An OpenMP-based Circuit Design Tool: Customizable Bit-width

Timothy F. Beatty, Eric E. Aubanel, and Kenneth B. Kent
Faculty of Computer Science, University of New Brunswick
Fredericton, New Brunswick, Canada
{tim.beatty,aubanel,ken}@unb.ca

Abstract

As transistor density grows, increasingly complex hardware designs may be implemented. In order to manage this complexity, hardware design can be performed at a higher level of abstraction. High level synthesis enables the automatic conversion of algorithms into hardware implementations, abstracting away the underlying complexities of hardware from the designer. A number of high level synthesis tools have recently been developed, including an OpenMP to HandelC translator. Improvements to the translator, including a new compiler directive allowing customizable register width, are described. Using a set of benchmark tests, the OpenMP to HandelC translator is evaluated on several criteria, with the goal of evaluating the variable bit-width effects and identifying further areas for improvement.

1 Introduction

As transistor density grows with Moore's law, larger and more complex designs can be implemented, increasing the difficulty of Field Programmable Gate Array (FPGA) and ASIC design. To manage this complexity, hardware design can be performed at a higher level of abstraction. High level synthesis, enabling the automatic conversion of algorithms into hardware implementations, has been examined by several authors [10,1]. A recent study demonstrates the gains in productivity [21].

Synthesis of hardware from general-purpose languages has been examined in several studies [4,3,9]. Though a number of challenges have been identified, some progress has been made. Wong et al. have achieved favorable results with a high level synthesis tool that accepts a C program, annotated with OpenMP pragmas, and outputs synthesizeable HandelC [12].

HandelC is a behavioral hardware description language designed by [2]. The language contains a subset of C language elements that are suitable for hardware design as well as extensions to support concurrency. Intercommunication between parallel processing elements is provided through a communicating sequential process based model. Signed and unsigned integer datatypes are supported natively and support for fixed and floating point numbers is provided through a set of libraries. Width of variables may be specified at declaration time.

OpenMP provides an application program interface (API) for shared-memory parallel programming in C/C++ and Fortran [18]. The OpenMP API employs a fork-join model of execution by which the programmer can direct the main thread of execution to fork a pool of worker threads for work sharing purposes. When these threads complete their execution, they are joined together, and the main thread of execution resumes.

The OpenMP API provides a set of directives and clauses that are realized in C/C++ through pragma directives. Using theses directives, the programmer can specify parallelism within their program. A set of runtime functions is also provided by the OpenMP library for specifying and querying environment settings such as the number of threads.

The OpenMP to HandelC translator described in [12] is based on a project called C-Breeze. C-Breeze, an infrastructure for building C compilers [7], parses a C program into an abstract syntax tree. The C-Breeze lexer and parser have been modified to accept OpenMP directives and new abstract syntax tree nodes were added to represent most OpenMP constructs. After a series of pre-processing steps, including a check to ensure OpenMP nesting and binding rules are followed, the abstract syntax tree is translated into a HandelC program via a set of algorithms described in [17].

2 Customizable Bit-width

The C language, that OpenMP and HandelC utilize, targets general purpose processors. As a result, it can handle 8-, 16-, and 32-bit values well. These are common widths for memories, registers, and functional units in general purpose processors. There is no reason to be tied to these specific data widths when building custom hardware, so HandelC provides a variable width data type. When defining variables of this type in HandelC, the minimum width should be specified to minimize hardware usage. For example, if a variable will only hold values between 0 and 31, a 5 bit integer should be used.

The OpenMP to HandelC translator maps variables of type `int` in C to type `int 32` in HandelC. The C type `int` is a 32 bit value, so consistency is maintained between hardware and software implementations. If the values being held in a variable of this type were smaller, a 32 bit register would use more resources than necessary. Consider the example C program:

978-1-4244-4561-5/09/$25.00 ©2009 IEEE

```
int my_flag; // my_flag = {0,1}
if (my_flag == 0)
    // Do task A
else
    // Do task B
```

When translated to HandelC, the variable `my_flag` is translated to a 32 bit integer. If the possible values of `my_flag` were known to be 0 or 1, then it could be represented by a single bit. In addition, if `my_flag` was held in a single bit, then a single bit comparator could be used instead of a 32 bit comparator as well.

