Generating Hardware From OpenMP Programs

Y.Y. Leow, C.Y. Ng, and W.F. Wong

Department of Computer Science
National University of Singapore
3, Science Drive 2, Singapore 117543
wongwf@comp.nus.edu.sg

Abstract— **Various high level hardware description languages have been invented for the purpose of improving the productivity in the generation of customized hardware. Most of these languages are variants, usually parallel versions, of popular software programming languages. In this paper, we describe our effort to generate hardware from OpenMP, a software parallel programming paradigm that is widely used and tested. We are able to generate FPGA hardware from OpenMP C programs via synthesizable VHDL and Handel-C. We believe that the addition of this medium-grain parallel programming paradigm will bring additional value to the repertoire of hardware description languages.**

I. Introduction

Along with Moore's Law and the need to contain recurring engineering cost as well as a quick time to market, there has been a significant push for the use of high-level hardware description languages for the purpose of hardware synthesis. The holy grail in this front is to be able to automatically generated hardware from programs written and debugged in traditional software programming languages. Years of development in software engineering and programming have made it possible for a large number of software designers to collaborate and produce very large applications. However, even after overcoming the barriers of behavioral synthesis [11], there is still the issue that much of the traditional software code is written in sequential languages. Research in automatic parallelization has only met limited success.

To overcome the above issues, many high-level hardware description languages have been proposed. Many of these are essentially parallel versions of traditional programming languages such as C together with bit-width extensions.

In this paper, we describe the use of OpenMP [10] as a hardware description language. OpenMP works on the basis of pragmas – or "active" comments – added to a sequential program with the aim of parallelizing the code. There are several key advantages in taking this approach. First, the programs themselves will execute "as is" on any processor. This helps enhance the productivity of the software development process by leveraging on software functional and performance debugging facilities. Secondly, OpenMP is supported on a wide range of multicore, shared memory processors, clusters as well as compilers, including forthcoming versions of GCC. With the rise of multicore processing in the main stream processor market, one would expect that a parallelization paradigm such as OpenMP that already enjoyed significant support will be gaining further grounds.

We have created backends that generate synthesizable VHDL [12] or Handel-C [13] code from OpenMP C programs. We have successfully executed these on a FPGA platform. We shall first give a very brief introduction of OpenMP. This will be followed by the descriptions of our Handel-C and VHDL backends. In the Section 5, we will describe results from experimenting with our tool. This will be followed by a description of the related works and the conclusion.

II. OpenMP

OpenMP [10] is a specification jointly defined by a number of major hardware and software vendors. It is a set of compiler directives, runtime library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. It uses a thread-based shared memory programming model where objects in an OpenMP program are either shared or private to each thread. Shared objects are accessible by all threads but private objects are only accessible to the thread that owns the objects. Any data transfer between the shared memory and the thread's local memory is transparent to the programmer and most synchronization are implicit.

Every OpenMP directive has the following grammar in general:

```
#pragma omp directive_name [clause[[,]clause]…]
```

The directives, shown in Table I can be divided into the following categories: parallel constructs, worksharing constructs, parallel worksharing constructs, master and synchronization directives. If a directive allows some clauses, then these clauses can control the number of threads executing a parallel region, the scheduling for a loop construct and the OpenMP data environment. The memory model used by OpenMP follows a memory consistency model where each thread has its own view of the shared memory. Synchronization can take place to make a thread's view of memory consistent.

Parallelism in OpenMP can be modeled as a fork and join model where a parallel region of code is replicated for each thread executing the parallel region. The number of threads executing the parallel region is determined on entry to the parallel region. The master thread spawns a team of slave

