Estimation of Sample Mean and Variance for Monte-Carlo Simulations

David B. Thomas and Wayne Luk
Imperial College London
{dt10,wl}@doc.ic.ac.uk

Abstract

Monte-Carlo simulations are able to provide estimates of solutions for problems that are otherwise intractable, by examining the aggregate behaviour of large numbers of random simulations. Because these simulations are independent and embarrassingly parallel, FPGAs are increasingly being used to implement Monte-Carlo applications. However, as the number of simulation runs increases, the problem of accurately accumulating the aggregate statistics, such as the mean and variance, becomes very difficult. This paper examines three accumulation methods, adapting them for use in FPGA applications. In particular, we develop a mean and variance calculator based on cascading accumulators, which is able to process streams of floating-point data in one pass, while operating in fixed-point internally. This method has the advantage that it calculates the exact sample mean and an accurate numerically stable sample variance, while using few logic resources and providing performance to match commercial floating-point operators: clock rates of 434MHz are achieved in Virtex-5, 1.46 times faster than a double-precision accumulator, while using one eighth of the resources.

1. Introduction

Monte-Carlo simulations are a class of applications that often map particularly well to FPGAs, due to the embarrassingly parallel nature of the computation. The huge number of independent simulation threads allow FPGA-based simulators to be heavily pipelined [6], and also allow multiple simulation instances to be placed within one FPGA, providing a roughly linear increase in performance as the size of the FPGA is increased. A second advantage is the relatively low IO required per simulator instance, meaning that low performance buses can support very large amounts of computation without becoming a bottleneck [8].

A side-effect of high-performance FPGA Monte-Carlo simulators is that they generate huge numbers of results per second, and statistics on these results must be accurately accumulated to provide the overall answer. To keep up with the simulations this accumulation must happen at full-speed, processing large streams of data in one pass, and using constant resources to process one result per clock-cycle. The accumulators must also be able to maintain accuracy over large numbers of samples, to ensure that numerical stability problems do not bias the overall simulation results.

This paper examines methods for the calculation of mean and variance statistics over large data sets using FPGAs. Our contributions are:

- Identification of accumulation methods used in software that can be adapted to work in hardware.
- An adaptation of the cascading accumulators method for hardware, allowing floating-point data to be efficiently processed in fixed-point format.
- An empirical evaluation of the accuracy of five different hardware accumulation methods, showing that only the cascading accumulator and double-precision methods maintain sufficient accuracy.
- A comparison of area and speed, showing that the cascading accumulators method using one eighth of the logic resources of the double precision method, while operating 1.46 times faster.

2. Motivation

At an abstract level Monte-Carlo simulations can be described as a tuple $(\mathbf{P}, \mathbf{O}, \mathbf{A}, f, a, p, d_0)$:

$\mathbf{P}$: The set of possible simulation parameters, including environmental parameters and the starting state.

$\mathbf{O}$: The set of possible simulation run outputs, containing the results produced by a single simulation run.

$\mathbf{A}$: The set of accumulator states.

$f : \mathbf{P} \mapsto \mathbf{O}$: A stochastic simulation function which maps a simulation input to one possible result. This function must involve some randomness, so two executions using the same simulation input will not provide the same output.

$a : \mathbf{A} \times \mathbf{O} \mapsto \mathbf{A}$: A deterministic accumulation function that combines the current accumulator state with the result from a simulation run. The function should be order independent, i.e. $a(a(d,e_1),e_2) = a(a(d,e_2),e_1)$.

$p \in \mathbf{P}$: An element from the simulation input set giving the starting point for the application.

$d_0 \in \mathbf{A}$: The initial state of the accumulator.

The simulation is then executed by generating a series of simulation results, and accumulating the results:

$$d_i = a(d_{i-1}, e_i) \qquad e_i = f(p) \tag{1}$$

As $i \to \infty$ the estimate accumulated into d_i will asymptotically converge on the average properties of $f(p)$.

In a hardware implementation it is convenient to view each simulator instance as a black box, with one input that accepts p, and one output that produces a stream of results derived from independent simulation runs. The rate of result generation will vary between different simulations, and between different architectures implementing a given simulation. In the extreme case the simulator will generate one new result every cycle [4], while in other cases the simulator will on average produce less than one result per cycle.

