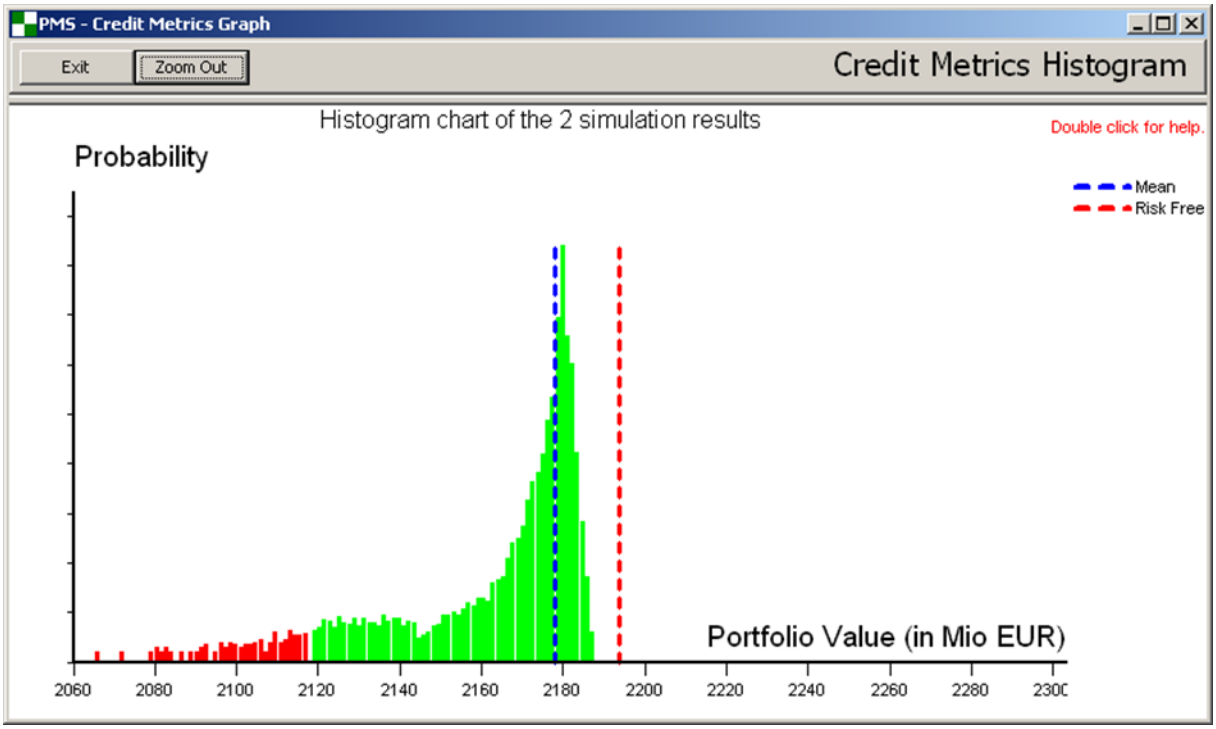
CreditMetrics

CreditMetrics is a credit risk model that quantifies the loss caused by a change in the obligor's creditworthiness. Credits are subject to an interest rate risk and counter party risk. The main goal of the credit risk evaluation is the computation of the credit risk exposure profile, showing the notional amount that can be lost when a counterparty defaults its obligations. Credit risk of counterparties is measured by single value, the Value-at-Risk(VaR). VaR represents unexpected losses of the portfolio value due to credit events at given confidence level and horizon. The measure of credit risk depends on the estimation of a transaction's mark-to-market value on different future sampling time points, where the counterparty is expected to default. It is assumed that in case of default at a future time point all remaining positive cash flows will be lost, so the loss can be expressed as the probability-weighted mark-to-market value of these cash flows.



