Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations

Gerhard Lienhart, Andreas Kugel and Reinhard Männer
Dept. for Computer Science V, University of Mannheim, B6-26A, D-68131 Mannheim, Germany
{lienhart,kugel,maenner}@ti.uni-mannheim.de

Abstract

This paper investigates the usage of floating-point arithmetic on FPGAs for N-Body simulation in natural science. The common aspect of these applications is the simple computing structure where forces between a particle and its surrounding particles are summed up. The role of reduced precision arithmetic is discussed, and our implementation of a floating-point arithmetic library with parameterized operators is presented. On the base of this library, implementation strategies of complex arithmetic units are discussed. Finally the realization of a fully pipelined pressure force calculation unit consisting of 60 floating-point operators with a resulting performance of 3.9 Gflops on an off the shelf FPGA is presented.

1. Introduction

In the scientific community there is a particular interest for special purpose computers, which can compute a specific problem at a very high performance. Since the simulation of biological, chemical or physical effects became an important aspect of natural science, there has always been an insatiable demand for computation power. Examples for arbitrary large computing problems can be found in the field of theoretical astrophysics where the motion of millions of stars need to be simulated in order to understand how the structures in the universe have developed.

For very specific problems special purpose computers based on ASICs can be built. There, arithmetic units are implemented which are optimally suited for the computation pattern of the problem. An example for such an approach is the GRAPE project where a single formula for getting the gravitational interaction force is calculated in a massively parallel manner (see publication of Ebisuzaki et al. [1]). These GRAPE Computers reach a performance up to the range of Tflops, and the Gordon-Bell Price has been awarded several times.

The gain of computation speed compared to general-purpose machines is directly related to the amount of specialization on an application. But in science the exact mathematical formulation of a simulation problem is often subject to change. This aspect foils the approach of building an ASIC for the majority of applications.

FPGA based machines are somewhere between general-purpose computers and ASIC-based computers. They enable building of application specific computation units but preserve the possibility of redesigning these structures at any time. Due to the limits of programmable logic resources on an FPGA they are to date mainly used for integer and bit-level processing tasks. But with the recent progress in FPGA technology these chips are more and more interesting for scientific applications where floating-point arithmetic is needed.

Although the clock frequency of typical FPGA designs (50-200 MHz) is much below the frequencies of state of the art CPUs (1 GHz) a computational speedup is possible for some algorithms, mainly for two reasons. First, optimally suited calculation units with a high degree of parallelization can be designed, where a minimum amount of silicon is wasted for modules in wait state. This is possible if the underlying algorithm has a simple computation pattern which allows to design a special purpose calculation structure. Furthermore the required precision of operators must be small enough to enable an implementation. Secondly, a possible speedup is obtained if the environment surrounding the FPGA and the controllers on the FPGA can be designed for providing a very high bandwidth of data transfer. Both aspects together allow for a sustained calculation with maximized utilization of the operators.

There have been several investigations of using FPGAs for implementing floating-point arithmetic. Fagin and Renard [17], Shirazi et al. [15], Lourca et al. [12] and Ligon et al. [16] investigated ways of implementing floating-point adders and multipliers. Shirazi et al. [15] and Louie et al. [13] published division implementations. Li and Chu [11] presented their implementation of a square root unit on FPGA. All these investigations have in common that they were done for a particular floating-point precision. In contrast, we focus on the impact of different floating-point representations on the resulting hardware design. Additionally we use different integer

Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)
1082-3409/02 $17.00 © 2002 **IEEE**

operator implementations in order to take advantage from extended capabilities of modern FPGA technology. Jaenicke and Luk [18] presented parameterized floating-point adders and multipliers. In this paper we not only analyze the scaling of the resource utilization more detailed but also discuss the square root and division operators.

Cook et al. [3] and Hamada et al. [4] have investigated the use of FPGA processors for N-Body algorithms with basic gravitational force interaction.

In this paper we first give an impression of the application we implemented with FPGA technology. The application is part of a project in which current astrophysical simulation platforms consisting of GRAPE boards and a host workstation shall become accelerated by the reconfigurable computing approach - the goal is to overcome the current bottleneck of processing the gasdynamical equations of the N-Body algorithm (AHA-GRAPE project, see [8]). We present the SPH method (see below) for treating gasdynamical effects in N-Body simulations, and show the formulas we have to calculate. Then we introduce the floating-point arithmetic with reduced accuracy and show the effects of the precision on the design for the floating-point operators. We present our implementation of a floating-point arithmetic library, which is parameterized in precision, and show the resulting numbers of resource utilization and design frequency as a function of the accuracy of the different operators. After a more general view on implementation strategies for complex arithmetic units we describe our design implementation of the SPH formulas.

This application exhibits structural similarities with many other particle based scientific simulation problems, e.g. in the wide field of molecular dynamics, and our realization gives an image of the capabilities of current FPGA technology in the area of floating-point arithmetic.