The OpenMP to HandelC translator needs a way to take advantage of HandelC's arbitrary bit width capability to avoid generation of unnecessary hardware [22]. A new OpenMP pragma for specifying bit widths is proposed.

2.1 Design

A new OpenMP-like compiler directive has been implemented that allows the bit width of variables to be specified for the HandelC translation of a program. This new feature is implemented as a `pragma` to maintain a consistent code style with OpenMP. The keyword “handelc" is used instead of “openmp" when invoking the pragma to denote that it is a hardware-only construct that will be ignored when compiling to software.

The bit width pragma comes in two forms. The first form is for specifying bit width when defining variables. It is defined by the following grammar:

```
#pragma handelc width number | (number {, number}) new-line
```

Several examples of variable definitions using this form of the pragma are given below. Lines 1 and 2 specify that `x` should be an 8-bit integer when translated to hardware. Lines 4 and 5 state that `a`, `b`, and `c` should be 8-, 16-, and 32-bit integers, respectively.

```
#pragma handelc width 8
int x;

#pragma handelc width (8, 16, 32)
int a, b, c;
```

The second form of the bit width pragma is for specifying the width of function return values and parameters. It is defined by the following grammar:

```
#pragma handelc function return [()]number[()] [params [()]number[()] | (number{, number})] new-line
```

Example function defintions using this form of the pragma are given below. Lines 1 and 2 specify the function `foo()` should return an 8-bit `int`. The second example (lines 4 and 5) specifies the function `bar()` should have a 16-bit `int` return value and two 8-bit `int` parameters.

```
#pragma handelc function return 8
int foo();

#pragma handelc function return (16) params (8, 8)
int bar (int x, int y);
```

3 Implementation

Some changes to the translator's lexer and parser were required to implement the two forms of the bit width pragma. The translator's lexer is generated by the GNU lexer generation tool *Flex* [6]. Its behavior is defined by a lexer specification file, to which several new terminal symbols were added. The parser, generated by the GNU parser generation tool *Bison* [5], is defined by a parser definition file in which the language's grammar is specified. With the new terminal symbols in place, grammatical rules specifying the bit width pragmas for variables and functions were added to the parser definition file.

3.1 Bit Width for Variable Declarations

Two new terminal symbols were added to the lexer. The first new symbol represents the first part of the pragma -- `#pragma handelc` -- and the second represents the keyword `width`. Symbols to represent constant numbers, parentheses, and the new-line character are required to specify the rest of the pragma, but these symbols are already present in the lexer specification file.

A new non-terminal symbol representing the bit width pragma grammar for variable declarations was added to the parser definition file. The new symbol was then inserted into the existing grammar for variable declarations to allow an optional bit width to be specified.

A record of requested bit widths must be maintained in the translator's abstract syntax tree. A new field was added to the syntax tree class `typeNode` for this information. If no bit width is requested, variables of type `int` and `long long` default to 32 and 64 bits respectively.

Once a program has been successfully parsed, the HandelC code generator is applied to the program's abstract syntax tree. Modifications were made to this phase to output bit widths associated with variable declarations.

3.2 Bit Width for Function Declarations

Additional terminal symbols were added to the lexer definition in order to implement the bit width pragma for functions. These symbols represent the tokens `function` and `params` from the previously stated grammar. Lists of constants, parentheses and a symbol for the token `return` are already defined in the lexer specification.

A new non-terminal symbol was defined in the parser specification to represent the bit width pragma for functions. The grammar for function definitions was modified to include this symbol allowing the pragma to be optionally specified when functions are declared.

