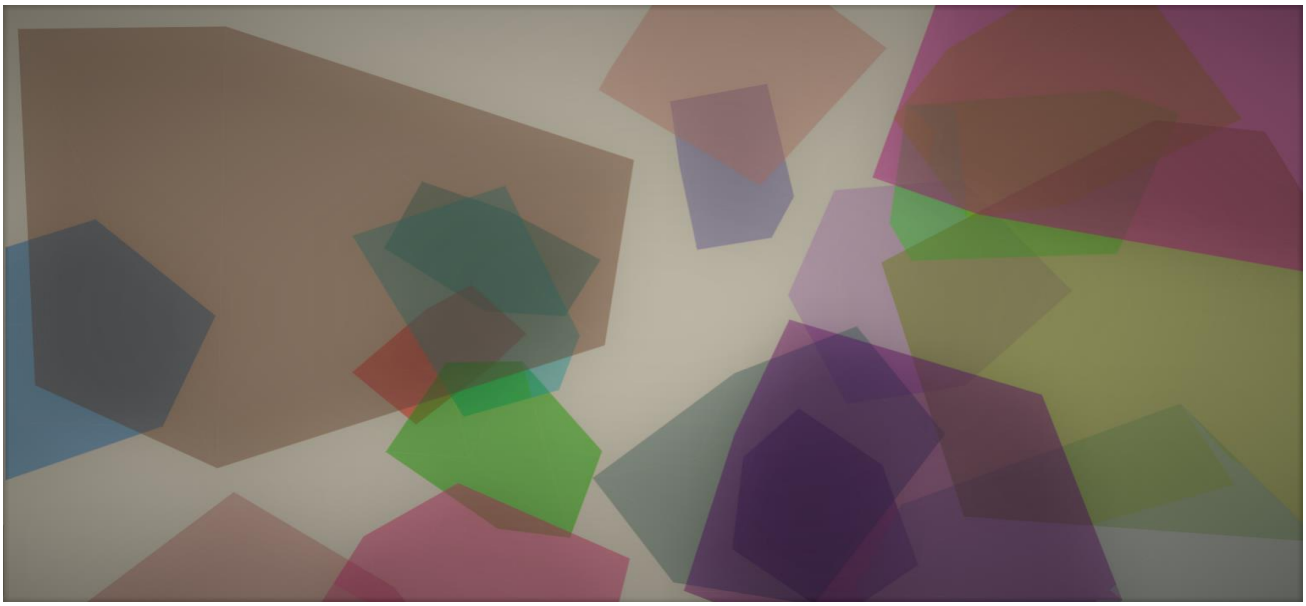


SPEED UP OF NUMERIC CALCULATIONS USING A GRAPHICS PROCESSING UNIT (GPU)

NIKOLA VASILEV, DR. ANATOLIY ANTONOV

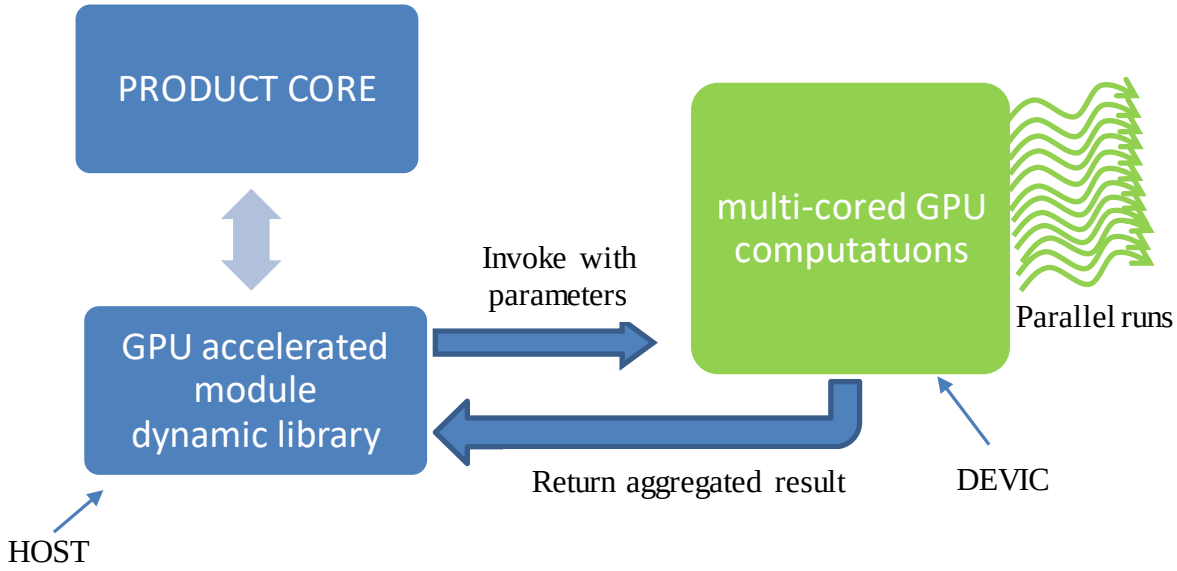
Eurorisk Systems Ltd.
31, General Kiselov str.
BG-9002 Varna, Bulgaria
Phone +359 52 612 367
Mobile+359 52 612 371
info@eurorisksystems.com
www.eurorisksystems.com