2. Target application

The application we implemented on FPGA is part of astrophysical N-Body simulations where gasdynamical effects are treated by the so-called smoothed particle hydrodynamics method (SPH). A review of this method has been published by W. Benz [6]. The principle of this method is, that the gaseous matter is represented by particles, which have a position, velocity and mass. To form a continuous distribution of gas from these particles they become smoothed by folding their discrete mass distribution with a smoothing kernel W. This means that the point masses of the particles become smeared so that they form a density distribution. At a given position the density is calculated by summing the smoothed densities of the surrounding particles. Mathematically this can be written as follows:

$$\rho_i = \sum_{j=1}^{N} m_j W(\vec{r}_i - \vec{r}_j, h) \tag{1}$$

Commonly used smoothing kernels are strongly peaked functions around zero and are non-zero only in a limited area. The parameter *h* for the smoothing kernel W is the so-called smoothing length which specifies the radius of this area. In our application we used the following spherical kernel function:

$$W(\vec{r}_i - \vec{r}_j, h) = \frac{1}{h^3} B\left(x = \frac{\left|\vec{r}_i - \vec{r}_j\right|}{h} \right)$$
$$with\ B(x) = \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & for\ 0 < x \le 1 \\ \frac{1}{4}(2-x)^3 & for\ 1 < x < 2 \end{cases} \tag{2}$$

The important point of SPH is that any gasdynamical variables and even their derivatives can be calculated by a simple summation over the particle data multiplied by the smoothing kernel or its derivative.

The motion of the SPH particles is determined by the gasdynamical force calculated via the smoothing method. According to this force the particles then move like Newtonian point masses. Following is the physical formulation of the velocity derivative given by the pressure force and the so-called artificial viscosity. This artificial viscosity is needed to be able to simulate the behaviour of shock waves.

$$\frac{d\vec{v}_i}{dt} = -\frac{1}{\rho_i}\nabla P_i + \vec{a}_i^{\,visc} \tag{3}$$

The SPH method transforms this equation to the following formulation, which is only one of several possibilities.

$$\frac{d\vec{v}_i}{dt} = -\sum_j m_j \left(\frac{p_i}{\rho_i^{\,2}} + \frac{p_j}{\rho_j^{\,2}} + \Pi_{ij} \right) \nabla_i W\left(\left|\vec{r}_{ij}\right|, h_{ij} \right)$$
$$\Pi_{ij} = \begin{cases} \frac{\left(-\alpha\, c_{ij}\mu_{ij} + \beta\, \mu_{ij}^{\,2} \right)}{\rho_{ij}} & for\ \vec{v}_{ij}\vec{r}_{ij} \le 0 \\ 0 & for\ \vec{v}_{ij}\vec{r}_{ij} > 0 \end{cases}$$
$$\rho_{ij} = \frac{\rho_i + \rho_j}{2}, \quad f_{ij} = \frac{f_i + f_j}{2}, \quad \vec{r}_{ij} = \vec{r}_i - \vec{r}_j$$
$$c_{ij} = \frac{c_i + c_j}{2}, \quad h_{ij} = \frac{h_i + h_j}{2}, \quad \vec{v}_{ij} = \vec{v}_i - \vec{v}_j$$
$$\mu_{ij} = \frac{h_{ij}\, \vec{v}_{ij}\vec{r}_{ij}}{\vec{r}_{ij}^{\,2} + \eta^2 h_{ij}^{\,2}} f_{ij} \tag{4}$$

Due to optimization of the simulation behaviour under specific conditions of the considered astrophysical system there are different approaches of symmetrisizing the formulation. For example the mean value of the density ρ,

the sound speed c, the smoothing length h and the variables f, which may be different for each particle, are taken here for further calculation.

This formula for the velocity derivative is the same as it was published by W. Benz [6], and it is used at the Max Planck Institute for Astrophysics in Heidelberg [7].

All variables with index i (j) are related to particle number i (j). The sum over j is in fact a sum over all neighbouring particles in the area of radius ($2 \cdot h_i$) around particle i. In typical applications the depth of this sum is of O(50). The variable Π is due to the artificial viscosity in the SPH simulation.

The abstract computing structure is like in many other simulation applications too. For each variable i, a summation over j terms depending on variables with index i and j has to be performed.

$$y_i = \sum_{j \in N(i)} F\left(x_i^{\,1} .. x_i^{\,n}, x_j^{\,1} .. x_j^{\,m}\right) \qquad (5)$$

This structure of computation leads to the hardware scheme shown in Figure 1, where the data x^k is stored locally in RAM.

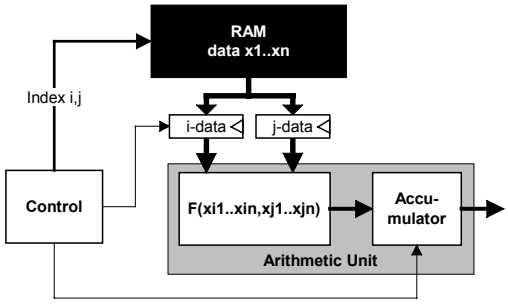

Figure 1. Hardware scheme for abstract computation pattern.

For the arithmetic unit floating-point calculation is needed, because the astrophysical variables span an enormous range . Using logarithmic arithmetic could have been considered too, but was not implemented because of the extensive use of ROMs for logarithmic adders [14].

The number of floating-point operations required for a full parallel implementation is very high. In our application we have 60 floating-point operations to perform during one summation step. These operations include highly expensive arithmetic functions like division and square root.

To make a straightforward pipelined implementation on an off-the-shelf FPGA it is necessary to work with reduced precision. For our application, test simulations showed that it is sufficient to utilize a floating-point format with only 16 bit for the significand compared to 24 bit for the IEEE-754 standard for single precision numbers. This is due to discretization errors of the underlying SPH model which are higher than the floating-point calculation errors. We will discuss our floating-point representation more detailed in the next section.

The target platform is an FPGA processor PCI-card, which has been developed in our institute (see Figure 2, [9]). The card is based on the 66 MHz, 64 bit PCI bus specification. It contains a modern Virtex2 FPGA, four 36 bit wide 133 MHz SRAM banks and a connector to a standard 133 MHz SDRAM bank. Moreover it has different connectors to daughter cards and board-to-board interfaces for multiple-board designs.

