the MA output back to its input or to select between multiple coefficients are eliminated. Between 1 and 32 stages, this overhead grows with the number of MAs, resulting in an overall growth in area, with the design consuming 86% of the chip's ALMs at 32 stages. The drop in Fmax from 221 MHz at 4 stages to the lowest 142 MHz at 32 also results in a sub-linear growth in throughput at that range. The net result is a low compute density for all but the fully unrolled implementation (Figure 5(c)).



(a) TILT-System (left) and OpenCL HLS (right) throughput.

(b) TILT-System (left) and OpenCL HLS (right) area.

(c) TILT-System (left) and OpenCL HLS (right) compute density.

Fig. 5: Scaling 64-tap FIR on TILT-System and OpenCL HLS.

In comparison, the small FIR TILT configuration of Table IV provides near-linear scaling in throughput and area, also shown in Figure 5. The TILT-System was scaled by connecting multiple TILT-SIMDs to the Fetcher in parallel, each with a maximum of 24 TILT cores. In Figure 5(c), the sharp increase in the TILT compute density from 1 to 8 cores is due to the amortization of the Fetcher area. TILT enables small, area efficient design choices, scaling down to 2.7k eALMs with 1 TILT core compared to the 52k eALMs of the unrolled OpenCL design. The OpenCL design is smallest at 2 stages, consuming 35k eALMs but has a low throughput of 8.3 M inputs/sec. For the same area, a 25 core TILT-System achieves a throughput of 96 M inputs/sec. Further, we are able to exceed the throughput of 239 M inputs/sec of the unrolled OpenCL FIR design with roughly 70 TILT cores but at 1.9x more area.

VII. Future Work

The discussion on tuning the TILT-System thus far was limited to a target application. However, most applications can be grouped into categories that are representative of their compute

and memory access patterns [30]. This observation suggests we can customize the TILT-System to a class of applications, with a single configuration that will perform reasonably well across multiple benchmarks. This is useful because it will reduce the need to recompile the TILT-System if the computation is modified or replaced with a similar application. Instead, we will only need to regenerate the schedules and update the instruction memories which is much faster than recompiling the overlay (Table IX). We plan to investigate this in future work. We would also like to integrate the TILT overlay as an optional OpenCL HLS compute component. A few small TILT cores can be used to perform specialized calculations for a larger computation, enabling the OpenCL HLS to take advantage of TILT's high FU reuse capability and its ability to scale down to very small implementations.

VIII. Conclusion

The TILT overlay is an area-efficient method to implement shared operator, application customizable, execution units. We extend the TILT architecture of [15] to allow use of off-chip memory with the scalable Memory Fetcher. We also enable the generation of more area-efficient designs with new custom units to support loops and indirect addressing. These are optionally generated for applications that benefit from them, improving compute density by 6.1x for the Mandelbrot and 2.9x for the FIR applications respectively. We also provide designers the ability to quickly explore a large design space of throughput and area trade-offs without requiring the overlay to be synthesized with our new Predictor tool. Further, TILT can be configured for higher throughput or lower area without requiring the application code to be changed. Configuration of an application onto an existing TILT design is also fast, taking an average of 38 seconds. In contrast, OpenCL kernels must be recompiled into hardware to observe the changes in performance and area after any application code change, lengthening design time considerably. TILT's higher productivity comes at a reasonable price; it achieves 41% to 80% of the compute density of our best OpenCL HLS designs.

We recommend Altera's OpenCL HLS for the generation of high throughput systems. OpenCL HLS maximizes throughput at the cost of more resources by generating a heavily pipelined, spatial design and by executing many threads in parallel. Deep FIFOs and interconnect buffer the thread data and stream it into the compute units. The resulting designs are large, about 3.2x (HDR) to 19x (FIR) bigger than the top (most area-efficient) single core TILT designs. If a kernel requires more modest throughput, the OpenCL HLS has difficulty generating an area-efficient, lower throughput system. Therefore when a low to moderate throughput is sufficient, we instead recommend TILT as it is capable of generating smaller but still computationally dense designs. The top TILT-Systems with 1 core require 2.7k to 14k eALMs (with 1.9k to 5.6k for minimal designs) compared to 26k to 52k eALMs for the smallest, computationally dense OpenCL HLS systems. Hence, we see the TILT overlay paradigm as an useful complement to OpenCL HLS.