When the bit width pragma for functions is specified, requested bit widths must be recorded. The abstract syntax tree node for functions contains a node that represents the return type of the function as well as a list of nodes for the parameters. The requested return value width is stored in the former, and parameter widths are stored in the latter. The length of the parameter list matches the length of the parameter bit width specification list. If the bit width pragma is not specified when a function is declared, default bit widths are used: 32 bits for `int` and 64 bits for `long long`.

Finally, the HandelC translation phase was modified to output the HandelC width associated with any return value or parameter.

3.3 Automatic Width Adjustment

It is likely that a program will contain an expression with operands of different width. In C, automatic type conversions convert narrower operators to wider ones which enables the operation to be executed without losing any information. However, HandelC does not have such automatic conversions, and expressions with operators of different widths cause the compiler to fail.

The HandelC standard library provides macro functions for adjusting the width of integers. The `adjs()` macro sign-extends signed integers to a specific width and the `adju()` macro zero-pads unsigned integers. Using these library functions, a proper translation is generated.

Particular attention must be paid to assignment statements when operand widths are mixed. In C, conversions can take place across assignments. HandelC does not have conversion across assignments, so operands with mismatching widths in assignment statements must be adjusted to ensure that the translated HandelC code will compile correctly. The translator adjusts the widths of the operands on the right hand side of the assignment statement to the width of the left hand side.

In C, conversions across assignments where information is lost are also legal, but may draw a compiler warning. The same idea is applied to the translator. When the width of right hand side operands is less than the width of the left hand side, right hand side operands are adjusted to the width of the left hand side, regardless of potential information loss. It is up to the programmer to be aware of the bit widths of variables and to know when automatic width adjustment is being applied.

Automatic width conversion is currently implemented only for integers. The HandelC reference manual recommends the use of integers instead of other logic types (which are just integers with fixed widths) anyway [2], so this is believed to be reasonable.

Automatic type conversions in C also allow expressions to mix signed and unsigned operands. There are, however, no such conversions in HandelC. Values must be cast between types explicitly to ensure that the programmer is aware that a conversion is occurring that may result in the meaning of a value being changed. Since the programmer is required to explicitly convert types in HandelC when signed and unsigned operands are mixed, the onus is left with the programmer to explicitly specify any necessary conversions.

3.4 Shift Operators

Operations in C are defined for most combinations of operand types. This is not the case with HandelC, which has operations that are sensitive to the width of the operands. Complications of variable bit width and the shift operations (<< and >>) are addressed in this section.

In a shift operation, the width of the rightmost operand must be no larger than the width necessary to express the range of possible values by which the leftmost operand may be shifted. Widths of the rightmost operand in a shift operation are therefore adjusted accordingly in the translator. Special cases arise when the rightmost operand's width is insufficient to express the range of possible shift values and when the value of the rightmost operand is greater than the largest possible shift value. When the rightmost operand's width is insufficient to express the range of possible shift values, it is expanded to the correct width. When the rightmost operand's value is greater than the largest possible shift value, the operand is adjusted down to the correct width anyway, but the result of the shift operation is undefined.

Finally, the rightmost operand in a shift operation must be unsigned. If it is signed, the value is automatically cast to a signed type. The programmer must use caution when shifting by values of signed type or face unexpected results.

4 Benchmarks

While the results shown in [21] demonstrate a correct implementation, performance and resource usage statistics are not examined in significant detail. Further benchmarking of the translator is required to collect detailed performance and resource usage data. An analysis of these results will allow any inefficiencies in the translation to be identified. Furthermore, a baseline for performance and resource usage can be established which may be used in demonstrating the performance gains or resource usage improvements obtained in future improvements to the translator [13,8]. Overlooked implementation issues causing incorrect results may also be identified.

Each test program is first translated to HandelC and then imported into an Agility DK 5.0 workspace, where a simulation is generated and clock cycles are counted. The generated HandelC source file is then converted to synthesizable VHDL and a count of NAND gates is recorded. The VHDL file is then synthesized with Xilinx ISE 9.1 for a Xilinx Spartan-3 FPGA (model XC3S5000-4FG900). Logic slice and 4-input look-up table counts are collected from the map report and minimum clock periods and maximum clock frequencies are collected from the post-place-and-route timing report.