TABLE I
OPENMP DIRECTIVES

OpenMP	Directive/Clause Explanation
`#pragma omp parallel`	Parallel construct
`#pragma omp for`	Worksharing construct
`#pragma omp sections`	Worksharing construct
`#pragma omp parallel for`	Parallel Worksharing construct
`#pragma omp parallel sections`	Parallel Worksharing construct
`#pragma omp single`	Worksharing construct
`#pragma omp master`	Master construct
`#pragma omp critical`	Critical construct, used for critical sections of code
`#pragma omp atomic`	Atomic construct, similar to critical construct, but a critical section has only one statement
`#pragma omp ordered`	Ordered Construct, for ordered execution in a loop or parallel loop construct
`#pragma omp flush`	Flush directive, for memory synchronization
`#pragma omp barrier`	Barrier directive, for thread execution synchronization
`#pragma omp threadprivate`	Threadprivate directive, for global private variables to a thread
`if(expression)`	Decides whether parallel region is serialized
`num_threads(expression)`	Number of threads for a parallel region
`schedule(kind,expression)`	Scheduling for a loop or parallel loop construct
`ordered`	Indicates the presence of an ordered region for a loop and parallel loop construct
`nowait`	Indicates that a worksharing construct has no implicit barrier
`private(variable_list)`	Declares local variables for each thread
`firstprivate(variable_list)`	Same as private clause, but local variables are initialized
`lastprivate(variable_list)`	Same as private clause, but original variables are updated
`copyprivate(variable_list)`	The thread that executes the single construct broadcasts the values of its copies of the variables in variable_list to other threads
`default(shared | none)`	Indicates sharing attributes for visible data
`shared(variable_list)`	Indicates that variables in variable_list are shared
`reduction(op:variable_list)`	Reduction operation
`copyin(variable_list)`	A mechanism to copy the master thread's threadprivate variables to other threads.

threads to execute the parallel region. The code in the parallel region is replicated for each thread executing the region. On exit from the parallel region, the threads are synchronized and the slave threads are kept in a thread pool to be reused for another parallel region.

OpenMP is ideally suited for medium granularity, loop parallelism. Given that a lot of behavioral synthesis works have focused on loop-level parallelism, we believe that this is the right level of approach the problem from a high level perspective.

III. GENERATING HANDEL-C CODE FROM OPENMP PROGRAMS

Handel-C is a C-based behavioral hardware description language sold by Celoxica. The advantage of using Handel-C is that it comes with various board support packages and tools that enables us to perform rapid prototyping on FPGA boards. Since we chose the C-variant of OpenMP, the exercise is therefore one of translation. To facilitate this, we used C-Breeze [7], a C compiler infrastructure. C-Breeze parses a C program into an abstract syntax tree (AST). The lexer and parser in the C-Breeze was extended to parse OpenMP C pragmas in accordance to the OpenMP API 2.5 standard. New AST node classes are added to represent each OpenMP construct, including the data clauses that come with some of the constructs. We shall now highlight some interesting cases in the translation process.

We start with a parallel construct. The following is an OpenMP parallel construct:

```
#pragma omp parallel
{
 /* Code executed in parallel */
}
```

This is translated simply as:

```
par(tid = 0;tid < np;tid++) {
seq {
   {
     /* Code executed in parallel */
   }
  }
}
```

The number of threads is determined by the OMP_NUM_THREADS environment variable. At the exit from the parallel construct, there is an implicit barrier. This is already indirectly taken care of by Handel-C's par replicator, which has an implicit barrier on exit from the par replicator. Nested parallelism and dynamic adjustment of threads are not supported at the moment.

The parallel-for construct is used to describe parallel loops in OpenMP:

```
#pragma omp for
for (i = 0;i < 50;i++)
{
    /* parallel code */
}
```

This is translated into the following Handel-C code:

```
int 32 start; /* start of chunk */
int 32 end; /* end of chunk */

end = 50/np;
start = (0@tid)*end;
end = end * np;
end = 50 - end;

if ((unsigned)tid < (unsigned)end<-2)
{
   if (start != 0)
      start += 0@tid;
   end = start + 50/np + 1;
}
else if (end != 0 &&
      (unsigned)tid >= (unsigned)end<-2)
{
   start += end;
   end = start + 50/np;
}
else if (end == 0)
   end = start + 50/np;

for (i = start;i < end && i < 50;i++)
{
   /* parallel code */
}

barrier(tid);
```

The iteration space is divided among the threads as follows:

- Each thread is assigned one chunk of iterations.
- The size of each chunk assigned to each thread is approximately the same.
- Since the `for` construct does not have the `nowait` clause, a barrier function call is inserted.

