Multi-GPU Computing for Achieving Speedup in Real-time Aggregate Risk Analysis

A. K. Bahl
Center for Security, Theory and Algorithmic Research
International Institute of Information Technology
Hyderabad, India
aman.kumar@research.iiit.ac.in

O. Baltzer, A. Rau-Chaplin, B. Varghese and A. Whiteway
Faculty of Computer Science
Dalhousie University
Halifax, Canada
{*obaltzer, arc, varghese*}*@cs.dal.ca, aaron.whiteway@dal.ca*

***Abstract*—Stochastic simulation techniques employed for portfolio risk analysis, often referred to as Aggregate Risk Analysis, can benefit from exploiting state-of-the-art high-performance computing platforms. In this paper, we propose parallel methods to speedup aggregate risk analysis for supporting real-time pricing. To achieve this an algorithm for analysing aggregate risk is proposed and implemented in C and OpenMP for multi-core CPUs and in C and CUDA for many-core GPUs. An evaluation of the performance of the algorithm indicates that GPUs offer a feasible alternative solution over traditional high-performance computing systems. An aggregate simulation on a multi-GPU of 1 million trials with 1000 catastrophic events per trial on a typical exposure set and contract structure is performed in less than 5 seconds. The key result is that the multi-GPU implementation of the algorithm presented in this paper is approximately 77x times faster than the traditional counterpart and can be used in real-time pricing scenarios.**

***Keywords*-GPU computing; high-performance computing; aggregate risk analysis; catastrophe event risk; real-time pricing**

I. Introduction

Large-scale simulations in the risk analytics [1], [2] domain which are both data and computationally intensive can benefit from exploiting advances in high-performance computing. While a large number of financial engineering applications, for example [3], [4], [5] are benefitting from the advancement of high-performance computing, there are relatively fewer insurance and reinsurance applications exploiting parallelism. In this paper, we explore parallel methods and their implementations for aggregate risk analysis [6], [7], [8], [9] required in portfolio risk management and real-time pricing.

Aggregate risk analysis is a form of Monte Carlo simulation performed on a portfolio of risks that a reinsurer holds rather than on individual risks. A portfolio may comprise thousands or even thousands of contracts that cover risks associated with catastrophic events such as earthquakes, hurricanes and floods. Generally, contracts have an 'eXcess of Loss' (XL) [10], [11], [12] structure and may provide coverage for (a) single event occurrences up to a specified limit with an optional retention by the insured, or (b) multiple event occurrences up to a specified aggregate limit and with an optional retention by the insured, or (c) a combination of both features. Each trial in the aggregate risk analysis simulation represents a view of the occurrence of catastrophic events and the order in which they occur within a predetermined period, (i.e., a contractual year) and how they will interact with complex treaty terms to produce an aggregated loss. A pre-simulated Year Event Table (YET) containing between several thousand and millions of alternative views of a single contractual year is used as input; this provides actuaries and decision makers with a consistent lens through which to view results. The output of aggregate analysis is a Year Loss Table (YLT). From a YLT, a reinsurer can derive important portfolio risk metrics such as the Probable Maximum Loss (PML) [13], [14], [15] and the Tail Value at Risk (TVAR) [16], [17] which are used for both internal risk management and reporting to regulators and rating agencies.

In this paper, firstly, a sequential aggregate risk analysis algorithm is implemented in C on a CPU, followed by parallel implementations using C and OpenMP on multi-core CPU and using C and CUDA on many-core GPU platforms. The algorithms ingest large data for aggregating risks, and therefore, challenges such as efficiently organising input data in limited memory, and defining the granularity at which parallelism can be applied to the problem for achieving speedup are considered. Optimisations, such as chunking, loop unrolling, reducing the precision of variables used and the usage of kernel registries over shared and global memories of the GPU are performed to improve the speedup achieved on the GPU. A maximum speedup of 77x is achieved for the parallel implementation on the multi-GPU. The results indicate the feasibility of employing aggregate risk analysis on multi-GPUs for real-time pricing scenarios. The implementations presented in this paper are cost effective high-performance computing solutions over traditional clusters and full-blown supercomputers. The GPU implementations takes full advantage of the high levels of parallelism, some advantage of fast shared memory access, and relatively little advantage of the fast numerical performance all offered by the machine architecture of GPUs.

The remainder of this paper is organised as follows. Section II presents the aggregate risk analysis algorithm, its inputs and outputs. Section III considers the implementations

of the algorithm on a multi-core CPU and a many-core GPU. Section IV highlights the results obtained from an analysis of the performance of the algorithm. Section V compares and contrasts the algorithms and the results obtained from the experiments. Section VI concludes the paper by considering future work.

