Accounting for Secondary Uncertainty: Efficient Computation of Portfolio Risk Measures on Multi and Many Core Architectures

Blesson Varghese and Andrew Rau-Chaplin
Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
{varghese, arc}@cs.dal.ca

ABSTRACT

Aggregate Risk Analysis is a computationally intensive and a data intensive problem, thereby making the application of high-performance computing techniques interesting. In this paper, the design and implementation of a parallel Aggregate Risk Analysis algorithm on multi-core CPU and many-core GPU platforms are explored. The efficient computation of key risk measures, including Probable Maximum Loss (PML) and the Tail Value-at-Risk (TVaR) in the presence of both primary and secondary uncertainty for a portfolio of property catastrophe insurance treaties is considered. *Primary Uncertainty* is the the uncertainty associated with whether a catastrophe event occurs or not in a simulated year, while *Secondary Uncertainty* is the uncertainty in the amount of loss when the event occurs.

A number of statistical algorithms are investigated for computing secondary uncertainty. Numerous challenges such as loading large data onto hardware with limited memory and organising it are addressed. The results obtained from experimental studies are encouraging. Consider for example, an aggregate risk analysis involving 800,000 trials, with 1,000 catastrophic events per trial, a million locations, and a complex contract structure taking into account secondary uncertainty. The analysis can be performed in just 41 seconds on a GPU, that is 24x faster than the sequential counterpart on a fast multi-core CPU. The results indicate that GPUs can be used to efficiently accelerate aggregate risk analysis even in the presence of secondary uncertainty.

Keywords

Primary and Secondary Uncertainty, Aggregate Risk Analysis, GPU Computing, Parallel Computing, Risk Analytics, Risk Management

1. INTRODUCTION

Reinsurance companies who insure primary insurance companies against losses caused by catastrophes, such as earthquakes, hurricanes and floods, must quantify the risk related to large portfolios of risk transfer treaties. In reinsurance a portfolio represents complex insurance contracts covering properties against losses due catastrophes, and contracts include Per-Occurrence eXcess of Loss (XL), Catastrophe XL and Aggregate XL treaties. Aggregate Risk Analysis is performed on portfolios to compute risk measures including Probable Maximum Loss (PML) [1] and the Tail Value-at-Risk (TVaR) [2]. Such an analysis is central to treaty pricing and portfolio/solvency applications. The analysis may involve simulations of over one million trials in which each trial consists of one thousand catastrophic events each of which may impact tens of thousands to millions of individual properties, for example buildings. Not only is the analysis computationally intensive but also data intensive, and therefore the application of High Performance Computing (HPC) techniques is desirable.

The analysis can be run weekly, monthly or quarterly on production systems based on the requirement for updating the portfolio. For example, based on the fluctuation of currency rates an entire portfolio a weekly update can be performed which requires more than twenty four hours. Often times it is not sufficient to run routine analysis but requires ad hoc analysis. For example, consider a real-time pricing scenario in which an underwriter can evaluate different contractual terms and pricing while discussing with a client over a telephone. This cannot be accommodated on production system that is committed to routine analysis and the response required for real-time cannot be achieved on these systems. Hence, achieving significant speed up using high-performance computing techniques in risk analysis is desirable.

In our previous work [3], we explored the design and implementation of a parallel Aggregate Risk Analysis algorithm which was significantly faster than previous sequential solutions. However, it was limited for its use in portfolio wide risk analysis scenarios since the algorithm could only account for *Primary Uncertainty* - the uncertainty whether a catastrophic event occurs or not in a simulated year. It was not able to account for *Secondary Uncertainty*, the uncertainty in the amount of loss incurred when the event occurs. In this paper, secondary uncertainty is taken into account whereby loss distributions flow through the simulation rather than mean loss values due to an event.

In practice there are many sources of secondary uncertainty in catastrophic risk modelling. For example, the exposure data which describes the buildings, their locations, and construction types may be incomplete, lacking sufficient detail, or may just be inaccurate. Also physical modelling of

WHPCF'13 November 18 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2535557.2535562 ...$15.00.

hazard, for example an earthquake, may naturally generate a distribution of hazard intensity values due to uncertainty in energy attenuation functions used or in driving data such as soil type. Lastly, building vulnerability functions are simplifications of complex physical phenomenon and are therefore much better at producing loss distributions than accurate point estimates. An aggregate risk analysis algorithm that accounts only for primary uncertainty uses only mean loss values and fails to account for what is known about the loss distribution. Therefore, aggregate analysis needs to take both primary and secondary uncertainty into account by considering event loss distributions represented by the event occurrence probability, mean loss, and independent and correlated standard deviations. This captures a wide range of possible outcomes.

The research reported in this paper proposes an aggregate risk analysis algorithm capable of capturing both primary and secondary uncertainty. The algorithm is designed to run efficiently on both multi-core CPUs and many-core GPUs. A distribution of losses is used in the simulation rather than just mean loss values and efficient statistical operations, such as Cumulative Distribution Functions and Quantiles of the Normal and Beta distributions need to be performed. A significant challenge to not only balance the workload across threads performing fixed time operations (for example, addition operation), but also to balance the workload when individual numerical operations require variable time (for example, computations in iterative methods) for achieving effective parallelism is addressed. The implementation and optimization of the algorithm for multi and many core architectures is presented along with experimental evaluation. Since performance of the algorithm is dependent on the underlying statistical operations, four different statistical libraries were explored for use on the GPU and an additional three statistical libraries for use on the CPU. A parallel simulation of 800,000 trials with 1,000 catastrophic events per trial on an exposure set and on a contract structure taking secondary uncertainty exhibits a speedup of 24x over the sequential implementation on the CPU.

The remainder of this paper is organized as follows. Section 2 proposes an algorithmic framework for performing Aggregate Risk Analysis with Primary and Secondary Uncertainty. Section 3 presents how secondary uncertainty is applied within the inner loop of the risk analysis algorithm. Section 4 describes the implementation of the proposed algorithm on the GPU platform. Section 5 highlights the results obtained from experimental evaluation. Section 6 concludes the paper by considering future work.

2. AGGREGATE RISK ANALYSIS WITH PRIMARY AND SECONDARY UNCERTAINTY

Stochastic Monte Carlo simulations are required for portfolio risk management and contract pricing. Such a simulation in which each trial of the simulation represents a distinct view of which catastrophic events occur and in what order they occur in a contractual year is referred to as Aggregate Risk Analysis [4, 5, 6]. One merit of performing such an analysis is that millions of alternative views of a single contractual year can be obtained. This section considers the inputs required for performing aggregate risk analysis, proposes an algorithm for aggregate risk analysis, considers the financial terms employed in the algorithms, and presents the output of the analysis.