Contents

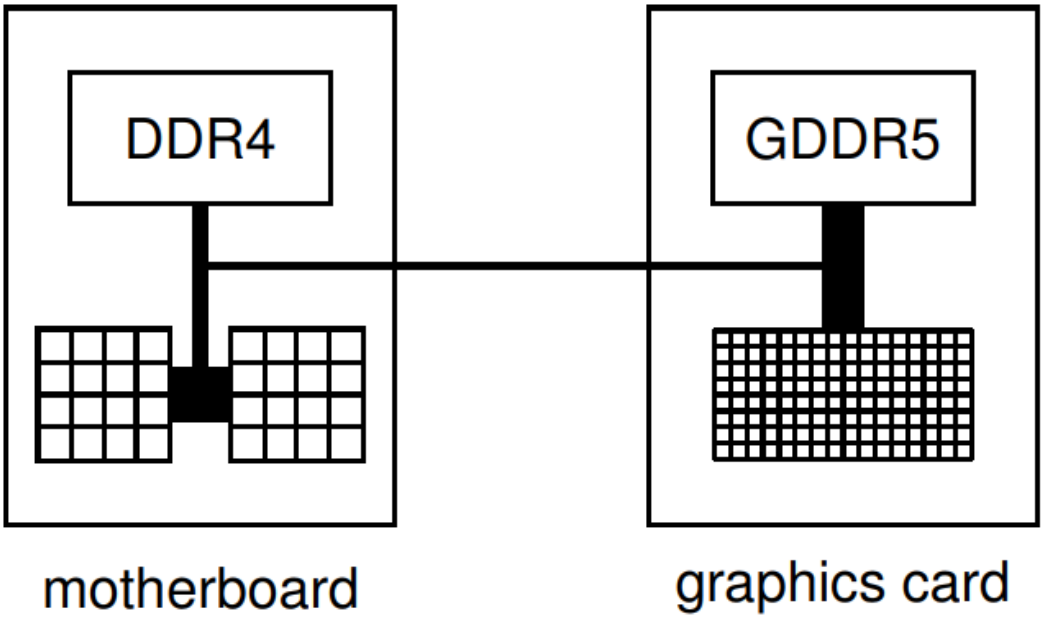
Introduction

The GPU is a graphics processing unit and represents a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines". In the financial sector, the potential of this resource could be used to improve financial computing. This makes sense, because faster pricing gains more revenue, better modeling contributes to less risk and maximizing resources enhances the efficiency. Nowadays, GPUs enable the calculation, simulation, as well as prediction of prices and risk for complex options, OTC derivates, complex exotic option instruments, in seconds, rather than minutes or even hours. The architecture of the GPUs makes it possible to run several simulations simultaneously, thereby increasing the quality of the results. With more confidence in the data, it is possible to offer tighter spread and gain competitiveness. GPU even provides the means to run complex models, that up until now could not possibly be ran. Results of very complex models can be obtained in real/near time, rather than overnight, which provides deeper insight into exposures, making it possible to rapidly adjust positions and reduce risk.

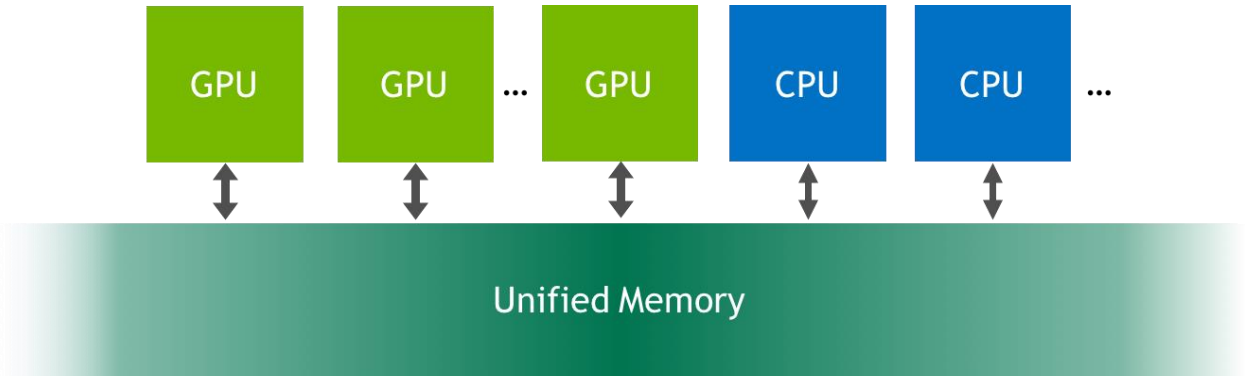
The basic concept of GPU accelerated modules



The concept of CPU accelerated models is developed separately from the product and is invoked with input parameters. GPU accelerated modules are designed in two parts – host and device. 'Host' represents tasks for the CPU, such as preparing algorithms, whilst main computing is intergraded into the 'device' part and will deliver the results. GPU acceleration is only used for massively paralleled algorithms, i.e. for algorithms with at least 1,000 threads. The hardware overview is shown below:



Another advantage is the multi-GPU support, that theoretically enables much better performances. For example, systems that have eight identical GPUs will provide results eight times faster than systems with only one GPU. The access of data between CPU and GPU is much faster, because of the unified memory architecture. Calculations within GPU accelerated modules are at least 10 times faster, but it is possible to achieve a speed of up 200 times, depending on the applied parallel algorithm.



In addition, the number of graphical processor cores enables the running of complex analysis on large portfolios more efficiently, because the saturation of the GPU is much more complicated, as opposed that of the CPU.

GPU represents a Heterogeneous Parallel Computing (HPC). The difference between CPU and GPU is shown below:

- CPU is optimized for fast single-thread execution
- CPU cores are designed to execute two or more threads concurrently
- GPU is optimized for high multi-thread throughput

- GPU cores are designed to execute several parallel threads concurrently and are optimized for data-parallel throughput computation

The system's key feature is that cores of every GPU block are Single Instruction Multiple Thread (SIMT) cores, i.e. a group of 32 cores will execute the same number of instructions simultaneously using different data (for example, elements in matrices). SIMT is a natural choice for many core chips, as it simplifies each core. Threads on each GPU block execute in groups of 32, called "warps"; execution alternates between "active" warps, with warps becoming temporarily "inactive" when waiting for data.