The code snippet below illustrates how a log(*n*) barrier is implemented.

```
unsigned int 1 comm[np][np] = {0};
macro proc barrier(tid) {
   unsigned int 2 i;
   unsigned int 3 j;
   unsigned int 1 input;

   /* Each thread synchronizes with tid + 2^i
      where i is the round number*/
   i = 0;
   while(i != 2) {
      // precompute tid + 2^i
      // decode computes 2^i
      j = (tid<-3) + (decode(i)<-3);
      while(comm[i][j<-2])
        delay;
      // set partner's flag
      if (j < np)
        comm[i][j<-2] = 1;
      else
        comm[i][(j-np)<-2] = 1;
      while(!comm[i][(unsigned)(tid<-2)])
          delay;
        comm[i][(unsigned)(tid<-2)] = 0;
        i++;
    }
}
```

In addition to the above, we have also implemented translation for parallel sections, master-slave, critical sections, data sharing and log(*n*) reduction constructs.

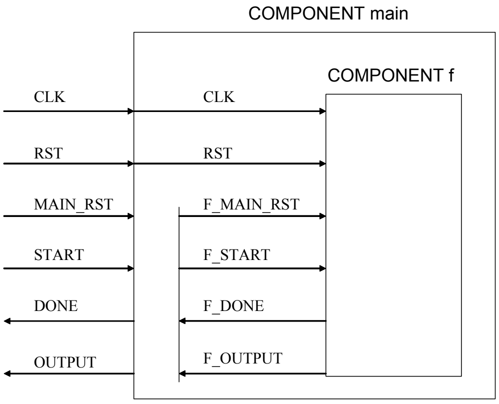

Fig. 1. Function call implementation.

The implementation in Handel-C shows how readily OpenMP code can be mapped to an existing C-based hardware description language. There are still some components missing in OpenMP when we compare it to a hardware description language like Handel-C that was designed from “ground up” for the purpose of specifying hardware. In particular, there is an absence of bit-width specifications, bit-wise and bit-parallel operations. We believe these operations may be too low level when coding in OpenMP. However, in the spirit of OpenMP, it is possible to achieve bit width specifications by means of pragmas. Bit-wise and bit-parallel operations can be realized as macros or (inlined) procedure calls.

IV. Generating Synthesizable VHDL From OpenMP Programs

In order to widen the choices of platform, we implemented a translation to synthesizable VHDL as well. Our implementation of OpenMP in VHDL uses the one-hot method. Each C function is initialized as a VHDL component using a pass-by-value implementation. The input parameters of a function are specified as input ports to the component. In addition to these ports, the ports in Table II are common to all generated components.

Fig. 1 is a visual representation of the generated VHDL component for a function call in which `main()` calls `f()`. The CLK and RST signals are globally routed to all components, while F_MAIN_RST, F_START, F_DONE and F_OUTPUT are internal signals of the `main()` component used to control its child component `f()`.

A. Private and Shared Variables

We have only implemented the `int` type, which represent 32-bit integers. This corresponds to the `integer` type in VHDL. Arrays of integers are also allowed, but only up to two dimensions, as the synthesis tool we used only allowed a maximum of three dimensional bit arrays (two dimensions for actual array and one dimension of 32 bits for integer).

Global variables are currently not supported due to the absence of a memory system in VHDL. However, a restricted form of shared variables internal to a function is allowed, namely shared variables written to by different threads. Due to the fact that in actual hardware registers, only a single signal input source is allowed, a register cannot be directly written to by more than one thread. To overcome this limitation, we implemented explicit multiplexing. A thread writing to a shared variable sends request signals together with their data to a multiplexer which we call a *coordinator*. The coordinator chooses a thread request in a clock cycle and sends an acknowledgement signal to the chosen thread.

Array writes are implemented as exclusive writes across all elements in an array. That is, only a particular element of an array may be written in a clock cycle. For arrays, the thread needs to supply the indices as the address to select the element of the array to write to. This is illustrated in the following code snippet.

TABLE II

COMMON INPUT AND OUTPUT SIGNAL FOR ALL VHDL COMPONENTS

CLK	a global clock signal used to drive all components for the circuit;
RST	a synchronous global reset signal which is set to high for 1 clock cycle at the start of the execution of a VHDL program to reset all FSMs to their initial states;
MAIN_RST	a local reset signal used to reset a child component or process to its initial state;
START	a signal sent by a parent component to a child component to signal the start of its execution;
DONE	a signal sent by a child component to a parent component to signal the end of its execution;
OUTPUT	return value of a component, which is set together with the DONE signal when a child component completes execution.