Acknowledgement

We thank NSERC and Altera for funding support and Kalin Ovtcharov, Ilian Tili and Charles Eric LaForest for their feedback on this work.

References

[1] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[2] H. Wong, V. Betz, and J. Rose, "Comparing fpga vs. custom cmos and the impact on processor microarchitecture," in *FPGA*. ACM, 2011, pp. 5–14.

[3] D. Chen and D. P. Singh, "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms." in *ASP-DAC*, 2013, pp. 297–304.

[4] A. Papakonstantinou, K. Gururaj, J. A. Stratton *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors*. IEEE, 2009, pp. 35–42.

[5] A. Canis, J. Choi, M. Aldham *et al.*, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[6] T. Feist, "Vivado design suite," *White Paper*, 2012.

[7] Altera. (2014) Altera SDK for OpenCL. [Online]. Available: http://www.altera.com/literature/lit-opencl-sdk.jsp

[8] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of OpenCL applications on FPGA platforms," in *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2013, pp. 26–30.

[9] C. H. Chou, A. Severance, A. D. Brant *et al.*, "VEGAS: Soft vector processor with scratchpad memory," in *FPGA*. ACM, 2011, pp. 15–24.

[10] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2012, pp. 245–245.

[11] N. Kapre and A. DeHon, "VLIW-SCORE: Beyond C for sequential control of spice FPGA acceleration," in *Field-Programmable Technology (FPT)*. IEEE, 2011, pp. 1–9.

[12] R. Dimond, O. Mencer, and W. Luk, "CUSTARD-a customisable threaded fpga soft processor and tools," in *Field Programmable Logic and Applications*. IEEE, 2005, pp. 1–6.

[13] J. Coole and G. Stitt, "Fast and flexible high-level synthesis from OpenCL using reconfiguration contexts," 2013.

[14] K. O. W. Group. (2009, October) The OpenCL specification. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[15] K. Ovtcharov, I. Tili, and J. G. Steffan, "TILT: A multithreaded VLIW soft processor family," in *Field Programmable Logic and Applications (FPL)*, Sept 2013, pp. 1–4.

[16] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 2008, pp. 61–70.

[17] C. Nvidia, "Programming guide," 2008.

[18] Z. Zhang, Y. Fan *et al.*, "AutoPilot: A platform-based ESL synthesis system," in *High-Level Synthesis*. Springer, 2008, pp. 99–112.

[19] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 216–225.

[20] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *FPGA*. ACM, 2010, pp. 41–50.

[21] I. Tili, "Compiling for a multithreaded horizontally-microcoded soft processor family," Master's thesis, University of Toronto, Nov 2013.

[22] J. E. Smith, "Decoupled access/execute computer architectures," *Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.

[23] N. C. Crago and S. J. Patel, "OUTRIDER: efficient memory latency tolerance with decoupled strands," in *Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 117–128.

[24] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *The journal of political economy*, pp. 637–654, 1973.

[25] S. Mann and R. W. Picard, "On being 'undigital' with digital cameras: Extending dynamic range by combining differently exposed pictures," in *Proceedings of IS&T*, 1995, pp. 442–448.

[26] Altera. (2014) OpenCL design examples. [Online]. Available: http://www.altera.com/support/examples/opencl/opencl.html

[27] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," vol. 117, no. 4. Blackwell Publishing, 1952, p. 500.

[28] A. R. James Lebak and E. Wong. (2006) HPEC challenge benchmark suite. [Online]. Available: http://www.omgwiki.org/hpec/files/hpec-challenge/

[29] D. Lewis and T. Vanderhoek, "Stratix V block areas," personal communication, January 2014.

[30] K. Asanovic, R. Bodik, B. C. Catanzaro *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.