Examining Implementations of a Computationally Intensive Problem in GF(3)

Joey C. Libby
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
g6x2d@unb.ca

Jonathan P. Lutes
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
f9dz2@unb.ca

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
ken@unb.ca

ABSTRACT

Computing the irreducible and primitive polynomials under GF(3) is a computationally intensive task. A hardware implementation of this algorithm should prove to increase performance, reducing the time needed to perform the computation. Previous work explored the viability of a co-designed approach to this problem and this work continues addressing the problem by moving the entire algorithm into hardware. Handel-C was chosen as the hardware description language for this work due to its similarities with ANSI C used in the software implementation. A hardware design for the algorithm was developed and optimized using several different optimizations techniques before arriving at a final design.

Optimization; Handel-C; Galois Fields

1. INTRODUCTION

The performance of many software systems can be improved by the creation of custom hardware circuits that are capable of performing some or all of a software systems processing in a native hardware environment [8,9,10,11]. One major reason that software is implemented in hardware is the core features that a hardware implementation offers a system designer. The most important of these features is the inherent parallelism that is found in hardware systems such as Field Programmable Gate Arrays (FPGA).

The work presented in this paper is a continuation of work started in [1] and centers around the creation [14] of migrating a software system for the computation of irreducible and primitive polynomials over GF(3) completely to hardware, the issues that surrounded the migration and optimizations that were applied to the final hardware design using an automated parallelism extraction tool. The original work [1] concentrated only on implementing the computation intensive *multmod* function of the GF3 algorithm in hardware.

Further work presented includes further optimization of the design described in [14] and a discussion of several optimization techniques that were used to perform these optimizations.

This paper begins by presenting a brief background on some of the subject matter deemed relevant to the proper understanding of this paper. Following this the original research that lead to this project is presented, as well as work from another project on automated extraction of parallelism from Handel-C hardware definitions. The full hardware design is then presented followed by a discussion of the different optimization techniques used to optimize the design, as well as benchmarking results for each of the techniques.

2. BACKGROUND

This section will discuss the background information that is necessary for understanding this paper. This discussion includes Handel-C [2], Galois Fields [3] and the previous work that was completed.

2.1 HANDEL-C

The hardware implementation for this work was implemented in Handel-C [2]. Handel-C is a high level hardware description language that bears much resemblance to the ANSI C programming language. While Handel-C is very similar to ANSI C in many respects, there are some major differences between the two languages. Handel-C does not support the entire ANSI C specification. One of the more important features removed from Handel-C is support for runtime recursion. Handel-C, along with support for a subset of the ANSI C

specification, includes extra support for hardware descriptions. Included in this extended support are constructs for input and output, communications, and control flow constructs for controlling the parallelism of a design. Parallelism in a Handel-C program is defined by using the `par{}` and `seq{}` statement blocks. Sequential instructions wrapped in a `par{}` statement will be executed in parallel, while statements wrapped in a `seq{}` statement will be forced to execute sequentially. Example 1 shows `par` and `seq` statements in a simple Handel-C design.

The absence of runtime recursion support in Handel-C proved to be one of the more challenging aspects of this work. In most cases recursive algorithms can be easily converted to a non-recursive, loop based algorithm. This would prove to be problematic during the course of this work as several of the recursive functions written in the C algorithm proved to be resistant to conversion loops.

```
int 8 a,b,c,d,e,f,g,h;
a = 1; b = 2; c = 3; d = 4;
seq {
        d = a + b;
        e = c + d;
}
par {
       f = d+e;
       g = d*e;
}
```

Example 1: Example of `par` and `seq` Statements

2.2 GALOIS FIELDS AND THE ALGORITHM

A Galois Field is a finite order denoted by GF(p), where p is a prime or a power of primes [3]. A Galois Field of order p has only p elements, 0 though p-1. The focus of the algorithm implemented for this paper is Galois Fields of the order GF(3). These fields are of interest due to their application in pairing based cryptographic systems [4].

The C algorithm discussed in this paper describes the problem of enumerating all of the primitive and irreducible polynomials of a given order [5]. Irreducible polynomials are polynomials such that p(x) in F(x) is called irreducible over F if it is non-constant and cannot be represented as the product of two or more non-constant polynomials from F(x) [3]. A primitive polynomial is a polynomial such that F(X), with coefficients in GF(p) = Z/pZ, is a primitive polynomial if it has a root α in GF(pm) such that $\{0, 1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{p^m-2}\}$ is the entire field GF(p^m), and moreover, $F(X)$ is the smallest degree polynomial having α as root [3].

The C algorithm consists of a number of functions that will now be detailed. Where applicable functions that are recursive are noted.

Add: Adds two polynomials under GF(3).
Subtract: Subtracts two polynomials under GF(3).
Mod: Takes the modulus of two polynomials under GF(3).
GCD: Find the greatest common divisor of two polynomials under GF(3) (recursive).
Multmod: Multiplies two polynomials under *mod* p.
Powmod: Finds the result of one polynomial raise to the power of another polynomial under GF(3).
Minpoly: Finds the minimum polynomial given a necklace.
Gen: Controls execution of the algorithm by cycling over all possible necklaces (recursive).

2.3 THE CO-DESIGNED SOLUTION

The previous implementation of the C algorithm did not attempt to migrate the entire software algorithm into a hardware system. Instead it was decided to explore a co-designed approach [1] where only a portion of the software would be translated into a hardware design and this hardware module would be called from the software running on a general purpose CPU.