Coordinator	Thread Process `i`

State 0:
1. Scans request signal array, chooses thread process `i`
2. if (`has_request` == 1)
`ack[i]` 🡐 1;
`x[x_dim1[i],`
`x_dim2[i]]`
🡐 `x_data[i];`
Go to State 1.
State 1:
1. Do nothing. Wait for chosen thread process to de-assert request signal.
2. Go to State 0.
	State *n*:
	1. `req[i]` 🡐 1;
	`x_data[i]` 🡐 10;
	`x_dim1[i]` 🡐 1;
	`x_dim2[i]` 🡐 2;
	2. if `ack[i]` == 1 then
	`req[i]` 🡐 0;
	go to State *n*+1;
	State *n*+1:
	(next statement to execute)
	....
	....

In Fig. 2, we show the clock cycle analysis diagram for the following array write to a shared array `x`. Thread 0 is attempting to write the value 22 to `x[1][2]` while thread 1 wants to write 44 to `x[5][6]`.

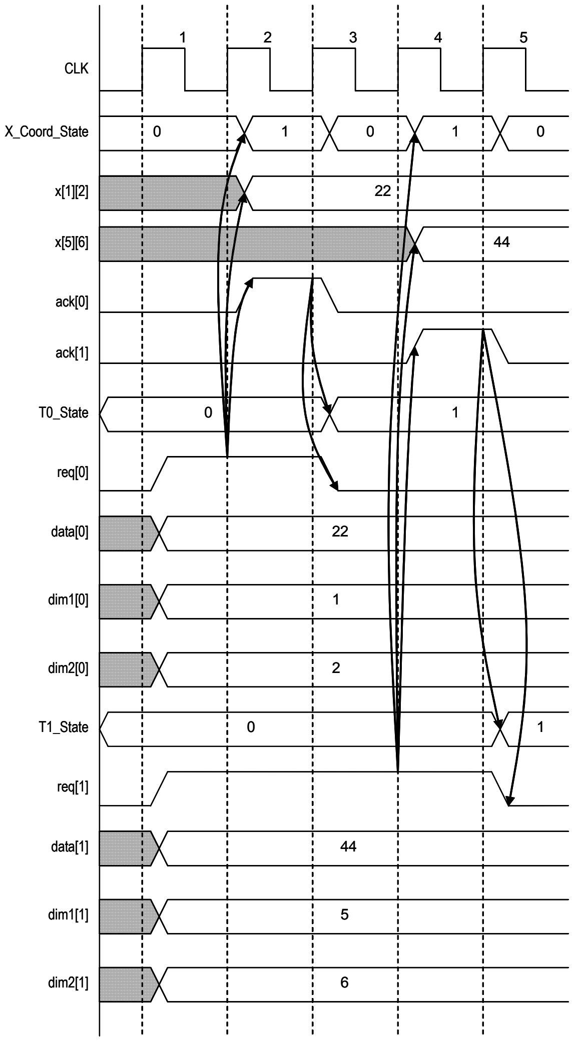

Fig. 2. Clock cycle analysis of a shared array write.

At the end of clock cycle 1, the coordinator sees that both thread 0 and thread 1 has `req[0]` and `req[1]` asserted. It selects the value of thread 0 to write, and at the start of the clock cycle 2, it asserts `ack[0]` to inform thread 0 that the write is complete. `x[1][2]` is also updated with the value of `data[0]` by the coordinator. At the end of clock cycle 2, thread 0 sees that the coordinator has asserted `ack[0]`, hence at the start of clock cycle 3 it de-asserts `req[0]` to stop its request, and proceed on to the next state. At the end of clock cycle 3, thread 1 finally gets a chance to write to the array. The coordinator selects thread 1 as the next write, asserts `ack[1]` and write `data[1]` to `x[5][6]`. At the end of clock cycle 4, thread 1 sees that `ack[1]` is asserted, hence at the start of clock cycle 5 it de-asserts `req[1]` to stop requesting, and moves to the next state.

B. Dealing with control flows

Due to the differences in the paradigms, there is a need to reduce control flows in the original C program into state machine operations. This is illustrated by the following code template:

1) Original C code
 	
if (a) then stmt1 else stmt2;
	2) C-Breeze's MIR code
	after dismantling
	
	If (a) then goto label1;
	goto label 2;
	label1:
	stmt1;
	goto label3;
	label2:
	stmt2;
	label 3:

3) Our modified MIR code
after dismantling
 	
Label0: {
if (a) then goto label1;
else goto label2;
}
Label1: {
stmt1;
goto label3;
}
Label2: {
stmt2;
goto label3;
}
Label3:
…….
	4) Final VHDL code
	
	case state is
	when S0=>
	if (a) then
	state <= S1;
	else
	state <= S2;
	end if;
	when S1=>
	stmt1;
	state <= S3;
	when S2=>
	stmt2;
	state <= S3;
	when S3 =>
	……..

