ACM/IEEE Design Automation Conference (DAC) 2013

FPGA Code Accelerators - The Compiler Perspective

Walid Najjar
University of California Riverside
Computer Science & Engineering
najjar@cs.ucr.edu

Jason Villarreal
Jacquard Computing Inc.
Riverside, CA
jason@jacquardcomputing.com

ABSTRACT

FPGA-based accelerators have repeatedly demonstrated superior speed-ups on an ever-widening spectrum of applications. However, their use remains beyond the reach of traditionally trained applications code developers because of the complexity of their programming tool-chain. Compilers for high-level languages targeting FPGAs have to bridge a huge abstraction gap between two divergent computational models: a temporal, sequentially consistent, control driven execution in the stored program model versus a spatial, parallel, data-flow driven execution in the spatial hardware model. In this paper we discuss these challenges to the compiler designer and report on our experience with the ROCCC toolset.

Categories and Subject Descriptors

D.3.4 [Programming Languages]]: Processors—Retargetable compilers; optimizations; B.5.2 [Register-Transfer-Level Implementation]: Design Aids

General Terms

Algorithms, Design, Performance.

Keywords

FPGAs. Compiler. Hardware Accelerators.

1. INTRODUCTION

In recent years we have witnessed a dramatic widening of the scope of use of FPGAs as computing devices. It is driven by a variety of factors including their larger size enabling a very large degree of parallelism, a richer set of embedded functionalities (RAM, DSP etc.), high efficiency, as compared to software, coupled with re-programmability, lower energy per task, high I/O bandwidth that eliminates the need for memory off-loading of data, etc. A general-purpose use of FPGAs as accelerators was already described a few years after the introduction of the first device [2]. However, the main obstacle facing a wider use of FPGAs as code accelerators is their poor programmability using high-level programming languages (HLL). The challenge lies in the translation of a stored-program machine, the high-level language, to a spatial and parallel computing structure with no instruction set architecture and no pre-determined control structures.

In this paper we revisit the earliest documented use of FPGAs as code accelerators and discuss various aspects of the challenge of compiling HLLs to accelerators mapped onto FPGAs (Section 2). Section 3 describes the difficulties inherent in bridging the abstraction gap between high-level languages and hardware circuits. In Section 4 we describe the ROCCC (Riverside Optimizing Compiler for Configurable Computing) [3] approach to compiling code accelerators for FPGAs. The programming model and code optimizing features of ROCCC are described in Section 5 with an emphasis on its use for high-level design space exploration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC '13, May 29 - June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00

2. THE HISTORICAL PERSPECTIVE

The first documented use of FPGAs as code accelerators appeared, to our knowledge, just four years after the introduction of the first SRAM-based FPGA device (Xilinx, 1985). The PAM (Programmable Active Memory) [2], built at the DEC Paris Research Lab, is described as "universal hardware co-processor closely coupled to a standard host computer." Ten benchmark codes were implemented and evaluated on PAM [3], including: long multiplication, RSA cryptography, Ziv-Lempel compression, edit distance calculations, heat and Laplace equations, N-body calculations, binary 2D convolution, Boltzman machine model, 3D graphics (including translation, rotation, clipping and perspective projection) and discrete cosine transform. The authors' conclusions were that PAM delivered a performance comparable to that of ASIC chips or supercomputers, of the time, and was one to two orders of magnitude faster than software. They also state that because of the PAM's large off-chip I/O bandwidth (6.4 Gb/s) it was ideally suited for "... on-the-fly data acquisition and filtering, ..."

What has changed in the nearly 25 years since the first PAM? FPGAs are much larger and faster; the application domains have grown in scope following the growth in size and speed of the devices. However, the main challenge to FPGAs as code accelerators, namely the abstraction gap between application development and FPGA programming, not only remains unchanged but has probably gotten worse due to increase in complexity of the applications enabled by the larger device sizes.

3. THE ABSTRACTION GAP

In this section we discuss two issues that define the complexity of compiling HLLs to hardware circuits: (1) the semantic gap between the sequential stored-program execution model implicit in these languages and (2) the effects of virtualization, or lack thereof, on the complexity of the compiler.

