# Arbitrary Function Approximation in HDLs with Application to the N-body Problem

C.H. Ho[1], K.H. Tsoi[1], H.C. Yeung[1], Y.M. Lam[1], K.H. Lee[1], P.H.W. Leong[1], R. Ludewig[2], P. Zipf[2], A.G. Ortiz[2], M. Glesner[2]

[1]Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin NT HK.
{chho2,khtsoi,hcyeung,ymlam,khlee,phwl}@cse.cuhk.edu.hk
and
[2]Institute of Microelectronic Systems
Darmstadt University of Technology, Germany.
{ludewig,zipf,agarcia,glesner}@mes.tu-darmstadt.de

## Abstract

*A module generator is described that allows for the generation of synthesizable VHDL modules which implement arbitrary functions in fixed point precision using the Symmetric Table Addition Method (STAM). This module generator was interfaced to a high level synthesis tool "fly" which automatically generates fully-pipelined circuits from a Perl-like language. The resulting system was applied to the N-body problem and results are presented. It was found that a function generator module is a very useful addition to a hardware description language.*

## 1 Introduction

The field of reconfigurable computing has benefited greatly from the development of efficient hardware description languages as well as improved field programmable gate array (FPGA) device capacity in recent years. These two developments have greatly improved our ability to develop FPGA based hardware accelerators for computing applications. A large number of computing applications require the use of real arithmetic, and FPGA designers use both fixed and floating point representations in such systems.

In software based systems, libraries of commonly used mathematical functions are available, and programmers need not be familiar with the details of their implementation. In hardware design, although there are libraries available to perform specialized functions, they are normally restricted to a small fixed set of functions e.g. implementations of the CORDIC algorithm.

In this paper, an efficient table lookup generation system for supplementing a hardware description language (HDL) is proposed. In particular, an implementation of the Symmetric Table Addition Method (STAM) which acts as a module generator for an arbitrary twice differentiable function is described. This module generator was integrated with our high level synthesis tool "*fly*" [3] to produce a very flexible design environment which allows the specification of arbitrary functions in a high level manner. The resulting environment was used to develop a coprocessor for the computation of the N-body problem.

The N-body problem is computationally intensive and involves a large number of floating point operations. This together with the fact that relatively low precision is required makes it a good candidate for hardware acceleration using FPGAs. Using a module generator provides further advantages by allowing the designed system to be adapted for different application or different algorithms with very little effort. The design productivity of the module generator approach was much higher than that which could be expected from a VHDL designer.

The rest of the paper is organized as follows. In Section 2, an introduction to the N-body problem is given and the need for a function approximation justified. In Section 3, the STAM table lookup algorithm is described and extensions made to VHDL to allow for the simple interfacing of the STAM code detailed. An example which describes the application of the resulting STAM system to the implementation of the floating point function $f(x) = x^{\frac{-3}{2}}$ is also given. Section 4 describes the implementation of the N-body problem. Results obtained from this implementation are described in Section 5 and finally, conclusions are drawn in Section 6.

## 2 The N-body Problem

A wide range of physical systems can be studied by modeling them as an N-Body problem and are used in fields of science such as astrophysics and molecular biology. In the N-body problem, particles are modeled as points in space. The potential of the system can be expressed as a function of the properties and positions of all particles in the system and the force exerted on a particle is the first derivative of this potential with respect to the position of the particle. Different applications of the N-body problem share the same basic structure but differ in the physical law that governs the force between particles and the exact equation for calculating the potential and force depends on the application. By integrating the force acting on a particle, its position can be computed as a function of time.

There is no known analytic solution for the N-body problem for $N \geq 3$ so N-Body problems are solved numerically in practice via simulation. At each time step, forces exerted on each particle are computed and the positions of the particles are updated at the end of the time step by integrating all forces acting on all particles. In N-body simulations, most computation time is spent on force calculation and the number of interactions between particles grows as O($n^2$), where $n$ is the number of particles. For large $n$, this calculation becomes very expensive and poses a limit on the size of system that can be realistically studied.

Since the force calculation part consumes most of the execution time, and at the same time has a rather simple algorithm, it is a good candidate for hardware acceleration. In fact, this has been done in many systems, usually employing a heterogeneous architecture consisting of a general purpose host computer and a special purpose hardware. The special purpose hardware handles the force calculation while the host computer handles all other computations. Most notable of those is the GRAPE (Gravitational Pipeline) computer for the gravitational N-body problem [5].