GPU vs. CPU benchmark example

We created heavy random filler algorithms, to push the limits of the GPU application. The results were satisfying. Following benchmarks is an exponential problem, to which the CPU isn't the sufficient answer, because of the lack of random numbers and prolonged computation times. The CPU does produce results, but with complex modifications of random numbers, whereby the entire process takes longer than a day.

The benchmark task is derived from a medicine application that simulates breast cancer and looks as follows:

1. Six types of spheres should be randomly placed within a half cylinder container, where the total volume for every sphere type should be about the same:

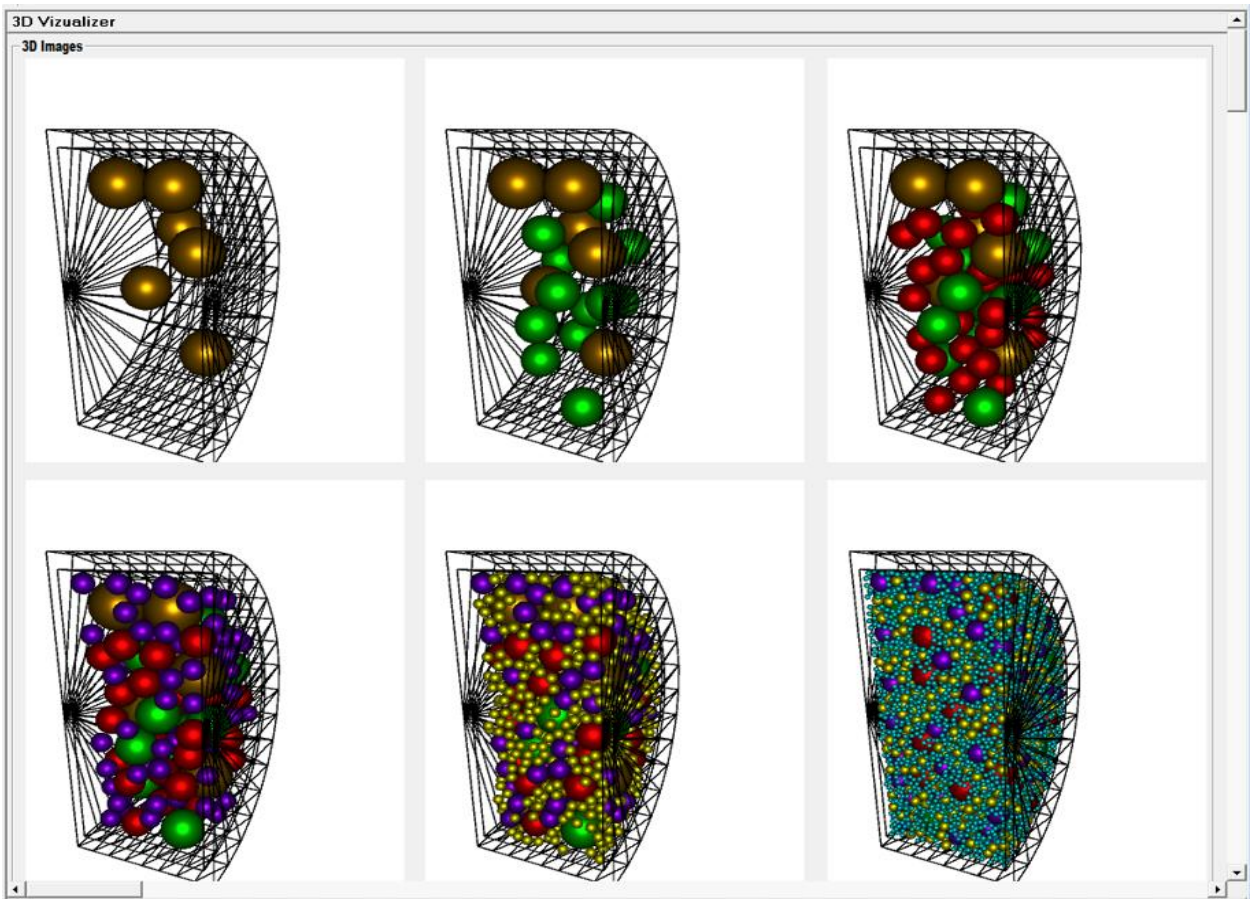
$$\sum_{i=1}^{n} V_{1i} = \sum_{j=1}^{m} V_{2j} = \sum_{k=1}^{o} V_{1k} = \sum_{r=1}^{p} V_{1p} = \sum_{s=1}^{t} V_{1s} = \sum_{u=1}^{h} V_{1u}$$

   where:

   n = 6 - number of spheres with radius r1 = **7.94**
   m = 11- number of spheres with radius r2 = **6.35**
   o = 25- number of spheres with radius r3 = **4.76**
   p = 83- number of spheres with radius r4 = **3.175**
   t = 775- number of spheres with radius r5 = **1.59**
   h = 6775 - number of spheres with radius r6 = **0.79**