2.1 Inputs

Three data tables are input to Aggregate Risk Analysis. The first table is the Year Event Table (YET), denoted as YET, which is a database of pre-simulated occurrences of catastrophic events from a catalogue of stochastic events. Each record in a YET called a "trial", denoted as T_i, represents a possible sequence of event occurrences for any given year. The sequence of events is defined by an ordered set of tuples containing the ID of an event and the time-stamp of its occurrence in that trial

$$T_i = \{(E_{i,1}, t_{i,1}, z_{(Prog,E)_{i,1}}), \ldots, (E_{i,k}, t_{i,k}, z_{(Prog,E)_{i,k}})\}.$$

The set is ordered by ascending time-stamp values. Program-and-Event-Occurrence-Specific random number, $z_{(Prog,E)}$ is considered in Section 3. A typical YET may comprise thousands to a million trials, and each trial may have approximately between 800 to 1500 'event time-stamp' pairs, based on a global event catalogue covering multiple perils. The YET can be represented as

$$YET = \{T_i = \{(E_{i,1}, t_{i,1}, z_{(Prog,E)_{i,1}}), \ldots, (E_{i,k}, t_{i,k}, z_{(Prog,E)_{i,k}})\}\},$$

where $i = 1, 2, \ldots$ and $k = 1, 2, \ldots, 800 - 1500$.

The second table is the Extended Event Loss Tables, denoted as $XELT$, which represents a collection of specific events and their corresponding losses with respect to an exposure set. In addition, a few parameters, namely the Event-Occurrence-Specific random number ($z_{(E)}$), the independent standard deviation of loss (σ_I), the correlated standard deviation of loss (σ_C), and the maximum expected loss (max_l) are represented within the $XELT$. The loss associated with an event E_i is represented as μ_l is required for the analysis with secondary uncertainty. Applying secondary uncertainty using the XELT is presented in Section 3.

Each record in an XELT is denoted as 'eXtended' event loss

$$XEL_i = \{E_i, l_i, z_{(E)_i}, \sigma_{I_i}, \sigma_{C_i}, max_{l_i}\}.$$

and the financial terms associated with the XELT are represented as a tuple

$$\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2, \ldots).$$

A typical aggregate analysis may comprise 10,000 XELTs, each containing 10,000-30,000 extended event losses with exceptions even up to 2,000,000 extended event losses. The XELTs can be represented as

$$XELT = \left\{ \begin{array}{lcl} XEL_i & = & \{E_i, \mu_{l_i}, z_{(E)_i}, \\ & & \sigma_{I_i}, \sigma_{C_i}, max_{l_i}\}, \\ \mathcal{I} & = & (\mathcal{I}_1, \mathcal{I}_2, \ldots) \end{array} \right\}$$

with $i = 1, 2, \ldots, 10,000 - 30,000$.

The third table is the Portfolio, denoted as PF, which contains a group of Programs, denoted as P and represented as

$$PF = \{P_1, P_2, \cdots, P_n\}$$

with $n = 1, 2, \ldots, 10$.

Each Program in turn covers a set of Layers, denoted as L, which covers a collection of XELTs under a set of layer terms. A single layer L_i is composed of two attributes. Firstly, the set of XELTs

$$\mathcal{E} = \{XELT_1, XELT_2, \ldots, XELT_j\},$$

and secondly, the Layer Terms, denoted as

$$\mathcal{T} = (\mathcal{T}_{OccR}, \mathcal{T}_{OccL}, \mathcal{T}_{AggR}, \mathcal{T}_{AggL}).$$

A typical Layer covers approximately 3 to 30 individual XELTs. The Layer can be represented as

$$L = \left\{ \begin{array}{rcl} \mathcal{E} & = & \{XELT_1, XELT_2, \ldots, XELT_j\}, \\ \mathcal{T} & = & (\mathcal{T}_{OccR}, \mathcal{T}_{OccL}, \mathcal{T}_{AggR}, \mathcal{T}_{AggL}) \end{array} \right\}$$

$$\text{with } j = 1, 2, \ldots, 3 - 30.$$

2.2 Algorithm

The basic algorithm (line no. 1-17 shown in Algorithm 1) for aggregate analysis has two stages. In the first stage, data is loaded into local memory what is referred to as the preprocessing stage in this paper. In this stage YET, $XELT$ and PF, are loaded into memory.

Algorithm 1: Aggregate Risk Analysis with Primary and Secondary Uncertainty

Input : YET, $XELT$, PF
Output: YLT

```
1  for each Program, P, in PF do
2      for each Layer, L, in P do
3          for each Trial, T, in YET do
4              for each Event, E, in T do
5                  for each XELT covered by L do
6                      Lookup E in the XELT and find
                       corresponding loss, l_E
7                      Apply Secondary Uncertainty to l_E
8                      Apply Financial Terms to l_E
9                      l_T ← l_T + l_E
10                 end
11                 Apply Occurrence Financial Terms to l_T
12                 Apply Aggregate Financial Terms to l_T
13             end
14         end
15     end
16 end
17 Populate YLT using l_T
```

In the second stage, the four step simulation executed for each Layer and for each trial in the YET is performed as shown below and the resulting Year Loss Table (YLT) is produced.

In the first step shown in line no. 6 in which each event of a trial and its corresponding event loss in the set of XELTs associated with the Layer are determined. In the second step shown in line nos. 7-9, secondary uncertainty is applied to each loss value of the Event-Loss pair extracted from an XELT. A set of contractual financial terms are then applied to the benefit of the layer. For this the losses for a specific event's net of financial terms $\mathcal{I}$ are accumulated across all XELTs into a single event loss shown in line no. 9. In the third step in line no. 11 the event loss for each event occurrence in the trial, combined across all XELTs associated with the layer, is subject to occurrence terms. In the fourth step in line no. 12 aggregate terms are applied. The next sub-section will consider how the financial terms are applied.

2.3 Applying Financial Terms

The financial terms applied on the loss values combined across all XELTs associated with the layer are Occurrence and Aggregate terms. Two occurrence terms, namely (i) Occurrence Retention, denoted as $\mathcal{T}_{OccR}$, which is the retention or deductible of the insured for an individual occurrence loss, and (ii) Occurrence Limit, denoted as $\mathcal{T}_{OccL}$, which is the limit or coverage the insurer will pay for occurrence losses in excess of the retention are applied. Occurrence terms are applicable to individual event occurrences independent of any other occurrences in the trial. The occurrence terms capture specific contractual properties of 'eXcess of Loss' treaties as they apply to individual event occurrences only. The event losses net of occurrence terms are then accumulated into a single aggregate loss for the given trial. The occurrence terms are applied as $l_T = min(max(l_T - \mathcal{T}_{OccR}), \mathcal{T}_{OccL})$.

Two aggregate terms, namely (i) Aggregate Retention, denoted as $\mathcal{T}_{AggR}$, which is the retention or deductible of the insured for an annual cumulative loss, and (ii) Aggregate Limit, denoted as $\mathcal{T}_{AggL}$, which is the limit or coverage the insurer will pay for annual cumulative losses in excess of the aggregate retention are applied. Aggregate terms are applied to the trial's aggregate loss for a layer. Unlike occurrence terms, aggregate terms are applied to the cumulative sum of occurrence losses within a trial and thus the result depends on the sequence of prior events in the trial. This behaviour captures contractual properties as they apply to multiple event occurrences. The aggregate loss net of the aggregate terms is referred to as the trial loss or the year loss. The aggregate terms are applied as $l_T = min(max(l_T - \mathcal{T}_{AggR}), \mathcal{T}_{AggL})$.