In this work, an FPGA based co-processor for evaluating gravitational forces in N-body simulations was built using a module generator approach. The architecture of the co-processor is similar to the GRAPE-1 system. GRAPE-1 is the first in a series of specialized processors evaluating gravitational forces or acceleration in a gravitational N-body simulation. Equation 1, 2, 3 shows the force evaluation in this system.

$$\mathbf{a}_i = \sum_{j=1}^{N}(\mathbf{a}_{ij}) \tag{1}$$

$$\mathbf{a}_{ij} = (\mathbf{x}_j - \mathbf{x}_i)(r_{ij}^2 + \epsilon^2)^{-3/2} \tag{2}$$

$$r_{ij}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 \tag{3}$$

The equations are the same as that implemented in the GRAPE-1 system. $\mathbf{a}_i$ is the gravitational acceleration at the position of particle $i$, $\mathbf{x}_i$ is the position of vector particle $i$, $r_{ij}$ is the distance between particles $i$ and $j$ and $\epsilon$ is the artificial potential softening used to suppress the divergence of the force at $r_{ij} \to 0$. Implementation of the above equations is reasonably straightforward using multipliers and adders, with the exception of a function to compute $x^{-3/2}$. The implementation of arbitrary functions is a tedious task and hardware description languages such as VHDL do not offer support to aid their implementation. In the following sections, we describe the STAM table lookup algorithm which can approximate any twice differentiable functions to a desired accuracy and then describe its integration into VHDL.

## 3 Table Lookup Approximations

### 3.1 Taylor Expansion

If a function $f(x)$ has continuous derivatives up to $(n+1)^{th}$ order, it can be expanded using a Taylor series:

$$f(x) = \sum_{i=0}^{n} \frac{f^{(i)}(a)(x-a)^i}{i!} + R_n \tag{4}$$

where

$$\begin{aligned} R_n &= \int_a^x f^{(n+1)}(u)\frac{(x-u)^n}{n!}\mathrm{d}u \\ &= \frac{f^{(n+a)}(\xi)(x-a)^{n+1}}{(n+1)!} \quad \text{for} \quad a < \xi < x \end{aligned}$$

To reduce the required hardware resources and/or computation power, only the first few terms in the Taylor series can be used to approximate the function.

### 3.2 Symmetric Bipartite Table Method (SBTM)

The SBTM uses the first two terms of the Taylor series to approximate a function $f(x)$ as $\widetilde{f(x)}$ [6]. In the SBTM, two lookup tables are constructed and the precision of the output is maximized.

Assume that the $n$-bit input, $x$, of the function to be approximated has the range $[0, 1)$. It is first partitioned into 3 segments as shown in Fig 1 where $x = x_0 + x_1 + x_2$.

The ranges of $x_i$ are:

$$\begin{aligned} 0 &\leq x_0 &&\leq 1 - 2^{-n_0} \\ 0 &\leq x_1 &&\leq 2^{-n_0} - 2^{-n_0-n_1} \\ 0 &\leq x_2 &&\leq 2^{-n_0-n_1} - 2^{-n_0-n_1-n_2} \end{aligned}$$

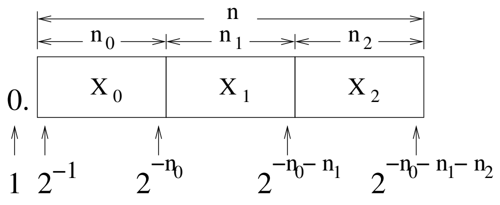

**Figure 1. Input partition of SBTM.**

Two lookup tables which return the values $a_0$ and $a_1$ are then constructed. The sum of these two values will be an approximation to the function. We first select mid points in the ranges of $x_1$ and $x_2$:

$$\begin{aligned}\delta_1 &= (2^{-n_0} - 2^{-n_0-n_1})/2 \\ \delta_2 &= (2^{-n_0-n_1} - 2^{-n_0-n_1-n_2})/2\end{aligned} \tag{5}$$