II. Aggregate Risk Analysis

There are three inputs to the procedure that aggregates risk. The first input is a database of pre-simulated occurrences of events from a catalog of stochastic events, which is referred to as the Year Event Table (YET). A possible sequence of catastrophe event occurrences for any given year comprises a record in the YET is a 'trial' (T_i). The sequence of events is defined by a set of tuples containing the ID of an event and the time-stamp of its occurrence in a trial $T_i = \{(E_{i,1}, t_{i,1}), \ldots, (E_{i,k}, t_{i,k})\}$ which is ordered by ascending time-stamp values. A typical YET may comprise thousands to millions of trials, and each trial may have approximately between 800 to 1500 'event time-stamp' pairs, based on a global event catalog covering multiple perils. The YET is represented as $YET = \{T_i = \{(E_{i,1}, t_{i,1}), \ldots, (E_{i,k}, t_{i,k})\}\}$, where $i = 1, 2, \ldots$ and $k = 1, 2, \ldots, 800 - 1500$.

The second input is collections of specific events and their corresponding losses with respect to an exposure set referred to as the Event Loss Tables (ELT). An event may be part of multiple ELTs and associated with a different loss in each ELT. For example, one ELT may contain losses derived from one exposure set while another ELT may contain the same events but different losses derived from a different exposure set. Each ELT is characterised by its own metadata including information about currency exchange rates and terms that are applied at the level of each individual event loss. Each record in an ELT is denoted as event loss $EL_i = \{E_i, l_i\}$, and the financial terms associated with the ELT are represented as a tuple $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2, \ldots)$. A typical aggregate analysis may comprise 10,000 ELTs, each containing 10,000-30,000 event losses with exceptions even up to 2,000,000 event losses. The ELTs are represented as $ELT = \left\{ \begin{array}{c} EL_i = \{E_i, l_i\}, \\ \mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2, \ldots) \end{array} \right\}$, with $i = 1, 2, \ldots, 10,000 - 30,000$.

The third input are the Layers, denoted as L, which cover a collection of ELTs under a set of layer terms. A single layer L_i is composed of two attributes. Firstly, the set of ELTs $\mathcal{E} = \{ELT_1, ELT_2, \ldots, ELT_j\}$, 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 ELTs. $L = \left\{ \begin{array}{c} \mathcal{E} = \{ELT_1, ELT_2, \ldots, ELT_j\}, \\ \mathcal{T} = (\mathcal{T}_{OccR}, \mathcal{T}_{OccL}, \mathcal{T}_{AggR}, \mathcal{T}_{AggL}) \end{array} \right\}$, with $j = 1, 2, \ldots, 3 - 30$ is a representation of the Layer.

The principal algorithm (line no. 1-32 shown in Algorithm 1) for aggregate analysis consists of 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, ELT and L, are loaded into memory.

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.

Algorithm 1 Aggregate Risk Analysis

```
1:  procedure AggregateRiskAnalysis(YET, ELT, L)
2:      for all a ∈ L do
3:          for all b ∈ YET do
4:              for all c ∈ (EL ∈ a) do
5:                  for all d ∈ (Et ∈ b) do
6:                      x_d ⇐ E ∈ d in El ∈ f, where f ∈ ELT and (EL ∈ f) = c
7:                  end for
8:                  for all d ∈ (Et ∈ b) do
9:                      l_{x_d} ⇐ ApplyFinancialTerms(I)
10:                 end for
11:                 for all d ∈ (Et ∈ b) do
12:                     lo_{x_d} ⇐ lo_{x_d} + l_{x_d}
13:                 end for
14:             end for
15:             for all d ∈ (Et ∈ b) do
16:                 lo_{x_d} ⇐ min(max(lo_{x_d} − T_{OccR}, 0), T_{OccL})
17:             end for
18:             for all d ∈ (Et ∈ b) do
19:                 lo_{x_d} ⇐ Σ_{i=1}^{d} lo_{x_i}
20:             end for
21:             for all d ∈ (Et ∈ b) do
22:                 lo_{x_d} ⇐ min(max(lo_{x_d} − T_{AggR}, 0), T_{AggL})
23:             end for
24:             for all d ∈ (Et ∈ b) do
25:                 lo_{x_d} ⇐ lo_{x_d} − lo_{x_{d−1}}
26:             end for
27:             for all d ∈ (Et ∈ b) do
28:                 lr ⇐ lr + lo_{x_d}
29:             end for
30:         end for
31:     end for
32: end procedure
```

Line no. 4-7 shows the first step in which each event of a trial its corresponding event loss in the set of ELTs associated with the Layer is determined.

Line no. 8-10 shows the second step in which a set of financial terms is applied to each event loss pair extracted from an ELT. In other words, contractual financial terms to

the benefit of the layer are applied in this step. For this the losses for a specific event's net of financial terms $\mathcal{I}$ are accumulated across all ELTs into a single event loss shown in line no. 11-13.

Line no. 15-20 shows the third step in which the event loss for each event occurrence in the trial, combined across all ELTs associated with the layer, is subject to occurrence terms (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. 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.

Line no. 21-29 shows the fourth step in which the aggregate terms (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. 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 and stored in a Year Loss Table (YLT) as the result of the aggregate analysis.