2.4 Output

The output of the algorithm for performing aggregate risk analysis with primary and secondary uncertainty is a loss value associated with each trial of the YET. A reinsurer can derive important portfolio risk metrics such as the Probable Maximum Loss (PML) and the Tail Value-at-Risk (TVaR) which are used for both internal risk management and reporting to regulators and rating agencies. Furthermore, these metrics flow into a final stage of the risk analytics pipeline, namely Enterprise Risk Management, where liability, asset, and other forms of risks are combined and correlated to generate an enterprise wide view of risk.

Additional functions can be used to generate reports that will aid actuaries and decision makers. For example, reports presenting Return Period Losses (RPL) by Line of Business (LOB), Class of Business (COB) or Type of Participation (TOP). Further, the output of the analysis can be used for estimating Region/Peril losses and for performing Multi-Marginal Analysis and Stochastic Exceedance Probability (STEP) Analysis.

3. APPLYING SECONDARY UNCERTAINTY

The methodology to compute secondary uncertainty heavily draws on industry-wide practices. The inputs required for the secondary uncertainty method and the sequence of

steps for applying uncertainty to estimate a loss are considered in this section.

3.1 Inputs

Six inputs are required for computing secondary uncertainty which are obtained from the Year Event Table (YET) and the 'eXtended ELT' (XELT). The first input is $z_{(Prog,E)} = P_{(Prog,E)} \in U(0,1)$ referred to as the Program-and-Event-Occurrence-Specific random number. Each Event occurrence across different Programs have different random numbers. The second input is $z_{(E)} = P_{(E)} \in U(0,1)$ referred to as the Event-Occurrence-Specific random number. Each Event occurrence across different Programs have the same random number. The third input is μ_l referred to as the mean loss. The fourth input is σ_I referred to as the independent standard deviation of loss and represents the variance within the event-loss distribution. The fifth input is σ_C referred to as the correlated standard deviation of loss and represents the error of the event-occurrence dependencies. The sixth input is max_l referred to as the maximum expected loss.

3.2 Combining standard deviation

Given the above inputs, the independent and correlated standard deviations need to be combined to reduce the error in estimating the loss value associated with an event. This is done in a sequence of five steps. In the first step, the raw standard deviation is produced as $\sigma = \sigma_I + \sigma_C$.

In the second step, the probabilities of occurrences, $z_{(Prog,E)}$ and $z_{(E)}$ are transformed from uniform distribution to normal distribution using

$$f(x;\mu,\sigma^2) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \mathrm{d}x$$

This is applied to the probabilities of event occurrences as

$$\begin{aligned} v_{(Prog,E)} &= f(z_{(Prog,E)};0,1) \in N(0,1) \\ v_{(E)} &= f(z_{(E)};0,1) \in N(0,1) \end{aligned}$$

In the third step, the linear combination of the transformed probabilities of event occurrences and the standard deviations is computed as

$$LC = v_{(Prog,E)}\left(\frac{\sigma_I}{\sigma}\right) + v_{(E)}\left(\frac{\sigma_C}{\sigma}\right)$$

In the fourth step, the normal random variable is computed as

$$v = \frac{LC}{\sqrt{\left(\frac{\sigma_I}{\sigma}\right)^2 + \left(\frac{\sigma_C}{\sigma}\right)^2}}$$

In the fifth step, the normal random variable is transformed from normal distribution to uniform distribution as

$$z = \Phi(v) = F_{Norm}(v) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{v} e^{\frac{-t^2}{2}} \mathrm{d}t$$

The model used above for combining the independent and correlated standard deviations represents two extreme cases. The first case in which $\sigma_I = 0$ and the second case in which $\sigma_C = 0$. The model also ensures that the final random number, z, is based on both the independent and correlated standard deviations.

3.3 Estimating Losses

The loss is estimated using the Beta distribution since fitting such a distribution allows the representation of risks quite accurately. The Beta distribution is a two parameter distribution, with an upper bound for the standard deviation. The standard deviation, mean, alpha and beta are defined as

$$\begin{aligned} \sigma_\beta &= \frac{\sigma}{max_l} \\ \mu_\beta &= \frac{\mu_l}{max_l} \\ \alpha &= \mu_\beta\left(\left(\frac{\sigma_{\beta_{max}}}{\sigma_\beta}\right)^2 - 1\right) \\ \beta &= (1-\mu_\beta)\left(\left(\frac{\sigma_{\beta_{max}}}{\sigma_\beta}\right)^2 - 1\right) \end{aligned}$$

An upper bound is set to limit the standard deviation using $\sigma_{\beta_{max}} = \sqrt{\mu_\beta(1-\mu_\beta)}$, if $\sigma_\beta > \sigma_{\beta_{max}}$, then $\sigma_\beta = \sigma_{\beta_{max}}$. In the algorithm reported in this paper, for numerical purpose a value very close to $\sigma_{\beta_{max}}$ is chosen.

To obtain the loss after applying secondary uncertainty beta distribution functions are used as follows

$$\begin{aligned} Loss &= max_l * InvCDF_{beta}(z;\alpha,\beta) \\ InvCDF_{beta}(z;\alpha,\beta) &= \left(\frac{B(z;\alpha,\beta)}{B(\alpha,\beta)}\right)^{-1}, \text{where} \\ B(z;\alpha,\beta) &= \int_0^z t^{\alpha-1}(1-t)^{\beta-1}\mathrm{d}t \end{aligned}$$

4. IMPLEMENTATION

In this section, the hardware platforms used for the experimental studies are firstly considered, followed by the implementation of the data structures required for aggregate risk analysis with uncertainty and the implementation of the methods for computing secondary uncertainty. Optimizations incorporated in the implementations are further considered.

Two hardware platforms are used for implementing a sequential and parallel aggregate risk analysis algorithm. Firstly, a multi-core CPU is employed whose specifications are a 3.40 GHz quad-core Intel(R) Core (TM) i7-2600 processor with 16.0 GB of RAM. The processor has 256 KB L2 cache per core, 8MB L3 cache and maximum memory bandwidth of 21 GB/sec. The processor supports hyperthreading on the physical cores making eight virtual cores available. The experiments consider virtual cores as hyperthreading is beneficial for data intensive applications. Both sequential and parallel versions of the aggregate risk analysis algorithm were implemented on this platform. The sequential version was implemented in C++, while the parallel version was implemented in C++ and OpenMP. Both versions were compiled using the GNU Compiler Collection g++ 4.7 using '`-O3`' and '`-fopenmp`' when OpenMP is used.

Secondly, an NVIDIA Tesla C2075 GPU, consisting of 448 processor cores (organized as 14 streaming multi-processors each with 32 symmetric multi-processors), each with a frequency of 1.15 GHz, a global memory of 5.375 GB and a memory bandwidth of 144 GB/sec was employed in the GPU implementations of the aggregate risk analysis algorithm. The peak double precision floating point performance is 515 Gflops whereas the peak single precision floating point performance is 1.03 Tflops. The implementation of the algorithm is compiled using the NVIDIA CUDA Compiler (nvcc), version 5.0[1].

[1] `https://developer.nvidia.com/cuda-toolkit`

The following implementations for aggregate risk analysis with uncertainty are considered in this paper: (i) a sequential implementation on the CPU, (ii) a parallel implementation on the multi-cores of the CPU, and (iii) a parallel implementation on the many-cores of the GPU. Four libraries are used for applying secondary uncertainty on the many-core GPU and four additional libraries on the multi-core CPU.