Three benchmarks are used for testing & evaluation:

- Mandelbrot Set [15] - is a set of points in the complex plane whose boundary forms a fractal.
- Miller-Rabin Primality Test [19] - An integer n is called a *strong pseudoprime* to the base b if either $b^t = 1$ **mod** n or there exists an r with $0 \leq r < s$ such that $b^{2^r t} = -1$ **mod** n.
- Edit Distance [14] - A systolic array implementation of string comparison using edit distance.

5 Results

The benchmark tests were implemented and data was collected according to the methodology in Section 4. The resulting data is presented here along with a discussion.

5.1 Mandelbrot Set

Two versions of the Mandelbrot set generator were implemented. Using the bit width pragma, variables were set to 32 and 16 bits wide in the first and second versions respectively. Half of these widths were reserved for fractional quantities in the fixed-point representation. Both versions produce the same classification for each pixel.

Figure 1 shows a comparison of NAND gate counts as the number of threads increases. Significantly fewer NAND gates are required to implement the Mandelbrot set generator with 16 bit registers than with 32 bit registers. This is because smaller registers and arithmetic operations on smaller operands require less hardware to implement. As the number of threads increases, the difference in NAND gate counts grows. This is due simply to the fact that more hardware is generated per thread for the 32 bit version than the 16 bit version. NAND gates are mapped to lookup tables by the synthesis tool and then assigned to logic slices in the FPGA. A comparison of 4-input lookup tables (Figure 2) further demonstrate reduced resource usage when smaller registers are used.

Figure 3 compares maximum clock frequencies between versions. Clock frequencies are at least twice as fast for all samples in the 16 bit version than the 32 bit version. This difference is due to operand size and the type of operations performed. Double width multiplication is required due to binary scaling and since this operation is the slowest and most complicated, it has the largest impact on maximum clock frequency.