Let $a = x_0 + x_1 + \delta_2$ and use the first two terms of the Taylor Expansion:

$$\begin{aligned}f(x) &= f(x_0 + x_1 + x_2) \\ &\approx f(x_0 + x_1 + \delta_2) + f'(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2) \\ &= a_0(x_0, x_1) + a_1(x_0, x_2) \\ &= \widetilde{f(x)}\end{aligned} \tag{6}$$

Not all bits from $a_1$ are required to be in the table as the carefully selected $\delta_2$ results in a large number of leading 0s or 1s in the $a_1$ table. Since $\delta_2$ is located in the center of $x_2$'s range, then

$$|x_2 - \delta_2| \le \frac{x_{max}}{2} \quad \Rightarrow \quad |x_2 - \delta_2| < 2^{-n_0-n_1-1} \tag{7}$$

The upper bound of $a_1$ is

$$\begin{aligned}|a_1(x_0, x_2)| &= |f'(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2)| \\ &< |f'(\xi_1)|2^{-n_0-n_1-1}\end{aligned} \tag{8}$$

where

$$\xi_i \in [0, 1)|(\forall x \in [0, 1)|f^{(i)}(\xi_i) > f^{(i)}(x))$$

Note that even though the domain of $f(x)$ is $[0, 1)$, the range of $a_0 = f'(x)(x - a)$ may include negative numbers. From equation (8), one can calculate the number of leading 0s (or leading 1s for a negative number) to be

$$n_0 + n_1 + 1 + \log_2(|f(\xi_0)/f'(\xi_1)|). \tag{9}$$

To maximize the precision of the algorithm and optimize the resource utilization, errors in approximation must be controlled. In the SBTM algorithm [6], the following inequality ensures that the error will be less than the LSB of the result:

$$|f''(\xi_2)|2^{-2n_0-n_1-2}(1 + 2^{-n_1}) + 2^{-p_f-g-1} \le 2^{-p_f-1} \tag{10}$$

The first term is the error contributed by the Taylor approximation and the rest are from rounding intermediate results. This inequality limits the choice of partition size and required guard bits. If the configuration parameters of the algorithm are selected according to the following constraints, inequality (10) will be satisfied.

$$\begin{aligned}2n_0 + n_1 &\le p_f + log_2(f''(\xi_2)) \\ g &\le 2\end{aligned} \tag{11}$$

The size of table $a_1$ is $2^{n_0+n_2} \times (p_f + g)$ bits. This size can be reduced by half by exploiting symmetry in the table. First, we notice that $2\delta_2 - x_2$ equals the 1's complement of $x_2$. Then we notice that $a_1(x_0, 2\delta_2 - x_2)$ equals the bitwise inversion of $a_1(x_0, x_2)$. Replacing $x_2$ by $2\delta_2 - x_2$ in $a_1 = f'(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2)$ gives the result. The resulting implementation is shown in Fig 2.

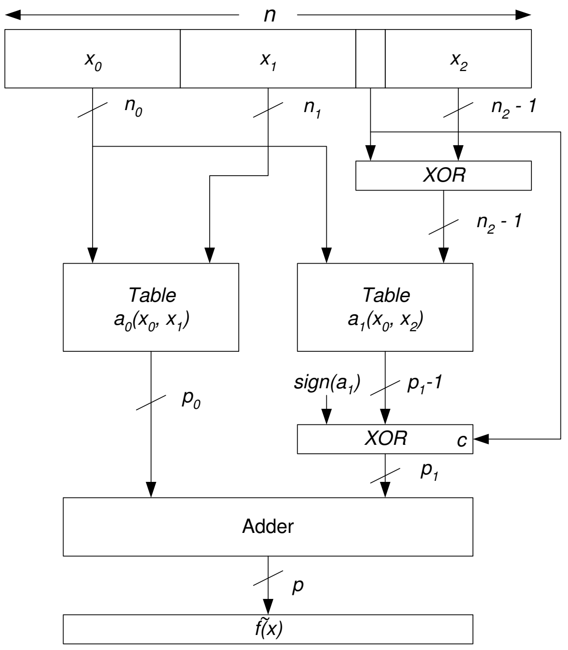

**Figure 2. Structure of optimized SBTM.**

The size of the table can be further reduced by eliminating the leading zeros or ones according to equation (9). Since all values in table $a_0$ have the same sign, the sign bit of all values can be moved outside the table.