Figure 2. Target platform - PCI plug in card with Virtex2 FPGA.

3. Floating-point arithmetic with reduced precision

A general floating-point number can be expressed as

$$f = \pm s \cdot 2^e \qquad (6)$$

with a significand s in the half open interval [1,2[and an integer exponent e. To represent s by an unsigned integer value, only the fractional part s-1 needs to be saved (Fract). The exponent e can be expressed as e = Exp-bias.

Figure 3 shows the definition of single precision floating-point numbers according to the IEEE 754 Standard. This standard also defines double precision numbers in a similar way.

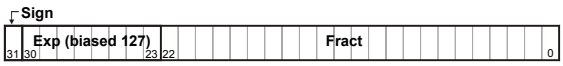

Figure 3. IEEE 754 floating-point format.

The real number f which is represented by the 32 bit word is calculated as follows:

$$f = \begin{Bmatrix} +1 \; for \; \text{Sign} = 0 \\ -1 \; for \; \text{Sign} = 1 \end{Bmatrix} \cdot \left(1 + \text{Fract} \cdot 2^{-22}\right) \cdot 2^{\text{Exp}-127} \qquad (7)$$

Fract and Exp are treated as unsigned integers.

We used a generalized floating-point representation with ExpBits bits for the exponent and SignifBits bits for the significand as shown in Figure 4.

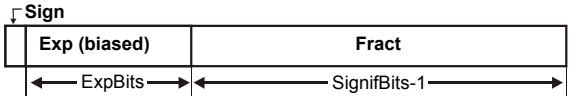

Figure 4. Generalized floating-point format.

With the generalized floating-point number format a real number f is given as:

$$f = \left\{ \begin{matrix} +1 \; for \; \text{Sign} = 0 \\ -1 \; for \; \text{Sign} = 1 \end{matrix} \right\} \cdot \left(1 + \text{Fract} \cdot 2^{-(\text{SignifBits}-1)}\right) \cdot 2^{\text{Exp}-\text{bi}} \quad (8)$$

$$with \; \text{bias} = 2^{(ExpBits-1)} - 1$$

In the following we will discuss the impact of the floating-point representation on the realization of the operator implementation. Our discussion covers the basic binary operations addition, multiplication, division and the unary operation square root.

We can generally divide the task of a floating-point operator into three steps. The first step is the operand preparation where the input data is transformed so that the operation can be done in integer arithmetic. This is the so-called denormalization. The second step is concerned with performing the calculation and is called operation step. The third step transforms the calculation results to a normalized floating-point output. There rounding and overflow detection is also done. This is called normalization. Figure 5 shows the flow-graph of an abstract floating-point operator divided into these steps.

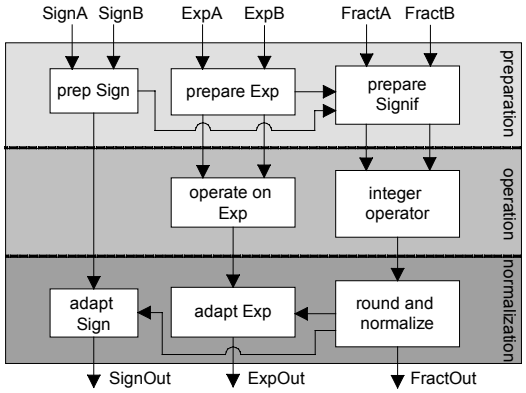

Figure 5. Abstract structure of a floating-point operator.

All operations on sign and exponent are simple logical operations or integer calculations with low bit width and consume little resources. The resource utilization of these operations scales linearly with the number of exponent bits. We will not describe these operations in detail but concentrate on the modules 'prepare signif', 'integer operator' and 'round and normalize'.

3.1. Floating-point adder

The floating-point addition of two values f_1 and f_2 where $f_1 > f_2$ can be formulated as follows:

$$\begin{aligned} \overbrace{\pm s \cdot 2^{Exp-bias}}^{f} &= \overbrace{\left(\pm s_1 \cdot 2^{Exp_1-bias}\right)}^{f_1} + \overbrace{\left(\pm s_2 \cdot 2^{Exp_2-bias}\right)}^{f_2} \\ &= \pm \underbrace{\left(\pm s_1 + s_2 \cdot 2^{Exp_2-Exp_1}\right)}_{\text{integer operator}} \cdot 2^{Exp_1-bias} \end{aligned} \quad (9)$$

For the floating-point adder the integer operator for the prepared significand is a simple adder, which can be implemented very cheaply on FPGA. Several times more expensive is the implementation of the modules for significand preparation, rounding and normalization. Before the significands can become added they have to be aligned in order to base on the same exponent. Therefore the significand of the operand with the smallest exponent has to be right-shifted by $|Exp_2-Exp_1|$ positions. Before shifting a swap unit has to be employed to distribute the number with the smaller exponent to the aligning unit. The shift module we need, must be able to shift a significand by any number of bits between 0 and SignifBits to the right. Figure 6 shows the number of LUTs used for the shifter operator as a function of the bit width of the argument. The shifter was implemented in behavioral VHDL code and mapped by Synplicity Synplify 7.0 on a XC2V3000 FPGA. The function is not smooth because the packing efficiency of the logic on the slices varies for different implementations.

Note that for the Virtex2 FPGAs the block multipliers can be used as barrel shifters, so many LUTs for the shifters can be saved. We did not use this option in our implementation in order to save the block multipliers for the multiplication operators.