Even when generating less than one result per cycle, the simulator may exhibit bursty behaviour, where the simulator produces one result per cycle for a short period of time. This may be somewhat predictable: if for example the simulator uses a circular pipeline of depth k, and each simulation run requires m circuits of the pipeline, then there will be a burst of k results followed by a gap of $k(m-1)$ cycles [6, 9]. However, more complicated simulations may take a random number of cycles to complete, such that the length of each burst follows a probability distribution [7]. The tails of this distribution may be very long, making the probability of a very long burst small, but still non-zero.

In such a situation it is important not to drop any results in the burst, as this may bias the overall result of the simulation. For this reason it is necessary to design accumulators that can handle the worst case, so that they can accept a sustained rate of one input value per cycle. This has the desirable side-effect of making the interface between the simulation black-box and the accumulator modular, as the simulator can simply throw results out whenever they become ready, without needing to worry about whether the downstream accumulator is ready to accept them.

So far we have not specified exactly what the accumulator does, as there are a number of choices. In some applications it is necessary to estimate the whole empirical distribution function of the simulation results, requiring complex histogram operations, but this is relatively uncommon. A slightly simpler case is when it is only necessary to estimate one quantile of the results; this occurs, for example, in Value-at-Risk calculations, where one wishes to estimate the 1% quantile of loss (how much one might lose every one in a hundred days).

In this paper we consider the simplest and most common type of accumulator: the arithmetic mean. This statistic estimates the expected value of the stochastic process as the sample count increases to infinity, and is used in many applications: for example, in pricing financial instruments one typically requires the average price over all possible future outcomes, while in Monte-Carlo integration it is necessary to estimate the average occupancy over the integration domain.

If we assume that the output of each simulation is a scalar value, then the sample mean ($\overline{x}$) of the first n simulation runs is defined as:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2}$$

For example, if the samples were generated by a Monte-Carlo simulation for option pricing, where each value x_i is the price of the option at the end of one simulation trial, then $\overline{x}$ is an estimate of the true price of the option.

By itself the value of $\overline{x}$ is not very helpful, as it gives no idea how much confidence can be placed in the estimate. The second key statistic is the sample variance (s^2), which provides an estimate of how much the individual samples are spread around the mean:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{x})^2 \tag{3}$$

An important related statistic is the sample standard deviation (s), which is simply the square root of the variance. The advantage of the standard deviation is that it is measured in the same units as the data samples, while the variance is measured in squared units.

The definitions of mean and variance are straightforward, but actually calculating the statistics is more complex, particularly for the large numbers of samples generated by Monte-Carlo simulations: an FPGA might be configured with 10 or more simulation cores, each generating one result per cycle at 500MHz, so even one second of execution provides over 2^{32} samples, which imposes a number of requirements on the accumulators.

The first requirement is that the accumulator must be able to operate in a streaming mode, using constant resources, and able to accept one new floating-point value per cycle at the same clock rate as the floating-point cores used to calculate the value. In the formal definition of variance (Equation 3) it is assumed that the mean has already been calculated. This is a problem for streaming data applications, as it requires all the intermediate samples to be stored between the calculation of

the mean and the variance. This is a particular problem in an FPGA, as a RAM that is both large enough and fast enough to buffer all the samples is likely to be very expensive, so the accumulator must be able to operate on just one pass through the data.

A second requirement is that the estimates of the mean and variance are accurate, particularly when the results are floating-point values that may range over many orders of magnitude. Using the wrong method for accumulating the mean may lead to significant biases in the result, while some simple methods for calculating the variance can even lead to nonsensical results such as negative variance. Any realistic method needs to produce a very accurate mean, and must at least provide a tight upper bound on the variance.

A third requirement is that the IO requirements of the accumulator are very low. Most numerical Monte-Carlo simulations are organised in a system that contains both a CPU and an FPGA (or more than one of each). The CPU is responsible for managing jobs and transferring simulation inputs to the FPGA. Within the FPGA multiple simulation instances all work in parallel on the same simulation instance, producing an aggregate estimate. As the simulations proceed the CPU occasionally checks the state of each instance, combining the results to produce one overall result. Based on the standard error of this result the CPU can then decide whether the answer is sufficiently accurate, or whether more simulation runs are needed.