3.1 Semantics of the Execution Model

CPUs and GPUs are inherently stored-program machines (or von Neumann machines) and so are the programming languages used on these, essentially most of the languages in use today. As such they are bound by the sequential consistency of that model, both at the language and machine levels. While CPU and GPU architectures exploit various forms of parallelism, instruction, data

and thread-level, they do so circumventing the sequential consistency implied in the source code internally (branch prediction, out-of-order execution, SIMD parallelism, etc.), while preserving the appearance of a sequentially consistent execution externally (reorder buffers, precise interrupts etc.). Von Neumann, or imperative, languages are even more constrained by sequential consistency: sequential execution, pre-determined control flow structures, etc. The compiling of a HLL code to a CPU or GPU is therefore the translation from one stored program machine model, the HLL, to another, the machine's ISA. A digital circuit, on the other hand, is inherently parallel, spatial, with distributed storage, timed behavior etc. the abstraction and semantic gap between the hardware and software computing models is summarized in Table 1. Translating a HLL to a circuit requires a transformation of the sequential to a spatial/parallel, with the creation of custom sequencing, timed synchronizations, distributed storage, pipelining, etc.

Table 1. Features of hardware and software computing models

	Stored Program	Spatial Computing
Storage & data access	Central, large, virtualized. Memory resident, multi-level caches	Distributed, small, physical. Streaming, limited memory and caching. No virtualization
Parallelism	Dynamic - separate ILP, DLP, TLP	Static - combined ILP, DLP, TLP
Sequencing	Central, static, appearance of SC	Data-flow, asynchronous
Data-Path	Pre-designed, one size fits all. Dynamic data dependencies	Customized, very deep pipelines. No dynamic data dependencies

Raising the abstraction level of FPGA programming to that of CPU or GPU programming is a daunting task that is yet to be fully completed. It is of critical importance in the programming of accelerators as opposed to the high-level design of arbitrary digital circuit, which is the focus of high- level synthesis.

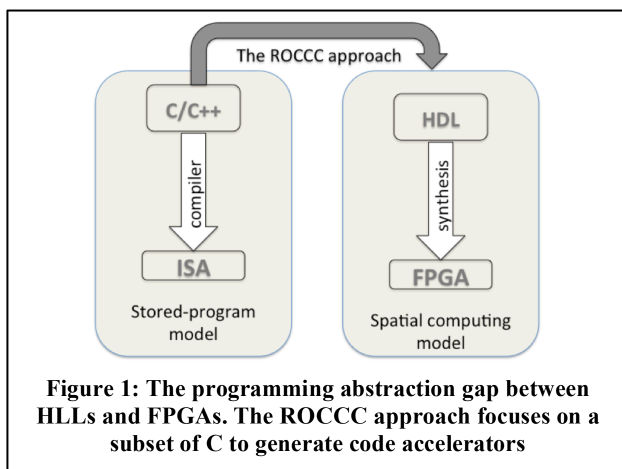

Figure 1: The programming abstraction gap between HLLs and FPGAs. The ROCCC approach focuses on a subset of C to generate code accelerators

Code accelerators differ from general purpose logic design in one important way: the starting point of logic design is a device whose behavior is specified by a hardware description code implemented in a hardware description language (HDL) such as VHDL, Verilog, SystemC, SystemVerilog, or Bluespec. The starting point of a code accelerator is an existing software application a subset of which, being frequently executed, is translated into hardware. That subset is, quasi by definition, a loop nest. Hopefully that loop nest is parallelizable and can therefore exploit the FPGA resources. By focusing on loop nests, the task of compiling HLLs to FPGAs is simplified and opportunities for loop transformations and optimizations abound. This is the approach taken by the ROCCC compiler (Figure 1) and is described in the rest of this paper.

3.2 Virtualization

Virtualization is probably one of the greatest achievements of modern computer system design: when a CPU issues a *load* instruction to an address in memory it is not aware of its actual physical location: the loaded data may not be in its cache or physical memory, it may not even be in the same time zone as the CPU! Thanks to multiple layers of hardware and software support, the world of the CPU is a single dimensional memory as large as its address space[1]. Obviously, this storage model is a perfect fit to the one implicit in all commonly used HLLs: one large flat array of bytes.