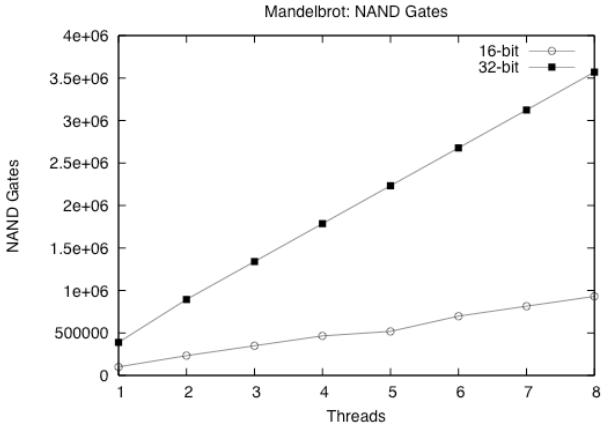

Figure 1: Mandelbrot NAND Gates

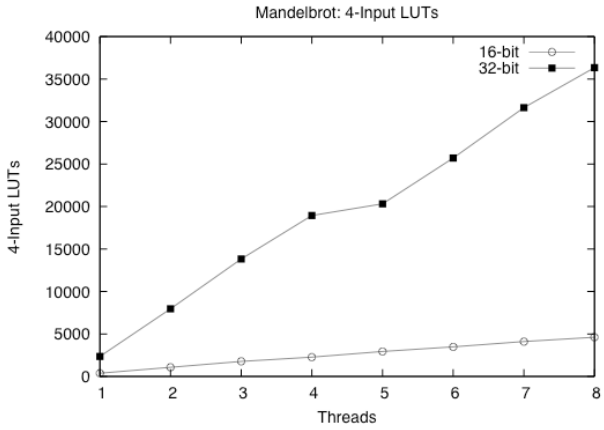

Figure 2: Mandelbrot 4-input LUTS

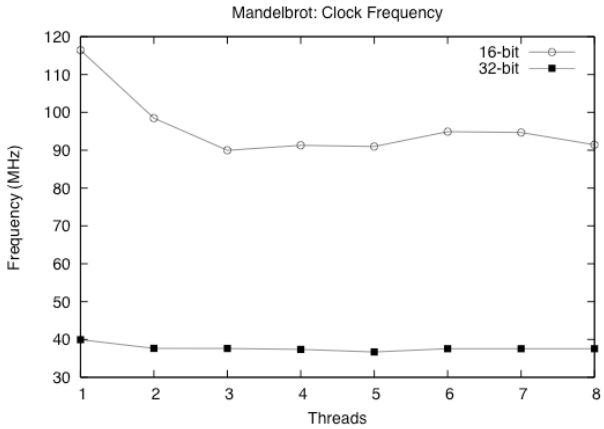

Figure 3: Mandelbrot Maximum Clock Frequency

Figure 4 shows the execution times of both versions of the algorithm. In both versions, execution times decrease as the number of threads increases. Clock cycle counts are nearly the same so the faster execution times of the 16 bit version is a result of the higher clock rates.

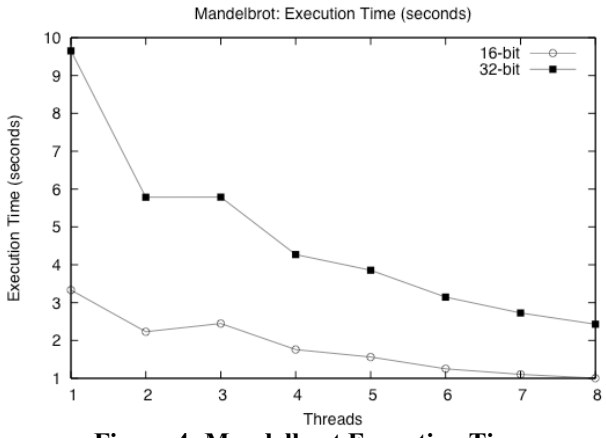

Figure 4: Mandelbrot Execution Time

5.2 Miller-Rabin Primality Test

Two versions of the Miller-Rabin primality test were synthesized -- one with 32 bit registers and one with 16 bit registers. Both versions were tested with a single fixed input and produce the same result.

A maximum of only two threads could be synthesized for this test due to resource limitations on the FPGA. During translation, all functions are marked as `inline`. Each function call in a program is therefore expanded inline at compile time, resulting in the generation of a new piece of hardware for each function call. Non-`inline` functions in HandelC are shared by parallel threads, and access to this shared hardware must be explicitly controlled by the programmer. This explicit control cannot be realized by the translator because the OpenMP library lock routines are not yet implemented. Translated functions are therefore made to be `inline` to avoid concurrency issues. In this algorithm, two calls to `modpow()` (which performs modular exponentiation) are required per iteration and therefore two hardware blocks are generated for the function for each thread.

The NAND gate and LUT usage for the Miller-Rabin primality test, Table 1, shows less than half of the resources required to implement the 32 bit version were required to implement the 16 bit version. However, many more numbers can be tested with the 32 bit version. This reduction in resource usage is due to narrower operands and less hardware being required to implement arithmetic operations on these operands.

In addition to a savings in resources, smaller operands result in faster clock rates. Table 2 compares maximum clock frequencies and clock cycles between versions. The double width modulo operation has the largest effect on maximum clock rate. Two factors account for the difference in clock cycles: implementation of the modular exponentiation function and automatic width adjustment. The modular exponentiation function uses binary exponentiation and therefore the number of steps is proportional to the bit width of operands [20]. Automatic width adjustment uses HandelC's `adjs()` and `adju()` macro functions which are designed to handle arguments of any given width. Some control logic is therefore implied and the number of cycles required to execute these functions depends on the width of the arguments.

The execution times of both versions of this algorithm are shown in Table 3. The 16 bit version requires fewer clock cycles and has higher maximum clock rates resulting in faster execution times than the 32 bit version.