2.3.1 THE CO-DESIGN

The first task was to determine how to partition the hardware and software for this project. Timing analysis of the original software showed that the multmod function, which computes multiplication between polynomials, is the most processing intensive portion of the software. It was found that 80% of the total processing time was spent in the multmod function. It was decided that a suitable solution to improve the performance of the system would be to implement the multmod function in hardware.

The design of the hardware was created based on careful analysis of the operations that are performed in the multmod function. These operations were then placed in a module called the Arithmetic Computation Unit (ACpU). Figure 1 shows a flowchart of the operations performed when executing a call to the multmod function.

The hardware partition also contains a module responsible for controlling the execution of calls to the multmod hardware. This module is called the Arithmetic Control Unit (ACtU). The ACtU is a finite state machine that is responsible for sequencing the operations that take place in the ACpU. The main requirement of this state

machine is to increase the performance of the hardware module by leveraging as much parallelism as is possible in the multmod operation. The design of the state machine for the ACtU can be seen in Figure 2.

Two different implementations of this co-designed hardware were then created. The first implementation was designed to be housed on an FPGA located on a board connected to its host computer via a PCI bus. Using this configuration software running on the host machine communicates with the hardware device via the PCI bus. One of the issues with this design, however, was communication delays between the hardware and software. In order to remove this communications gap, it was decided to explore another option using a System-on-chip architecture. In this design the FPGA is very closely integrated with an embedded processor. This allows the software partition to communicate with the custom hardware directly, without the use of the time costly PCI bus.

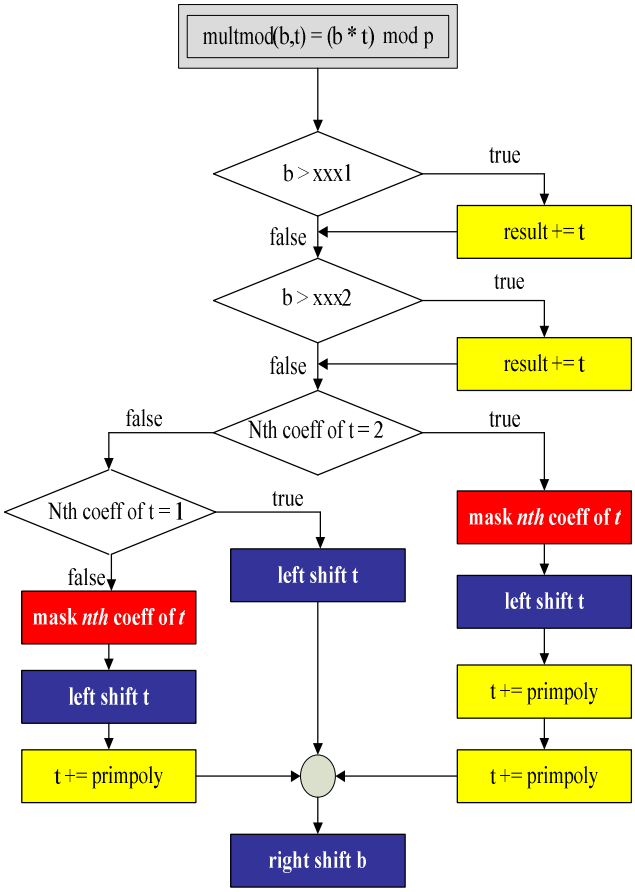

Figure 1: Arithmetic Computation Unit

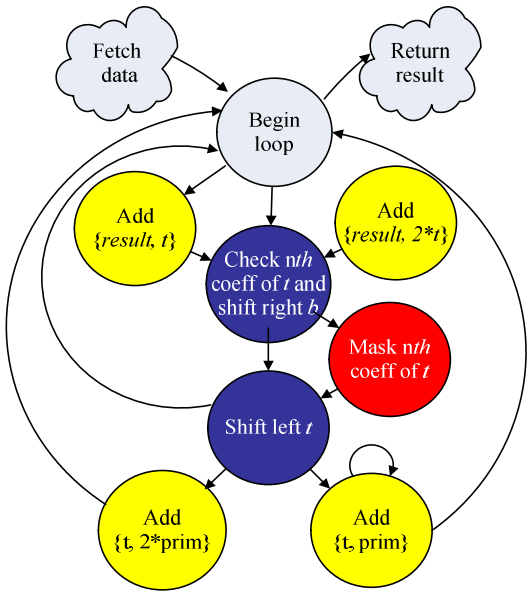

Figure 2: Arithmetic Control Unit

2.3.2 THE PCI-BASED SOLUTION

For the PCI-based solution the reconfigurable hardware receives and sends data to the software partition over a PCI bus. For this implementation the software was running on a Windows 2000 PC with a 1.8 Ghz Intel Xeon processor with 1 GB of RAM. The FPGA development board used for this project is an APEX PCI Development board with an Altera Apex 20KC1000CF672 FPGA [15] supporting 32 and 64 bit PCI communications at 33 and 66 Mhz [16]. The software running on the host machine is a modified version of the original software implementation. The modifications allow the software to call the hardware when necessary for completing computations. A high level view of the structure of this loosely coupled co-designed system can be seen in Figure 3.

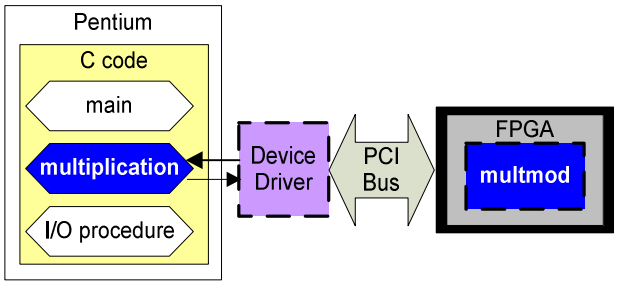

Figure 3: Overview of loosely coupled system

In order to interface the software partition with the hardware device it was necessary to develop a PCI device driver for the system [17].