C. Dealing with OpenMP pragmas

The subsections below give the details on how we dealt with the OpenMP pragmas. In most cases, one-hot finite-state machines are used.

Each computational expression is assigned a state. When the control FSM goes into that state, the logic circuit implementing that expression will be executed. Currently, each state only has one executing statement, but future optimization efforts can greatly reduce the number of states so as to reduce the clock cycles needed to run the program.

1) #pragma omp parallel

Each thread is implemented as a separate FSM, with the main thread controlling the execution of its child threads. There is an implicit barrier at the end of the parallel region. Currently a parallel region is restricted to its static region; it does not apply to functions called within a parallel region. Also, nested parallelism is currently not supported.

2) #pragma omp for

This construct distributes the iteration of a for-loop across the threads in a parallel region based on the schedule type specified. Currently, unlike the Handel-C implementation, only static schedule with a chunk size of one is implemented. For a region of n threads, thread i gets the $(I + n \times j)$ iterations of the loop where $j = 0,1,2,3,\ldots$

3) #pragma omp sections

Each block of code is moved to its respective thread at compile time, and since each thread has different sections of code, the circuit generated for each thread is more compact than in the case of duplicating all sections of the code for each thread and testing which section should be executed for a specific thread at run-time. There is an implicit barrier at the end of the sections region.

4) #pragma omp single

In our implementation, the first thread which arrive at the single construct will execute the single region. In the event that two or more threads arrive at the `single` contruct in the same clock cycle, the thread with the smaller thread number will execute the single region. A coordinator performs this selection.

5) #pragma omp critical

This is implemented in a similar manner as the single construct except that every thread will get a turn to execute the code in the critical region.

6) #pragma omp master

This construct specifies that a block of code is only executed by the master thread. In our implementation, only the master thread process contains this block of code, which means that only one copy of the generated circuit exists.

7) #pragma omp atomic

This construct specifies that a storage location is updated atomically.

```
#pragma omp atomic
x = x + expr
```

This is handled by transforming it into a critical section.

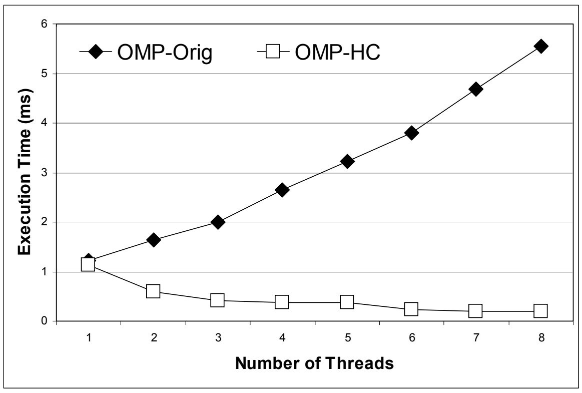

Fig. 3. Parallel matrix multiply in Handel-C.

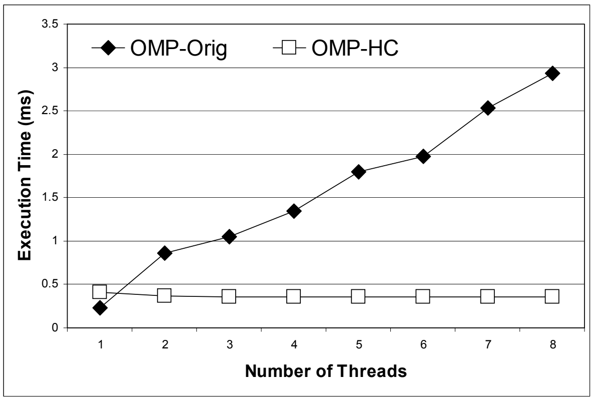

Fig. 4. Parallel Sieve of Eratosthenes in Handel-C.