4.1 Implementing Data Structures for the Algorithm

In aggregate risk analysis, the losses of events in a trial need to be determined by looking up losses in the XELT. The key design question is whether the data structure containing the event-loss pairs of all trials need to be a sparse matrix in the form of a direct access table or a compact representation. While fast lookups can be obtained in the sparse matrix representation, this performance is achieved at the cost of high memory usage. Consider a YET with 1,000,000 events and one Layer with 16 XELTs, and each XELT consisting of 20,000 events with non-zero losses. The representation using a direct access table would require memory to hold $16,000,000$ event-loss pairs (without considering the data required for secondary uncertainty calculations). While such a large data structure is held in memory, 15,700,000 events represent zero loss value.

Though the sparse representation requires large amount of memory it is chosen over any compact representation for the following reason. A search operation is required to find an event-loss pair even in a compact representation. If sequential search is adopted, then $O(n)$ memory accesses are required to find an event-loss pair. If sorting is performed in a pre-processing phase to facilitate a binary search, then $O(log(n))$ memory accesses are required to find an event-loss pair. If a constant-time space-efficient hashing scheme, such as cuckoo hashing [7] is adopted then an event-loss pair can be accessed with a constant number of memory accesses. However, this can be only be achieved at the expense of a complex implementation and overheads depreciating runtime performance. Further, such an implementation on the GPU with a complex memory hierarchy is cumbersome. Although large memory space is required for a direct access table, looking up event-loss pairs can be achieved with fewer memory accesses compared to the memory accesses in a compact representation.

Two data structure implementations of 16 XELTs were considered. In the first implementation, each XELT is considered as an independent table; therefore, in a read cycle, each thread independently looks up its events from the XELTs. All threads within a block access the same XELT. In the second implementation, all the 16 XELTs are combined into a single table. Consequently, the threads then use the shared memory to load entire rows of the combined XELTs at a time. The second implementation performs poorly compared to the first implementation. This is because of the memory overheads for the threads to collectively load rows from the combined XELT.

In the implementation on the multi-core CPU platform the entire data required for the algorithm is processed in memory. The GPU implementation of the algorithm uses the GPU's global memory to store all data structures. The parallel implementation on the GPU requires high memory transactions. This is surmounted by utilising shared memory over global memory.

4.2 Implementing Methods to Compute Secondary Uncertainty

Three statistical functions are required in the method for applying secondary uncertainty. They are (i) the Cumulative Distribution Function (CDF) of Normal distribution, (ii) the Quantile of the Normal distribution, and (iii) the Quantile of the Beta distribution. The Quantile of the Beta distribution is a numerically intensive function since it is an iterative method which converges to the solution within a certain error.

Seven different libraries are used for implementing the secondary uncertainty methodology on the multi-core CPU. The first is the Boost statistical library offered by the Boost C++ libraries[2]. The statistical functions are available inside the namespace `boost::math`. In order to use the distributions the header `<boost/math/distributions.hpp>` needs to be included. For example, `boost::math::normal_distribution<> NormDist (0.0L, 1.0L)` will create a Standard Normal distribution with mean equal to 0 and standard deviation equal to 1. The Quantile function of the Normal distribution can be obtained by as `quantile(NormDist, double value)`. The CDF of the Normal distribution is obtained by `cdf (NormDist, double value)`. Similarly, an Assymetrical Beta distribution with alpha and beta values can be created using `boost::math::beta_distribution<> BetaDist (double alpha, double beta)` and the Quantile can be obtained from `quantile(BetaDist, double cdf)`[3].

The second is the IMSL C/C++ Numerical Libraries offered by Rogue Wave Software[4]. The mathematical functions are obtained from the `imsl.h` header file and the statistical functions are obtained from the `imsls.h` header file [8, 9]. The CDF for the Normal distribution with mean equal to 0 and standard deviation equal to 1 is obtained from `imsl_f_normal_cdf (double value)` and the Quantile is obtained from `imsl_f_normal_inverse_cdf (double cdf)`. The Quantile for the Beta distribution with alpha and beta values are obtained as `imsl_f_beta_inverse_cdf (double cdf, double alpha, double beta)`.

The third is PROB which is a C++ library that handles the Probability Density Functions for various discrete and continuous distributions[5]. In order to use the distributions the header `prob.hpp>` needs to be included. The CDF for the Normal distribution with mean equal to 0 and standard deviation equal to 1 is obtained from `normal_01_cdf (double value)` or `normal_cdf (double x, double a, double b)` (where $a = 0$ and $b = 1$) and the Quantile from `normal_01_cdf_inv (double cdf)` or `normal_cdf_inv (double cdf, double a, double b)` (where $a = 0$ and $b = 1$). The Quantile for the Beta distribution with alpha and beta values are obtained as `beta_cdf_inv (double cdf, double alpha, double beta)` [10].

The fourth is DCDFLIB which is a C library adapted from Fortran for evaluating CDF and inverse CDF of discrete and continuous probability distributions[6]. In order to use the distributions the header `dcdflib.c` needs to be included.

[2] `http://www.boost.org/`

[3] `http://www.boost.org/doc/libs/1\_35\_0/libs/math/doc/sf\_and\_dist/html/math\_toolkit/dist/dist\_ref/dists/beta\_dist.html`

[4] `http://www.roguewave.com/products/imsl-numerical-libraries/c-library.aspx`

[5] `http://people.sc.fsu.edu/~jburkardt/cpp\_src/prob/prob.html`

[6] `http://www.netlib.org/random/`

The CDF and the Quantile for the Normal distribution with mean stored in `mean` and standard deviation stored in `sd` can be obtained from `cdfnor (int *which, double *p, double *q, double *value, double *mean, double *sd, int *status, double *bound)` [11]. '`which`' is set to 1 to obtain the CDF value `p` and `q = 1.0 - p`. '`which`' is set to 2 to obtain the Quantile in `value`. `status` and `bound` are variables to report the status of the computation. The Quantile of the Beta distribution `x` and `y = 1.0 - x` for `alpha` and `beta` can be obtained from `cdfbet (int *which, double *p, double *q, double *x, double *y, double *alpha, double *beta, int *status, double *bound)` when `which` is set to 2, `p` is the CDF and `q = 1.0 - p` [12].

The fifth library is ASA310 or the Applied Statistics Algorithm 310[7] which is a C++ library for evaluating the CDF of the Noncentral Beta distribution [13]. The include file is `asa310.hpp`. The iterative algorithm for achieving convergence of the solution to compute the Quantile calls the function for computing the tail of the Noncentral Beta distribution `betanc (float value, float alpha, float beta, float lambda, int *ifault)`, where `lambda`, the noncentrality parameter is set to 0 for the standard Beta distribution and `ifault` is an error flag.

The sixth library is ASA226 or the Applied Statistics Algorithm 226[7] which is a C++ library similar to ASA310 [14, 15]. The include file is `asa226.hpp`. The iterative algorithm for achieving convergence of the solution to compute the Quantile is used to call the function for computing the tail of the Noncentral Beta distribution.

The seventh library is BETA_NC another C++ library that can evaluate the CDF of the Noncentral Beta distribution [16]. The include file is `beta_nc.cpp`. The iterative algorithm that achieves convergence of the solution to compute the Quantile calls the `beta_noncentral_cdf (double alpha, double beta, double lambda, double value, double error_max)`, where `lambda`, the noncentrality parameter is set to 0 for the standard Beta distribution and `error_max` is is the error control in the computation.