Threads	16 bit version		32 bit version	
	NANDs	LUTs	NANDs	LUTs
1	282829	6742	1121805	43007
2	569097	13540	2240217	72844

Table 1: Miller-Rabin NAND Gates and LUT Usage

Threads	16 bit version		32 bit version	
	Freq.	Cycles	Freq.	Cycles
1	9.702	899	3.053	1219
2	9.533	507	2.433	667

Table 2: Miller-Rabin Maximum Clock Frequency and Clock Cycles

Threads	16 bit version	32 bit version
1	92 usecs	399 usecs
2	52 usecs	247 usecs

Table 3: Miller-Rabin Execution Time

5.3 String Comparison

Two versions of systolic string comparison were tested. Default register widths were used in the "standard" version and "minimal" register widths were used for the second version. The horizontal axes of the graphs in this section represent thread numbers or, equivalently, the number of processing elements. Strings of equal length are used as inputs and must be hard coded as the HandelC channel and interface types cannot be used from C to read input. The length of input strings for any point is given by $(t+1)/2$, where t is the number of threads.

Figure 5 shows NAND gate counts for both versions of the algorithm. The difference in NAND gate counts is the result of a number of minimizations. Since addition and subtraction mod 4 may be used (as stated in [14]), distance values and processor state values are held in 2 bit registers in the minimal version. A shared counter, used to reconstruct the edit distance, is held in a 32 bit register in the standard version and an 8 bit register in the minimal version (only short input strings are anticipated). A flag indicating a processing element has been initialized may be held in a single bit register in the minimal version while a 32 bit register must be used in the standard version. More hardware is generated per thread in the stan-

dard version and accounts for the growing difference in NAND gate counts as the number of threads increases.

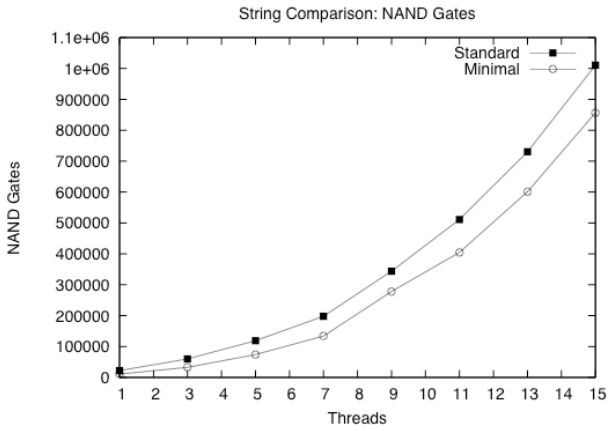

Figure 5: String Comparison NAND Gates

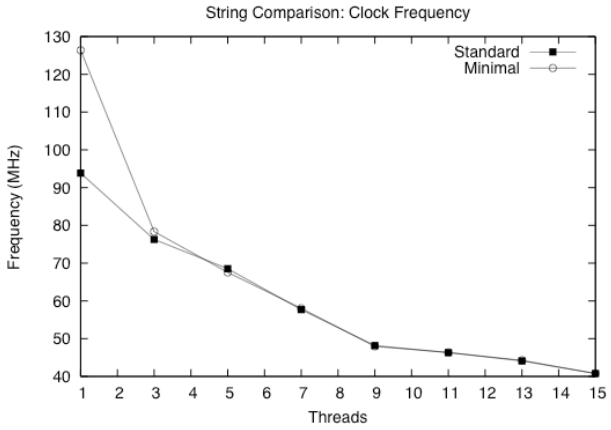

Figure 6: String Comparison Clock Frequency

The number of clock cycles are the same in both versions. Maximum clock frequencies are shown in Figure 6. The maximum clock frequencies of the minimal version were expected to be slightly higher than the standard version, however, in some cases, the reverse is true. This difference is attributed to differences in the outcome of the place-and-route phase of circuit synthesis. The differences in clock frequencies are marginal and since the number of clock cycles required are the same between versions, differences in execution times are also marginal.