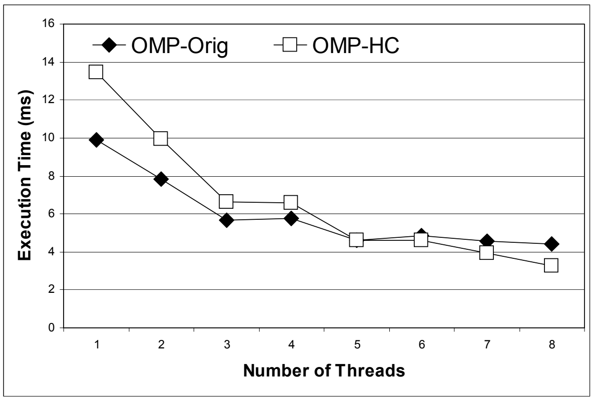

Fig. 5. Parallel Mandelbrot in Handel-C.

8) #pragma omp barrier

This is implemented as a logical-and of a bit array. Threads arriving at the barrier set its own bit to one, and test whether the logical-and of the bit array is equal to 1 at every clock cycle. Once the logical and is asserted, a thread will then set its bit to 0 and proceed to the next state.

.

V. Experimental Evaluation

Both Handel-C and VHDL implementation were tested on the Celoxica RC100 FPGA development board [2]. The RC100 board has a 200,000 gate Xilinx Spartan II FPGA, two off-chip Synchronous Static RAM (SSRAM) banks, Flash RAM, video decoder, video output subsystem as well as PS/2 ports for a keyboard and mouse. The board can be connected to the host computer using a parallel port cable where parallel port I/O can be performed between the host computer and the board. The Celoxica and Xilinx tool chains were needed to complete the synthesis and test cycles.

As a comparison, we also executed the same OpenMP code on a Sun Microsystem SunFire 4800 server. This Symmetrical Multiprocessor machine has eight UltraSPARC III 1.2-GHz processors and 8GB of shared memory. The Sun C compiler supports OpenMP natively. Optimizations were turned off. We believe this gives a fairer picture as our OpenMP implementations do not do much optimization at the moment either.

A. Performance of Handel-C Implementation

For the Handel-C implementation, we chose the matrix multiply, summation, reduction, prime testing and Mandelbrot set generation as our tests. Table III shows the actual hardware resources utilized on the FPGA for each of our Handel-C test programs. It shows the minimum clock period achieved, maximum clock rate achieved as well as the number of clock cycles used for program execution excluding clock cycles used for I/O and the calculated execution time on the FPGA.

For the parallel matrix multiply (see Fig. 3), we tested it on a 62×15 by 15×7 array multiply. We were limited by the resources of the RC100 board. The Handel-C implementation achieved a speedup of 5.9 for 8 threads. The software implementation, on the other hand, on the Sun server showed a slight slowdown. In the summation tests, because of the use of critical sections, the Handel-C implementation only achieved a speedup of 1.14.

For the Sieves of Eratosthenes (Fig. 4), the Handel-C implementation showed a speedup of about 14% while the software OpenMP version showed a siginificant slow-down.

For Mandelbrot (Fig. 5), the speedups for Handel-C and the software versions were 4.09 and 2.18, respectively.

B. Performance of VHDL Implementation

For the VHDL implementation, we included one more benchmarks – the infinite impulse response filter. For the other benchmarks, the results were similar as that for Handel-C. Across the tests, we achieved clock speeds of around 23 to 45 MHz.

Table IV shows the execution time for the prime sieving benchmarks. A fixed 20MHz clock was used in all the VHDL designs.

Table V shows the execution time for the infinite impulse filter benchmarks.

It should be pointed out that the aim of these tests is not to show that hardware implementation will always be better. An

TABLE III.

ACTUAL HARDWARE RESOURCES REQUIRED BY HANDEL-C BENCHMARKS

Program	Gates	FFs	LUTs	Slices	Clock Cycles	Clock Period (ns)	Clock Rate (MHz)	Execution
omp_mm.hcc	46052	1363	4685	2350	327	46.123	21.681	0.015082
sumup.hcc	40911	1339	3953	2350	217	39.694	25.193	0.008613
sumup_reduce.hcc	42045	1440	4011	2350	187	40.236	24.853	0.007524
PrimesOMP.hcc	43714	756	4021	2350	466	79.472	12.853	0.037034
mandel.hcc	46793	809	4576	2350	16933	42.152	23.724	0.713749

important caveat is that because of the sizes of our problems are limited by the FPGA board we are working on, they translate to communication and synchronization bound programs on the Sun server. Therefore, the comparison is not altogether fair for the software version. The tests do reveal however, that (a) our implementation is correct; and (b) it is physical evidence that the primary proposal of this paper, namely the use of OpenMP as a high level hardware description language is a feasible one.