The algorithm will provide an aggregate loss value for each trial denoted as lr in line no. 28. Financial functions or filters are then applied on the aggregate loss values.

III. Experimental Studies

The experimental studies investigate the sequential and parallel implementation of the aggregate risk analysis on three hardware platforms. 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 had 256 KB L2 cache per core, 8MB L3 cache and maximum memory bandwidth of 21 GB/sec. 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++ 4.4 using the "-O3" and "-m64" flags.

Secondly, a NVIDIA Tesla C2075 GPU, consisting of 448 processor cores (organised 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.

Thirdly, a multiple GPU platform comprising 4 NVIDIA Tesla M2090 GPUs, and each GPU consists 512 processor cores (organised as 14 streaming multi-processors each with 32 symmetric multi-processors) and 5.375 GB of global memory with a memory bandwidth of 177 GB/sec is employed for implementing the fastest aggregate risk analysis algorithm reported in this paper. The peak double precision floating point performance is 665 Gflops whereas the peak single precision floating point performance is 1.33 Tflops. CUDA is employed for the basic GPU implementation of the aggregate risk analysis algorithm and the optimised implementations.

Five variations of the algorithm are implemented, they are: (i) a classic sequential implementation, (ii) a parallel implementation for multi-cores CPUs, (iii) a parallel GPU implementation, (iv) an optimised parallel implementation on the GPU, and (v) an optimised parallel implementation on a multi-GPU.

In all implementations a single thread is employed per trial, T_{id}. The key design decision from a performance perspective is the selection of a data structure for representing Event Loss Tables (ELTs). ELTs are essentially dictionaries consisting of key-value pairs and the fundamental requirement is to support fast random key lookup. The ELTs corresponding to a layer were implemented as direct access tables. A direct access table is a highly sparse representation of a ELT, one that provides very fast lookup performance at the cost of high memory usage. For example, consider an event catalogue of 2 million events and a ELT consisting of 20K events for which non-zero losses were obtained. To represent the ELT using a direct access table, an array of 2 million loss are generated in memory of which 20K are non-zero loss values and the remaining 1.98 million events are zero. So if a layer has 15 ELTs, then 15×2 million $= 30$ million event-loss pairs are generated in memory.

A direct access table was employed in all implementations over any alternate compact representation for the following reasons. A search operation is required to find an event-loss pair in a compact representation. If sequential search is adopted, then $O(n)$ memory accesses are required to find an event-loss pair. Even 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 [18] is adopted then only a constant number of memory accesses may be required but this comes at considerable implementation and run-time performance complexity. This overhead is particularly high on GPUs with

their complex memory hierarchies consisting of both global and shared memories. Compact representations therefore place a very high cost on the time taken for accessing an event-loss pair. Essentially the aggregate analysis process is memory access bound. For example, to perform aggregate analysis on a YET of 1 million trials (each trial comprising 1000 events) and for a layer covering 15 ELTs, there are 1000×1 million $\times 15 = 15$ billion events, which requiring random access to 15 billion loss values. Direct access tables, although wasteful of memory space, allow for the fewest memory accesses as each lookup in an ELT requires only one memory access per search operation.

Two data structure implementations of the 15 ELTs were considered. In the first implementation, each ELT is considered as an independent table; therefore, in a read cycle, each thread independently looks up its events from the ELTs. All threads within a block access the same ELT. By contrast, in the second implementation, the 15 ELTs are combined as a single table. Consequently, the threads then use the shared memory to load entire rows of the combined ELTs at a time. The second implementation has comparatively poorer performance than the first because for the threads to collectively load from the combined ELT each thread must first write which event it needs. This results in additional memory overheads.

The data structures in the basic sequential and parallel implementations are: (i) a vector consisting of all $E_{i,k}$ that contains approximately 800M-1500M integer values requiring 3.2GB-6GB memory, (ii) a vector of 1M integer values indicating trial boundaries to support the above vector requiring 4MB memory, (iii) a structure consisting of all El_i that contains approximately 100M-300M integer and double pairs requiring 1.2GB-3.6GB, (iv) a vector to support the above vector by providing ELT boundaries containing approximately 10K integer values requiring 40KB, and (v) a number of smaller vectors for representing $\mathcal{I}$ and $\mathcal{T}$.

In the basic implementation on the multi-core CPU platform the entire data required for the algorithm is processed in memory. The GPU implementation of the basic algorithm uses the GPU's global memory to store all of the required data structures. The basic parallel implementation on the GPU requires high memory transactions and leads to inefficient performance on the GPU platform. To surmount this challenge shared memory can be utilised over global memory.