Depending on the interface between the CPU and the FPGA, the bandwidth may be low and the latency high, making it expensive for the CPU to read each accumulator's state. This is compounded by the fact that the CPU has to manage multiple simulation instances within the FPGA, as well as performing other work such as network operations and tasks. Because of this, it is critical that the accumulator is able to process data for a large number of samples between communications with the host. A practical minimum is that each accumulator should be able to operate independently for at least one second; assuming a maximum clock rate of 500MHz, that corresponds to the processing of 2^{30} inputs.

The requirements for our accumulator are to:

1. calculate the mean and variance for a stream of scalar floating-point values.
2. sustain a rate of one input per cycle.
3. use constant resources (i.e. resources do not increase with number of inputs processed).
4. provide accurate estimates for the mean and variance for sample sizes up to at least 2^{30}, without any intermediate IO.
5. operate at the same rate as floating-point units.
6. have low resource usage.

3. Existing Methods

In this section we identify a number of existing algorithms for calculating the mean and variance of data, and identify those that can be adapted for use in a hardware accumulator.

Textbook algorithm : By rearranging the definitions of mean and variance, it is possible to develop a one-pass algorithm by tracking three values: T, the sum of all the sample values; Q, the sum of all the squared sample values; and n, the total number of samples accumulated.

The values are updated for each new datum x_i:

$$T_i \leftarrow T_{i-1} + x_i \qquad Q_i \leftarrow Q_{i-1} + x_i^2 \tag{4}$$

Using these three values the statistics can be calculated for all the samples seen so far:

$$\overline{x} = T_n/n \qquad s^2 = \left[Q_n - T_n^2/n\right]/n \tag{5}$$

This method requires few operations per sample, but is also very numerically unstable when implemented using floating-point. The main problem is in the calculation of the variance in Equation 5, as the values of Q and T^2/n may be similar in magnitude if the variance is small compared to the mean, leading to large cancellation errors or even a negative estimated variance.

Updating algorithm : The instability can be treated by calculating the mean and variance using the *updating* (or online) method [1]. The updating algorithm tracks M, the sample mean, and S, the sum of squared differences about M:

$$M_i \leftarrow M_{i-1} + \delta/i, \qquad \text{where } \delta = x_i - M_{i-1} \tag{6}$$

$$S_i \leftarrow S_{i-1} + \delta(x_i - M_i) \tag{7}$$

The disadvantage of this method is that the computational cost is significantly higher: the number of additions has increased from two to four, and there is now also a division operation.

Pairwise summation : One method used in the general floating-point summation problem is pair-wise summation, which allows the use of floating-point operations while retaining numerical stability. The central idea is to form a binary tree of intermediate sums and sums of squares, from the input samples at the leaves up to the final overall sum and sum of squares at the root. Because each node's children accumulate a similar number of samples, the chance of cancellation is greatly reduced. However, this method requires a non-constant number of resources, as the storage and computational requirements grow with $O(\log n)$, so we do not consider it here.

Cascading accumulators : A general technique for reducing cancellation in floating-point adders is to make

sure that only terms of a similar magnitude are added together. In the pairwise summation approach this is achieved implicitly, as it is assumed that the magnitude of sums at different nodes in the tree will grow with depth, so two nodes at the same level will have approximately the same magnitude. Cascading accumulators explicitly group together terms of similar magnitude, by taking advantage of the floating-point exponent.

Each floating-point number can be viewed as $s \times f \times 2^e$, where $s \in \{-1,1\}$, $f \in [1,2)$, and $e \in \mathbb{Z}$ (ignoring denormals and zero). If we want to group together numbers with similar magnitudes, then any two numbers with the same exponent will be within one binary order of magnitude of each other. The idea behind the cascading accumulator is to maintain a table of accumulators, with one accumulator for each exponent value. In practice the set of possible exponents is finite, so the table size is a constant determined by the number of exponent bits in the floating-point format.