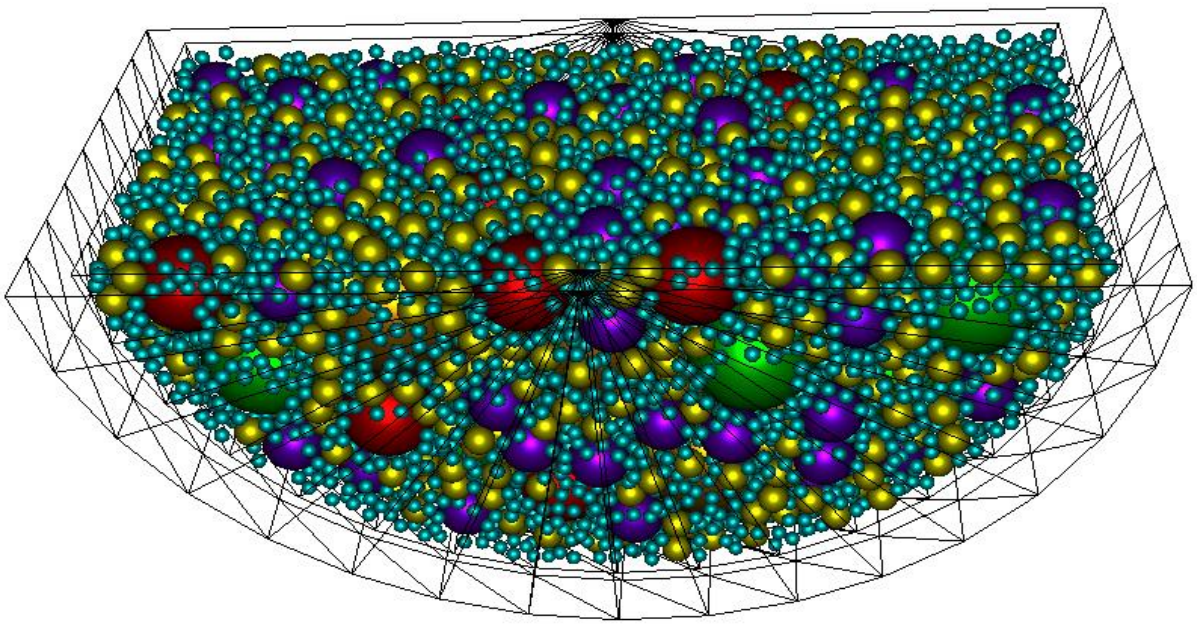
2. A sphere should not include other spheres
3. The random sphere packing, that is giving a 64% filling quote, is described in the following link: http://mathworld.wolfram.com/SpherePacking.html

The random sphere packing task can become very complex and may require large computation power. The reason is that during the filling of the container, free spaces, in which the next random sphere is placed, become small and thus a large number of random trials will be discarded. Sphere types are generated starting with the largest radius. The filling phases for the 6 different sphere types is shown below.



One final random sphere packing, that is giving a 64% filling quote, is presented on the next page.

The random sphere packing task has been run on three hardware platforms: CPU with 8 cores, GPU with 160 cores and GPU with 512 cores. The comparison is performed against



a CPU whit 1 core, resulting in a calculation time of 30 hours. The sphere packing time for the container above, enclosing 6 000 spheres, has been 20 sec. on the third platform.

Processor unit	Cores@Clock rate	Time	Faster then single core
CPU – intel i7 6700K	8 cores@4.0GHz	300 sec.	x 8
GPU – nVidia GTX960M	5 blocks x 32 cores@1097MHz	80 sec.	X 50
GPU - nVidia GTX1070	16 blocks x 32 cores@1506MHz	20 sec.	X 200

The random sphere packing algorithm was transformed for a parallel run on the GPU/CPU cores, using the following approach:

1. The filing of the container sequentially, for all sphere types, is performed using a number of stages.
2. Within every stage, each processor core (or core thread) generates a random sphere.
3. The generated sphere is then checked against all spheres within the container, that don't change during the current stage. An attempt to place the sphere in the container is performed. Steps 2 and 3 are performed by the cores in parallel.
4. All successfully placed spheres are then checked against each other and placed into the container. This is a sequential task, but the number of successfully placed spheres is small.
5. This process is then repeated, from point 2 onward, during the following stage. The algorithm ends when the number of needed spheres is achieved.

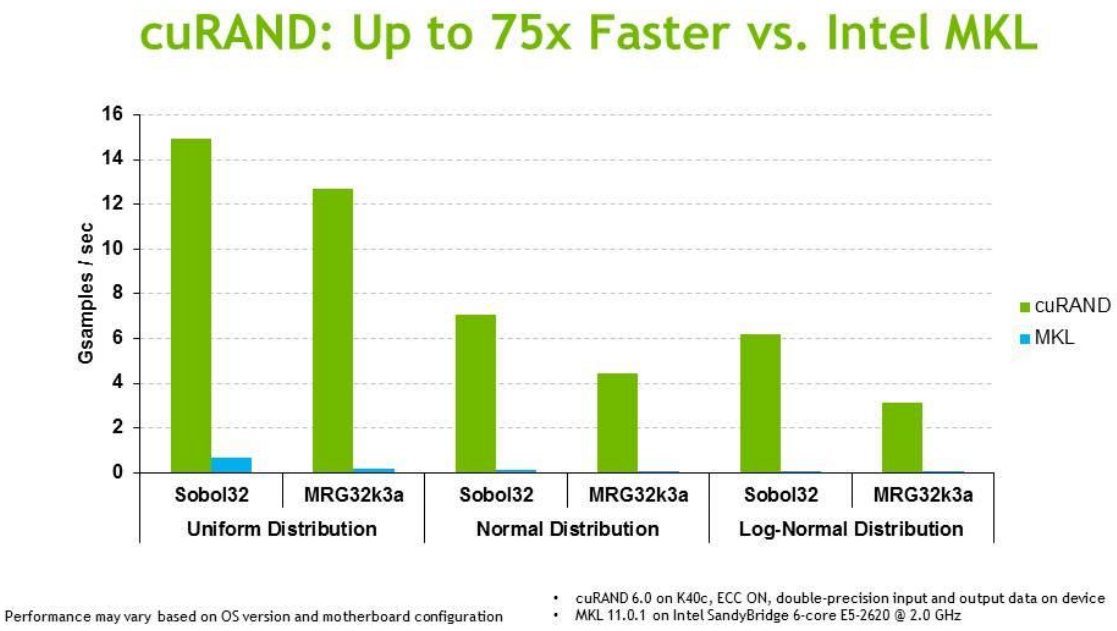
Recent speed tests, performed on a benchmark with 30 000 spheres on a 1920 core GPU (s. bellow, **NVidia GeForce GTX 1070**), demonstrate that the filling of spheres is executed under 5 minutes, which is about 800 time faster then a single core.