The optimised implementation on the GPU builds on the parallel implementation and considers the following:

i. Chunking, which refers to processing a block of events of fixed size (or chunk size) for the efficient use of shared memory. The four steps (lines 4-29 in the basic algorithm, i.e., events in a trial and both financial and layer terms computations) of the algorithm are chunked. In addition, the financial terms, $\mathcal{I}$, and the layer terms, $\mathcal{T}$, are stored in the streaming multi-processor's constant memory. In the basic implementation, l_{x_d} and lo_{x_d} are represented in the global memory and therefore, in each step while applying the financial and layer terms the global memory has to be accessed and updated adding considerable overhead. This overhead is minimised in the optimised implementation by (a) chunking the financial and layer term computations, and (b) chunking the memory read operations (line no. 4-7) for reading events in a trial from the SimGrid, represented by Et_{id}. Chunking reduces the number of global memory update and global read operations. Moreover, the benefits of data striding can also be used to improve speedup.

ii. Loop unrolling, which refers to the replication of blocks of code included within for loops by the compiler to reduce the number of iterations performed by the for loop. The for loops provided in lines 5 and 8 are unrolled using the pragma directive, thereby reducing the number of instructions that need to be executed by the GPU.

iii. Reducing precision of variables, whereby the double variables are changed to float variables. 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 on GPUs.

iv. Migrating data from both shared and global memory to the kernel registry. The kernel registry has the lowest latency compared to all other memory.

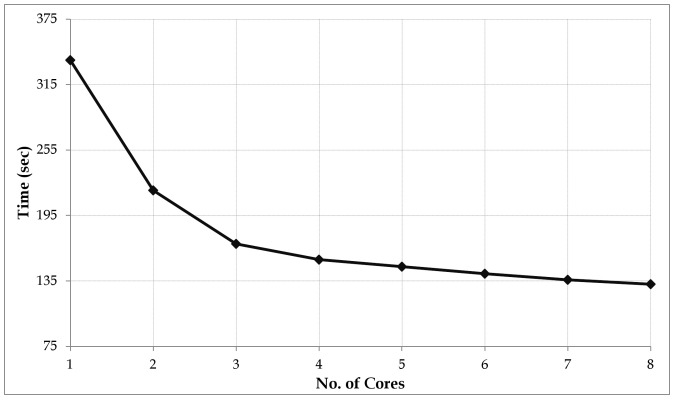
The optimised aggregate risk analysis algorithm was also implemented on a multi-GPU platform. This implementation was achieved by decomposing the aggregate analysis workload among the four available GPUs. For this a thread on the CPU invokes and manages a GPU. The CPU thread calls a method which takes as input all the inputs required by the kernel (the three inputs are presented in Section II) and the pre-allocated arrays for storing the outputs generated by the kernel. The CPU threads are invoked in a parallel manner thereby contributing to the speedup achieved on the multiple GPU platform.

IV. Performance Analysis

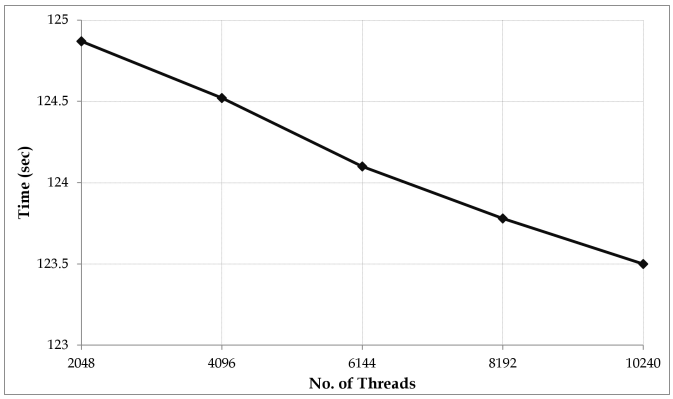
The results obtained from the three platforms, namely the multi-core CPU, many-core GPU and multiple GPUs are presented in this section. The classic sequential implementation is considered on the CPU, and the parallel implementation are considered on the multiple cores of the CPU, the many-core GPU and the multiple GPU.

A. Results from the multi-core CPU

The size of an aggregate analysis problem is determined by four key parameters of the input, namely (i) number of Events in a Trial, $|Et|_{av}$, which affects computations in line no. 5-29 of the algorithm, (ii) number of Trials, $|T|$, which



(a) No. of cores vs execution time



(b) Total No. of threads vs execution time

Figure 1: Performance of the parallel implementation of the aggregate analysis algorithm on a multi-core CPU

affects the loop in line no. 3 of the algorithm, (iii) average number of ELTs per Layer, $|ELT|_{av}$, which affects line no. 4 of the algorithm, and (iv) number of Layers, $|L|$, which affects the loop in line no. 2 of the algorithm.

In the experiments for the sequential implementation it was observed that there is a linear increase on running time of executing the sequential version of the basic aggregate analysis algorithm on a CPU using a single core when the number of the number of events in a trial, number of trials, average number of ELTs per layer and number of layers is increased. For a typical aggregate analysis problem comprising 1 million trials and each trial comprising 1000 events the sequential algorithm takes 337.47 seconds, with over 65% of the time for look-up of Loss Sets in the direct access table, and with only over 31% of the time for the numerical computations. This indicates that in addition to improving the speed of the numerical computations, techniques to lower the time for look-up can provide significant speedup in the parallel implementations.