4. Hardware Architecture for Textbook and Updating Algorithm

Both the textbook and updating methods maintain a three element state, with each element of the state updated for every new value that is accumulated. The state update functions require one or more floating-point operations, which in software can be executed sequentially, allowing a single accumulation state to be updated in place. However, in hardware these floating-point operations must be pipelined for efficiency, and the loop carried dependency cannot be removed.

The solution to this problem is to use a C-Slow approach, which uses multiple independent accumulation states, all cycling through the same pipelined loop [5]. For example, the textbook algorithm contains a dependency between T_i and T_{i-1} via an addition. If the pipelined addition operator requires p cycles, then we introduce p completely independent accumulation states, which rotate through the pipeline.

A minor disadvantage of using this approach is that the aggregate mean and variance must be calculated in software, using the values from the p separate accumulators. For the textbook method this is very simple: if (T_n, Q_n) and (T_m, Q_m) are the states of two accumulators, then the combined accumulator state is simply:

$$T_{n+m} = T_n + T_m \qquad Q_{n+m} = Q_n + Q_m \tag{8}$$

By repeatedly combining the individual states, eventually one combined state is produced, which estimates the mean and variance over all the samples.

In the case of the updating method more work is required to update the running estimates of the sum of squared differences (S), as each accumulator will have taken squared differences around a slightly different sample mean (M):

$$M_{n+m} = (nM_n + mM_m)/(n+m) \tag{9}$$

$$S_{n+m} = S_n + S_m + \frac{m}{n(m+n)}(nM_m - mM_n)^2 \tag{10}$$

An important difference between the textbook and updating methods is that in the updating method, it is necessary to know the number of samples associated with each accumulator state, as the values of n and m are needed to combine individual states. However, in the textbook method it is only necessary to know the *total* number of samples across all the states, without needing to track the number of samples contributed to each value of T and Q. Tracking the total number of samples means that the textbook method can save a small amount of hardware, as it can use a single counter shared amongst all the accumulator states, rather than requiring storage for p separate counters (where p is the C-Slow factor).

Figure 1 gives an example of a pipelined textbook accumulator, using floating-point operators with three cycles of latency. Floating-point values are streamed into the accumulator via *iX*, with the *iValid* input indicating on each cycle whether *iX* is valid or not. The input value is then squared, with the associated value and valid flag buffered to keep synchronised. The valid flag is used to control whether a counter is incremented, which tracks the total number of samples across all accumulator states. The sample value and squared sample value are added to whichever accumulator state happens to be exiting the bottom of the adders when they arrive.

On successive cycles the outputs of the design will cycle through the different accumulator states, which could in principle be used for online monitoring purposes. However, the more likely scenario is that in the client-server application model the client (software) will occasionally request an update to see whether the results have reached the desired accuracy. We can take advantage of this occasional updates to clear the accumulator states, as the fewer samples that are combined in each accumulator state, the more accurate the estimate will be. In the design shown, when *iRead* is asserted for three cycles the whole state is streamed out, while at the same time all the accumulators are cleared.

5. Hardware Architecture for Cascading Accumulators

In software the cascading accumulators method is usually implemented using floating-point operations for each accumulator, but again we run into the problem

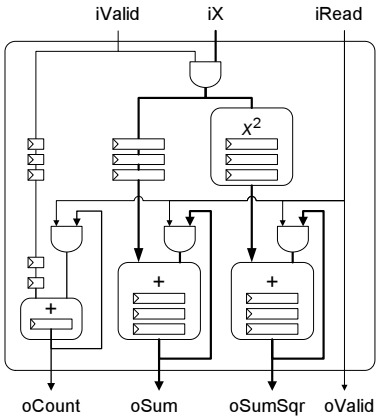

Figure 1. Example of textbook accumulator using floating-point adders with a latency of 3.