FPGA-based accelerators do not enjoy, yet, such sophisticated levels of virtualization. Rather, the compiler must be aware of all data placements, on and off-chip, and actively manage the interfaces to one or more memory modules as well as data streaming interfaces (e.g. PCI, USB, Ethernet etc.). Furthermore, most HLLs do not support streams as programming constructs or indications of physical data locations. The compiler must therefore manage the data location, both off and on-chip, with no support in the HLL, through pragmas or GUI-based indications from the user.

Each of these interfaces, to physical memories or streams, implies a preset data width, addressing modality (bursts or singletons) and mapping to a single or multiple data channels on the circuit. None of these parameters is supported in the HLL let alone in the intermediate representation (IR) the compiler uses to generate the code. On a CPU, or GPU, all data values entering the data-path come from the L1 cache with a pre-determined timing pattern. Consider a loop body that accesses four arrays, or streams, from two separate memory modules and two streams. Its data-path on an FPGA requires four physical data interfaces each with its own timing patterns that could raise very significant timing and synchronization issues.

4. THE ROCCC APPROACH

As mentioned above, the objective of ROCCC [1] is the generation of efficient customized hardware accelerators for frequently executing code segments, namely loop nests. Its target audience is application code developers with hardly any training as hardware designers. The objective being to make FPGA-based code accelerators accessible to a wider spectrum of users. As such, ROCCC is not a general-purpose high-level logic design tool, rather its focus is on generating hardware accelerators from existing C codes with minimal modifications to the source code. The same code can be compiled for software execution or translated to hardware.

The ROCCC compiler supports an extensive set of loop optimizations and transformations. One of the driving

[1] This has not always been the case; there was a time when users, or compilers, had to actively manage the memory allocation of data and code.

philosophies of ROCCC is that there should be one source code description of an accelerator that could be compiled, using different transformations, into multiple hardware implementations. All transformations are therefore done in the GUI by the user, and not by re-implementing the source code. Users are given the flexibility to choose which optimizations and transformations to perform, on each individual loop and exactly how to apply them to different parts of their application. Control of optimizations is given to the user as options in the GUI and each can have a dramatic effect on the generated hardware [7][5][4]. It would have been possible, in some limited instances, to let the compiler automatically decide which transformations should be applied. This would imply having, in the compiler, knowledge of all possible FPGA platforms, i.e. system and board architectures, current and future.

The objectives of ROCCC in generating code accelerators is maximizing throughput through (1) parallelism, (2) minimizing the area occupied by the circuit, (3) the reuse of data fetched off-chip [6] and (4) pipelining to reduce clock cycle time. ROCCC favors throughput over space, so, under user control, it could generate as much hardware as necessary to maximize parallelism. The data-path generated is purely data driven with no FSM created for resource sharing: data is pushed onto the top and flows through without any control. There is minimal control logic generated to keep track of which pipeline stages are active so the hardware can output values at the correct clock cycle.

4.1 The ROCCC Programming Model

ROCCC code is a subset of C. All ROCCC code can be compiled and run with a normal software compiler such as *gcc* and will generate the same output as the ROCCC-generated hardware from the same source. The limitations of ROCCC, compared to C, are (1) no recursion, (2) no arbitrary use of pointers that the compiler cannot un-alias statically. The use of dynamic pointers inside loop bodies would result in multiple memory de-referencing accesses being serialized, for consistency reasons, and would eliminate the parallelism.

4.1.1 Bottom-up design and code reuse

Just as in software construction, designs for hardware accelerators can benefit from opportunities for code reuse and raising the abstraction level. ROCCC is designed to support a modular approach to hardware accelerator design, enabling reuse of components and ease of design space exploration [8][9].

C code compiled by ROCCC falls under one of two categories: modules or systems. Both modules and systems are represented in C as a function call and can be compiled with *gcc* to perform the same operations in software as in hard- ware.

Module code describes components, which perform a computation on scalar inputs and generate a set of scalar outputs. They are translated into pipelined hardware structures that can take a set of inputs every clock cycle and generate a set of outputs every clock cycle. Each module is itself a complete hardware implementation and may be used by itself as a complete design, or may be included as a component in larger modules or systems.