The modifications to the original C code allowed all calls made to the multmod function to be redirected to the hardware device. In order to do this, input data for the calls to the multmod function are redirected to the PCI device driver. The device driver is then responsible for initiating communications with the FPGA and sending all required data for the call. In this implementation the configuration is static and must be downloaded to the FPGA before runtime. This configuration contains four modules: the ACpU, ACtu, PCI Core and PCI control unit.

The hardware implementation of the *multmod* algorithm was implemented in Verilog and was targeted to an Amirix AP1000 development board [6]. This development board was chosen as the target platform because of its on-chip PowerPC processor that is directly connected to the FPGA fabric. This feature allowed the software to be executed on a platform that is more tightly coupled with the FPGA and removed the need to create a PCI bus driver for the work. Figure 4 shows an overview of the co-designed system and Table 1 shows the benchmarking results for this implementation.

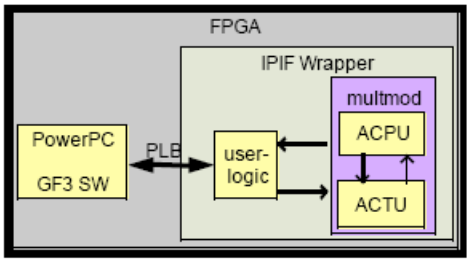

Figure 4: Co-Designed System Overview[1]

While a performance increase was realized by moving to the co-designed system, it was found that several factors severely limited the overall performance of the system. The slow speed of the embedded processor running the software portion of the system was one issue that arose. The 200mhz clock speed of this processor simply was not fast enough to hold pace with the faster general purpose processors that would normally run the full software implementation [1].

Also a major problem, more so than the slow clock rate of the embedded processor running the software portion of the system, is the communications between the hardware and the software system. Communications prove to be the Achilles heel of this work, as well as many other co-design works [7]. The amount of data communications that is necessary between the hardware and the software is so great that it limits the maximum throughput of the system, which has a huge impact on performance.

The only solution to this problem is to move the entire system into hardware, completely eliminating the communication channels. This will allow the system to operate at full speed, only having to access communication channels when retrieving jobs and reporting results.

2.4 AUTOMATED EXTRACTION OF PARALLELISM

Identification of simple parallelism, that is sequential blocks of hardware code that can be executed in parallel, can have a huge impact on the performance of the hardware system that is being designed. The tool that will be used to apply optimizations for the purpose of this paper can be found in [12].

Given a HandelC source file, this tool is capable of parsing and extracting simple parallelism from the source file. This information is then relayed to the hardware designer who can implement the proposed changes in order to build a more optimized version of the original hardware design.

Figure 5 shows an overview of the operation of the automated parallelism extraction tool. The tool operates by taking, as input, a HandelC hardware definition file provided by a software programmer or hardware designer. From this source file the tool creates an abstract syntax tree, annotated with additional information that is required to compute the dependency graph from the source file. Upon completion of the syntax tree, it is used to generate a dependency graph structure for the hardware design. This dependency graph structure is then used to determine which individual lines of source can potentially be executed in parallel. Currently the tool then applies a greedy algorithm which builds as large and as many parallel blocks as possible from the remaining available lines of source code. This approach generates large parallel blocks which in turn reduce the overall run time of computations on the hardware.

Once the tool has determined where parallel blocks may be added to the source hardware design, it produces a report for the developer that details the necessary modifications that must be performed in order to exploit the available parallelism. Table 2 illustrates the report output of the tool. After potentially several iterations with the tool, the developer can then input their HandelC specification into the typical tool flow starting with the Agility DK tool suite [2].

Degree	SFW @1.8GHz (sec)	Altera @66MHz (sec)	%	Xilinx @80MHz (sec)	%
2	0.00004	0.00002	*37.6*	0.00001	*24.8*
3	0.00018	0.00009	*46.6*	0.000056	*30.7*
4	0.00076	0.00034	*44.2*	0.000228	*30.0*
5	0.00298	0.00145	*48.6*	0.001005	*33.7*
6	0.01495	0.00519	*34.7*	0.003664	*24.5*
7	0.04043	0.02005	*49.6*	0.01439	*35.6*
8	0.14300	0.07123	*49.8*	0.051788	*36.2*
9	0.54600	0.25595	*46.9*	0.188174	*34.5*
10	1.89800	0.89412	*47.1*	0.663513	*35.0*
11	6.24000	3.08083	*49.4*	2.302203	*36.9*
12	22.30800	10.58020	*47.4*	7.963267	*35.7*
13	74.78900	35.12896	*47.0*	26.53804	*35.5*
14	263.53600	120.09697	*45.6*	91.323996	*34.7*
15	888.30300	400.77343	*45.1*	306.075094	*34.5*
16	2985.50200	1343.56091	*45.0*	1049.41517	*35.9*
17	10192.85900	4424.87400	*43.4*	n/a	n/a
18	32658.34090	14642.10675	*44.8*	n/a	n/a

Table 1: Co-designed Performance Results [1]

In testing, this tool has proven that it is capable of finding, on average, 78% of the simple straight line parallelism that exists in a hardware design. Tables 3 and 4 shows manual and automatic optimization benchmark results for AES encryption and LZ77 decryption hardware circuits optimized using this tool. [12]