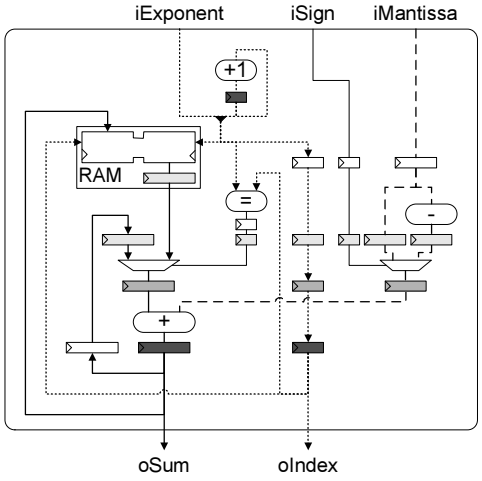

Figure 2. Cascading accumulators architecture for calculating the sum of floating-point values.

of high-latency floating-point operators combined with loop-carried dependencies. In principle the same C-Slow approach could be used, but this leads to excessive consumption of memory. To support the 8-bit exponents of single-precision floating-point numbers, we must use a RAM with 256 entries. However, to support the 12 cycle delay of a Virtex-5 floating-point adder, we must replicate each entry 12 times, requiring 3072 entries. Each entry must contain both a running sum and sum of squares, so the total number of 36x1024 Virtex-5 RAM primitives is 6.

As well as the RAM usage, another problem with single-precision accumulators is that it does not necessarily lead to a more accurate calculation of mean and variance. For example, if all the input samples happen to have the same exponent (e.g. they were all between one and two), then the cascading accumulator would effectively become the textbook accumulator. So such an arrangement would require all the resources of the textbook accumulator, plus a number of block RAMs, without improving accuracy.

Our solution to this problem is to observe that much of the logic associated with floating-point adders is due to the shifting of values, both when aligning the input arguments, and when normalising the result. However, the whole point of the cascading accumulators method is that values added to a given accumulator will have the same exponent, so we can add the mantissas together without any shifting. This has a number of advantages:

- Logic resource utilisation is greatly reduced, as the floating-point adders reduce to fixed-point adders.
- The fixed-point adders have lower latency, reducing the RAM resources needed to support C-Slow.
- In the absence of overflow, each accumulator will be exact, so the aggregated sum is also exact.

This last point is particularly interesting, as it eliminates one of the key worries when attempting to extract accurate results from a huge numbers of simulations. This is particularly important for the calculation of the mean, as it is the mean that is the "answer" that the simulation provides, while the variance only estimates the accuracy of that answer.

In order to achieve exact accumulation, we make sure that the fixed-point accumulators cannot overflow. A floating-point number with m_w mantissa bits, including the implicit bit, can be converted into a fixed-point twos-complement number with $1+m_w$ bits. If we wish to accumulate up to n samples without overflowing, then the fixed-point accumulators must have at least $1+log_2(n)+m_w$ bits. However, even with the reduced latency, the accumulator update must still be C-Slow by some factor p, which means that the accumulator (i.e. each RAM entry) must have $1+log_2(n/p)+m_w$ bits.

Figure 2 shows how the fixed-point cascading accumulator is constructed, showing just the summation part (no sum of squares or counter). A C-Slow factor of $p=4$ is used, as this provides enough stages to pipeline, while mapping well into the block RAMs of contemporary architectures. The different stages of the pipeline are shown as different shades of grey on each register, from white through to dark grey, then back to white.

On the right, the dashed data-path shows the conversion of the sign-magnitude mantissa into twos-complement. On the left, the solid data-path shows the fixed-point accumulation pipeline, where values are read from the accumulator table, added to the incoming value, then stored again. In the centre, the dotted data-path shows the flow of addresses used to select the accumulators. This is formed from a combination of the

value's exponent, and a 2-bit slot counter that is incremented every cycle. The slot counter is to make sure that over a four cycle period it is impossible for each RAM value to be read or written more than once.

An important practical consideration in modern FPGAs is the presence of pipeline registers in the block RAMs. If these registers are not used, then the performance of the RAMs is severely limited, so it is essential to support them. However, this consumes a valuable pipeline stage, meaning that all the entry update logic must be completed in a single stage. To recover this stage we added collision detection, which is able to check when the address currently being written is also being read. When a collision happens, the value just calculated is forwarded, rather than using the stale value from memory. This emulates Write-Before-Read behaviour between the two ports, a feature that is usually only possible when using just one port.

