**Introduction**

Spatial Computing (SC) (most often referred to as reconfigurable computing) is a computing paradigm where the computation is formed as a circuit where data is streamed rather than a collection of instructions (as in temporal computing). The increased popularity of this model in the past few years is mostly due to the increases in both size and speed of commercially available FPGAs. In its most common form one or more FPGA is used as a code accelerator: frequently executed code segments (loops) are formed into a circuit that is mapped onto the FPGA(s) while the host CPU manages the streaming of data through that circuit. Reported speed-ups have ranged from one over three orders of magnitude [1][2][3]. These speedups are due to two main factors (1) The very large scale of parallelism that can be mapped on a typical FPGA and (2) The efficiency of the SC model as compared to the classical von Neumann model [4]. While initially this model seemed to be confined to the high-performance embedded computing market, in the past year we have seen a rapid growth in interest in the SC model and the temporal, or von Neumann, model are:

- SC is inherently parallel while temporal computing (TC) is sequential.
- TC relies on two centralized storage locations that are both explicitly addressed by the code: the register file and the memory. In SC storage is distributed throughout the circuit and is accessed implicitly rather than explicitly.

Scheduling in TC is driven by control flow while it is driven by data flow in SC. The main challenge in translating from a HLL to an HDL is in overcoming these fundamental differences.

**ROCCC**

ROCCC (Riverside Optimizing Configurable Computing Compiler) is a second-generation compilation tool targeting FPGA-based code acceleration and leveraging on our prior experience with SA-C. It takes high-level code, such as C or FORTRAN, as input and generates RTL VHDL code for FPGAs. Its objectives are (1) To bridge the semantic gap between algorithms and circuits and thereby improve the productivity of application code developers, and (2) To leverage extensive compiler transformation in order to maximize the performance of the FPGA-based codes. ROCCC is a compiler tool that maps the most frequently executed regions of software written in C, onto configurable hardware. Figure 1 shows an overview of the ROCCC framework. We have

Several projects have developed tools that translate a high-level language to an HDL, among them [8][9][10][11][12][13][14][15]. Their main advantage is an increased productivity in developing SC code. The main difficulty in this process is not the language translation, such as C to VHDL, rather it is the generation of optimized and highly efficient code (circuit) that can extract and exploit the parallelism inherent in the computation. The fundamental differences between the SC model and the temporal, or von Neumann, model are:

- SC is inherently parallel while temporal computing (TC) is sequential.
- TC relies on two centralized storage locations that are both explicitly addressed by the code: the register file and the memory. In SC storage is distributed throughout the circuit and is accessed implicitly rather than explicitly.

One of the most important challenges facing the SC model is its programmability. FPGAs are programmed with Hardware Description Languages (HDLs) such as VHDL and Verilog. Traditionally trained application program developers are familiar with HDL programming. Even when used in a behavioral style, HDLs require the programmer to develop a relatively low level description of the circuit as opposed to a high level description of the algorithm. As a consequence, the development of large-scale application accelerators becomes a lengthy and tedious task.

---

**Figure 1 - ROCCC system overview**

![Diagram showing the ROCCC system overview](image-url)
separated the front and back ends to achieve some modularity and eventually allow the use of other tools for either end. ROCCC is built on the SUIF2 and Machine-SUIF platforms. It compiles C code into VHDL code for mapping onto the FPGA fabric. Information about loops and memory accesses is visible in SUIF2's high-level Intermediate Representation (IR). Accordingly, most loop level analysis and optimizations are done at the SUIF2 level. ROCCC performs a very extensive set of loop analysis and transformations aiming at maximizing parallelism and minimizing area. The compiler also performs scalar replacement at the SUIF2 level. All memory loads are moved to the top of the loop body and all memory stores are moved to the bottom of the loop body. Machine-SUIF is an infrastructure for constructing the back end of a compiler. We modified Machine-SUIF's virtual machine (SUIFvm) IR to build our data flow. All arithmetic opcodes in SUIFvm have corresponding functionality in IEEE 1076.3 VHDL with the exception of division. Machine-SUIF's existing passes, like the Control Flow Graph (CFG) library, Data Flow Analysis library and Static Single Assignment library provide useful optimization and analysis tools for our compilation system.