TABLE IV

PARALLEL SIEVE OF ERATOSTHENES IN VHDL

No. of threads	VHDL time (ms)	OpenMP time (ms)
1	0.5725	0.283
2	0.4301	3.906
3	0.32985	7.976
4	0.26185	15.085
5	0.28485	22.461
6	0.28085	29.332
7	0.25285	36.841
8	0.24985	44.287

TABLE V

INFINITE IMPULSE FILTER IN VHDL

No. of threads	Clock period (ms)	Clock freq (MHz)	VHDL time (ms)	OpenMP time (ms)
1	12.463	80.238	0.02085	0.184
2	14.95	66.89	0.0241	1.003
3	15.537	64.362	0.02435	1.64
4	23.195	43.113	0.02525	1.755
5	24.065	41.554	0.02395	2.945
6	26.677	37.485	0.0241	3.866

VI. RELATED WORKS

The need for high-level descriptions of hardware has been recognized widely by the industry and academia [14]. The hope is that this will enable reuse with the same ease as software, resulting in greater productivity. Besides the very popular Verilog and VHDL, several hardware description languages based on Java [1], Lola [4], C [5], ML [6], and Ruby [8] have been proposed. There are also new languages like Pebble [9].

Dziurzanski and Beletskyy [3] first proposed the subseting and extending OpenMP for the purpose of hardware description. However, as far as we know, ours is the only actual implementation of OpenMP as a high level hardware description language via translations to Handel-C and VHDL. We showed actual benchmarks using our implementations.

VII. CONCLUSION

In this paper, we proposed the use of OpenMP as a hardware description language. OpenMP leverages on the idea of "active comments" as the key to transit from sequential to (shared-memory) parallel programs. An OpenMP program should, in principle, compile and run transparently on a sequential or a parallel processor. We believe that this is an attractive philosophy for a medium grain, rapid prototyping high level hardware description language. It is unlikely that an OpenMP hardware description will yield the most optimal hardware. But it helps speed up the process of obtaining *a* working prototype, especially if there is already some software counter-part. We hope that this will be a useful addition to the repertoire of hardware description languages available to hardware designers.

REFERENCES

[1] P. Bellows and B. Hutchings, "JHDL - an HDL for recongurable systems", in *Proc. FCCM98*, IEEE Computer Society Press, 1998.

[2] Celoxica Inc., *The RC100 Hardware Reference Manual.*

[3] P. Dziurzanski and V. Beletskyy, "Defining Synthesizable OpenMP Directives and Clauses", M. Bubak et al. (Eds.): *ICCS 2004*, LNCS 3038, pp. 398–407, 2004.

[4] S. Gehring and S. Ludwig, "The Trianus system and its application to custom computing", in Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, LNCS 1142, Springer, 1996.

[5] M. Gokhale and E. Gomersall, "High-Level compilation for fine-grained FPGAs", in Proc. FCCM97, IEEE Computer Society Press, 1997.

[6] Y. Li and M. Leeser, "HML: an innovative hardware description language and its translation to VHDL", in *Proc. CHDL'95*, 1995.

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

[8] W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A framework for developing parametrised FPGA libraries", in *Field-Programmable Logic, Smart Applications*, New Paradigms and Compilers, LNCS 1142, Springer, 1996.

[9] W. Luk and S. McKeever, "Pebble: a language for parametrised and reconfigurable hardware design", in *Field-Programmable Logic and Applications*, R.W. Hartenstein and A. Keevallik (editors), LNCS 1482, pp. 9-18, Springer, 1998.

[10] OpenMP. http://www.openmp.org.

[11] I. Page, "Constructing hardware-software systems from a single description". *Journal of VLSI Signal Processing*, 12(1), pp. 87-107, 1996.

[12] D. L. Perry, *VHDL*, 2nd edition. McGraw-Hill series on computer engineering) ; New York, 1994.

[13] C. Peters, "Overview: Hardware Compilation and the Handel-C language".
http://web.comlab.ox.ac.uk/oucl/work/christian.peter/overview_handelc.html.

[14] R.A. Walker, R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, Boston, Ma, 1991.