Portfolio histogram and distribution

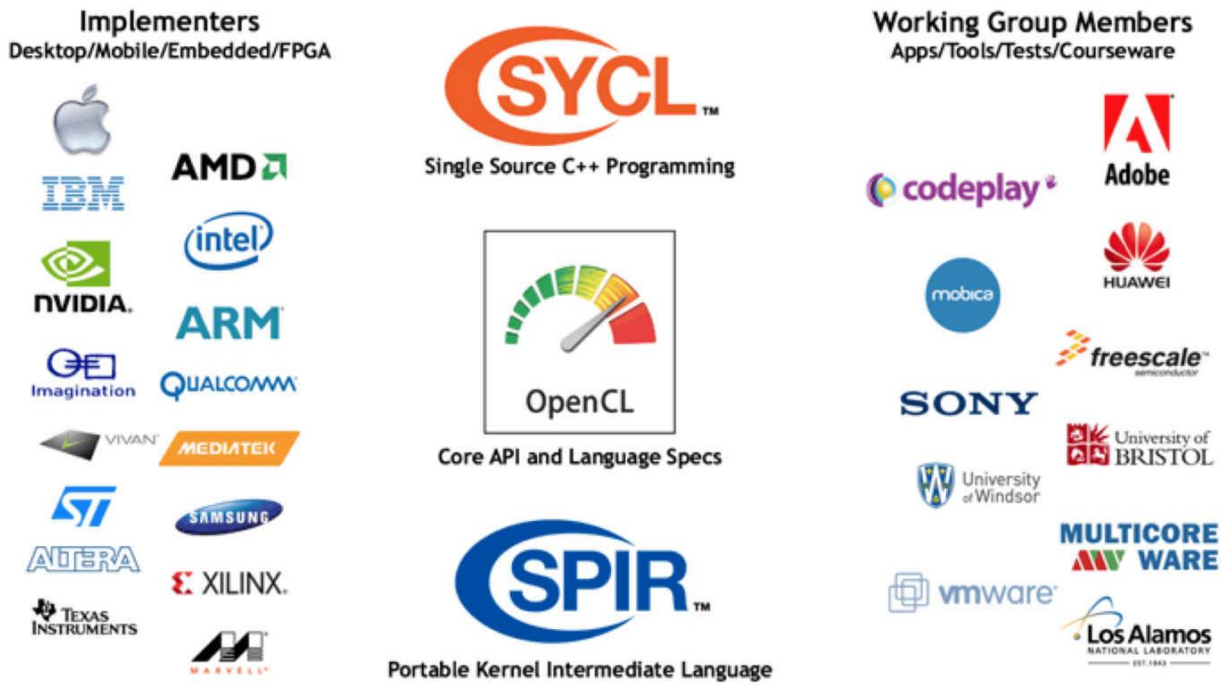
The consolidation of the counterparty's credit risk distribution is based on the structured Monte Carlo simulation which uses the Monte Carlo simulation engine, given the non-normal credit risk distribution. The main calculation target is the generation of appropriate credit rating movement scenarios in accordance with the counterparty credit risk distribution

OpenCL™

Open Computing Language(OpenCL™) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms. Using the OpenCL, users can launch computation on high-performance devices – mostly Graphical Processor Units(GPU). GPU is graphical processor unit. It presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines", but in finance we could use the potential of this resource to better financial

computing. It's make sense because faster pricing gains more revenue, more modeling gains less risk and maximizing resources gives more efficiency. Nowadays GPUs enables you to calculate, simulate and predict pricing and risk for complex options, OTC derivate, complex EOT types instruments in seconds, rather than minutes or even hours. The architecture of GPUs allows you to run more simulations that would increase the quality of your results. With more confidence in your data, you are able to offer tighter spread and gain competiveness. GPU even make it possible to run complex model, that were even impossible. You can obtain results of very complex models in real/near time, rather than overnight, and also provide deeper insight into your exposures enabling you to rapidly adjust positions and reduce risk.

Functions executed on an OpenCL device are called "kernels". OpenCL is mostly used on AMD, NVidia and Intel devices, which are central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. This platform greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories, including finance computations. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. Conformant implementations are available from Altera, AMD, Apple, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS.



A single compute device typically consists of several compute units, which in turn comprise multiple processing elements (PEs). A single kernel execution can run on all or many of the PEs in parallel. Vendors subdivide compute device into compute units and PEs. Compute unit is abstract of a "core", because the notion of core is hard to define across all the types of supported OpenCL devices. Also number of compute units may not correspond to the number of cores claimed in vendor marketing literature. OpenCL Platform model is based on host device and one or more compute devices. Another advantage is multi-GPU support, theoretically provides as much more performance. For example, system with 8 same GPUs will provide 8 times faster than

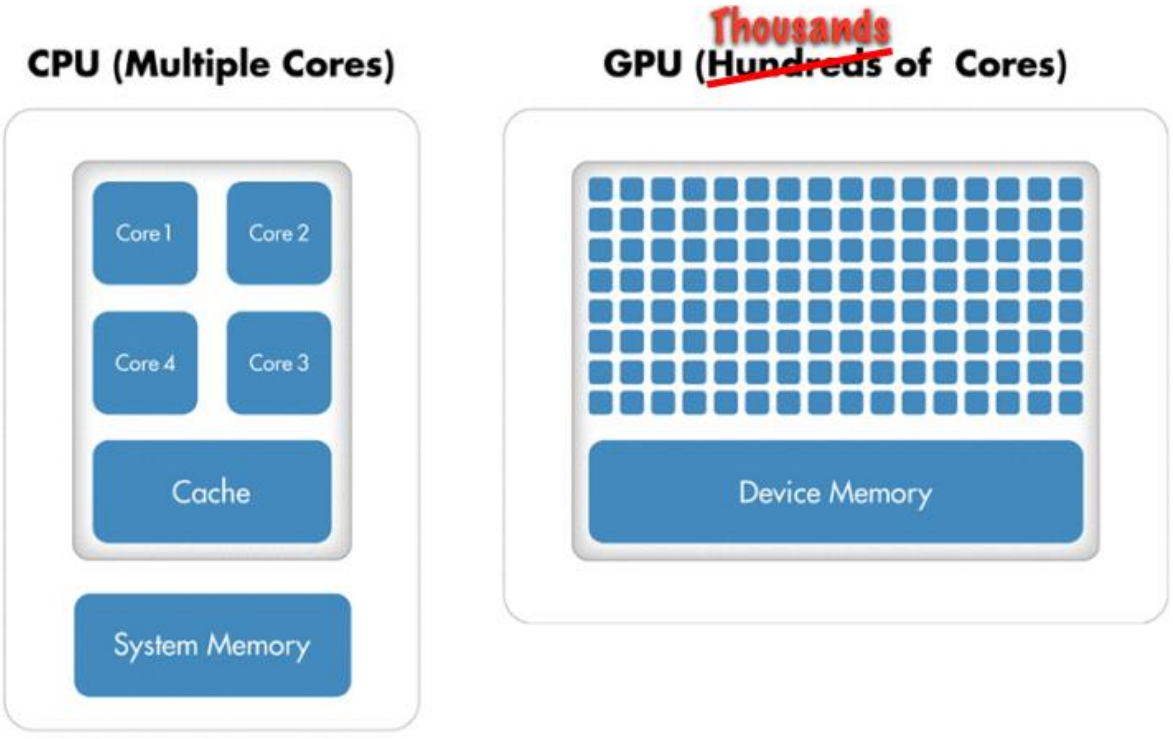
one. The access of data between CPU and GPU is much faster, because unified memory architecture. The GPU accelerated modules will provide at least 10 times faster calculation and it is quite possible to achieve 200 times speed up depending of applied parallel algorithm