Each module may have different optimizations performed in order to best suit the user's specific needs with regard to clock speed or area. Modules included in larger designs are treated as black boxes by the compiler so as not to affect any implementation decisions made at the lower level. Treating module instantiations as black boxes could obscure some optimization opportunities, so inlining is given as an option if the user wants to take advantage of coding at a higher level but has no fixed requirement for the low level components. All modules and systems are stored in a database, supported in the GUI, from where the user can drag and drop them in other projects. An example module that sorts two values is shown in Figure 2. In C, this code takes two integers and returns two integers in sorted order. When compiled with ROCCC, this generates a pipelined component that can take two integers every clock cycle and generated two sorted integers every clock cycle. The generated hardware is purely computational and consists of a pipeline that performs a comparison and two multiplexors.

```
void BitonicSort2(int a, int b, int& o1, int& o2)
{ if (a < b) { o1 = a; o2 = b;}
  else {o1 = b; o2 = a;} }
```

Figure 2: Bitonic sort module for two numbers

System code describes computational kernels that may apply large amounts of computation on input streams of data and generate output streams of data. Streams connections in hardware are inferred from array accesses in C. Figure 3 shows an example system that performs the Median filter operation on a 3x3 window of an NxN stream. The call to BitonicSort9 is a function call in C, but is translated into an instantiation of the BitonicSort9 module and placed in a pipeline when converted to hardware. The BitonicSort9 module is not shown, but is constructed by instantiating many copies of the BitonicSort2 module in a butterfly network. An input stream and an output stream are inferred from the parameters A and Out respectively, and result in hardware that communicates with memory in order to feed the pipeline elements from A and stores output to Out.

```
#include "roccc-library.h"
void MedianFilter(int** A, int N, int** Out) {
  int i, j ;
  int s1, s2, s3, s4, s5, s6, s7, s8, s9 ;
  for (i = 0 ; i < N ; ++i) {
   for (j = 0 ; j < N ; ++j) {
    BitonicSort9(A[i][j], A[i][j+1],
          A[i][j+2], A[i+1][j], A[i+1][j+1],
          A[i+1][j+2], A[i+2],[j], A[i+2][j+1],
          A[i+2][j+2], s1, s2, s3, s4, s5,
          s6, s7, s8, s9) ;
    Out[i][j] = s4; } } }
```

Figure 3: Median filter on a 3X3 window using the bitonic sort module

The generated data-path with no optimizations specified requires nine elements from A each clock cycle in order to generate one output each clock cycle. The first iteration, all of these values must be fetched from memory, but subsequent iterations only need fetch three new elements from memory and can reuse six elements.

The parameter N to the function in Figure 3 is translated into an input scalar. A connection is made in the generated hardware to a register that is read once at the beginning of execution and then kept constant.

4.1.2 *Data types*

In addition to the standard C data types (*char, int, long, float, double)* ROCCC supports variable bit width data types both integer and fixed-point.

ROCCC does not assume a fixed target data-path width so operations such as addition and multiplication do not need to be truncated after every step. The user can elect to maximize precision or adopt a C-like truncation model. For example, the addition of two eight-bit numbers in software will result in an eight-bit value, but in the generated hardware the result can be stored and used as a nine-bit value. Floating-point operations are assumed to be present in software, but require hardware components. Different FPGA platforms may have varying levels of support for floating point operations and since the hardware generated is not specific to a certain platform, there can be no assumptions made about the target platform's resources. As a solution, ROCCC gives the user the ability to manage a library of intrinsics, which include floating point operations and integer division. These libraries are reflections of hardware libraries such as cores generated by Xilinx Core Generator [12] and generate connections to include platform specific cores to handle the floating-point operations. Changing the libraries can affect the performance of the hardware, but is purely done through the GUI and has no effect on the source code.

ROCCC supports user-defined tables to be accessed by the data-path in some instances. These can be read-only or random access. Some operations are more efficient when implemented as a look-up table rather than an actual circuit; division is used as an example later in this paper. These tables are implemented using BRAMs when available. Random access tables may be written once per loop iteration but may be read as many times as necessary in each loop iteration.