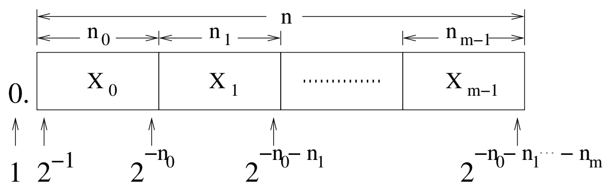

**Figure 3. Input partition of STAM.**

## 3.3 Symmetric Table Addition Method (STAM)

The logic in the SBTM is very simple and only two tables are required. The STAM algorithm uses more tables of smaller size to significantly reduce the overall memory required [7].

As shown in Fig 3, the $n$-bit input is partitioned into $m$ segments instead of 3. The input is now $x = \sum_{i=0}^{m-1} x_i$.

The ranges of $x_i$ become:

$$\begin{aligned} 0 &\le x_0 &&\le 1 - 2^{-n_0} \\ 0 &\le x_i &&\le 2^{-p_{i-1}} - 2^{-p_i} \end{aligned} \tag{12}$$

where $p_i = \sum_{k=0}^{i} n_k$ and the $\delta_i$ are chosen as following:

$$\delta_i = (2^{-p_{i-1}} - 2^{-p_i})/2 \tag{13}$$

To apply the Taylor approximation, let $a = x_0 + x_1 + \sum_2^m \delta_i$. The approximation function is now:

$$\begin{aligned} \widetilde{f(x)} &= f(x_0 + x_1 + \sum_2^m \delta_i) + \\ &\quad f'(x_0 + \delta_1 + \sum_2^m \delta_i)(\sum_2^m x_i - \sum_2^m \delta_i) \\ &= a_0(x_0, x_1) + \sum_2^m a_{i-1}(x_0, x_i) \end{aligned} \tag{14}$$

where

$$a_{i-1}(x_0, x_i) = f'(x_0 + \delta_1 + \sum_2^m \delta_k)(x_i - \delta_i) \quad 2 \le i \le m$$

The error analysis of the STAM is similar to the SBTM algorithm. The constraints for the parameter configuration are:

$$\begin{aligned} 2n_0 + n_1 &\le p_f + log_2(|f''(\xi_2)|) && (15) \\ g &\le 2 + log_2(m-1). && (16) \end{aligned}$$

## 3.4 Input Range Scaling

The above analysis was based on the input range $x \in [0, 1)$. Both SBTM and STAM can be adapted to other input ranges by first making a linear transformation of the actual input range to this range without changing the value of $f(x)$.

For an $n$-bit input $x_i$, let $\widehat{x_i}$ be the transformed value and assume the decimal point is right of the LSB. If the input range is $[x_{min}, x_{max})$, then

$$\begin{aligned} \frac{x_i - x_{min}}{x_{max} - x_{min}} &= \frac{\widehat{x_i}}{2^n} \\ \Rightarrow x_i &= \frac{\widehat{x_i}}{2^n}(x_{max} - x_{min}) + x_{min} \end{aligned} \tag{17}$$

## 3.5 VHDL Extension

To allow easy usage of the STAM algorithm in VHDL designs, some simple extensions were introduced by making use of the comment section inside VHDL code segments. The user includes the name and the body of the target function as well as some configuration parameters. The following example demonstrates the instantiation and usage of a *sin* function in the VHDL source.

```
architecture ...
...
-- __STAM_BEGIN__
-- my_function(x) = Sin(x)
-- range_min = 0
-- range_max = 1
-- segments = 8 2 2 2 2
-- decimal_point = 16
-- __STAM_END__
component my_function is port (
clk: in std_logic;
x: in std_logic_vector(15 downto 0);
fx: out std_logic_vector(20 downto 0));
end component;
...
begin
...
f0: my_function
  port map (clk=>clk, x=>x, fx=>fx);
...
```

Four tables are generated for the 16-bit input. The *decimal_point* statement indicates that decimal point is located at the left side of the most significant bit. The output will be ready after the next rising clock edge and will be valid as long as the input *x* is valid. The clock signal is required since synchronous RAMs are used to store the tables.