Figure 1 illustrates the performance of the basic aggregate analysis algorithm on a multi-core CPU. In Figure 1a, a single thread is run on each core and the number of cores is varied from 1 to 8. Each thread performs aggregate analysis for a single trial and threading is implemented by introducing OpenMP directives into the C++ source. Limited speedup is observed. For two cores we achieve a speedup of 1.5x, for four cores the speedup is 2.2x, and for 8 cores it is only 2.6x. As we increase the number of cores we do not equally increase the bandwidth to memory which is the limiting factor. The algorithm spends most of its time performing random access reads into the ELT data structures. Since these accesses exhibit no locality of reference they are not aided by the processors cache hierarchy. A number of approaches were attempted, including the chunking method described later for GPUs, but were not successful in achieving a high speedup on our multi-core CPU. However a moderate reduction in absolute time by running many threads on each core was achieved.

Figure 1b illustrates the performance of the basic aggregate analysis engine when all 8 cores are used and each core is allocated many threads. As the number of threads are increased an improvement in the performance is noted. With 256 threads per core (i.e. 2048 in total) the overall runtime drops from 135 seconds to 125 seconds. Beyond this point we observe diminishing returns as illustrated in Figure 1a.

B. Results from the many-core GPU

In the GPU implementations, CUDA provides an abstraction over the streaming multi-processors, referred to as a CUDA block. When implementing the basic aggregate analysis algorithm on a GPU we need to select the number of threads executed per CUDA block. For example, consider 1 million threads are used to represent the simulation of 1 million trials on the GPU, and 256 threads are executed on a streaming multi-processor. There will be $\frac{1{,}000{,}000}{256} \approx 3906$ blocks in total which will have to be executed on 14 streaming multi-processors. Each streaming multi-processor will therefore have to execute $\frac{3906}{14} \approx 279$ blocks. Since the threads on the same streaming multi-processor share fixed size allocations of shared and constant memory there is a real trade-off to be made. If we have a smaller number of threads, each thread can have a larger amount of shared and constant memory, but with a small number of threads we have less opportunity to hide the latency of accessing the global memory.

Figure 2 shows the time taken for executing the parallel version of the basic implementation on the GPU when the number of threads per CUDA block are varied between 128 and 640. At least 128 treads per block are required to efficiently use the available hardware. An improved performance is observed with 256 threads per block but beyond that point the performance improvements diminish greatly.

The optimised implementation of the aggregate risk analysis algorithm on the GPU platform aims to utilise shared

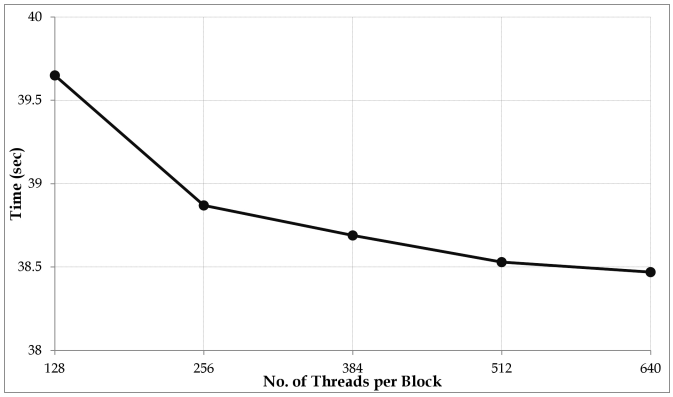


Figure 2: Graphs plotted for number of threads vs the time taken for executing the parallel implementation on many-core GPU

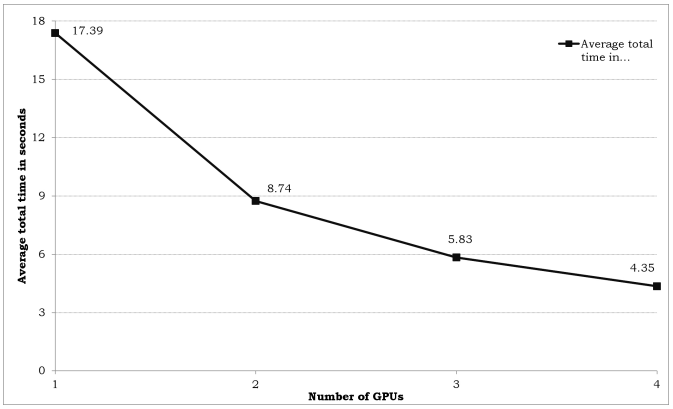


Figure 3: Graphs plotted for number of GPUs vs the time taken for executing the optimised parallel implementation on multiple GPUs

and constant memory as much as possible by processing "chunks", blocks of events of fixed size (referred to as chunk size), to improve the utilisation of the faster shared memories that exist on each streaming multi-processor. Further to optimise the implementation, loops are unrolled, the precision of variables are reduced by changing the double variables to float variables, and data from both shared and global memory are migrated to the kernel registry. The optimised algorithm has a significantly reduced runtime from 38.47 seconds down to 20.63 seconds, representing approximately a 1.9x improvement.