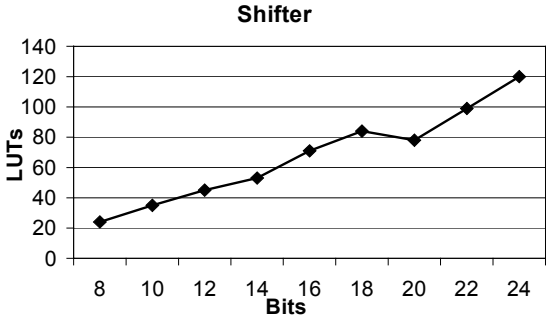

Figure 6. Resource utilization of shift operator on XC2V3000 FPGA.

We now distinguish between adders for signed and unsigned floating-point numbers. For the unsigned adders the normalization step has to shift the adder result only by 0 or 1 bits to the right, because the resulting sum is in the interval [1,4[. After rounding which is basically an

addition of 1, also a shift by one position may be needed. These three operations scale linearly with SignifBits. We can estimate that the corresponding hardware design together with the swap unit described above, utilizes 4-5 times the resources of the significand adder.

For the signed adders one of the input significands has to be 2's-complemented before the operation when the signs of the input are different. During addition of signed integers bit cancellation may occur. So the normalization step needs an additional shifter for arbitrary numbers of positions. Moreover a second 2's-complement unit must be implemented to transform the possibly negative normalized addition result into an unsigned significand. Rounding and second normalization are done like in the unsigned adder architecture. The preparation and normalization step in the signed adder takes about 2 times more resources than the corresponding units of the unsigned adder.

3.2. Floating-point multiplier

The floating-point multiplication of two values f_1 and f_2 can be formulated as follows:

$$\begin{aligned} \overbrace{\pm s \cdot 2^{Exp-bias}}^{f} &= \overbrace{\left(\pm s_1 \cdot 2^{Exp_1-bias}\right)}^{f_1} \cdot \overbrace{\left(\pm s_2 \cdot 2^{Exp_2-bias}\right)}^{f_2} \\ &= \pm \underbrace{\left(s_1 \cdot s_2\right)}_{\text{integer operator}} \cdot 2^{(Exp_1+Exp_2-bias)-bias} \end{aligned} \tag{10}$$

In the floating-point multiplier the integer operator step is dominant for the resource utilization. The step of preparing the significand before multiplication can be neglected, because it consists only of prefixing the hidden 1 to FractA and FractB. The normalization is similar to the design of the unsigned adder described above.

The most resource efficient but also slowest way to implement an integer multiplier in modern FPGA logic is to build an unrolled bit-serial shift-and-add multiplier structure by the use of mult-and-add primitives. These primitives can perform both a one-bit multiplication and a one bit addition in one LUT. Figure 7 shows such a multiplier for four bit integers. The scaling of the resource utilization is quadratic.

The multipliers we implemented have small shift-and-add multipliers and a partial product adder tree. Figure 8 shows the resulting resource utilization depending on the bit width of the arguments.

Note that the Virtex2 FPGAs contain dedicated block multiplier resources. The use of these elements dramatically reduces the utilization of slice-logic as we show in the succeeding section.

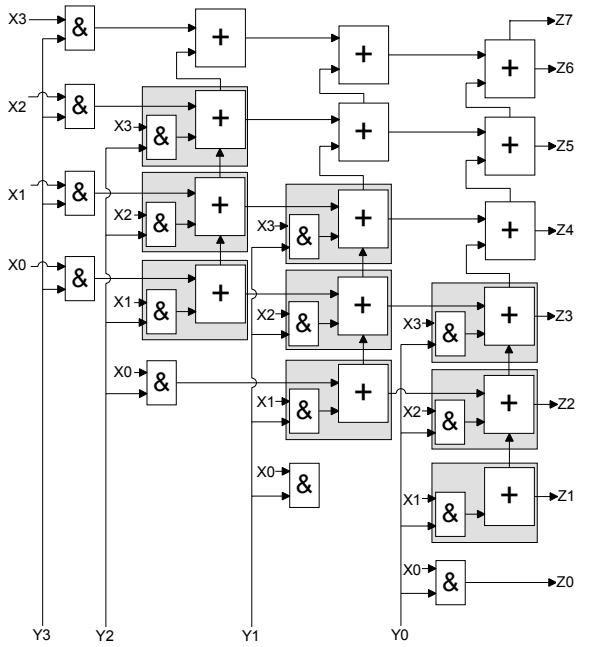

Figure 7. Multiplication of 4 bit integers with one bit mult-and-add primitives (gray blocks).

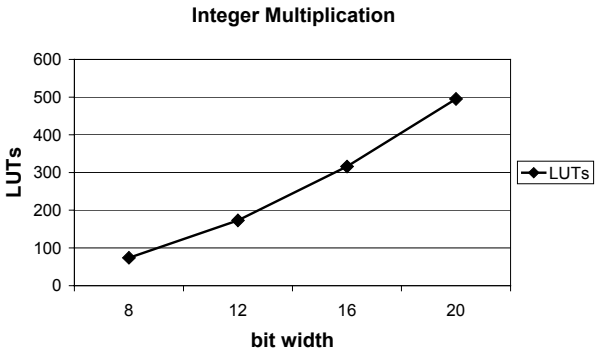

Figure 8. Resource utilization of integer multipliers on XC2V3000 FPGA.