For implementing the secondary uncertainty methodology on the GPU statistical functions provided by the CUDA Math API are employed by including the `math.h` header file[8]. The CDF of the Normal distribution `normcdf` and the Quantile of the Normal Distribution `normcdfinv` are fast methods and included in the implementation. The CUDA Math API currently does not support Beta distribution functions. Therefore, four libraries, namely the PROB, ASA310, ASA226 and BETA_NC are incorporated in the implementation for the many-core GPU. These libraries are ported for the GPU platform and all the functions in the libraries are implemented as `__device__` functions for the GPU.

4.3 Optimising the Implementation

The implementations were optimised for better performance in three ways. Firstly, by incorporating loop unrolling, which refers to the compiler replicating of blocks of code within 'for loops' to reduce the number of iterations performed by for loops. The `for` loops are unrolled using the `pragma` directive; the `for` loops in line nos. 1-5 of Algorithm 1 can be unrolled as each iteration is a mutually independent iteration.

[7] `http://lib.stat.cmu.edu/apstat/`

[8] `http://docs.nvidia.com/cuda/cuda-math-api/index.html`

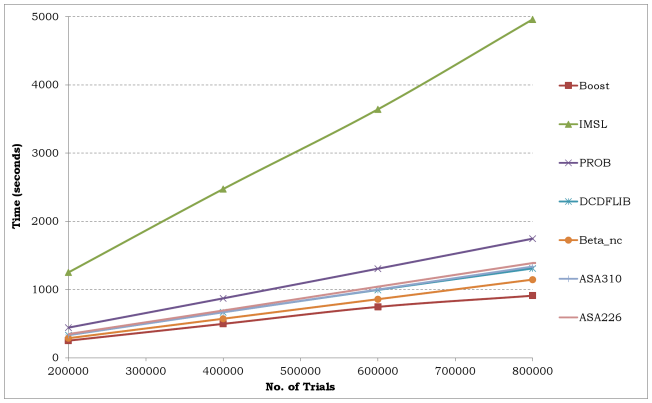

Figure 1: Applying secondary uncertainty using different statistical libraries for sequential implementation on CPU

Secondly, in the case of the GPU by migrating data from both shared and global memory to the kernel registry. The kernel registry has the lowest latency compared to all other memory.

Thirdly, by reducing the precision of variables used in the algorithm, whereby the double variables are changed to float variables. In the case of the GPU, read operations are faster using float variables as they are only half the size of a double variable. Furthermore, the performance of single precision operations tend to be approximately twice as fast as double precision operations. The CUDA Math API supports functions for floating point operations and the full acceleration of CUDA Math API can be achieved by using the compiler flag `-use_fast_math`.

5. EXPERIMENTAL RESULTS

In this section, the results obtained from the sequential implementation on the CPU, the parallel implementation on the multi-core CPU and the many-core GPU, and the summary of the experimental results are presented.

Figure 1 shows the graph plotted for the time taken for sequentially performing aggregate risk analysis using trials varying from 200,000 to 800,000 with each trial comprising 1,000 events on the CPU when secondary uncertainty is applied using the Boost, IMSL, PROB, DCDFLIB, BETA_NC, ASA310 and ASA226 libraries. The experiments are performed for one Layer and 16 XELTs. The Boost library provides the fastest functions for secondary uncertainty followed by the BETA_NC, DCDFLIB, ASA310 and ASA226 libraries. The PROB library is approximately 2 times slower and the IMSL numerical library is approximately 5 times slower than the Boost library.

Figure 2 shows the graph plotted for the time taken for applying secondary uncertainty for trials varying from 200,000 to 800,000 with each trial comprising 1,000 events on the CPU when Boost library is used. The results from the Boost library are chosen since it provides the fastest functions for applying secondary uncertainty. The experiments are performed for one Layer and 16 XELTs. In each case of trials shown in the graph the time for applying secondary uncertainty is nearly 2.5 times the time taken for aggregate risk analysis. The mathematical functions employed for applying secondary uncertainty are fast methods with the exception

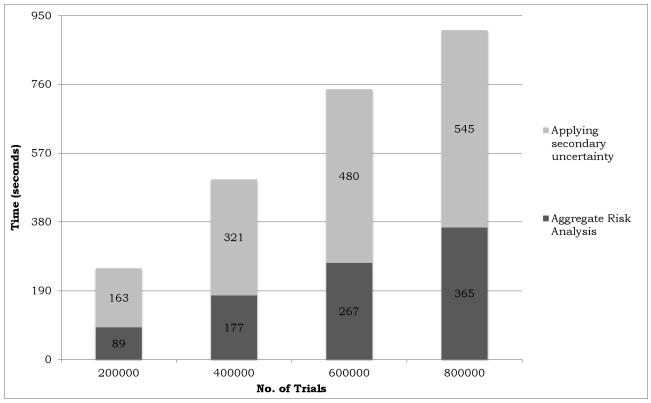

Figure 2: Time taken for aggregate risk analysis and applying secondary uncertainty for different trials in sequential implementation on CPU

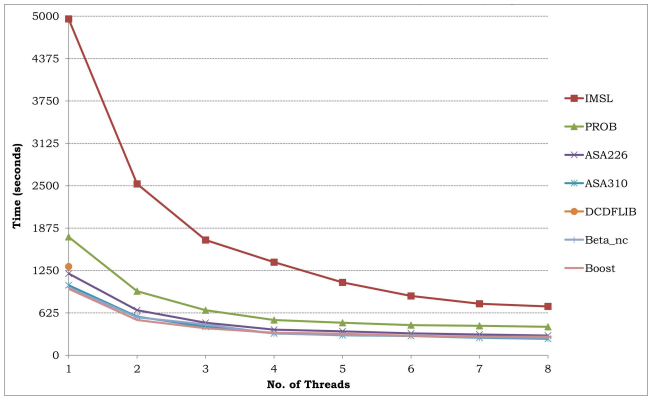

Figure 3: Time taken for aggregate risk analysis and applying secondary uncertainty for one thread on each virtual core of the CPU

of the inverse cumulative distribution function of the beta distribution which takes majority of the time.

The time taken both for performing aggregate risk analysis with only primary uncertainty and for applying secondary uncertainty with increasing number of trials should scale linearly and this is observed both in Figure 1 and Figure 2.

Figure 3 and Figure 4 show the graphs plotted for the time taken for performing parallel aggregate risk analysis and applying secondary uncertainty for 800,000 trials on the multi-core CPU using the Boost, IMSL, PROB, DCDFLIB, BETA_NC, ASA310 and ASA226 libraries.