C. Results from the multiple GPU

Figure 3 illustrates the performance of the optimised aggregate analysis algorithm on multiple GPUs. A CPU thread is used to employ an available GPU for executing the aggregate analysis problem which is decomposed. In the experiments performed the aggregate analysis algorithm is executed using one, two, three and four GPUs. A much higher speedup is achieved on the multi-GPU over single GPU; the time taken for look-up of Loss Sets in the direct access table drops from 20.1 seconds to 4.25 seconds and the time for all Financial-Term and Layer-Term computations drop from 0.11 seconds to 0.02 seconds. The results from the multiple GPU show approximately 100% efficiency. The best average time obtained for executing the optimised algorithm on four GPUs is 4.35 seconds which is around 5x times faster than the time taken on the many-core GPU and 4x times faster than the time taken by the implementation executing on a single GPU of the multi-GPU machine.

Figure 4 shows the performance of the optimised aggregate analysis algorithm on four GPUs when the number of threads per block is varied from 16 to 64. Experiments could not be pursued beyond 64 threads per block due to the limitation on the block size the shared memory can use. The best performance of 4.349 seconds is achieved when the number of threads per block is 32; this is so as the block

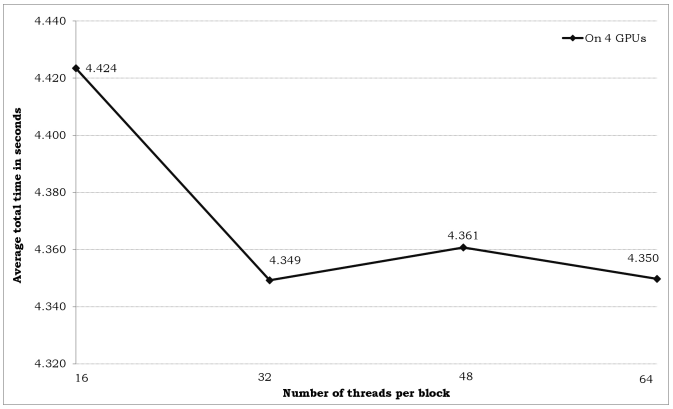


Figure 4: Graphs plotted for the number of threads per block size for four GPUs vs the time taken for executing the optimised parallel implementation on multiple GPUs

size is the same as the WARP size of the GPU whereby an entire block of threads can be swapped when high latency operations occur. Increasing the number of threads per block does not improve the performance owing to shared memory overflow.

V. Discussion

Figure 5 and Figure 6 summarises the results obtained from all the experiments of (a) a classic sequential implementation on the CPU, (b) a parallel implementation on the multi-core CPU, (c) a parallel implementation on the many-core GPU, (d) an optimised parallel implementation on the many-core GPU, and (e) an optimised parallel implementation on the multi-GPU.

Figure 5 shows the decrease in the total time taken for executing the aggregate analysis problem for 1 Layer, 15 Loss Sets and 1 million Trials with 1000 catastrophic Events per Trial from 337.47 seconds for a classic sequential implementation on the CPU to just 4.35 seconds for

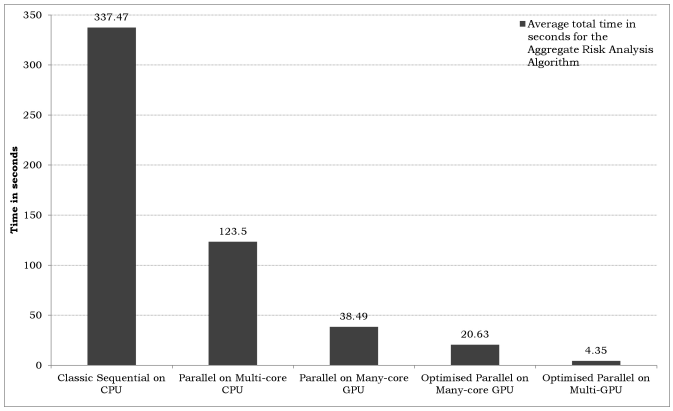


Figure 5: Bar graphs plotted for the average total time for executing (a) the sequential implementation on the CPU, (b) the parallel implementation on the multi-core CPU, (c) the parallel implementation on the many-core GPU, (d) the optimised parallel implementation on the many-core GPU, (e) and the optimised parallel implementation on the multi-GPU

an optimised parallel implementation on four GPUs. The parallel implementation on the multi-core CPU takes 123.5 seconds which is approximately $1/3^{\text{rd}}$ the time taken for the classic sequential implementation. This speedup is due to the use of multiple cores of the CPU, and there are memory limitations to achieve any further speedup. The time taken for executing the parallel implementation on the many-core GPU is reduced further by approximately $1/3^{\text{rd}}$ to 38.49 seconds over the multi-core CPU. This speedup is achieved due to the GPU architecture which offers lots of cycles for independent parallelism, fast memory access and fast numerical computations. The time taken for executing the optimised parallel implementation on the many-core GPU is reduced further by approximately half to 20.63 seconds over the many-core GPU. The speedup achieved in this case is attributed to four optimisations in the form of (i) chunking, (ii) loop unrolling, (iii) reducing the precision of variables and (iv) migrating data to kernel registry. The optimised parallel implementation on the multiple GPU takes 4.35 seconds which is approximately only $1/75^{\text{th}}$ the time taken by the CPU; the speedup in this case is achieved due to optimisations and the use of multiple GPUs.