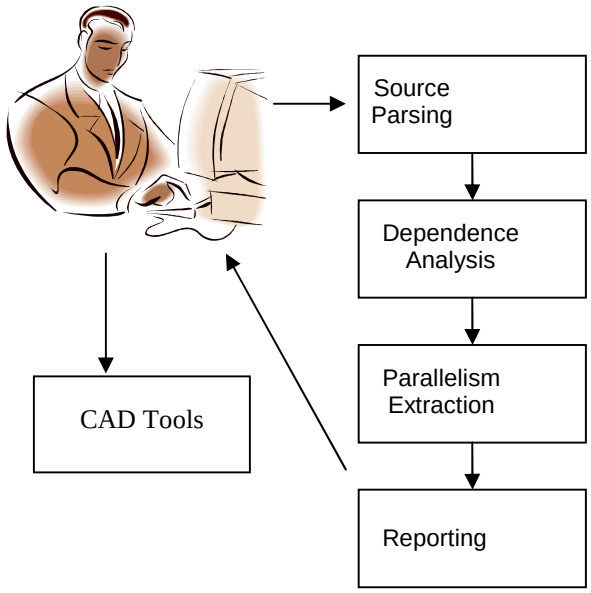

Figure 5: Automated Parallelism Extraction

Source Line#	Source	Action
*	Par {	Add
1	Statement 1	None
2	Statement 2	None
3	Statement 3	None
7	Statement 7	Move -
8	Statement 8	Move –
*	}	Add
4	While	Move +
5	Statement 5	Move +
6	Statement 6	Move +
9	Statement 9	Move +

Table 2: Tool Report [12]

Test	Par Blocks
LZ77	5
AES	13

Table 3: Manual Benchmark Statistics [12]

Test	Par Blocks Found	% of Total
LZ77	4	80%
AES	10	76%

Table 4: Automated Benchmark Statistics [12]

3. THE HARDWARE SOLUTION

In order to alleviate the performance degradation caused by communications between the hardware and software in the co-designed system, as well as the low performance of the general purpose processor, a full implementation was created in hardware. This implementation was written in Handel-C which allowed the hardware implementation to very closely mimic the software algorithm wherever possible.

Much of the ANSI C code that was created for the algorithm was capable of being directly translated into Handel-C. The code that was directly translated required only minimal modification to make it compatible with the Handel-C language. Some of these changes included re-definition of storage elements such as arrays to use static sizes instead of being dynamically allocated. Another trivial modification that was required in several places was the un-nesting of function calls. Handel-C does not

support the usage of nested function calls of the form `foo( bar(x,y),z)`. This necessitated rewriting some C code to call these functions sequentially using temporary variables to store the return value of the nested function call.

3.1 REMOVING RECUSION FROM THE ALGORITHM

Once the code was converted to Handel-C syntax all that remained was removing the recursion that exists in several of the functions in the software. The functions that required modification to remove recursion were the Gen and GCD functions. Both functions were translated to their loop based variants. Example 1 shows how the recursive function definition for the GCD was transformed into a loop.

```
Poly_GF3 gcd(Poly_GF3 a, Poly_GF3 b){
  if(!b.top && !b.bot) return a;
  return gcd(b, mod(a, b));
}
```

Example 1 (a): Recursive GCD Definition

Once the recursion was removed from the software functions they were implemented in Handel-C. Following the implementation in Handel-C, each function required verification to ensure that the hardware versions were equivalent to their software counterparts.

```
Poly_GF3 gcdx(Poly_GF3 a, Poly_GF3 b )
{
  Poly_GF3 c,zero;

  zero = {0,0};
  while (a.top || a.bot)
  {
     c = a;
     modx(b,a);
     a = modxResult;
     b = c;
  }
  return b;
}
```

Example 1 (b): Non Recursive GCD Definition

One of the larger challenges for this project was the removal of recursion from the Gen function. Example 2 shows the Gen function with a manual stack.

```
inline void Push(int *pos, int t, int
p, int j, int s) {
  stack[*pos][0] = t;
  stack[*pos][1] = p;
  stack[*pos][2] = j;
  stack[*pos][3] = s;
  (*pos)++
}

void Gen(unsigned t, unsigned p) {
  unsigned int j, top, state;

  top = 1;
  Push(top, 1, 1, 0, 0);
  top--;
  while (top > 0) {
    top--;
    t = stack[top][0];
    p = stack[top][1];
    j = stack[top][2];
    state = stack[top][3];
    if (t > N) {
      if (p == N) CheckIt();
    }
    else {
      switch (state) {
        case 0: {
          a[t] = a[t-p];
          Push(top, t, p, 0, 1);
          Push(top, t+1, p, 0, 0);
          break;
        }
        case 1: {
          j = a[t-p] + 1;
          if (j <= 2) {
            a[t] = j;
            Push(top, t, p, j, 2);
            Push(top, t+1, t, 0, 0);
          }
          break;
        }
        case 2: {
          j++;
          if (j <= 2) {
```

```
            a[t] = j;
            Push(top, t, p, j, 2);
            Push(top, t+1, t, 0, 0);
          }
          break;
        }
        default: { break; }
      }
    }
  }
}
```

Example 2: Non Recursive Gen Function

In order to remove recursion from the Gen function to make it compatible with hardware it was necessary to express the recursion in the form of a loop. One of the issues with recursion in hardware is the requirement for virtually unlimited memory within which the recursion can grow. It was found through careful analysis that the depth of the recursive calls to the Gen function was bounded by the input value N. Because of this upper bound it became possible to express the recursion as a loop by creating a stack on which the state of each "recursive call" would be saved.