The VHDL codes are first passed to a preprocessor before going to the synthesis stage. A flow chart of the preprocessor is shown in Fig 4. First, the function extractor

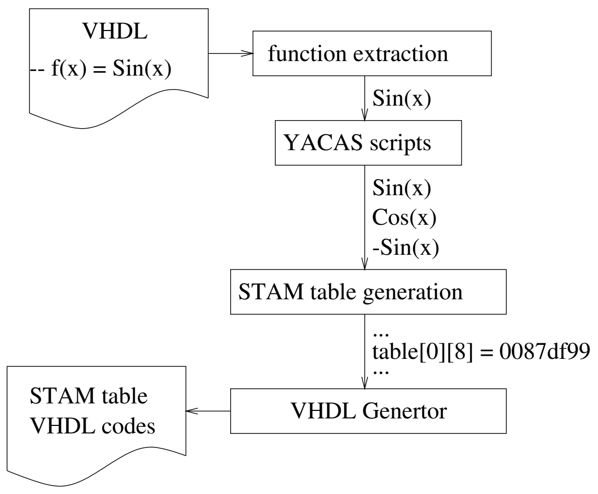

**Figure 4. Extended VHDL preprocessor.**

extracts the function body in the extended VHDL block and passes it to YACAS (YACAS is public domain software which performs symbolic arithmetic operations [2]). YACAS accepts the input function, finds the symbolic expression first and second derivatives and passes the results to the table generation program. The table generation program transforms the input strings to a sequence of arithmetic operations and generates the content of the lookup tables. These contents will be used in the VHDL generator to generate synthesizable VHDL code using Xilinx BlockRAM [8] as the lookup tables.

With this extension, an arbitrary twice differentiable function can be used in VHDL without knowledge of the detailed implementation. The default evaluation time is 1 clock cycle but this can be easily modified in the generated structural VHDL codes. This preprocessing method can be easily adapted for other HDL languages such as Verilog with minor modifications.

## 3.6 Floating Point Extension to *Fly*

The extended *fly* compiler supports basic floating-point operations, such as addition, subtraction and multiplication with different precisions. Transcendental functions such as square root and exponential are frequently required to evaluate the force or acceleration in the N-body problem. Such functions can be implemented using the modified STAM approach. This section will describe the development of a floating point coprocessor for the N-body problem which computes $v^{-3/2}$.

In the original STAM algorithm, the input value is considered to be a fixed point number within a predefined range. It is possible to modify the logic so that it can handle specific functions for floating-point arithmetic.

Range reduction and result correction are necessary in the floating point implementation. Consider the IEEE 754 binary floating point number representation:

$$v = 1.f \times 2^e \tag{18}$$

Then,

$$v^{-3/2} = (1.f \times 2^0)^{(-3/2)} \times (2^{(-3/2)e}) \tag{19}$$

When $e$ is even, let $e = 2N$, equation 19 becomes:

$$v^{-3/2} = (1.f \times 2^0)^{(-3/2)} \times (2^{-3N}) \tag{20}$$

Similarly, if $e$ is odd, let $e = 2N + 1$, equation 19 becomes:

$$v^{-3/2} = (1.f \times 2^0)^{(-3/2)} \times (2^{-3N}) \times (2^{-3/2}) \tag{21}$$

In both cases, the fractional part can be calculated using STAM with the input range $[1, 2)$, and the exponent part implemented using shift and add operations. If $e$ is odd, the final result should be multiplied by $2^{-3/2}$.

Normalization of the STAM output is required to convert it back into a floating point format. Since for $1 \le v < 2$ the output of STAM has a range $0.354 < v^{-3/2} < 1$, the location of the leading one must be one of the two most significant bits. The datapath of the calculation is shown in fig 5. Since it supports arbitrary size floating-point numbers, the precision can be modified to suit the application.

To implement the circuit on an FPGA, the *fly* [3] compiler was used to generate synthesizable VHDL code and a Pilchard board [4] was used as the reconfigurable platform. Pilchard uses a DIMM memory bus interface which provides high I/O performance compared to the PCI bus. *Fly* can take a Perl-like description of an algorithm and fully supports floating-point arithmetic of arbitrary precision. In addition, the mechanism of the *fly* compiler allows for easy integration of a block such as STAM. The *fly* compiler was modified such that it can handle the `_power15()` function using the built-in function mechanism.