3.3. Floating-point divider

The mathematical formulation of the floating-point divider is quite similar to the multiplication:

$$\begin{aligned} \overbrace{\pm s \cdot 2^{Exp-bias}}^{f} &= \overbrace{\left(\pm s_1 \cdot 2^{Exp_1-bias}\right)}^{f_1} / \overbrace{\left(\pm s_2 \cdot 2^{Exp_2-bias}\right)}^{f_2} \\ &= \pm \underbrace{\left(s_1 / s_2\right)}_{\text{integer operator}} \cdot 2^{(Exp_1-Exp_2+bias)-bias} \end{aligned} \tag{11}$$

Here the integer operator is the dominant part concerning the resource utilization, too. The normalization step only has to shift the integer division result by 0 or 1 bits to the left, because the quotient is in

the interval [1/2,2[. A very resource efficient way of implementing integer dividers in FPGA is the non-restoring division method. This iterative algorithm can be formulated as follows for the division s_1/s_2:

$$
\begin{aligned}
&rem_0 = s_1 \\
&rem_1 = rem_0 - s_2 \\
&rem_n = \begin{cases} 2 \cdot rem_{n-1} - s_2 \;\; for\; rem_n \geq 0 \\ 2 \cdot rem_{n-1} + s_2 \;\; for\; rem_n < 0 \end{cases} \\
&s^{\mathrm{SignifBits}-n} = \begin{cases} 1 \; for\; rem_n \geq 0 \\ 0 \; for\; rem_n < 0 \end{cases} \\
&s = s^{\mathrm{SignifBits}-1} .. s^0
\end{aligned}
\tag{12}
$$

We can implement these iteration steps by using the add-sub-primitives of modern FPGAs, which utilize 1 LUT per bit and stage. A parallel divider consists of an array of the unrolled iteration stages. We see that the required amount of logic resources scales quadratic with the size of the significand. Figure 9 shows the scheme of the division unit for 4-bit integer numbers.

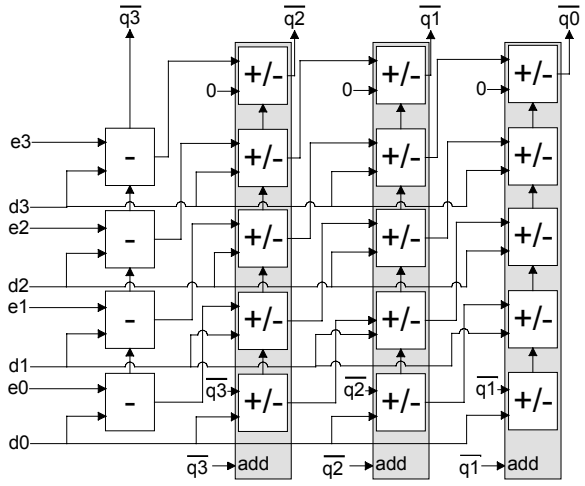

Figure 9. Division of 4 bit integers with add-sub primitives, q = e/d.

Pipeline stages can be inserted between any add-sub stage to speed up the design, but notice that the denominator also has to be pipelined. The amount of resource utilization and scaling with SignifBits is similar to that of the multiplier.

3.4. Floating-point square root

The following formula exhibits the calculation principle of the floating-point square root f of the number f_1:

$$
\begin{aligned}
\underbrace{s \cdot 2^{Exp-bias}}_{\mathrm{f}} &= \underbrace{(s_1 \cdot 2^{(Exp_1 - bias)})^{1/2}}_{\mathrm{f1}} \\
&= \underbrace{\sqrt{s_1'}}_{\text{integer operator}} \cdot 2^{(Exp_1' - bias)/2} \\
s_1' &= \begin{cases} s_1 \quad for\; even\; Exp_1 - bias \\ s_1 \cdot 2 \; for\; odd\; Exp_1 - bias \end{cases} \\
Exp_1' &= \begin{cases} Exp_1 \quad for\; even\; Exp_1 - bias \\ Exp_1 - 1 \; for\; odd\; Exp_1 - bias \end{cases}
\end{aligned}
\tag{13}
$$

We see that the preparation step includes a left shift of 0 or 1 bit depending on the exponent. No normalization step needs to be employed because the result of the square root of the shifted significand again lies in the interval [1,2[. As in the multiplier and division operator the integer operator is the most expensive part of the square root. For that operator the binary nonrestoring algorithm was implemented. Here is the iteration scheme for q=sqrt(z) as it has been described in a similar manner in [11] (here: $z = s_1'$, m = SignifBits+1):

$$
\begin{aligned}
&q_0 = 1, r_0 = (z^{(m-1)} z^{(m-2)} - 01) \\
&r_1 = r_0 z^{(m-3)} z^{(m-4)} - 0101 \\
&q_1 = \begin{cases} 11 \; for\; r_1 \geq 0 \\ 10 \; for\; r_1 < 0 \end{cases} \\
&r_{n+1} = \begin{cases} r_n z^{(m-2n-3)} z^{(m-2n-4)} - q_n 01 \; for\; r_n \geq 0 \\ r_n z^{(m-2n-3)} z^{(m-2n-4)} + q_n 11 for\; r_n < 0 \end{cases} \\
&note: z^{(n)} = 0 \; for\; n < 0 \\
&q_{n+1} = \begin{cases} q_n 1 \;\; for\; r_{n+1} \geq 0 \\ q_n 0 \;\; for\; r_{n+1} < 0 \end{cases}
\end{aligned}
\tag{14}
$$

For an efficient implementation on FPGA we can use the add-sub-primitives like for the integer divider. We can build a parallel square root by unrolling the iterations in a similar way as it has been shown for the divider design. Pipelining is also as simple as in the divider architecture.

Figure 10 shows the resource utilization of the integer square root operators depending on the bit width. There is a clear quadratic scaling but the total amount of resources lies about 50 % below the numbers given for the multiplier.