The comparison between GTX960M and GTX1070 suggests that there are 3 time more blocks, but the duration is linearly 4 times smaller, because of the Clock rate. This means that, if this test were ran on faster a GPU, the performance would be much better. Note: this code is not optimized as much as the CPU code. Pricings for the same high-class CPU and GPU are shown below:

Name	Cores@Clock rate	FP32 Performance	Price
Intel Core i7-6700K	8@4.0Ghz	0.109 TFLOPS	$340
NVidia GeForce GTX 1070	15x128@1506MHz	6.0 TFLOPs	$400
NVidia GeForce GTX1080	24x128@1733MHz	8.7 TFLOPs	$600
NVidia Quadro P5000	24x128@1500MHz	8.9 TFLOPS	$2,499
Nvidia Quadro P4000	16x128@1400MHz	5.3 TFLOPS	$769.00

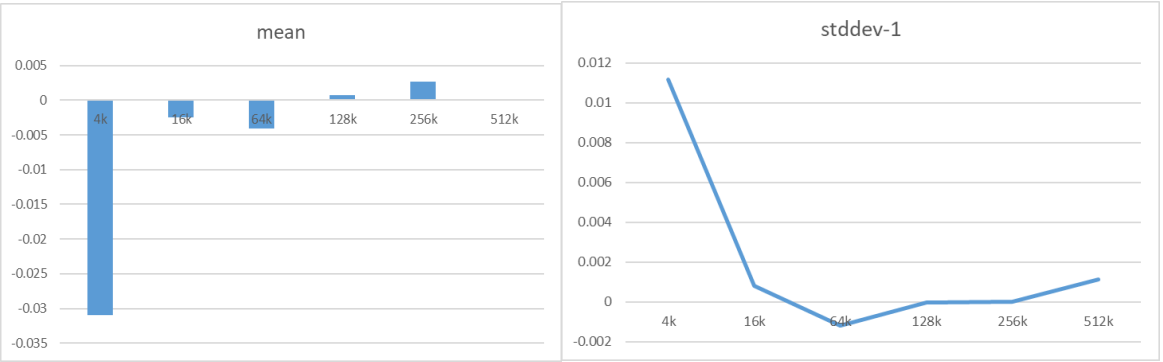
Providing high performance random number generation (RNG)

CuRAND is a library that provides Host API for the generation of random numbers in bulk on the GPU and has four high-quality RNG algorithms. After conducting tests on actual problems, the results have been generated two times faster, although this depends on the quality and the performance of random numbers. The result are shown in figure bellow. According to nVidia, the generation is up to 75x faster than with a standard intel MKL generator. This library supports Pseudo- and Quasi-RNGs — MRG32k3a, MTGP, Mersenne Twister, XORWOW, Sobol generators. It also provides several output distributions (e.g. uniform, normal, log-normal).



The table below shows mean and standard deviation delta to one of the generated simulation series:

Random numbers	mean	stddev-1
4k float	-0.030914356	0.011161298
16k float	-0.002508757	0.000820388
64k float	-0.004117639	-0.001202603
128k float	0.000718359	-3.47317E-05
256k 2 x float	-0.001716881	0.001135225
128k double	0.001197103	-0.002223814
256k 2 x float	-0.001716881	0.001135225
256k 2 x double	0.002675153	4.46802E-06
……	……	……



This test shows that, with the rise of random numbers, the mean is closer to 0 and the standard deviation is closer to 1, without correction.

Dense Linear Algebra

The NVIDIA cuBLAS library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS). NVBLAS is a GPU-accelerated version of BLAS, that further accelerates BLAS Level-3 routines by dynamically routing BLAS calls to one or more NVIDIA GPUs, as well as CPUs in the system, through the cuBLAS-XT interface.

Researchers and scientists use cuBLAS for developing GPU-accelerated algorithms in areas such as high performance computing, image analysis and machine learning. cuBLAS performs up to 5X faster than the latest version of the MKL BLAS on common benchmarks.