Each iteration of the loop would emulate a recursive call to the Gen function, pushing the previous state onto the stack and then beginning computation on the new state. Once computation is finished in the final recursive call, the result is computed by tracing back through the states stored on the stack to compile the final result.

This method provides a means to implement the Gen function in hardware, but it does have some limitations. In order for this method to work it is necessary to have a bound on the size of the computation. In this case the input value N is known at compile time, allowing the size of the stack to be sufficiently large. In some cases, however, the input size may also be limited by the available hardware resources, as a very large input value may cause the stack to consume all available resources, or more resources than are actually available for the design. For this particular application, the stack size grows to a rather small size, N, where the algorithm is exponentially bound by the same N. Thus, for this application we can easily fit the stack in the dedicated memory bits that are available on the FPGA device.

3.2 HARDWARE VERIFICATION

In order to verify that the hardware functions, especially the functions that were transformed from recursive to non-recursive, behaved as intended it was necessary to perform some verification tests. Test cases included boundary cases as well as a large number of randomly generated inputs to the functions.

Verification of the transformed recursive functions was performed in two stages. In the first stage, the non-recursive algorithm was tested as a software algorithm. Test cases were run against both the recursive and non-recursive versions of the functions and their return values were compared. Following running the test cases on both the recursive and non-recursive functions it was deemed that the recursive and non-recursive functions were both functionally equivalent and so passed verification.

Verification of the Handel-C hardware code was slightly more involved than testing software code against software code. The Handel-C hardware code was again tested using the same set of test cases used for testing the recursive functions. These test cases were first ran in the software version of the system, recording the results for each test. The same tests were then performed on each hardware function individually, running the hardware in a simulation environment. The results were also recorded and compared to those produced by the software for the same tests.

Following verification of the hardware definition it was deemed that the hardware definition is equivalent to the software algorithm so the work could proceed to benchmarking.

4. BENCHMARKING

In order to benchmark the hardware design of the GF(3) algorithm, it was necessary to synthesize the hardware definition to produce a hardware programming file. It was decided that the hardware would not be programmed onto a physical device for testing, but tests would be performed in a simulation environment in order to facilitate the gathering of statistics.

The Handel-C definition was first compiled using the Agility Handel-C compiler to produce both an executable simulation file as well as a synthesizable VHDL description file [2]. The execution simulation kernel was used to gather timing results for the hardware system and the VHDL description file was used to gather resource usage and clock speed statistics. Resource usage and clock speed statistics were gathered by synthesizing the VHDL specification in Xilinx ISE [18] targeting a Virtex II FPGA (XC2VP100). This FPGA is the same device used for gathering the results for the co-designed GF(3)

algorithm. The results in Table 5 show the resource usage and clock frequency for the design.

Clock Speed	Slices	Flip flops
68.523	23952	14579

Table 5: Resource Usage and Clock Frequency

Runtimes for the hardware were gathered by running the simulation kernel on different degrees ranging from 3 to 12. Cycle statistics were gathered for each run, and using the clock rate gathered from the Xilinx synthesis tool a run time was calculated. These run times are compared to the runtimes of the software in Table 6. Software run times were gathered on a 1.8 Ghz Pentium .

N	Cycles	HW Time (Seconds)	SW Time (Seconds)
3	15158	0.000212	0.061
6	899241	0.0131	0.075
8	13052272	0.1905	0.241
10	170959343	2.4949	2.102
12	2072543280	30.2495	26.603

Table 6: Runtime Comparison

On inspection of the results, it can be clearly seen that the hardware version of the algorithm, in its current form, does not surpass the performance of the software algorithm. While the hardware algorithm does not perform better than the software, the performance gap between the two is negligible when taking into account the speed grade difference between the hardware running at 68.523 Mhz and the software running on a 2.8 Ghz processor.

Taking this into account it was decided to attempt to improve the hardware design further by attempting to optimize the design for a hardware environment. Until this point the software had been converted to a hardware definition almost verbatim, ignoring any of the traditional hardware specific features such as parallelism.

5. OPTIMIZATION

The optimization that was chosen for this design was the addition of parallelism. The software design did not take into account any of the areas of parallelism that might lead to greater performance for the hardware system. For the purpose of this work, only simple optimizations were attempted. Individual statements that were capable of parallel execution were grouped into parallel blocks using the Handel-C `par` construct.

The parallel blocks were identified using a combination of both an automated parallelism detection tool [12] as well as manual optimization. This tool allows for the automatic identification of code that can potentially be executed in parallel. Currently the tool does not modify the Handel-C source directly and requires intervention from the designer to take advantage of code that is identified as parallel. The automated tool found a large portion of the available parallel blocks, and then manual code inspection was used to find more parallel blocks that the tool was unable to identify.

Clock Speed	Slices	Flip flops
68.909	23603	14383

Table 7: Resource Usage and Clock Frequency

The design was then simulated to gather clock cycle statistics for running the design at several different input values. These clock cycles were used, along with the clock rate statistic from Table 7 to generate the final runtime statistics for the new hardware which can be found in Table 8.