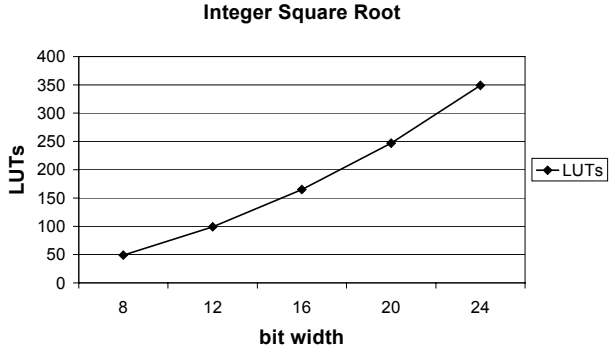

Figure 10. Resource utilization of integer square root on XC2V3000 FPGA.

4. Library of parameterized floating-point operators

In this section we present the performance results of the most important operators of our floating-point library implementation. All of the modules have parameters for the bit width of the floating-point numbers on input and output. Both floating-point adders for unsigned and signed arguments were built. Multiplier modules were designed for using FPGA logic based integer multipliers or Virtex2 block multipliers. We also show the performance results of our floating-point divider and floating-point square root designs.

For the resource and design frequency measurements, a fixed exponent width of 7 bits has been used, and the parameter SignifBits and the pipelining depth were varied. All modules were synthesized with Synplicity Synplify 7.0 and Xilinx Alliance 3.3.08i. The timing constraints of the place-and-route tool were set to 50 MHz. Therefore all frequency numbers much higher than 50 MHz are only lower bound estimates for the frequency which could be achieved with stronger constraints. This also explains that some design frequency numbers increase with an increasing value of SignifBits.

Note that one slice of the Xilinx Virtex2 FPGA basically contains two LUTs and two registers.

Table 1 shows the implementation results for the unsigned and signed adder operators. We see that signed addition utilizes almost twice as much logic resources as the unsigned version.

Table 2 summarizes the place-and-route results for our floating-point multipliers. The use of dedicated block multipliers of the Virtex2 FPGA dramatically reduces the amount of slices. For 16 bit significands only 1 block multiplier is used, for the other values of the significand bit width 4 block multipliers are utilized.

Table 3 and Table 4 show the resource utilization of the floating-point divider and square root for different depths of pipelining. Since each stage of these operators produces one result bit, the pipeline depth is given in relation to the number of result bits.

Figure 11 and Figure 12 show the graphs for the register, LUT and slice utilization for the square root and divider operators in order to demonstrate the effects of the pipeline depth more detailed.

Table 1. Resource utilization and speed of floating-point adder for XC2V3000 FPGA.

Unsigned addition			Signed addition		
Signif bits	Slices	Design freq. (MHz)	Signif bits	Slices	Design freq. (MHz)
16	111 (0.8%)	112	16	195 (1.4%)	76
18	119 (0.8%)	109	18	222 (1.5%)	93
20	126 (0.9%)	101	20	232 (1.6%)	68
22	137 (1.0%)	100	22	260 (1.8%)	74
24	152 (1.1%)	106	24	290 (2.0%)	85

Table 2. Resource utilization and speed of floating-point multiplier for XC2V3000 FPGA.

No block multipliers			With block multipliers		
Signif bits	Slices	Design freq. (MHz)	Signif bits	Slices	Design freq. (MHz)
16	193 (1.3%)	97	16	30 (0.2%)	76
18	246 (1.7%)	69	18	61 (0.4%)	61
20	287 (2.0%)	73	20	66 (0.5%)	66
			22	72 (0.5%)	66
			24	78 (0.5%)	63

Table 3. Resource utilization and speed of floating-point divider for XC2V3000 FPGA.

4 result bits per pipe stage			2 result bits per pipe stage		
Signif bits	Slices	Design freq. (MHz)	Signif bits	Slices	Design freq. (MHz)
16	224 (1.5%)	50	16	262 (1.8%)	73
18	269 (1.9%)	50	18	320 (2.2%)	70
20	328 (2.3%)	50	20	384 (2.7%)	61
22	382 (2.7%)	44	22	454 (3.2%)	70
24	452 (3.1%)	44	24	530 (3.7%)	61

Table 4. Resource utilization and speed of floating-point square root for XC2V3000 FPGA.

4 result bits per pipe stage			2 result bits per pipe stage		
Signif bits	Slices	Design freq. (MHz)	Signif bits	Slices	Design freq. (MHz)
16	129 (0.9%)	54	16	144 (1.0%)	93
18	152 (1.1%)	51	18	175 (1.2%)	86
20	188 (1.3%)	53	20	209 (1.5%)	86
22	216 (1.5%)	50	22	246 (1.7%)	76
24	255 (1.8%)	50	24	286 (2.0%)	78

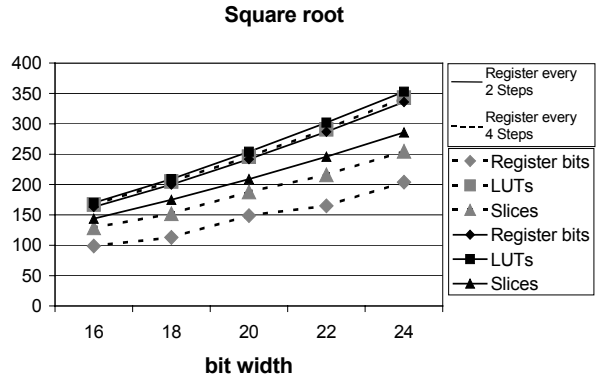

Figure 11. Scaling of resource utilization of floating-point square root operator.