4.1.3 *Importing external modules*

In many cases the development of hardware accelerators requires IP that was created outside the scope of the project and must be integrated into a larger design. Just as ROCCC is designed to integrate modules into larger designs, external IP can be imported and instantiated. Importing external IP requires the user provide a description of the inputs, outputs, and latency of the core through the GUI. A wrapper with default parameters such as stall and done signals connects the external IP to the generated data-paths. Calling external IP is identical to a module instantiation and appears as a function call in C. External IP calls, as well as module instantiations, can be inserted into application code directly through the GUI.

4.2 Transformations and Optimizations

A major goal of ROCCC is to enable the exploration of large design spaces through the tuning of optimizations on unchanging source code. Two types of transformations are exposed to the user in the GUI that facilitates design space exploration: High-level transformations control the overall structure of the generated hardware and can be used to create different memory configurations. These include inlining of modules, redundancy, loop optimizations, temporal common subexpression elimination, and systolic array generation. Low-level transformations control the utilization of the underlying hardware. These include pipelining control and fan-out tree generation.

Loop unrolling typically increases parallelism but also increases the necessary bandwidth to sustain a high throughput as well as the area used. For example, if the loop in Figure 3 is unrolled once so that two loop bodies are performed each iteration, the throughput is doubled as two values are generated every clock cycle. However, the resulting hardware requires two new data elements from the A input stream every clock cycle in order to maintain this. ROCCC provides fine-grained control over loop unrolling and stream connections to a degree not normally seen. Individual loops can be unrolled different amounts independently of one another, creating memory access requirements specific to individual streams.

By default, each input and output stream has one channel to memory through which all values must go. If there are multiple values generated in one clock cycle but only one output stream channel, the data must be serialized. The number of channels to memory may be configured on a stream by stream basis for each input and output stream. Each stream may be configured have the number of memory channels specified to support the highest possible throughput. Conversely, the streams can be tuned to read fewer elements per clock cycle on hardware platforms that cannot support the ideal bandwidth. For multidimensional streams support, the memory channels are further split up into address channels and data channels. Loop unrolling in multiple dimensions has different consequences on the resulting hardware depending on which loop is unrolled. Unrolling the outer loop results in more rows being fetched every clock cycle, which can be processed by increasing the number of data channels. Unrolling the inner loop, results in an increase to the size of each burst that is fetched but not the number of channels available. Unrolling either loop has the potential to increase parallelism.

Temporal common subexpression elimination [9] identifies computations that are identical across consecutive iterations of a loop and replaces those computations with a register. This can drastically reduce the area requirements by eliminating large blocks of hardware. A consequence of this optimization is that some memory fetches might be determined to be unnecessary, changing the stream interface.

Systolic array generation [5] completely transforms two-dimensional computation into a one-dimensional computation with much less area and high throughput. The memory connections of the generated hardware are changed by this optimization.

Different hardware platforms have different characteristics, such as number of inputs per LUT, which can have an effect on the relative cost of individual operations. When generating a hardware pipeline, the decision of how many basic operations to put into each level of the pipe is dependent on this information. As the compiler has no knowledge of the underlying restrictions, this control is again passed to the user.

The GUI provides both a basic slider to control the pipeline construction and the advanced capability to specify the relative cost of each basic operation on the underlying platform. Without changing the source code, many different pipelines can be created exploring the tradeoff of clock speed versus latency and area.

Another characteristic that differs from device to device is the amount of routing resources. While high fan-out is to be avoided in general, the specific limit on the amount of fan-out per element is platform specific. Again, this control is given to the user in order to control potential routing issues at the high level without rewriting the application.

5. DESIGN SPACE EXPLORATION

In this section we examine the effect of the high and low level transformations on clock speed and area on a concrete hardware

platform. The implementations were synthesized and placed and routed for a Xilinx Virtex 6 LX760 FPGA.

Median Filter – Loop Unrolling and Throughput.

Shown in Figure 3, the median filter works on a 3x3-sliding window of 8-bit data over a large 2D array. The 8-bit data is meant to be representative and not restrictive, similar results can be achieved for other bit widths. It uses the bitonic sort module (Figure 2) and has 50 cycles latency. The application is synthesized, placed and routed on the Xilinx Virtex 6 LX760, with a generic wrapper consisting of two sets of dual clock BRAMs connected to the I/O pins and acts as input and output to the ROCCC generated code.