6 Conclusions and Future Work

The number of arithmetic operations per thread, combined with the bit width of operands, has an effect on the resource usage of a design. This effect is even more pronounced when the operations require a lot of hardware to be generated (*e.g.* multiplication). The bit width of operands and the type of operations being performed have an impact on the clock frequency (and therefore execution time) of a design. This effect is less pronounced when the grain size of parallel computations is smaller.

7 References

[1] R. J. Carter, B. Shackleford, and G. Snider, Attacking the Semantic Gap between Application Programming Languages and Configurable Hardware, 9th International Symposium on Field Programmable Gate Arrays, pp. 115-124, 2001.

[2] Celoxica Inc., HandelC Language Reference Manual, Celoxica Inc., ver. 4, 2005.

[3]G. De Micheli, Hardware Synthesis from C/C++ Models, Conf. on Design, Automation and Test in Europe, p. 80, 1999.

[4] S. A. Edwards, The Challenges of Hardware Synthesis from C-like Languages, Conference on Design, Automation and Test in Europe, pp. 66-67, 2005.

[5] Free Software Foundation Inc., Bison, www.gnu.org/software/bison/manual, ver. 2.3, 2006.

[6] Free Software Foundation Inc., Flex, www.gnu.org/software/flex/manual, ver. 2.5, 1998.

[7] S. Z. Guyer, D. Jimenez, and C. Lin, The C-Breeze Compiler Infrastructure, Univ. of Texas at Austin, TR-01-43, Nov 2001.

[8] T. S. Hall, and K. B. Kent, A Hardware/Software Co-specification Methodology for Multiple Processor Custom Hardware Devices Based on OpenMP, University of New Brunswick, PhD Proposal, March 2008.

[9] R. Helaihel, and K. Olukoyun, Java as a Specification Language for Hardware-Software Systems, International Conference on Computer-Aided Design, p. 690, 1997.

[10] B. L. Hutchings, P. A. Jackson, and J. L. Tripp, Sea Cucumber: A Synthesizing Compiler for FPGAs, 12th Conference on Field Programmable Logic and Applications, p.51-72, 2002.

[11] D. E. Knuth, The Art of Computer Programming (3rd Ed.), vol. 2, Addison-Wesley, 1997.

[12] Y. Y. Leow, C. Y. Ng, and W. F. Wong, Generating Hardware from OpenMP Programs, IEEE International Conference on Field-Programmable Technology, pp.73-80,2006.

[13] J. C. Libby, F. Gharibian, and K. B. Kent, "Automatic Identification of Parallelism in HandelC, 2008 Euromicro Digital System Design Symposium, pp. 660-664, Sept 2008.

[14] R. J. Lipton, and D. Lopresti, A Systolic Array for Rapid String Comparison, Chapel Hill Conference on VLSI, pp. 363-376, 1985.

[15] B. B. Mandelbrot, The Fractal Geometry of Nature, W.H. Freeman and Company, 1982.

[16] S. B. Needleman, and C. D. Wunsch, A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins, Journal of Molecular Biology 48, 443-453, 1970.

[17] C. Y. Ng, Translating OpenMP Programs to Hardware via HandelC, National University of Singapore, Hons Thesis, 2006.

[18] OpenMP Architecture Review Board, OpenMP Application Program Interface, www.openmp.org, ver. 2.5, 2005.

[19] M. O. Rabin, Probabilistic Algorithm for Testing Primality, Journal of Number Theory 12, no. 1, pp. 128-138.

[20] B. Schneier, Applied Cryptography 2nd Ed., Wiley, 1996.

[21] B. Svensson, and Zain-ul-Abdin, A Study of Design Efficiency with a High-Level Language for FPGAs, International Parallel and Distributed Processing Symposium, pp. 1-7, 2007.

[22] T. F. Beatty, E. E. Aubanel, and K. B. Kent, Customizable Bit-width in an OpenMP-based Circuit Design Tool, 17th ACM International Symposium on Field Programmable Gate Arrays 2009, pp. 278, Feb 2009.