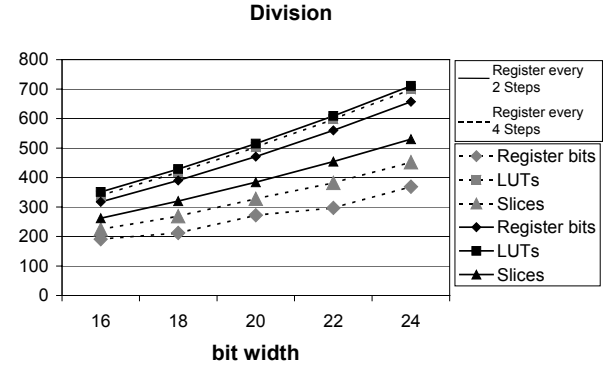

Figure 12. Scaling of resource utilization of floating-point division operator.

5. Implementation strategies for complex arithmetic units

On the base of the resource utilization numbers of the previous chapter we can estimate how much floating-point operations can be implemented in an FPGA design. For example, a 60 % utilization of a Xilinx XC2V3000 by pure arithmetic units with 16 bit significands allows for a maximum of 77 unsigned adders or 44 multipliers (without use of block multipliers) or 38 dividers.

If all the operators of the target application fit into the FPGA then a direct pipelined approach is the fastest implementation. For formulas with O(50) floating-point operations this design style is appropriate.

If the target formula does not fit into the FPGA with this method then some floating-point modules have to be shared for different calculations. If we share operators for two calculations at a time we can save approximately 50% of the logic with the effect of dividing the execution speed by a factor of 2. One approach to implement that is to build operator dispatchers, which switch the input and output values of an operator to different paths.

The most extreme method of sharing floating-point modules is to use an approach where the operators are connected by one or more buses. In that case a complex scheme of bus arbitration/reconfiguration has to be implemented.

6. Implementation of the arithmetic unit for SPH force calculation

We implemented the formulas for the force calculation of the SPH method, which we introduced above in formula (4), as one pipelined arithmetic unit. This was possible as we could use a floating-point representation with 16-bit significands. The pipeline is able to perform a complete force calculation of particle j on particle i at every clock cycle.

Figure 13 shows the flow-graph of the pipeline. To compensate the latency differences between the operators, delay units were introduced into the data paths. These delay elements are implemented very resource efficiently by configuring the LUTs of the FPGA as RAM based shift registers.

The total design fits in 49 % of the Logic resources of a Virtex XC2V3000-4 FPGA and is able to process the input data at 65 MHz design frequency.

The arithmetic unit is equivalent to 60 floating-point operations which results in a performance of 3.9 Gflops.

Modern general-purpose workstations typically achieve a floating-point performance in the order of 100 Mflops[1] for such application because of a memory-access bottleneck. So the FPGA application enables an arithmetic speedup of a factor around 30 to 40.

[1]On a Pentium III 1 GHz server workstation with 512 MB RAM a performance of 133 Mflops has been measured.