Figure 6 shows the percentage of time taken for (a) fetching Events from memory, (b) look-up of Loss Sets in direct access table, (c) Financial-Term computations, (d) Layer-Term computations, and (e) both Financial-Term and Layer-Term computations. The best time taken for fetching Events from memory in the sequential implementation on the CPU is over 10 seconds, in the parallel implementation on the multi-core CPU is nearly 6 seconds, in the parallel implementation on the many-core GPU is nearly 4 seconds, in the optimised implementation on the many-core GPU is less than 0.5 seconds and on the multi-GPU is less than 0.1 seconds. Precisely the most optimised implementation on the multi-GPU has an improvement of 100 times for the time taken in fetching Events from memory over the sequential implementation on the CPU.

The majority of the total time taken for executing the aggregate analysis problem is for the look-up of Loss Sets in the direct access table. While the classic sequential implementation requires 222.61 seconds for the look-up, the optimised implementation on the multi-GPU only requires 4.25 seconds, which is an improvement of 50 times. However, there is scope for improvement to bring down the time. Surprisingly, on the multi-GPU 97.54% of the total time (4.33 seconds) is for look-up. This calls for exploring optimised techniques for look-up in the direct access table to further reduce the overall time taken for executing the aggregate analysis problem.

The numerical computations, including both the Financial-Term and Layer-Term computations take 104.67 seconds for the sequential implementation on the CPU and only $1/10^{\text{th}}$ that time for the parallel implementation on the CPU. The most optimised implementation takes merely 0.02 seconds on the multi-GPU platform which is approximately 5000 times faster than the sequential implementation on the GPU. The cutting edge technology offered by GPU architectures for numerical computations is fully harnessed to significantly lower the computational time in the aggregate analysis algoritm.

To summarise, the results obtained by profiling of time indicates that the optimised implementation on the multi-GPU platform is a potential best solution for real-time aggregate risk analysis; the implementation is 77x faster than the sequential implementation.

VI. Conclusion and Future Work

In short, this paper has presented the aggregate risk analysis algorithm, and its sequential and parallel implementations on multi-core CPUs and many-core GPUs. Large data is provided as input for aggregating risks across the Layers, and therefore, challenges such as efficiently organising input data in limited memory available, and defining the granularity at which parallelism can be applied to the aggregate risk analysis problem for achieving speedup is considered. While the implementation of the algorithm on the multi-core CPU provides a speedup of nearly 3x times over the sequential implementation, the basic GPU implementation provides a speedup of approximately 9x times over the sequential implementation on the CPU. The most optimised implementation provides a speedup of 16x times on the GPU and a speed up of **77x** on a multi-GPU over the CPU. It is notable that the acceleration has been achieved on relatively low cost and low maintenance hardware compared to large-scale clusters which are usually employed. These results confirm