N	Cycles	Percent Reduction	HW Time (Secs)	SW Time (Secs)
3	10916	27.9%	0.000158	0.061
6	581396	42.4%	0.00843	0.075
8	8319569	36.3%	0.1207	0.241
10	108497030	36.5%	1.5745	2.102
12	1312560988	36.7%	19.0477	26.603

Table 8: Parallel Runtime Comparison

In order to emphasize the impact of using automated software for identifying parallelism optimizations in this hardware design it should be noted that the automated optimization process took less than a second to complete. Even in this case where it was found that several parallel blocks that were identified did not compile correctly the amount of time saved from doing a completely manual optimization is quite high. It was found in [13] that manually optimizing this design, with no previous knowledge of the available parallel blocks takes a skilled hardware designer approximately 8 hours of testing and refining. Even taking into account approximately one hour of testing to identify the two parallel blocks that did not behave properly after compilation, a time savings of approximately 7 hours was achieved.

After automated optimization of the hardware algorithm was complete, a brief manual inspection of the

remaining code was performed. This inspection yielded 8 more parallel blocks that were not found by the automated tool for a total of 32 `par` blocks of two or more sequential statements. This additional manual optimization took approximately two hours to complete. Parallel execution statements (`par{}`) were added to the design and the design was recompiled, again producing both a simulation kernel and a VHDL definition file for hardware synthesis. Table 9 shows the synthesis results gathered from the Xilinx ISE, again targeting the Virtex II FPGA (XC2VP100).

Clock Speed	Slices	Flip flops
68.813	23348	14245

Table 9: Resource Usage and Clock Frequency

Using the clock speed from Table 9 and the cycle statistics gathered from the simulation kernel the runtime statistics for the hardware algorithm can be calculated. Table 10 shows the new runtime statistics for the parallel hardware design. Also shown in Table 10 is the percentage reduction of clock cycles between the original non-parallel design and the parallel design.

Table 9 shows that a small increase, 0.290 Mhz, in clock speed was realized when moving from the non-parallel to the parallel design. The number of slices and flip flops utilized by the design was also reduced slightly. This is explained by the manner in which the handelC specification is synthesized. During synthesis the handelC code is transformed into a datapath and a finite state machine controller. By using `par{}` statements, states within the FSM controller are merged, thus reducing the size of the circuit. Contrary to some thoughts, the par{} statement does not add redundancy, multiple computational units, unless a shared function is used. Thus, exploiting parallelism typically gives the benefit of both faster computation and a smaller circuit. Figure 6 shows a comparison of the parallel and non-parallel hardware against the software implementation.

N	Cycles	Percent Reduction	HW Time (Secs)	SW Time (Secs)
3	8621	43.1%	0.000125	0.061
6	548189	39.0%	0.0079	0.075
8	8089562	38.0%	0.1176	0.241
10	106849548	37.5%	1.5527	2.102
12	1352768511	34.7%	19.6586	26.603

Table 10: Parallel Runtime Comparison

It can be seen in Figure 6 that the parallel version of the hardware outperforms the software implementation of the algorithm at all data points gathered for this work. It also appears that the hardware will continue to outperform the software even when computing orders higher than 12. Figure 7 illustrates the trend in the percentage difference between the hardware and software algorithms. This figure clearly shows that the rate of convergence between the hardware and software run times is slowing and that the hardware will continue to outperform the software.

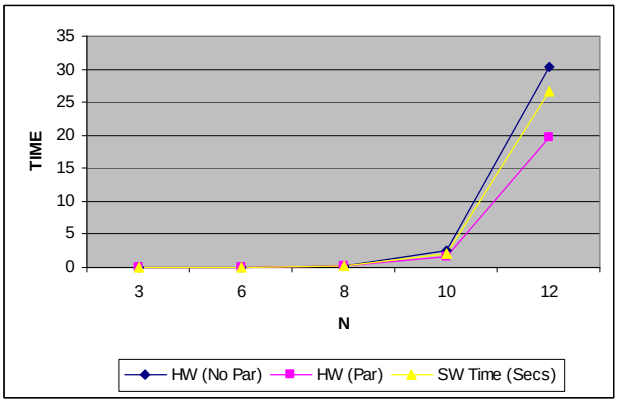

Figure 6: Results Comparison

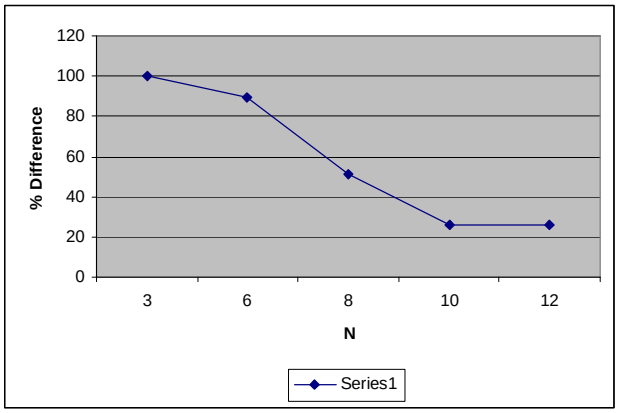

Figure 7: Execution Time Percentage Difference Between Parallel Hardware and Software

5.1 Further Optimizations

After completion of benchmarking of the parallelized design it was decided that an attempt would be made to identify any other optimizations that were possible in the design. After careful analysis of the design source code it was determined that there were two such types of optimizations that were suitable for this design.

The first optimization that was performed modified the method used to return values from function calls. In the original version of the hardware design, returns from non-recursive functions were stored in global variables,

and then assigned to the local variables as a result upon returning from the function call. This was done initially to reduce the amount of logic that was required in order to allow some functions to be called in parallel. Performing this modification entailed simply rewriting the function to return a value instead of writing it into a global variable.