The loop transformations and code optimizations built into ROCCC have the following objectives for the generated circuit that will be mapped on the FPGA:

- Maximize parallelism, clock frequency and throughput.
- Minimize the area and off-chip memory accesses.

These transformations fall in three main categories:

**Loop level transformations**: To help extract the parallelism and optimize the code for parallel execution ROCCC performs transformations both at the procedure and loop level. Loop level optimizations aim at extracting the parallelism thus increasing the throughput. ROCCC’s loop level optimizations include invariant code motion, forward substitution, induction variable substitution, full and partial loop unrolling, loop fusion, loop un-switching, loop tiling, strip mining and normalization. The optimization opportunities for FPGAs are very extensive and not affected by the side effects of one optimization removing the effects of other already applied optimizations such as unrolling many times causing increased register pressure and a negative impact on the instruction cache performance, which is usually faced in traditional compilers.

**Procedure level transformations**: These aim at reducing the circuit size and increasing throughput. This is achieved by approximating area-consuming arithmetic operators with low cost operators, removing redundancies and any computation where the result is computable at compile time and by saving the result any computation in registers for future reference wherever the computation is repeated in subsequent loop iterations. ROCCC’s extensive procedure level optimizations include constant propagation and folding, simplification of arithmetic identities, spatial and temporal common sub expression elimination, dead and unreachable code elimination, code hoisting/sinking and approximation of division by shift and addition sequences.

**Storage optimizations**: Off-chip memory accesses can have a large negative impact on the performance of FPGA-based circuits. The objectives of these optimization is to detect and exploit the reuse of off-chip data at compile time in order to minimize the number of off-chip memory accesses at run-time. The smart buffer [18] is a structure generated by the compiler that holds all fetched data that will be reused in future computations.
until it is not needed and then it is de-allocated. The smart buffer is particularly effective on windows-based operations that are very common in signal and image processing. We have shown that, for a 3x3 window, it reduces the number of memory accesses per pixel from 9 to 1.0625, a reduction of 88.2%.

**Circuit optimizations.** Their objective is to minimize the clock cycle time and maximize the throughput by employing efficient, albeit not always optimal, pipelining. Two previous works have compared compiler generated to hand-written VHDL codes for SA-C and StreamsC [16][17]. In both cases it was shown, independently and on different examples, that the hand-written VHDL achieved a clock cycle time twice as large as the compiler generated codes. Achieving a comparable clock rate is one of the objectives of ROCCC. We therefore compare the hardware performance of Xilinx IP cores and the equivalent ROCCC-generated VHDL code and shown that the clock rate is comparable, the are is larger by a factor of 2.5 [19].

**Acceleration of Smith-Waterman Programs**

The Smith-Waterman (SW) algorithm is a dynamic programming algorithm that is extensively used for both DNA and protein sequence matching. We compiled and mapped the C implementation of the matrix-fill, i.e. the compute-intensive, step of the Smith-Waterman algorithm on the FPGA using ROCCC.

The matrix-fill stage is a dynamic programming algorithm shown in Figure 2. The computation of each cell in the matrix is dependent upon the completion of the computation in its three neighboring cells that are the north, northwest and the west cells.

\[ A[i,j] = F(A[i,j-1], A[i-1,j-1], A[i-1,j]) \]

The function \( F \) above can be as simple as a min or a max but typically the function computed is selected by a cost matrix whose parameters are the top element in that column and the leftmost element in that row:

\[ F = CostMatrix(A[i,0], A[0,j]) \]

In other words the Cost Matrix indicates which function is used to compute the element in location \([i,j]\). For a given alphabet with cardinality \( C \) there can be up to \( C^2 \) distinct functions in the Cost Matrix.