As before, the current accumulator value is read by streaming all the individual accumulator values out over a number of successive cycles, performing a reset at the same time. During the read/reset period a small control circuit takes control of the address generator for the RAM, so that all the addresses of the RAM are examined in turn (not shown in the figure). This means that during the reset period any new values coming into the accumulator will be discarded. However, the reset period only takes as many cycles as there are entries in the RAM, and when viewed as a fraction of the total accumulation cycles, it is irrelevant. The ignored values will also not bias the results in any way, as the chance of missing any data item due to a reset will be independent of both the data generation process and the data values.

Figure 2 shows only the sum pipeline, and does not include the circuitry to calculate the sum of squares, or to track the total number of samples; as with the textbook method, it is not necessary to maintain a separate count for each accumulator state. The sum of squares is calculated by squaring the mantissa of the incoming value, and accumulating that in the same way as is shown for the basic sum. It is possible to make the sum of squares exact, but this requires much wider accumulators: the exact squared mantissa requires $2m_w$ bits (with no sign bit), so a width of $log_2(n/p) + 2m_w$ would be needed. To put this in context, for typical values of $n = 2^{30}$, $p = 4$, and $m_w = 24$, this would require the RAM entries and adders to be 77 bits wide, and would need a total of 130 bits per RAM entry to store both the sum and sum of squares.

Another choice is to round the squared mantissa to $1 + m_w$ bits before adding it to the accumulator, which is roughly equivalent to performing the square using the input floating-point precision. Although this means that the sum of squares is no longer exact, the only source of error is the per-value rounding error, with no additional error introduced when multiple values are combined. Using this scheme both the sum and sum of squares accumulators are 53 bits wide, requiring a total of 106 bits per RAM entry (i.e. 3×36-bit parallel block RAMs).

6. Evaluation

The cascading accumulator was described using FPGA-family independent VHDL, relying on HDL inference for the instantiation of block RAMs and DSPs. However, the arrangement of logic and registers is targeted specifically for the RAMB36 and DSP48E primitives, so performance and resource utilisation on other FPGA families would be less optimal.

The textbook and updating accumulators are described using Handel-C, with all floating-point components generated using Xilinx CoreGen 9.2. All cores are generated using the "*maximum pipelining*" option, with "*full usage*" of DSP48E resources for multipliers, and "*no usage*" for adders. The Handel-C code is compiled to VHDL using DK5 with default optimisations. [1]

All designs are synthesised using XST 9.2, with all optimisations for speed enabled. The designs are then placed and routed for the Virtex-5 xc5vlx110 using ISE 9.2, using all default settings. No timing constraints are used for any of the designs.

Table 1 shows the results for each of the accumulators, ordered from top to bottom by increasing resource utilisation, with figures in brackets indicating performance relative to the cascading accumulator. By far the smallest (in terms of logic resources) is the cascading accumulator, which is a third the size of even the basic single-precision textbook accumulator. The only drawback is that it needs 3 block RAMs, while none of the others need RAM resources.

The cascading accumulator is also the joint fastest method, achieving a clock rate of 434MHz, the same as the single-precision textbook accumulator. The performance limit is the 53-bit fixed-point adder used to update the accumulators, which is implemented using a non-pipelined ripple-carry adder. In principle the performance could be increased by pipelining the adder, but in practise it can already operate at the same speed as the floating-point cores that would be used to calculate the values it accumulates.

The final two columns of Table 1 provide an empirical estimate of the accuracy of each method. This was measured by generating 2^{30} normally distributed random samples, with a mean and variance of 1. The

[1] Note that the performance bottleneck of all Handel-C designs was the floating-point cores, so there is no performance penalty associated with using Handel-C as opposed to VHDL.

	Resources						Speed	Accuracy	
Method	Slices	LUT-FF Pairs	LUT	FF	DSP	RAM	MHz	Mean	Std
Cascading	223 (1.0)	601 (1.0)	473	575	2	3	434 (1.00)	∞	36.4
Textbook[Single]	768 (3.4)	2127 (3.5)	1732	1971	2	0	434 (1.00)	0.1	5.4
Updating[Single]	1441 (6.5)	4976 (8.3)	3458	4435	2	0	327 (0.75)	19.8	6.4
Textbook[Double]	1597 (7.2)	5094 (8.5)	4191	4728	12	0	297 (0.68)	46.4	38.8
Updating[Double]	3937 (17.7)	13985 (23.3)	9013	12428	12	0	288 (0.66)	48.5	48.5

Table 1. Resource utilisation and performance for accumulators in the Virtex-5 xc5vlx110.

same sequence was generated for each test, with the accuracy measured against a reference accumulator which uses 384-bit floating-point. The accuracy is measured as accurately calculated bits of precision for the mean and standard deviation: given a value x calculated by the accumulator, and x_*, the reference value, the bits of precision are calculated as $-log_2((x-x_*)/x_*)$.