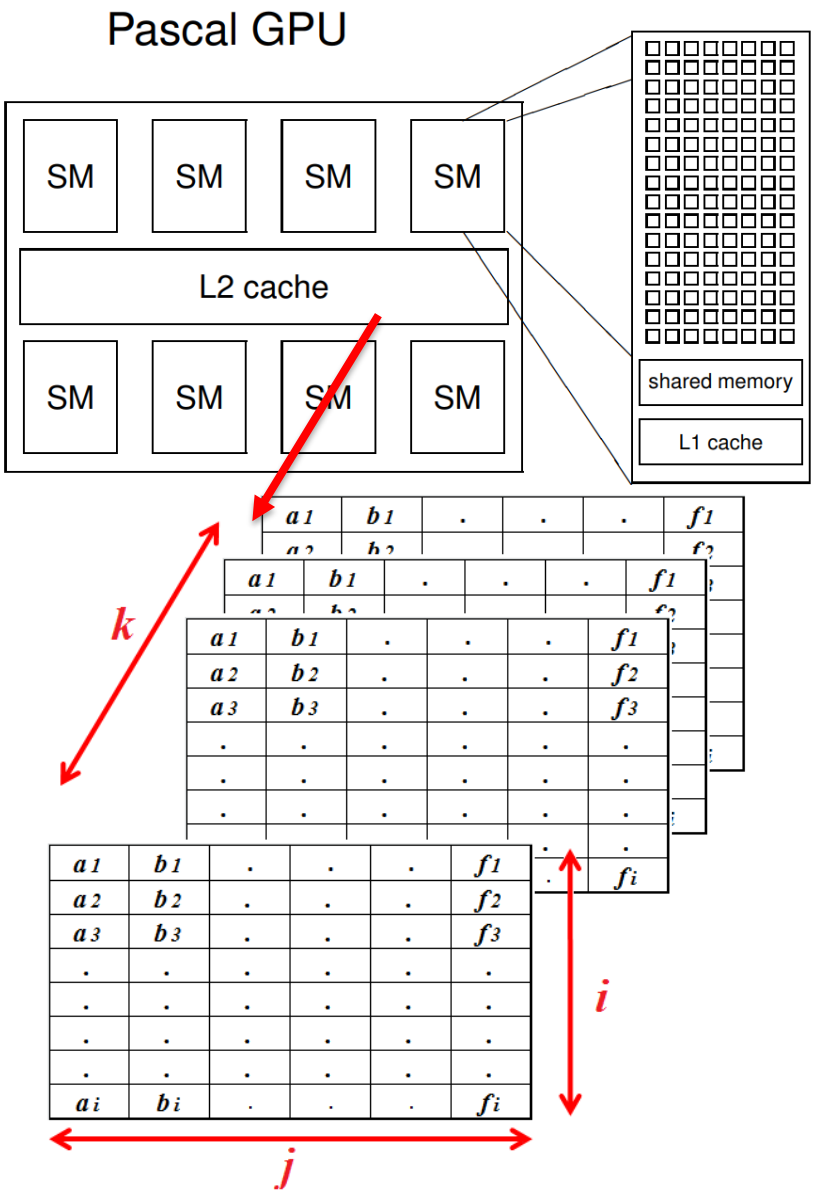
Parallel primitives and data structures – THRUST library

GPU-accelerated scan, sort, transform and reduce Improved developer productivity via C++ template library, allowing developers to focus on high-level tasks. Library simplifies memory management and data transfer.

Use cases of development

- Real-Time Options Analytics. Stochastic Volatility Modeling
- Stochastic Volatility + Jumps Modeling. Large-Scale Interest-Rate Swaps Risk
- Large-Scale Monte Carlo Risk, Large-Scale Parametric VaR
- Basket Barrier-Option (Monte Carlo)
- Derivatives simulation, ABS/SPV simulation. Complex exotic option simulation

Monte Carlo VaR simulation approach on GPU



In the figure above, a 3D matrix is displayed, representing an abstraction of the GPU processing unit. The dimensions express the following:

- dimension "k" represents each block;
- dimension "l" represents each core;
- dimension "j" represents each thread;

In conclusion, the length of dimension j should be at least 32, because of the warp size. The multiplication of k x i should give all CUDA cores in our hardware.

Monte Carlo Simulations correspond to an algorithm that generates random numbers, which are used to compute the distribution of a formula without a closed (analytical) form. This means that we need to proceed to some trial in picking up random numbers/events and

assess what the formula yields to approximate the solution. Drawing random numbers over a large number of times (a few hundred to a few million, depending on the problem at stake) will give a good indication of what the output of the formula should be. Computing VaR with Monte Carlo Simulations is very similar to Historical Simulations. The main difference lies in the first step of the algorithm – instead of using the historical data for the price (or returns) of the asset and assuming that this return (or price) can reoccur in the next time interval, we generate a random number that will be used to estimate the distribution of the return (or price) of the asset at the end of the analysis horizon. The algorithm for a multi-step Monte Carlo is described in the following steps:

1. Determine the time horizon t for our analysis and divide it equally into small time periods, i.e. (dt = t/n).
2. Draw a random number from a random number generator and update the price of the asset at the end of the first time increment.

   It is possible to generate random returns or prices. In most cases, the generator of random numbers will follow a specific theoretical distribution. This may be a weakness of Monte Carlo Simulations, compared to Historical Simulations that use empirical distributions. When simulating random numbers, we generally use normal distribution, but other distributions are possible as well.

   In this article, we use the standard stock price model to simulate the path of a stock price from the i-th day as defined by:

$$R_i = (S_{i+1} - S_i)/S_i = \mu\Delta t + \sigma\varepsilon\sqrt{\Delta t}$$

   - R_i – return of the stock on the i-th day
   - S_i – stock price on the i-th day
   - S_{i+1} – stock price on the i+1st day
   - μ – sample mean of the stock price
   - δt – timestep
   - σ – sample volatility (standard deviation) of the stock price
   - ϵ – random number generated from a normal distribution

   At the end of this step/day (δt = 1 day), we have drawn a random number and determined S_{i+1}, since all other parameters can be determined or estimated.

3. Repeat Step 2 until reaching the end of the analysis horizon T by walking along the N time intervals.
4. Repeat Steps 2 and 3 a number M of times to generate M different paths for the stock over T.
5. Rank the M terminal stock prices from smallest to largest, read the simulated value in this series that corresponds to the desired (1-α) confidence level (generally 95% or 99%) and deduce the relevant VaR, which represents the difference between the current price and the stock price at the confidence level.