The second optimization that was performed was also a side effect of applying the return value optimization. Changing the return from a global variable to an actual return value caused many function calls that could possibly be made in parallel to break. In order to fix this all of these functions were changed to inline functions. What this means is that instead of the compiler synthesizing control logic for a single logic core for each function, it creates a new instance of the logic for each function every time that function is called. This optimization has the effect of increasing the resource usage of the design, but decreasing the overall runtime of the design by reducing the amount of steps required to perform a function call.

Upon completion of these new optimizations the design was once again benchmarked, and the resource usage and clock rate statistics shown in Table 11 were gathered.

Clock Speed	Slices	Flip flops
58.998 Mhz	35991	34945

Table 11: Resource Usage and Clock Frequency

A simulation of the newly optimized hardware design was then performed, and cycle statistics for several input values were gathered. These cycle statistics, in conjunction with the clock rate given in Table 11 provide the actual runtimes shown in Table 12.

N	Cycles	Percent Reduction	HW Time (Secs)	SW Time (Secs)
3	7339	14.87	0.000124	0.061
6	464256	15.31	0.00786	0.075
8	6776626	16.23	0.11486	0.241
10	88118822	17.53	1.49356	2.102
12	1089884747	19.40	18.4732	26.603

Table 12: Parallel Runtime Comparison

As can be seen in Table 12, the newly optimized version of the hardware does indeed increase the performance of the design by an average factor of 1% over the parallelism only design.

Table 11 shows a comparison of the hardware run time of all four version of the hardware. This table illustrates the differences in performance gains along with the large amount of performance that was gained through the final round of optimizations.

N	Original	Automated Parallelism	Manual and Automated Parallelism	Final
3	0.000212	0.000158	0.000125	0.000124
6	0.0131	0.00843	0.0079	0.00786
8	0.1905	0.1207	0.1176	0.11486
10	2.4949	1.5745	1.5527	1.49356
12	30.2495	19.0477	19.6586	18.4732

Table 13: Comparison of all Designs

5.2 Comparison of Multmod Software and Hardware

This section will show a comparison of the hardware implemented for the Multmod function against the original software implementation. This section is meant to support that findings of this paper by showing that not only does the entire system outperform the software implementation of the GF(3) software, but also outperforms the implementation of the Multmod function which was implemented in hardware originally in [1].

Figure 8 shows the results that were gathered for these benchmarks. These statistics were gathered by adding cycle accurate counters to the final optimized version of the Multmod function, and then rerunning all of the simulations used previously. Statistics from these cycle counters were then gathered.

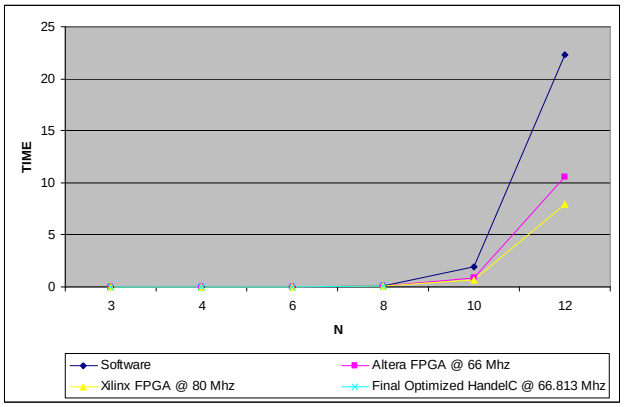

Figure 8: Multmod Comparison

Figure 8 clearly shows that while the optimized HandelC design runs at a slower clock rate then the Xilinx implementation, it still outperforms this implementation. This highlights the speed improvements that can be achieved by applying specific optimizations to a hardware design, as well as the speed improvements from moving from a co-designed environment to a full hardware implementation.

6. Conclusion

Based on the results gathered after optimizing the Handel-C design for the GF(3) primitive and irreducible polynomials algorithm it can be said that this work is a success. The entire algorithm was implemented in hardware and verified to function correctly. The results found in Section 5 highlight the performance of the hardware system, which outperforms the software on all test points up to order 12. It also appears that, based on Figure 2, the software will continue to outperform the hardware on higher orders. Figure 9 shows a comparison of the final results gathered. This figure clearly shows that the final optimized version of the hardware outperforms the original software version at all collected data points.

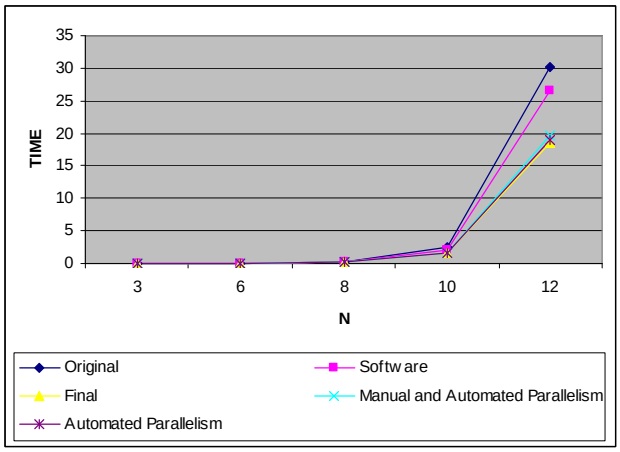

Figure 9: Final Results Comparison

Figure 10 highlights the amount of user effort that was required to complete each stage of the optimization process. The figure displays improvements based on the original, un-optimized version of the hardware, based on percentage increase in performance per minute of development time to achieve that performance increase.