The most striking figure is just how inaccurate the single-precision textbook method becomes, with complete loss of precision for the estimate of the mean. The problem is that once the running sums in the accumulator exceed 2^{25} many of the values being accumulated are too small in comparison to have any effect, with the problem becoming more serious the larger the accumulators become. Figure 3 charts the change in accuracy for the methods as the sample size is increased (missing data-points indicate no error), showing that the single precision methods gradually decline from about 26 bits down to about 18 bits, with the single-precision textbook method failing catastrophically at $n = 2^{29}$. By comparison all the double precision methods maintain over 45 bits of precision throughout. The cascading method is exact at all data points, so it is not plotted.

Figure 4 shows the change in accuracy for the standard deviation with increasing sample sizes. As before, the double precision versions are both accurate, but in this case the updating method has a clear advantage of 10 or more bits. The single-precision methods maintain approximately 25 bits of precision until a sample size of around 2^{18}, then the accuracy of both degrades at around 1 bit for every doubling of sample size, with the updating method maintaining an edge of about 5 bits.

The most interesting part of Figure 4 is the behaviour of the cascading method, as unlike the other methods, its accuracy actually *increases* with the sample size. Our hypothesis for explaining this behaviour is that the only error source is the rounding when each sample is squared, but then the accumulation of the squared values is exact. Because the input samples are random, the rounding error will also be random, so the central limit theorem comes into effect, meaning that as the sample size increases, the accuracy also increases.

If we combine the requirements of accuracy and resource efficiency, then the only two reasonable choices are the cascading accumulator and the textbook accumulator with double-precision. None of the single-precision methods are able to offer sufficient accuracy, while the increase in standard deviation accuracy provided by the double-precision updating method does not justify the huge number of resources it requires. Overall the cascading accumulator provides by far the best combination of speed, resource efficiency, and accuracy.

7. Related Work

Previous work on accumulation has focused on the floating-point summation problem, as this is used in the dot-product operations which form the core of matrix-matrix and matrix-vector operations. Recent work has examined the use of off-the-shelf floating-point operators, by constructing reduction circuits that use buffers to hold intermediate sums [10]. Another approach that attempts to deal with the inherent accuracy problems of floating-point summation splits the input data into groups, then aligns the values according to the largest value in each group [3]. These architectures are all driven by the need to reduce each summation to a single total, as the sum must be consumed elsewhere within the FPGA. However, in Monte-Carlo applications it is not necessary to reduce to a single value, as that step can be performed in software.

The most directly comparable method has been developed within the FloPoCo project [2], where the same idea of using a large fixed-point accumulator is used to provide increased accuracy without sacrificing speed. However, their approach is to choose a fixed-range for the floating-point accumulator, then to shift each floating-point mantissa to the correct position before adding to the accumulator. This means that the range of the fixed-point accumulator must be chosen before accumulation begins, so *a priori* knowledge of the number range is needed. The accumulation will also not be exact, and is limited to the precision chosen for the single accumulator. One benefit of the single-accumulator is that it can quickly be converted back into a floating-point number for consumption on chip,

but this is not important for Monte-Carlo applications. [2]

8. Conclusion

Gathering statistics is a key part of any Monte-Carlo simulation, and it is critical that these statistics are gathered accurately, without sacrificing performance or requiring large amounts of resources. In this paper we have examined three different methods for calculating mean and variance, showing how existing software methods can be adapted to provide high performance modules that are able to process large streams of floating-point data in one pass, without requiring any knowledge about the range of the input data.