Let us assume that we want the VaR with a 99% confidence interval. In order to obtain it, we first need to rank the M terminal stock prices from lowest to highest.

Then we read the 1% lowest percentile in this series. This estimated terminal price, Si+T1% means that there is a 1% chance that the current stock price Si could fall to Si+T1% or less during the considered period in and under normal market conditions.

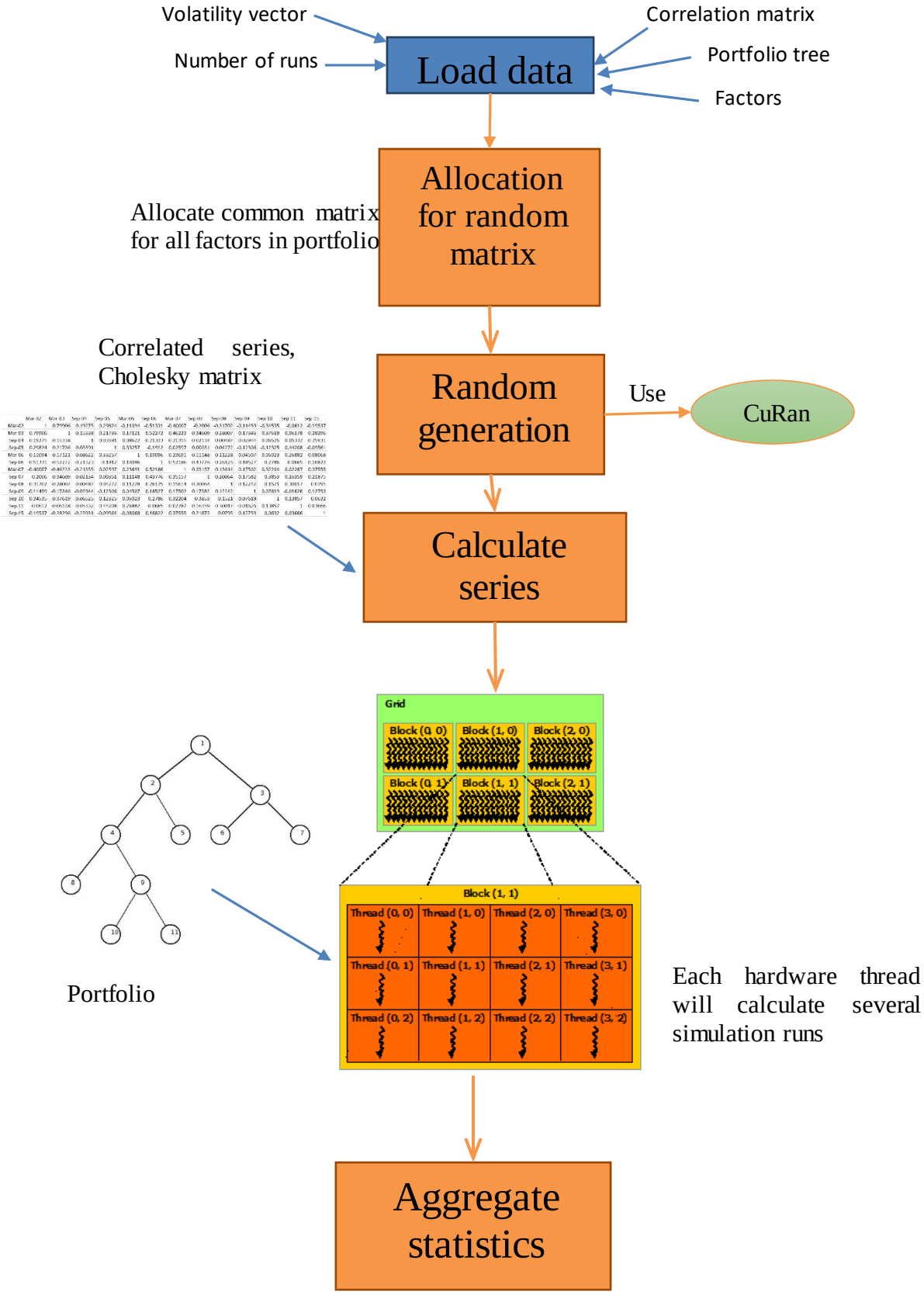
If Si+T1% is smaller than Si (which is the case most of the time), then Si – Si+T1% will correspond to a loss. This loss represents the VaR with a 99% confidence interval.

In the past decade, there have been a lot of improvements in VaR models. In reality, however, the greatest importance should be ascribed to computational constrains. Every time a trade takes places, for example, the position of two economic agents is updated, and two VaR computations are required. The most active futures exchanges in the world nowadays experience at least 1 000 000 in around 10 000 secs. On average, this requires at least 100 VaR computations per second. With the development of new hardware and improvements of processor speed, parallel computing has been broadly used in the finance area. One of this area's representatives is the GPU. GPUs are originally designed to efficiently manipulate computer graphics. Their parallel structure makes them highly effective at a variety of complex algorithms, in comparison to general-purpose CPUs.

Nowadays, GPU is widely used in financial computing, such as VaR estimating, option pricing, etc. A lot of general methods, that have been used in the area of finance, such as Finite Differences, Random number generation, Monte Carlo test case, dynamic programming, etc., can be greatly accelerated by the GPU. Michael Feldman, a HPCwire editor, states that GPU computing is one of the "new kids" on the Wall Street, a technology that is making inroads across nearly every type of HPC application.