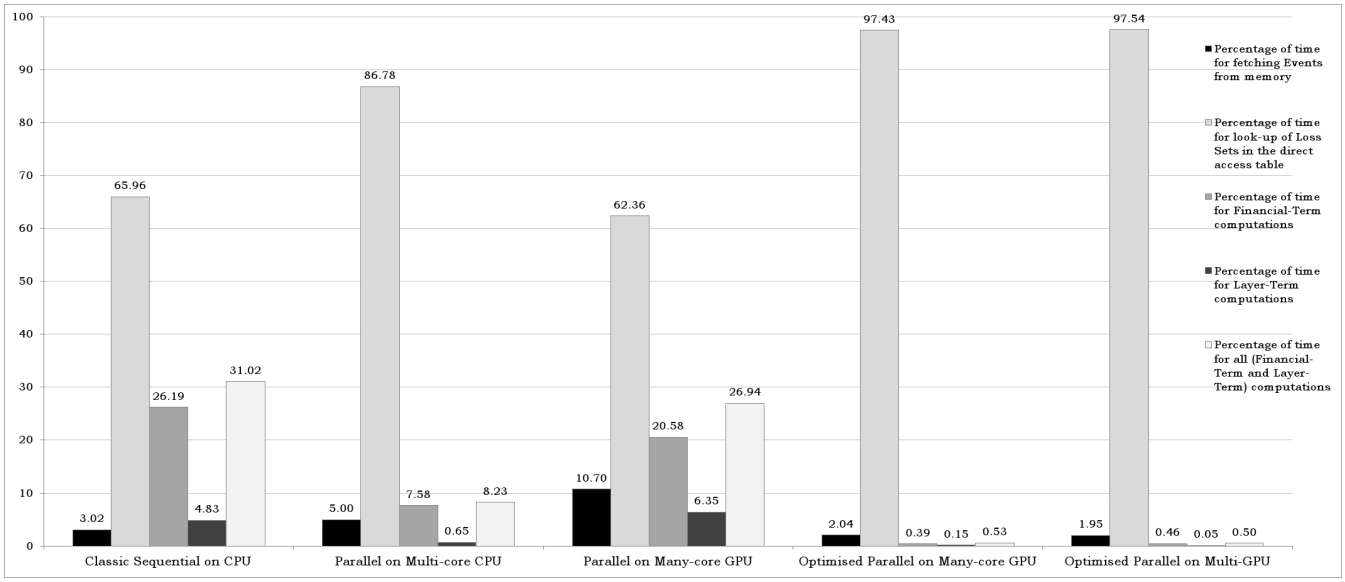


Figure 6: Bar graphs plotted for the percentages of (a) time for fetching Events from memory, (b) time for look-up of Loss Sets in the direct access table, (c) time for Financial-Term computations, (d) time for Layer-Term computations, and (e) time for all (Financial-Term and Layer-Term) computations

the feasibility of achieving high-performing aggregate risk analysis using large data for real-time pricing.

Future work will aim to investigate the use of compressed representations of data in memory, to optimise the GPU and CPU computations for a hybrid implementation, to evaluate the most optimised GPU implementation for handling larger data than what was employed in the research reported in this paper, and to incorporate secondary uncertainty in the computations.

REFERENCES

[1] T. S. Coleman and B. Litterman, *Quantitative Risk Management, + Website: A Practical Guide to Financial Risk*. Wiley, 2012.

[2] G. Connor, L. R. Goldberg, and R. A. Korajczyk, *Portfolio Risk Analysis*. Princeton University Press, 2010.

[3] M. B. Giles and C. Reisinger, "Stochastic finite differences and multilevel monte carlo for a class of spdes in finance," *SIAM Journal of Financial Mathematics*, vol. 3, no. 1, pp. 572–592, 2012.

[4] K. Smimou and R. K. Thulasiram, "A simple parallel algorithm for large-scale portfolio problems," *Journal of Risk Finance*, vol. 11, no. 5, pp. 481–495, 2010.

[5] A. J. Lindeman, "Opportunities for shared memory parallelism in financial modelling," in *Proceedings of the IEEE Workshop on High Performance Computational Finance*, 2010.

[6] G. G. Meyers, F. L. Klinker, and D. A. Lalonde, "The aggregation and correlation of reinsurance exposure," in *Casualty Actuarial Society Forum*, 2010, pp. 69–152.

[7] W. Dong, H. Shah, and F. Wong, "A rational approach to pricing of catastrophe insurance," *Journal of Risk and Uncertainty*, vol. 12, no. 2-3, pp. 201–218, 1996.

[8] R. M. Berens, "Reinsurance contracts with a multi-year aggregate limit," in *Casualty Actuarial Society Forum*, 1997, pp. 289–308.

[9] R. R. Anderson and W. Dong, "Pricing catastrophe reinsurance with reinstatement provisions using a catastrophe model," in *Casualty Actuarial Society Forum*, 1998, pp. 303–322.

[10] D. Cummins, C. Lewis, and R. Phillips, "Pricing excess-of-loss reinsurance contracts against catastrophic loss," in *The Financing of Catastrophe Risk*, K. A. Froot, Ed. University of Chicago Press, 1999, pp. 93–148.

[11] Y.-S. Lee, "The mathematics of excess of loss coverages and retrospective rating - a graphical approach," in *Casualty Actuarial Society Forum*, 1988, pp. 49–77.

[12] R. S. Miccolis, "On the theory of increased limits and excess of loss pricing," in *Casualty Actuarial Society Forum*, 1977, pp. 27–59.

[13] G. Woo, "Natural catastrophe probable maximum loss," *British Actuarial Journal*, vol. 8, 2002.

[14] M. E. Wilkinson, "Estimating probable maximum loss with order statistics," in *Casualty Actuarial Society Forum*, 1982, pp. 195–209.

[15] E. Kremer, "On the probable maximum loss," *Blatter der DGVFM*, vol. 19, no. 3.

[16] A. A. Gaivoronski and G. Pflug, "Value-at-risk in portfolio optimization: Properties and computational approach," *Journal of Risk*, vol. 7, no. 2, pp. 1–31.

[17] P. Glasserman, P. Heidelberger, and P. Shahabuddin, "Portfolio value-at-risk with heavy-tailed risk factors," *Mathematical Finance*, vol. 12, no. 3, pp. 239–269.

[18] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.