In Figure 3, a single thread is run on each virtual core of the CPU and the number of cores are varied from 1 to 8 (i.e., up to two threads on each of the four physical cores). Each threads performs the aggregate risk analysis for a single trial and applies secondary uncertainty. Multiple threads are used by employing OpenMP directive `#pragma omp parallel` in the C++ source. With respect to the overall time the ASA310 library performs the best requiring 232 seconds for the analysis. For the IMSL library, a speedup of nearly 1.9x is achieved for two cores, a speedup of nearly 3.6x is obtained for four cores and a speedup of 6.9x is obtained for 8 cores. While the performance keeps diminishing for the IMSL library, the efficiency of all the other libraries used

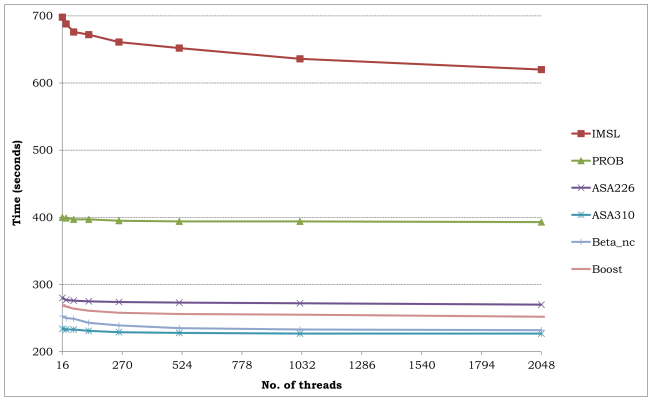

Figure 4: Time taken for aggregate risk analysis and applying secondary uncertainty for multiple threads on each virtual core of the CPU

in secondary uncertainty is significantly low. No more than 4x speedup is achieved on eight cores in the best case. The limiting factor is that the bandwidth to memory is not increased as the number of cores increase. The majority of the time taken in aggregate risk analysis is for performing random access reads into the data structure representing the XELT. The majority of the time in applying secondary uncertainty is consumed in the inverse cumulative distribution function of the beta distribution. The Boost library outperforms all the other libraries with respect to the overall time. The DCDFLIB library did not scale on multiple threads as the files are written as blocks of program with unconditional jumps using the `goto` statement.

In Figure 4, the performance on all eight virtual cores of the CPU is illustrated; multiple threads are run on each virtual core of the CPU. For example, when 16 threads are employed two threads run on each virtual core and when 2048 threads are employed 256 threads run on each virtual core of the CPU. A small drop is observed in the absolute time when many threads are executed on each core. When 256 threads run on a core, the overall runtime drops from 701 seconds (using two threads per core) to 620 seconds for the IMSL library and the runtime drops from 269 seconds (using two threads per core) to 252 seconds for the Boost library. When 2048 threads are employed, the ASA310 library performs better than the Boost library by 25 seconds.

Figure 5 shows the graph plotted for the time taken for applying secondary uncertainty using 2048 threads for trials varying from 200,000 to 800,000 on the eight virtual cores of the CPU when the Boost library is used. In each case of trials shown in the graph the time taken for applying secondary uncertainty increases with the number of trials. The time taken for applying secondary uncertainty on the multi-core is only $\frac{1}{6}^{th}$ the time taken in the sequential implementation.

Figure 6 shows the graph plotted for the time taken for aggregate risk analysis and for applying secondary uncertainty using the Boost library for 800,000 trials when the number of threads are varied from 1 to 2048 on the multi-core CPU. The lowest overall time is 252 seconds when 256 threads are employed per core of the CPU; 161 seconds are required for the aggregate risk analysis and 91 seconds for applying secondary uncertainty. However, the lowest time taken for applying secondary uncertainty is 75 seconds which

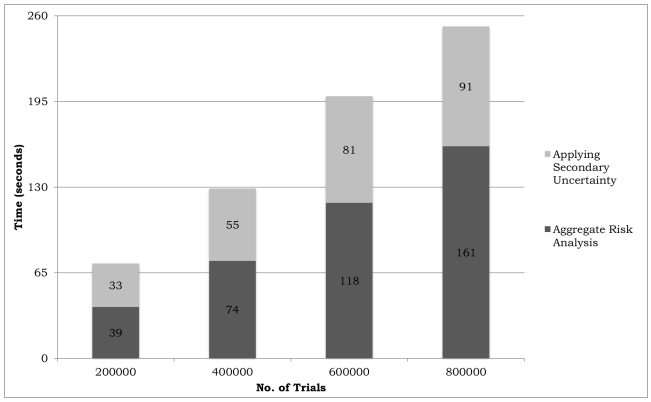

Figure 5: Time taken for aggregate risk analysis and applying secondary uncertainty for different trials in parallel implementation on multi-core CPU

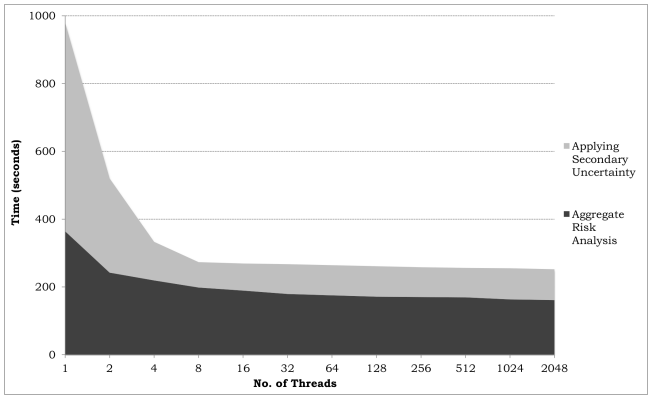

Figure 6: Time taken for aggregate risk analysis and applying secondary uncertainty using Boost library for 800,000 trials in parallel implementation on multi-core CPU

is achieved when one thread is used per core (8 threads on the CPU). While there is a decrease in the overall time taken as the number of threads increase, there is a gradual increase in the time taken for applying secondary uncertainty when more than 8 threads are employed on the CPU; 75 seconds when 8 threads are used, where as 91 seconds required when 2048 threads are used. This is due to the increasing overhead in swapping constants in and out of memory as the number of threads increase. The performance of Boost surpasses that of ASA only when one and two threads are used. Beyond two threads ASA310 has lower overall time.

Figure 7 shows the graph plotted for the optimal (best time) time taken for applying secondary analysis in aggregate risk analysis for 800,000 trials using different libraries. Optimality for overall time is achieved when 2048 threads are employed; in this graph the optimality for applying secondary uncertainty is considered. For the ASA310 library, the best time for applying secondary uncertainty is 66 seconds which is only $\frac{1}{8}^{th}$ the time taken in the sequential implementation.

Figure 8 and Figure 9 show the graphs plotted for the time taken for performing parallel aggregate risk analysis and applying secondary uncertainty for 800,000 trials on the

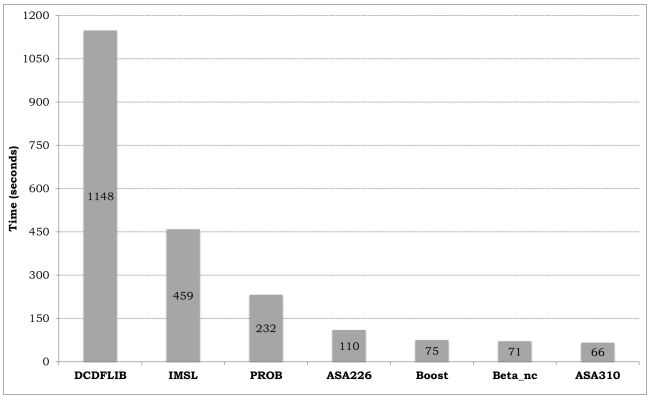

Figure 7: Optimal time taken for applying secondary uncertainty using different libraries for 800,000 trials in parallel implementation on multi-core CPU

many-core GPU using the PROB, ASA310, BETA_NC and ASA226 libraries. IMSL and Boost libraries are not available for GPUs. The DCDFLIB library was ported for the GPU but did not execute on the hardware.