Algorithm schedule:

1. **Read factors**, with static values at T0, the volatility, correlation matrix and Portfolio model
   Approach in this step is to extract risk factors from PMS and values to T0. In further steps, the triangular correlation matrix, which is extracted from the PMS, should also be extracted here.. The portfolio model in PMS for Monte Carlo VaR simulations is displayed in the form of a portfolio tree, which portrays a representation in memory of class based node relations. In the CUDA architecture, this tree cannot be computed. The approach here is to extract portfolio nodes only with necessary data (number, left, right, value, days) and aggregate it in CUDA dll library as arrays.
2. **Generate random vector series** using all available threads simultaneously.
   Random vectors can be generated via GPU cores in parallel, because it represents a multiplication of a Cholesky correlation matrix and normally distributed random floating point numbers. Here, the advantage of the GPU is that because all matrix elements can be computed in parallel and on the other side, the cuRand generator is more efficient.


Volatility vector
Number of runs
Load data
Correlation matrix
Portfolio tree
Factors
Allocate common matrix for all factors in portfolio
Allocation for random matrix
Correlated series, Cholesky matrix
Random generation
Use
CuRan
Calculate series
Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Portfolio
Each hardware thread will calculate several simulation runs
Aggregate statistics

3. **Synchronize threads**
   Before starting a computation, all threads should be synchronized in parallel. This step is major, because every thread should require the same length of time for the same number of simulation runs .
4. **Calculate all series in parallel**
   This is the main part of the algorithm, as it works with common constant resources – correlation matrix and portfolio tree – that are used for data in parallel.. The tree is a compute rule based model. Depending on the hardware, each thread will compute several runs. Result will be stored in an array, which cannot be collisional.
5. **Aggregate and return statistic to application**

Example: If a client wants the numbers of a run to be 5 000 000, in step 2 a matrix will be generated in advance, having the size "Number of factors" x 2 000 000. After that, the simulation will start and the GPU will allocate 61 440 threads (1920 x 32). Afterwards, various results will be calculated for each thread (~80 runs per thread). Finally, all results will be returned and aggregated.

With the proper hardware, one can expect an acceleration that is at least 100 times faster than with the CPU implementation. With the increasing number of runs, errors in final results will be minimal. For example, 5 000 000 runs with a confidence 99.9% will have a statistical error of 1.41% , compared to 10 000 runs with statistical errors of 31.62%.

Longstaff-Schwartz method on GPU

Option pricing is an important area of banks' activities. Nowadays, the most common type of options are American type options. These options represent contracts that give the option's buyer the right, but not the obligation, to buy or sell an underlying asset, where the right can be exercised at any point until the expiry date. This last condition implies that the pricing of an American option is much harder than that of an European version.

The value of the option over time can be described as a partial differential equation, called the Black-Scholes equation, which is:

$$\frac{\partial u}{\partial t} + rS\frac{\partial u}{S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 u}{\partial S^2} - ru = 0$$

Where:

- S - price of the underlying asset at time t
- u - value of the option
- r - risk-free interest rate
- t - time, where t = 0 denotes the present and T represents the expiration time
- σ - volatility of the underlying asset
- K - strike price

The entire LSM schedule is described below:

1. Load derivate parameters
2. Initialize paths $S_i(t)$, where $t=0,t_1 \ldots t_N$, $i=1,2,\ldots M$
3. Put $P_i = g(S_i(t_N))$ for all i
4. For each $t = (t_{N-1}$ to $t_1)$
   4.1. Find paths $\{ i_1 , i_2, \ldots i_N \}$ s.t. $g(S_i(t)) > 0$, *i.e.* that are in-the-money path (*itm_paths*)
   4.2. Let $x_i = S_i(t)$
   4.3. Let $y_i = e^{-r\Delta t}P_i$ for $i \subseteq itm\_paths$
   4.4. Perform regression on x, y to obtain beta coefficients
   4.5. Estimate the value of continuation $C(S_i(t))$ and calculate the value of immediate exercise $g(S_i(t))$ for $i \subseteq itm\_paths$
   4.6. For i from 1 to M *do*
      4.6.1. If $i \subseteq itm\_paths$ *and* $g(S_i(t)) > C(S_i(t))$ -> $P_i = g(S_i(t))$
      4.6.2. Else $P_i = y_i$
      4.6.3. Calculate price

$$price = \frac{1}{M}\sum_{i=1}^{M} e^{-r\Delta t} P_i$$

Where:

- M – number of paths
- N – number of time steps
- S – price of underlying asset
- g(S(t)) – payoff at time t
- P – current payoff
- C – value of continuation

The initial value of the Stock price at time T is:

$$S = S_0 . e^{T.\left(r-\frac{\sigma^2}{2}\right)+\sigma.\sqrt{T}.nrand\,(r)}$$, where r is 0 to the number of paths

Value of S is:

$$S = S_0 . e^{t(i).(r-\frac{\sigma^2}{2})+\sigma.\sqrt{\frac{T}{N}.t(i)}.nrand\,(r)}$$, where I is to 0 to the number of time steps

This algorithm will be suitable for Heterogeneous Parallel Computing on GPU, because it has a lot of data that is independent to each other. For example, each time step can be computed on a separate GPU core, and computations of stock price paths can be grouped in warps. This means that, for example, each thread warp should compute a minimum of 32 paths. The architecture provides the use of GPU cache memory to store matrices for each time step. The reason for this is that different values are computed for each step.

The performance of algorithms is increased by valuating large numbers of options in large slices of exercising time, because of high computation power and high level of parallelism. It is expected to achieve a speed up between 10 and 100 times, depending on the time slice of the exercising time.

Concluding words

Real time risk management is a problem for the financial industry today, where pushing GPU computing would provide faster analysis. GPUs are used by major financial institutions for quant finances. Performance gains will be at least 10x “dollar for dollar”. In general, developers would parallelize their code. This will provide finance software with the advances of many-core hardware.