Due to the limited amount of memory available on the FPGA chip, the STAM used 16-bit integers as input and the table size is (8,2,2,2,2). STAM was used to compute the function $f(x) = x^{-3/2}$ where $1 \le x < 2$. Note that since STAM requires an input $0 \le x < 1$, the transformation $\widehat{x} = x - 1$ was applied as described in equation 17. Thus the actual function approximated by the STAM module is $f(\widehat{x}) = (\widehat{x} + 1)^{-3/2}$ where $0 \le \widehat{x} < 1$. As the output of BlockRAM is 32-bit, the memory usage is $2^{(8+2)} \times 4 \times 32 =$ 131072-bits or 32 BlockRAMs.

## 4 Implementation

Figure 6 shows the datapath of the design which implements Equations 1, 2 and 3.

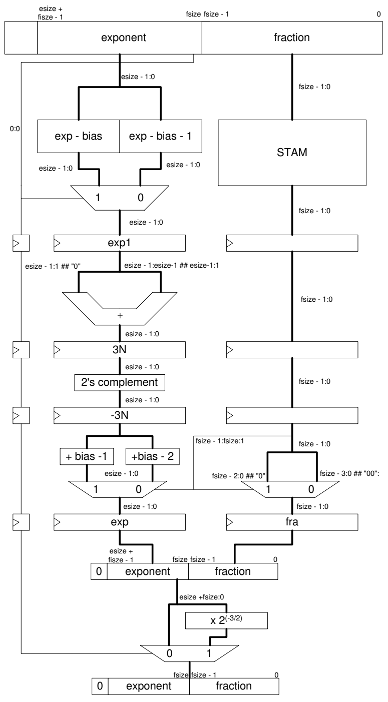
**Figure 5. Datapath of $v^{-3/2}$ using STAM for floating point arithmetic.**

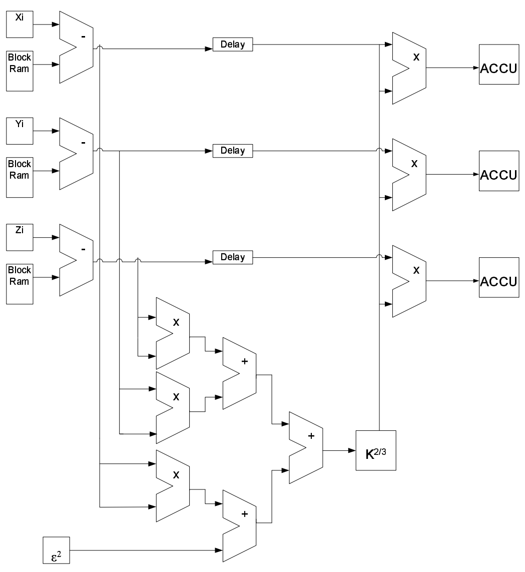
**Figure 6. Datapath of the N-body pipeline.**

A program written in the *fly* language was used to implement the equation 2, the inner loop of an N-body simulation. The input of the program is $\mathbf{x}_i$, $\mathbf{x}_j$ and $\epsilon$ while the output is the acceleration ($\mathbf{a}_{ij}$) for a particular value of $\mathbf{x}_j$. Most of the constructs are parallel in nature so that the vector manipulation can be processed simultaneously. For example, $\mathbf{x}_j - \mathbf{x}_i$ can be done at the same time for each scalar in $\mathbf{x}_j$ and $\mathbf{x}_i$. By appropriately inserting assignment statements in the program for pipelining, a fully-pipelined core can be generated using the *fly* compiler. The resulting VHDL code can produce a new value of $\mathbf{a}_{ij}$ every cycle. In addition, the *fly* code can used for simulation and verification by directly executing it under the Perl environment [3], saving time and reducing errors compared with manually translating the algorithm description into VHDL. The summation in equation 1 was done outside the *fly* core and a floating point accumulator was used to compute and store the value $\mathbf{a}_i$.

The floating point module supplied by the *fly* compiler is readily parameterized so the tradeoff between accuracy and resources is adjustable. Different sizes of fraction and magnitude can be implemented from the same description.