Results for Median Filter are shown in Table 2. Each row shows the effect on area, clock speed, throughput, and throughput per unit area resulting from unrolling the outer loop and adjusting the input and output memory channels appropriately. Throughput per unit area is reported in MB/s/slice and represents the gain in throughput with respect to the amount of area added with each transformation.

Table 2: Impact of loop unrolling on Median Filter

In/Out Channels	Clock (MHz)	Area (slices)	Throughput (MB/s)	Through put / area
1/1	225	735	75	0.102
3/1	225	766	225	0.294
4/2	225	1215	450	0.370
8/6	200	3160	1200	0.380

The first row of Table 2 represents the base configuration, where no transformations have taken place and the code was compiled with the default options. In this case ROCCC generates hardware that has only one input channel and one output channel. Before any input can be processed, the hardware has to read three elements from the one input channel, which takes three clock cycles, effectively cutting the throughput into one third of its potential.

The second row shows the effect of specifying three input memory channels with no other transformations. This allows all the necessary data to be read in one clock cycle, allowing the output to be generated every clock cycle resulting in a tripling of throughput. The area is slightly larger as the hardware has to deal with multiple connections, but some internal hardware components that serialized the incoming data are actually simplified in this implementation leading to a small increase in area.

The third and fourth row show the effect of unrolling the outer loop once and six times, corresponding to connecting to an interface of 32-bits and 64-bits respectively. Each unrolling allows the number of input and output channels to increase and still produce all output every clock cycle, resulting in a large increase in throughput and maximizing the throughput per unity area for this experiment.

Average Filter – Lookup Tables and Arithmetic Cores.

Average Filter computes the average of each 3x3-sliding window in the input array. We compare two versions where the division is either implemented as a look-up table or as an instantiation of an IP core generated by Xilinx Core Generator. Results are shown in Table 3.

For all transformations the achievable clock speed was 225 MHz. Again, the first row shows the default configuration with one input and one output, the second row shows three input channels but no transformations, and the third and fourth row show the configuration of unrolling the outer loop once and six times to interface with a 32-bit and 64-bit interface. In addition to loop transformations, the Average Filter example was synthesized using both a ROCCC compiled look up table (as reported in the column labeled Area Table) and an integer division core generated (as reported in the column labeled Area Divider).

Table 3: Average Filter implementations using table lookup or integer division core (clock in both cases is 225 MHz)

In/Out Channels	Area Table (slices)	Area Divider (slices)	Throughput (MB/s)	Through put / area
1/1	218	283	75	0.344
3/1	225	275	225	1.00
4/2	253	351	450	1.78
8/6	498	826	1350	2.71

The results of these transformations provide similar throughput while the Table implementation takes less area, even when unrolling causes duplication of the table to support multiple reads per clock cycle. The throughput per unit area reported in Table 3 is reported for the Table implementation, which is the more space efficient design. Using lookup tables and unrolling the loop provides nearly 8X improvement in terms of throughput per unit area over the default case.

Max Filter – Temporal Common Sub-expression Elimination.

Max Filter computes the maximum value in a sliding 3x3 window on a 2D array (image) of *height x width* as shown in Figure 4. We use it to show the impact of temporal common sub-expression elimination (TCSE), when combined with loop unrolling, in area and throughput.

The results are shown in Table 4. The original implementation, with no optimizations, is in the first row and has three input channels and generates one output element every clock cycle. It consists of four Max modules taking up 311 slices. When TCSE is applied, two of these components are removed and only one new data element is needed each cycle resulting in a lower area for the same throughput.

The third row of Table 4 shows the results when the outer loop is unrolled five times, taking in seven elements each clock cycle and generating five outputs. Applying TCSE (fourth row) results in smaller area, increased in clock speed and two variables being

```
void MaxFilterSystem(int** A, int N, int** Out) {
  int i, j ;
  int maxCol1, maxCol2, maxCol3, winMax ;
for (i = 0 ; i < N ; ++i) {
    for (j = 0 ; j < N ; ++j) {
      MaxFilter(A[i][j], A[i][j+1], A[i][j+2], maxCol1);
      MaxFilter(A[i+1][j],A[i+1][j+1],A[i+1][j+2], maxCol2);
      MaxFilter(A[i+2][j],A[i+2][j+1],A[i+2][j+2], maxCol3);
      MaxFilter(maxCol1, maxCol2, maxCol3, winMax);
     Out[i][j] = winMax ; } } }
```

Figure 4: Max filter on a 3X3 window

reused across iterations requiring only five input elements every

clock cycle. Assuming the necessary memory bandwidth is available, this exploration shows that a 48% increase in area results in 5X higher throughput and a 3.38X higher throughput per unit area.