Finally Figure 11 shows the increase in logic usage between the original and optimized versions of the hardware. This figure highlights how, while performance has increased, in the case of the final optimized version so have the resource requirements to implement the design in hardware.

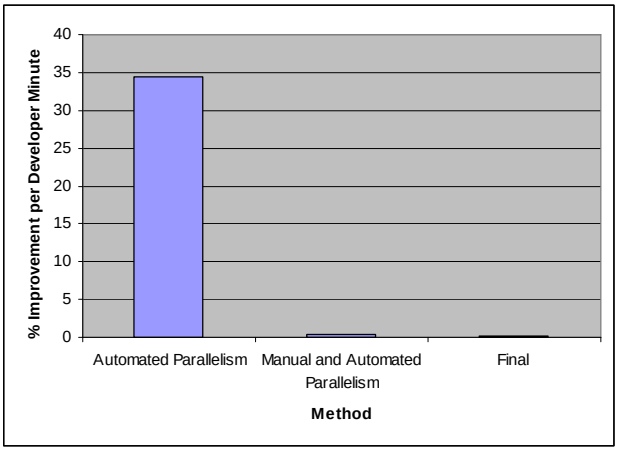

Figure 10: Developer Time Per Percentage Performance Increase

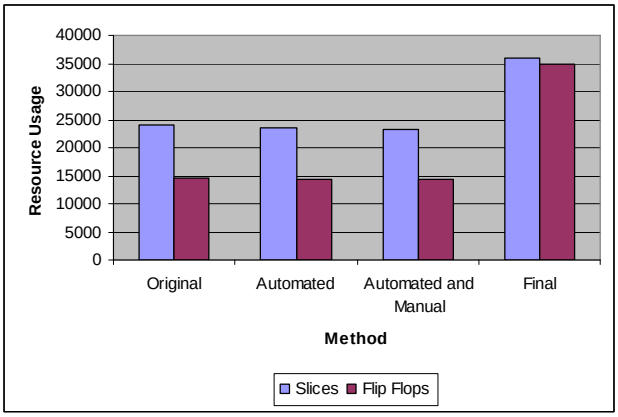

Figure 11: Resource Usage Comparison

7. Future Work

While the work can be considered a success, there is still much work to be done to further improve the performance of the system. At present only simple parallelism has been identified in the system. While parallelism between individual statements in a Handel-C program can greatly increase performance, there can be even greater performance gains from exploiting loop based parallelism or parallelism between different functional units.

Another optimization that may greatly benefit this work is the identification and implementation of a pipelined data path. A pipelined data path may increase the throughput of the algorithm by increasing the amount of work that is done per clock cycle by breaking the algorithm down into functional units that can operate in parallel much like an assembly line.

References

[1] K. Kent, B. Iaderoza, and M. Serra. Codesign of a Computationally Intensive Problem in GF(3), International Workshop on Rapid System Prototyping, pp. 10-16, May 2007.

[2] Agility Design Solutions, Handel-C Reference Manual, Website: www.agilityds.com. Accessed: May 2008

[3] G. Birkhoff, and S. Mac Lane. A Survey of Modern Algebra, 5th ed. New York: Macmillan, 1996.

[4] D. Page and N. P. Smart, "Hardware Implementation of Finite Fields of Characteristic Three". Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems, pp. 529-539, 2002.

[5] G. Lee, and F. Ruskey, Listing All Irreducible and Primitive Polynomials in GF(3),. Technical Report (University of Victoria, Canada), unpublished, 2006.

[6] AP1000 FPGA Development Board User Guide. User Guide Manual Version 2. AMIRIX Systems Inc, Halifax, Nova Scotia, Canada. 2005.

[7] M. Moazeni, A. Vahdatpour, K. Gururaj, and M. Sarrafzadeh, Communication Bottleneck in Hardware-Software Partitioning. In Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, pp. 262, 2008.

[8] R. Andraka, A Survey of CORDIC Algorithms for FPGA based Computers, 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays, pp. 191-200, 1998.

[9] M. Mylona, D. Holding, and K. Blow, DES Developed in Handel-C, London Communications Symposium, 2002.

[10] M. Serra, and K. B. Kent, Using FPGAs to Solve the Hamiltonian Cycle Problem, International Symposium on Circuits and Systems, pp. 228-231, vol. III, May 24-28, 2003.

[11] T. G. Noll, Application Specific eFPGAs for SoC Platforms, 2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test. April 2005.

[12] J. C. Libby, and K. B. Kent, Automatic Identification of Concurrency in Handel-C, International Symposium on Digital Systems Design, pp. 660-664, August 2008.

[13] J. C. Libby, and K. B. Kent, A Methodology for Rapid Optimization of HandelC Specifications, submitted to the 20th IEEE/IFIP International Symposium on Rapid System Prototyping, June 2009.

[14] J. C. Libby, J. P. Lutes, and K. B. Kent, A Handel-C Implementation of a Computationally Intensive Problem in GF(3), International Conference on Advances in Electronics and Micro-electronics, ENICS 2008, pp. 36-41, October 2008.

[15] APEX 20KC Programmable Logic Device. *Data Sheet Version 2.2*., Altera Corporation, 2002.

[16] APEX PCI Development Board Data Sheet. *Data Sheet Version 2.1*. Altera Corporation, 2002.

[17] J. Hannula, “Is High-Level Design Representation Worthwhile?”, *M.Sc thesis*, University of Victoria, Canada, 2004.

[18] Xilinx Incorporated, Website: http://www.xilinx.com/ , Accessed: 1/29/2009.