As the *fly* core used a register interface, in order to allow the design to run at maximum speed, a dedicated host interface, as shown in Fig 7, was developed so that it can provide a new $\mathbf{x}_j$ every clock cycle. There are two input buffers each for $\mathbf{x}_j$ and $\mathbf{x}_i$. Dual-port BlockSelect RAMs are used to construct these buffers in order to reduce area usage and isolate the clock domains of interface and core units. The host program first fills the $\mathbf{x}_j$ buffer. The design will loop through the $\mathbf{x}_j$ buffer for each $\mathbf{x}_i$ received from the host. By doing so, an N-body problem requires $N \times 2$ write operations instead of $N^2$ as long as the input buffers are large enough to store all N data. Also, the core design can start a new computation every clock cycle if the $\mathbf{x}_i$ buffer is not empty. Thus the pipelined design can be fully utilized to generate outputs at throughput of 1 result per clock cycle. If N is too large then not all $\mathbf{x}_j$'s can be stored in the input buffer and the host program is required to partition the problem and perform a final correction.

## 5 Results

The N-body simulation using the STAM for function approximation was implemented on a Xilinx XCV1000E-6 FPGA with a total of 10260 slices for the 32-bit configuration. The number of BlockSelect RAM used in the design was 32 for all configurations.

In order to test our design, the NEMO toolkit [1] was used for generating data and performing the N-body sim-

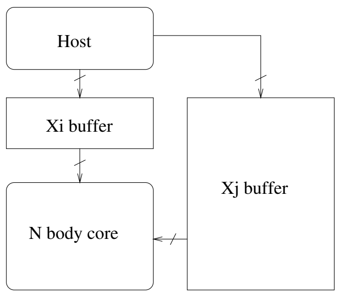

**Figure 7. Block diagram of the host interface. $\mathbf{X}_j$ is buffered to reduce required host bandwidth from $O(N^2)$ to $O(N)$.**

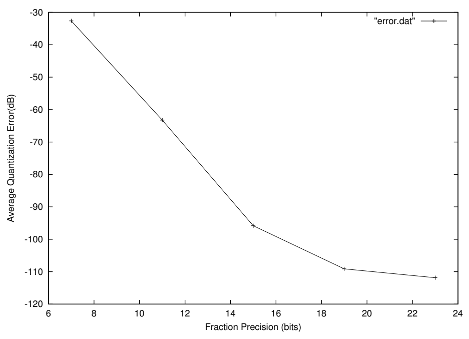

**Figure 8. Average quantization error versus precision.**

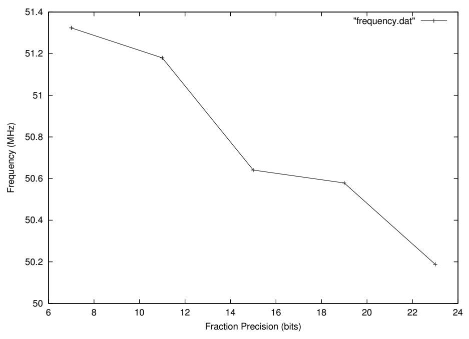

**Figure 9. Frequency versus precision for the N-body problem.**

ulation. The Nbody0 simulation program in the NEMO toolkit is modified to interface with our design. A test run using the modified NEMO code and a dataset of 256 particles was performed. The dataset was generated according to the Plummer model by a program in the NEMO toolkit. The Plummer model is widely used for modeling galaxies in Stellar Dynamics studies. To evaluate the quantization error incurred by the design, the force pipeline was used to calculate the force exerted on each of the 256 particles in the dataset at time zero. The average quantization error (QERR) is computed as:

$$QERR = \frac{1}{N} \sum_i 20 \log \left| \frac{out_i - ref_i}{ref_i} \right| \qquad (22)$$

where $out_i$ are the outputs from FPGA and $ref_i$ are the corresponding double precision reference outputs.

In figure 8, the average quantization error is plotted against the fraction length. As expected, for fraction lengths below 15-bits, the quantization error decreases linearly with the fraction length. The graph tails off at longer word length due to the limited precision of the STAM table used in this implementation. The precision required for an N-body simulation depends on the underlying problem being solved. Using our implementation, the user can choose the fraction length which best suits their requirements.