To overcome the limitation arising from the data dependence structure of SW algorithm, we structured the SW code as a collection of \( k \times k \) tiles using loop tiling. Loop tiling is a well-known traditional compiler optimization technique to improve cache utilization for codes that process large arrays in the face of loop-level dependencies. Loop tiling practically helps data to be kept in the cache until it is reused in traditional compilers. When compiling onto FPGAs loop tiling allows one to visualize the matrix in blocks, where the data-dependencies among the cells inside the blocks can later become wires on the FPGA whenever the blocks are fully unrolled.

Loop tiling transforms a given two-level nested for loop, operating over a 2D array, into a four-level nested for loop where the innermost two for loops operate over the array in tiles of \( k \times k \). Following the transformation, the outermost two loops perform the moving of the tile over the original 2D array. ROCCC uses loop tiling to map algorithms that are harder to parallelize (such as dynamic programming algorithms) due to loop-carried dependencies at all loop levels. ROCCC first tiles such loops, then generates the data path by fully unrolling the generated tile. Each tile accounts for one pipelined data path.

<table>
<thead>
<tr>
<th>Tile Size</th>
<th>FPGA area %</th>
<th>Clock (MHz)</th>
<th>Pipeline stages</th>
<th>Gcups/tile</th>
<th>Gcups/chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4</td>
<td>1%</td>
<td>126</td>
<td>3</td>
<td>0.672</td>
<td>189</td>
</tr>
<tr>
<td>8x8</td>
<td>1%</td>
<td>90</td>
<td>5</td>
<td>1.15</td>
<td>71.2</td>
</tr>
<tr>
<td>12x12</td>
<td>3%</td>
<td>97</td>
<td>8</td>
<td>1.75</td>
<td>41.1</td>
</tr>
<tr>
<td>16x16</td>
<td>5%</td>
<td>108</td>
<td>11</td>
<td>2.51</td>
<td>31.9</td>
</tr>
</tbody>
</table>

We mapped the C kernel code describing the matrix fill operation to the Xilinx Virtex II Pro XC2VP50-7, i.e. available on the Cray XD1, using ROCCC. The code was compiled for data widths of 2 bits and for tile sizes of 4x4, 8x8, 12x12 and 16x16. Table-1 shows our results. The area of a tile in slices, the tile area as percentage of the overall FPGA area, the clock rate in MHz, the number of pipeline stages in the tile data path, the number of Giga cell updates per second per tile (GCUPS/tile), and the GCUPS for the whole FPGA, assuming that we use 75% of the FPGA area.

Our results show that the code generated by our compiler can obtain 1 to 100 Giga cell updates per second (GCUPS). This translates to a two to four orders of magnitude speedup compared to a 2 GHz CPU with an ideal cache and no pipeline stalls.

**Productivity Speedup**

We have implemented and evaluated a number of other applications using ROCCC. These include image and signal processing codes such as FIR filters, DWT, motion estimation and compensation. Other examples include string-matching operations for in-router processing of packets such as virus signature detection using a Bloom filter.

<table>
<thead>
<tr>
<th>Code</th>
<th>C</th>
<th>VHDL</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>2</td>
<td>1,100</td>
<td>8x unrolled</td>
</tr>
<tr>
<td>DWT</td>
<td>18</td>
<td>16,500</td>
<td>8x8 unrolled</td>
</tr>
<tr>
<td>S-W</td>
<td>13</td>
<td>12,000</td>
<td>16x16 tile</td>
</tr>
<tr>
<td>B-F</td>
<td>11</td>
<td>3,400</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

---

1 The performance of a circuit degrades rapidly when its area exceeds about 80% of the FPGA area.
While the raw performance of the code generated by ROCCC is very important, the most crucial aspect of such a tool is the productivity speedup. On average, the code expansion of C to VHDL is about a factor of 1,000 as shown in Table 2.

References