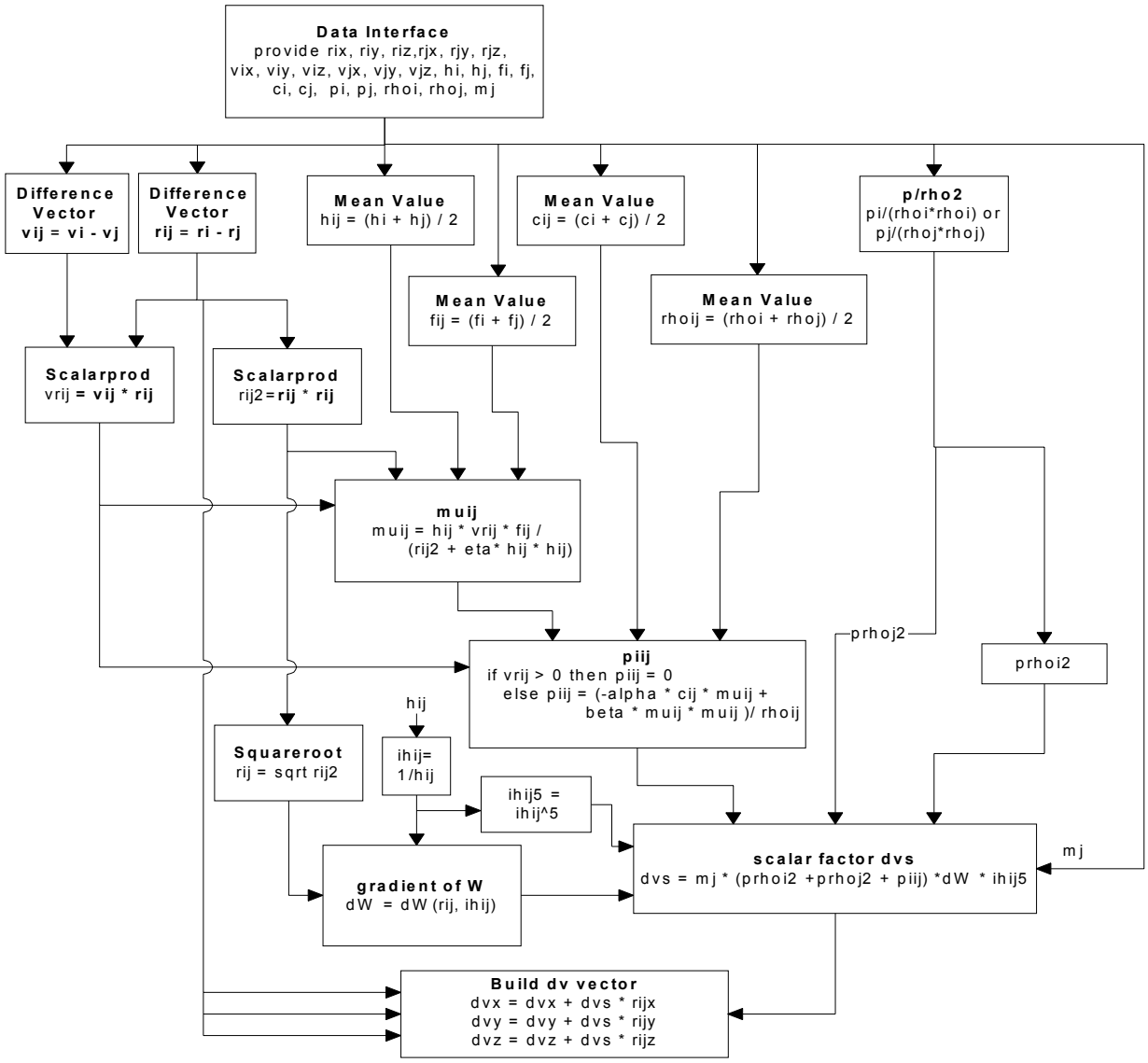

Figure 13. Structure of pressure force pipeline for processing the formula (4).

7. Conclusions

Assuming that scientific N-Body algorithms are calculated with sufficient accuracy by using reduced precision arithmetic we developed floating-point units which are parameterized in precision. The reduction of accuracy together with specializing the operators for different boundary conditions resulted in a significant reduction of resource requirements compared to standard single precision implementations. We have shown how much resources the basic floating-point operators utilize, depending on the precision of the floating-point number representation. We could demonstrate that modern FPGAs are capable of performing highly complex floating-point algorithms and gave a base of measurements for estimation of resource requirements of future implementations. We applied our floating-point library to a complex pressure force calculation step for SPH algorithms. The implementation fits on an off-the-shelf FPGA and resulted in a performance of 3.9 Gflops. As the implemented calculation step plays a central role in the SPH formulation this is a very important advance in our AHA-GRAPE project where the time-critical parts of astrophysical simulations shall become accelerated by FPGA platforms. We expect that a lot of similar

structured applications in science can become accelerated in the same way.

8. References

[1] Toshikazu Ebisuzaki et al., “GRAPE Project: An Overview”, Publications of the Astronomical Society of Japan, 1993, vol. 45, pp. 269-278.

[2] R. Spurzem, A. Kugel, “Towards the Million-Body Problem on the Computer - no news since the three-body-problem?”, Molecular Dynamics on Parallel Computers, Proc. Workshop NIC Juelich, World Scientific Press, Singapore, 1999.

[3] T. A. Cook, H.-R. Kim, L. Louca, “Hardware Acceleration of N-Body Simulations for Galactic Dynamics”, Proc. SPIE2607 on FPGAs for Fast Board Development and Reconfigurable Computing, 1995, pp. 42-53.

[4] T. Hamada, T. Fukushige, A. Kawai, J. Makino, “PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations”, Publ. of the Astronomical Society of Japan, 2000, vol. 52, pp. 943-954.

[5] J. J. Monaghan, “Smoothed Particle Hydrodynamics”, Annu. Rev. Astron. Astrophys, 1992, vol. 30, pp. 543-574.

[6] W. Benz, “Smooth Particle Hydrodynamics: A Review”, J.R.Buchler(ed), The Numerical Modelling of Nonlinear Stellar Pulsations, Kluwer Academic Publishers, 1990, pp. 269-288.

[7] M. R. Bate and A. Burkert, “Resolution Requirements for Smoothed Particle Hydrodynamics Calculations with Self-gravity”, Mon. Not. R. Astron. Soc., 1997, vol. 288, pp. 1060-1072.

[8] T. Kuberka, A. Kugel, R. Männer, H. Singpiel, R. Spurzem, R. Klessen, “AHA-GRAPE: Adaptive Hydrodynamic Architecture - GRAvity PipE”, Proc. Int´l Conf. on Parallel and Distributed Processing Techniques and Applications, 1999, vol. 3, pp. 1189-1195.

[9] A. Kugel, “RACE-1 - A PCI-64 based High Performance FPGA Co-Processor”, http://www-li5.ti.uni-mannheim.de/fpga/race/, Jan. 2002.

[10] Behrooz Parhami, “Computer Arithmetic, Algorithms and Hardware Designs”, Oxford University Press, 2000.

[11] Yamin Li, Wanming Chu, “Implementation of Single Precision Floating Point Square Root on FPGAs”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1997, pp. 226-232.

[12] L. Lourca, T. A. Cook and W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1996, pp. 107-116.

[13] M. E. Louie and M. D. Ercegovac, “Mapping Division Algorithms to Field Programmable Gate Arrays”, Proc. 26th Asilomar Conference on Signals, Systems, and Computers, 1992, pp. 371-375.

[14] J. N. Coleman and E. I. Chester, "A 32b Logarithmic Number System Processor and Its Performance Compared to Floating Point", Proc. 14th IEEE Symposium on Computer Arithmetic, 1999, pp. 142-152.

[15] N. Shirazi, A. Walters, and P. Athanas, “Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995, pp. 155-162.

[16] W. B. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, K. D. Underwood, “A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs”, Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1998, pp. 206-215.

[17] B. Fagin and C. Renard, “Field programmable gate arrays and floating point arithmetic”, IEEE Transactions on VLSI, 1994. vol. 2, no. 3, pp. 365-367.

[18] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900.