In figure 9, the clock frequency reported by the design tools is plotted against the precision. All configurations run successfully at 50 MHz. The performance of this design compares favorably to GRAPE-1 and GRAPE-3. A GRAPE-1 unit has a peak speed of 240 MFLOPs while a GRAPE-3 chip have a peak speed of 300 MFLOPs. Using the GRAPE convention of 30 floating-point operations per pipeline, this design running at 50 MHz has a peak speed of 1.5 GFlops. This equates to 5 times the performance of a GRAPE 3 chip [5]. The performance of this design also compares favorably with the software implementation in NEMO. The NEMO program running on a Pentium-III 800MHz machine requires 46 $\mu$s to compute each force value. Using this implementation it takes $10\mu$s to compute each force value.

In figure 10, the area in slices is plotted against the precision. It can be seen that the area increases linearly with fraction size. Since the design is parameterized, designs with different fraction size can be easily generated from the same description. If the application calls for a configuration with fraction width less than or equal to 15-bits, it is possible to achieve better performance by implementing 2 pipelines on the chip. With 2 pipelines running in parallel, the design can potentially achieve a peak speed of 3 GFLOPs.

Since the magnitude of force between particle pairs varies greatly, quantization error introduced by the accumulator in the long chain of summation operation can significantly degrade the accuracy of the result. It was found that using a longer accumulator fraction size greatly reduces quantization error without significantly increasing the area. Thus, 32-bit floating point accumulators were used for all configurations. Table 1 shows the average quantization er-

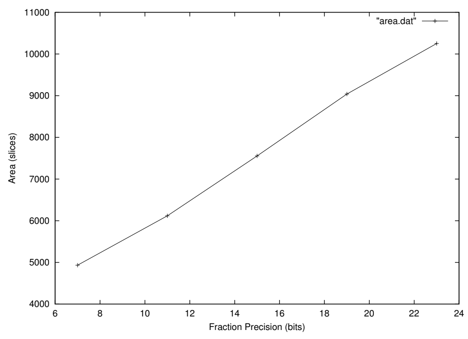

**Figure 10. Area versus precision for the N-body problem.**

**Table 1. Effect of different accumulator fraction size on area and quantization error.**

| FP Config (exp, frac) | ACC Config (exp, frac) | Area (Slices) | QERR (dB) |
|---|---|---|---|
| (8, 7) | (8, 7) | 4111 | -24 |
| (8, 7) | (8, 23) | 4934 | -32.6 |

ror and slices used by two implementations with different size accumulators.

## 6 Conclusions

A flexible framework for applying the STAM method has been presented. A preprocessor was developed which can be used to generate synthesizable VHDL modules which implement arbitrary user defined functions from comments inside VHDL source code. This function generator was integrated into the *fly* environment to extend its flexibility and efficiency. An N-body problem simulation was implemented using this framework to demonstrate the utility of such an approach. Without detailed knowledge of the STAM implementation, a fully pipelined N-body core was generated from 246 lines of *fly* source code. Furthermore, implementations with different floating point precision can be implemented from the same source code, facilitating the analysis of floating point precision, area utilization and performance. This framework can be used to solve real world problems with minimum design effort and with sacrificing performance.

## 7 Acknowledgements

The work described in the paper was substantially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No, CUHK4333/02E) and the Germany / Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and the German Academic Exchange Service (project No. G_HK006/02).

## References

[1] NEMO - A Stellar Dynamics Toolbox. In *http://bima.astro.umd.edu/nemo/*.

[2] Yet another computer algebra system (YACAS). In *http://yacas.sourceforge.net/*.

[3] C. Ho, P. Leong, K. Tsoi, R. Ludewig, P.Zipf, A. Ortiz, and M. Glesner. Fly - a modifiable hardware compiler. In *Proceedings of the twelfth International Workshop on Field-Programmable Logic & Applications*, 2002.

[4] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on FCCM*, 2001.

[5] J. Makino and M. Taiji. *Scientific Simulation with Special-Purpose Computers - the GRAPE systems*, pages 41–48. John Wiley & Sons Ltd, 1998.

[6] M. J. Schulte and J. Stine. Symmetric bipartite tables for accurate function approximation. In T. Lang, J.-M. Muller, and N. Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 175–183, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[7] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21:167–177, 1999.

[8] Xilinx. Using the Virtex Block SelectRAM+ Features. *Applications Note XAPP130*, 2000.