By modifying the existing cascading accumulators method we are able to create an FPGA specific design, which is able to accept floating-point input data while using fixed-point internally, without requiring expensive and slow normalisation steps. As well as providing good performance and low resource utilisation, this method also has the advantage that it is able to calculate the exact mean of large streams of data, eliminating concerns that poor statistical accumulators may bias the results of simulations. Numerical experiments also show that as well as accumulating the mean exactly, the method is able to calculate the standard deviation to a relative error of 2^{-36} on sample sizes of 2^{30}.

Future work will further investigate the numerical behaviour of the fixed-point cascading accumulator, both through empirical studies, and by determining theoretical worse case bounds for the error. In addition we will investigate methods for reducing the number of RAMs required, particularly for floating-point formats with larger exponent ranges. Another possibility is the online accumulation of higher order statistics, such as skewness and kurtosis.

References

[1] T. F. Chan, G. H. Golub, and R. J. Leveque. Algorithms for computing the sample variance. *The American Statistician*, 37(3):242–247, 1983.

[2] F. D. Dinechin, B. Pasca1, O. Cret, and R. Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. Technical Report ensl-00268348, Ecoles normales supérieures de Lyon, 2008.

[3] C. He, G. Qin, M. Lu, and W. Zhao. Group-alignment based accurate floating-point summation on FPGAs. In *ERSA*, pages 136–142, 2006.

[4] G. W. Morris and M. Aubury. Design space exploration of the European option benchmark using Hyperstreams. In *Proc. FPL*, pages 5–10, 2007.

[2] This work appears to have been developed in parallel with ours, and only came to light when preparing the camera ready version of this paper. Quantative comparisons will be made in future work.

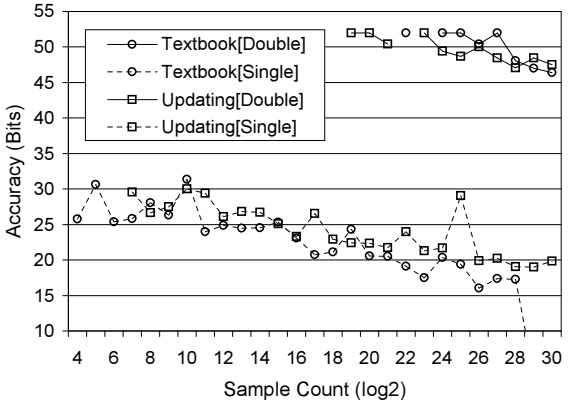

Figure 3. Accurately calculated bits of sample mean for increasing sample size.

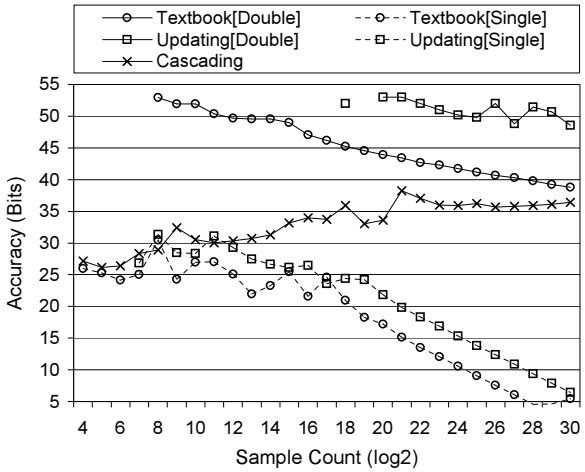

Figure 4. Accurately calculated bits of sample standard deviation for increasing sample size.

[5] H. Styles, D. B. Thomas, and W. Luk. Pipelining designs with loop-carried dependencies. In *Proc. FPT*, pages 255–263, 2004.

[6] D. B. Thomas, J. A. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations. In *Proc. ASAP*, pages 168–173, 2007.

[7] D. B. Thomas and W. Luk. A domain specific language for reconfigurable path-based Monte Carlo simulations. In *Proc. FPT*, pages 97–104, 2007.

[8] D. B. Thomas and W. Luk. Credit risk modelling using hardware accelerated Monte-Carlo simulation. In *Proc. FCCM*, 2008.

[9] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. In *Proc. FPT*, pages 215–224, 2005.

[10] L. Zhuo, G. R. Morris, and V. K. Prasanna. Designing scalable FPGA-based reduction circuits using pipelined floating-point cores. In *IPDP*, 2005.