CUDA provides abstraction over the streaming multi-processors of the GPU, which is often referred to as a CUDA block. The number of threads executed per CUDA block can be varied in aggregate risk analysis. For example, consider the execution of 800,000 Trials using 800,000 threads. If 256 threads are executed on one Streaming Multi-Processor (SMP), then 3125 CUDA blocks need to be executed on the 14 SMPs. Each SMP will have to execute 223 CUDA blocks. All threads executing on one SMP have a fixed size of shared and constant memory. Fewer the threads employed, then each thread will have a large size of shared and constant memory. But there is a trade-off when fewer threads are used since the latency for accessing the global memory of the GPU increases.

In Figure 8 the analysis is performed for 800,000 Trials on the GPU by varying the number of threads per block from 16 to 512 threads. An improvement in the performance is seen as the number of threads increase from 16 to 128 since the latency for accessing the global memory drops. Beyond 128 threads the performance starts to drop as the shared and constant memory available to each thread decreases. ASA226 library performs the best since the function used in computing secondary uncertainty has an optimal balance between the number of constants and the amount of computation required. This is vital when there is a trade-off between the size of shared and constant memory and latency in accessing global memory. The lowest time taken is 51 seconds when 128 threads per block are used.

Figure 9 shows the time taken to perform aggregate risk analysis for applying primary uncertainty and for applying secondary uncertainty using the ASA226 library on the GPU for 800,000 Trials. The time taken for performing aggregate risk analysis is nearly a constant. The time taken for applying secondary uncertainty first decreases from 16 to 128 threads per block and then increase beyond 128 threads per block. This is due to the trade-off between the size of the shared and constant memory and latency in accessing global memory. In the best case when 128 threads per block are employed the time taken for applying secondary uncertainty

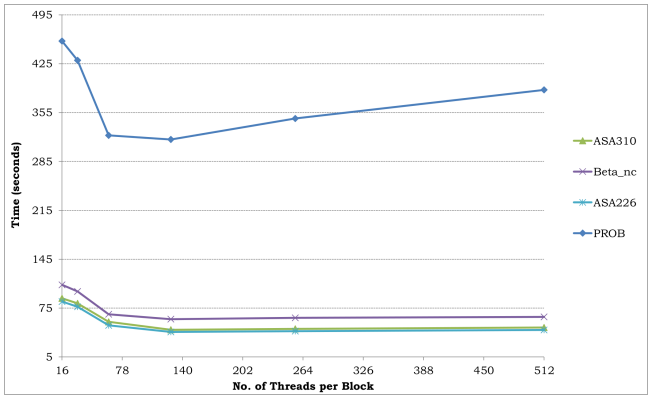

Figure 8: Time taken for aggregate risk analysis and applying secondary uncertainty using different threads per block on many-core GPU

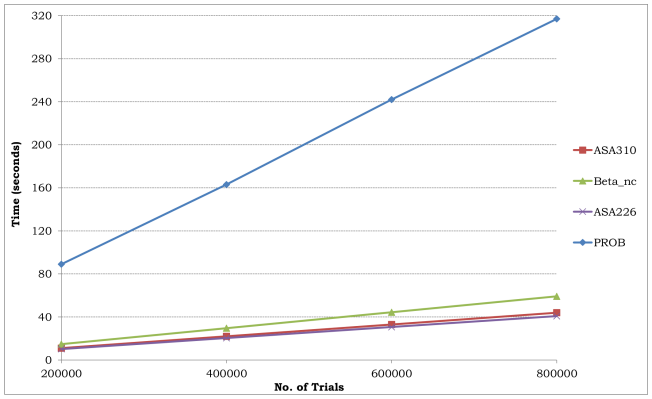

Figure 10: Time taken for aggregate risk analysis and applying secondary uncertainty for different trials in parallel implementation on many-core GPU

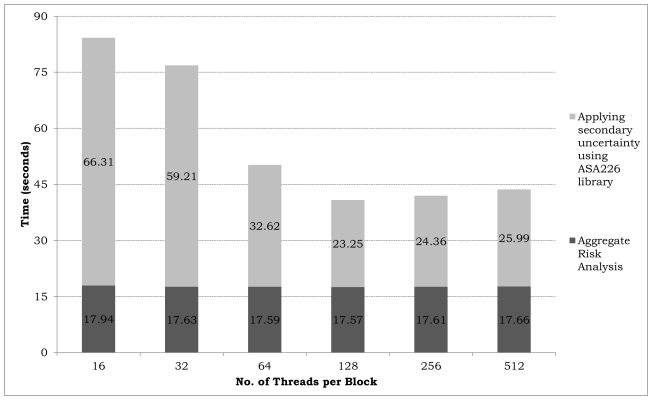

Figure 9: Time taken for aggregate risk analysis and for applying secondary uncertainty using different threads per block for ASA226 on many-core GPU

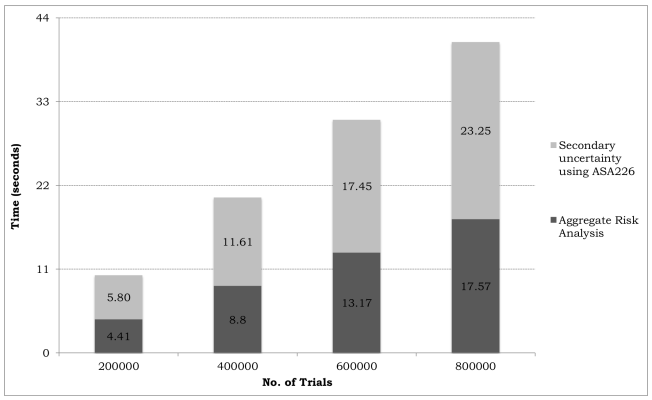

Figure 11: Time taken for aggregate risk analysis and for applying secondary uncertainty for different trials using ASA226 on many-core GPU

is nearly twice the time taken for performing aggregate risk analysis. The best time for applying secondary uncertainty using ASA226 on the GPU is only half the best time taken by ASA310 for applying secondary uncertainty using multiple threads on the CPU.

Figure 10 illustrates the performance of the ASA226, BETA_NC, ASA310 and PROB libraries on the GPU for different trials varying from 200,000 to 800,000. In each case of trials the time taken for applying secondary uncertainty increases with the number of trials. The ASA226 outperforms the BETA_NC, ASA310 and PROB libraries. Figure 11 shows the time taken to perform aggregate risk analysis for applying primary uncertainty and for applying secondary uncertainty using the ASA226 library on the GPU for trials varying from 200,000 to 800,000. Both times scale linearly.