Table 4: Impact of TCSE on Max Filter with loop unrolling

In/Out Channels	Clock (MHz)	Area (slices)	Through put (MB/s)	Through put / area
3/1	225	311	225	0.723
1/1 with TCSE	225	266	225	0.846
7/5	220	526	1100	2.092
5/5 with TCSE	225	460	1125	2.446

6. CONCLUSION

The automatic translation of programs written in HLLs to FPGA-based hardware accelerators is a daunting task. These tools have to (1) overcome a large semantic gap between temporal, sequential and control driven programs and spatial, parallel and data/event driven circuits; and (2) without any of the virtualizations commonly available with CPUs and GPUs. In this paper we describe the ROCCC C to VHDL compilation tool, one of over 40 similar tools developed in academia and industry. The focus of ROCCC is on compiling a subset of C into hardware accelerators while providing an extensive set of compiles time transformations and optimizations under user control via a GUI-based console. We report the experimental evaluation of the impacts of some of these transformations on the circuit costs (area) and performance (throughput).

7. ACKNOWLEDGMENTS

This work was supported in part by NSF Awards CCF-1219180 and IIS-1161997 and by AFRL Contract FA945309C0173.

8. REFERENCES

[1] ROCCC 2.0 - www.jacquardcomputing.com, 2013.

[2] Bertin, P., Roncin, D, and Vuillemin. J. *Introduction to programmable active memories*, pages 300–309. Prentice Hall, 1989.

[3] Bertin, P., Roncin, D, and Vuillemin. J. Programmable active memories: a performance assessment. In *Parallel Architectures and Their Efficient Use*. Paderborn, Germany, Nov. 1992, Lecture Notes in Computer Science, pages 119–130. Springer Verlag, 1992.

[4] Buyukkurt, B., Cortes, J., Villarreal, J. and Najjar, W. A. Impact of high-level transformations within the ROCCC framework. *ACM Trans. Architecture and Code Optimizations (TACO),* 7(4):17:1–17:36, Dec. 2010.

[5] Buyukkurt, B. and Najjar, W. A. Compiler Generated Systolic Arrays for Wavefront Algorithm Acceleration on FPGAs. In *Int. Conference on Field Programmable Logic and Applications (FPL)*, September 2008.

[6] Buyukkurt, B., Guo, Z., and Najjar, W. A. Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *In Proc. Int. Workshop On Applied Reconfigurable Computing (ARC)*, March 2006.

[7] Guo, Z., Buyukkurt, B. and Najjar, W. A. Input data reuse in compiling window operations onto reconfigurable hardware. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 249–256, New York, NY, USA, June 2004. ACM Press.

[8] Guo, Z., Najjar, W. and Buyukkurt, B. Efficient hardware code generation for FPGAs. *ACM Trans. on Architecture and Code Optimizations (TACO)*, 5(1):26, May 2008.

[9] Hammes, J., Bohm, A.P.W., Ross, C., Chawathe, M., Draper, B., Rinker, R., and Najjar, W. Loop Fusion and Temporal Common Sub-expression Elimination in Window-based Loops. In *Reconfigurable Architecture Workshop*, April 2001.

[10] Villarreal, J., Park, A., Najjar, W. and Halstead, R. Designing modular hardware accelerators in C with ROCCC 2.0. In *18th IEEE Ann. Int. Symp. on Field-Programmable Custom Computing Machines (FCCM),* 2010, pages 127 – 134, May 2010.

[11] Villarreal. J. *Compiled acceleration of C programs on FPGAs*. Ph.D. Thesis, U. California Riverside, USA, 2008. AAI3332643.

[12] Xilinx Core Generator System www.xilinx.com/tools/coregen.htm