5.1 Discussion

Figure 12 is a graph that summarises the key results from the experimental study. The set of three bars represents the time taken for (i) fetching Events from memory and for look up of Loss Sets in memory, (ii) applying Financial Terms and performing other computations in aggregate risk analysis, and (iii) applying secondary uncertainty on the sequential implementation on the CPU and the parallel implementations on both the multi-core CPU and many-core GPU when 800,00 Trials, with each Trial consisting 1,000 Events, and 16 XELTs are employed. In each case, parameters specific to the implementation, such as the number of threads, were set to the best value identified during experimentation. In the parallel implementations for the basic aggregate analysis, a speedup of 2.3x is achieved on the CPU and a speedup of 20x is achieved on the GPU when compared against the sequential implementation. A speedup of 24x is achieved in the overall time for the implementation on the GPU in contrast to the sequential implementation on the CPU. For applying secondary uncertainty, multiple threading on the eight virtual cores of the CPU is nearly five times faster than the sequential implementation and three times slower than the GPU. For the numeric computations on the GPU an accelaration of approximately 26x is achieved over the sequential implementation. Limited memory bandwidth is a bottleneck in the CPU resulting in approximately 27% and 53% of the time being spent for fetching Events and for look up of Loss Sets in memory for the sequential and parallel implementation on the CPU respectively. While the time for fetching Events and for look up of Loss Sets in memory have been significantly lowered on the GPU 39% of the total time is still used to this end.

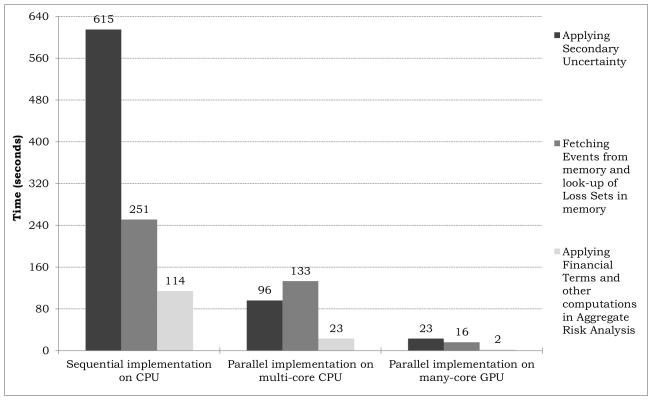

Figure 12: Time taken for fetching Events from memory and for look up of Loss Sets in memory, applying Financial Terms and performing other computations in aggregate risk analysis, and applying secondary uncertainty on sequential implementation on CPU and parallel implementations on CPU and GPU

In the sequential implementation on the CPU, in the parallel implementation on the multi-core CPU and in the parallel implementation on the many-core GPU approximately 62%, 38% and 56%, respectively, of the total time for aggregate risk analysis is required for applying secondary uncertainty. The majority of this time is required by the computations of the Inverse Beta Cumulative Distribution. This calls for not only the development of fast methods to apply secondary uncertainty in risk analytics, but also the development of fast methods for the underlying statistical functions. Fast methods have been implemented for computing the inverse CDF of the symmetrical beta distribution [17] which considers one shape parameter, but there are minimal implementations of fast assymetrical beta distribution that takes two shape parameters.

6. CONCLUSIONS

The research reported in this paper was motivated towards experimentally verifying whether GPUs can accelerate aggregate risk analysis with both primary and secondary uncertainty. To this end an algorithm for the analysis of portfolios of risk and a methodology for applying secondary uncertainty was proposed and implemented. A sequential and a parallel implementation on the CPU and a parallel implementation on the GPU were presented. Seven statistical libraries, namely Boost, IMSL, DCDFLIB, PROB, ASA310, ASA226 and BETA_NC were investigated for implementing the computations of secondary uncertainty. Numerous challenges in handling large data in limited memory of the GPU were surmounted; the resultant was a speedup of 24x which was achieved for the parallel analysis on the GPU over its sequential counterpart on the CPU.

In this research, the GPU performed well for the numerical computations of secondary uncertainty; the GPU was five times faster than the multiple threaded analysis on the multi-core CPU for applying secondary uncertainty. The CPU could have performed well had it not been for its limited memory bandwidth and the GPU could have performed better had it not been for its limited memory availability.

7. REFERENCES

[1] G. Woo, "Natural Catastrophe Probable Maximum Loss," British Actuarial Journal, Vol. 8, 2002.

[2] P. Glasserman, P. Heidelberger, and P. Shahabuddin, "Portfolio Value-at-Risk with Heavy-Tailed Risk Factors," Mathematical Finance, Vol. 12, No. 3, 2002, pp. 239-269.

[3] A. K. Bahl, O. Baltzer, A. Rau-Chaplin, and B. Varghese, "Parallel Simulations for Analysing Portfolios of Catastrophic Event Risk," in Workshop Proceedings of the International Conference of High Performance Computing, Networking, Storage and Analysis (SC12), 2012.

[4] G. G. Meyers, F. L. Klinker and D. A. Lalonde, "The Aggregation and Correlation of Reinsurance Exposure," Casualty Actuarial Society Forum, Spring 2003, pp. 69-152.

[5] W. Dong, H. Shah and F. Wong, "A Rational Approach to Pricing of Catastrophe Insurance," Journal of Risk and Uncertainty, Vol. 12, 1996, pp. 201-218.

[6] R. M. Berens, "Reinsurance Contracts with a Multi-Year Aggregate Limit," Casualty Actuarial Society Forum, Spring 1997, pp. 289-308.

[7] R. Pagh and F. F. Rodler, "Cuckoo hashing," Journal of Algorithms, Vol. 51, 2004.

[8] IMSL C Numerical Library, User Guide, Volume 1 of 2: C Math Library, Version 8.0, November 2011, Rogue Wave Software, USA.

[9] IMSL C Numerical Library, User Guide, Volume 2 of 2: C Stat Library, Version 8.0, November 2011, Rogue Wave Software, USA.

[10] G. W. Cran, K. J. Martin and G. E. Thomas, "Remark AS R19 and Algorithm AS 109: A Remark on Algorithms AS 63: The Incomplete Beta Integral and AS 64: Inverse of the Incomplete Beta Integeral," Applied Statistics, Vol. 26, No. 1, 1977, pp. 111-114.

[11] W. Cody, "Algorithm 715: SPECFUN - A Portable FORTRAN Package of Special Function Routines and Test Drivers," ACM Transactions on Mathematical Software, Vol. 19, 1993, pp. 22-32.

[12] A. R. DiDinato and A. H. Morris, "Algorithm 708: Significant Digit Computation of the Incomplete Beta Function Ratios," ACM Transactions of Mathematical Software, Vol. 18, 1993, pp. 360-373.

[13] R. Chattamvelli and R. Shanmugam, "Algorithm AS 310: Computing the Non-central Beta Distribution Function," Applied Statistics, Vol. 46, No. 1, 1997, pp. 146-156.

[14] R. Lenth, "Algorithm AS 226: Computing Noncentral Beta Probabilities," Applied Statistics, Vol. 36, No. 2, 1987, pp. 241-244.

[15] H. Frick, "Algorithm AS R84: A Remark on Algorithm AS 226: Computing Noncentral Beta Probabilities," Applied Statistics, Vol. 39, No. 2, 1990, pp. 311-312.

[16] H. Posten, "An Effective Algorithm for the Noncentral Beta Distribution Function," The American Statistician, Vol. 47, No. 2, 1993, pp. 129-131.

[17] P. L'Ecuyer and R. Simard, "Inverting the Symmetrical Beta Distribution," ACM Transactions on Mathematical Software, Vol. 32, No. 4, 2006, pp. 509-520.