OpenCL defines an application programming interface (API) that allows programs running on the host to launch kernels on the devices. Programs in the OpenCL language are intended to be compiled at run-time. It helps applications to be portable between implementations for various devices. Using the OpenCL API, developers can launch compute kernels written using a limited subset of the C99 programming language. For example:

- in OpenCL kernel is not allowed usage of recursion;
- Memory buffers reside in specific levels of the memory hierarchy, and pointers are annotated with the region qualifiers
- OpenCL C functions are marked __kernel to signal that they are entry points into the program to be called from the host program
- Function pointers, bit fields and variable-length arrays are omitted

Advantages of GPU devices over CPU are:

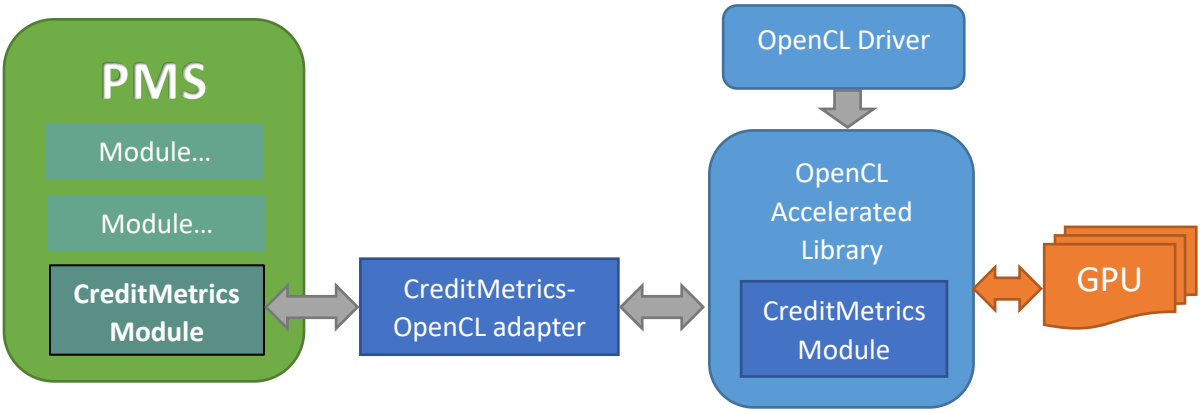
- Faster memory;
- More cores;
- Singe instruction multiple threads (SIMT) technology
- Cheap computing power(greater ratio of work per time)



OpenCL in PMS

OpenCL Concept

Implementation of CreditMetrics-OpenCL module in PMS is separated in separated DLL, because of better flexibility. Supported version for OpenCL driver is tested on version 1.2. Gathering data between Credit metrics and OpenCL is accomplished by adapter pattern. Main goal for adapter is get necessary data from PMS, like position and issuer data, and return MonteCarlo series from OpenCL accelerated library. PMS is linked explicitly to OpenCL library, because of easily management. To use this module, client must have OpenCL driver on GPU device. Currently OpenCL module is supported only on GPU module.



Turning on OpenCL acceleration of CreditMetrics in PMS, is accomplished by checkbox in CreditMetrics Panel. To use OpenCL you must set the OpenCLPlatform to desired choice in PMS Configuration panel in section “Analysis” and Section “CreditMetrics”. Mostly used configuration values for this configuration are:

- NVidia – for NVidia GPU’s with CUDA acceleration. In process of development we used mostly NVidia GTX card. Version of OpenCL is 1.2;
- AMD – for AMD GPU’s or AMD Accelerated Processing Unit (APU)
- Intel – not recommended, mostly Intel GPUs perform slower than OpenMP acceleration.

Module | **Section**

Module List: Analysis | Section List: CreditMetrics

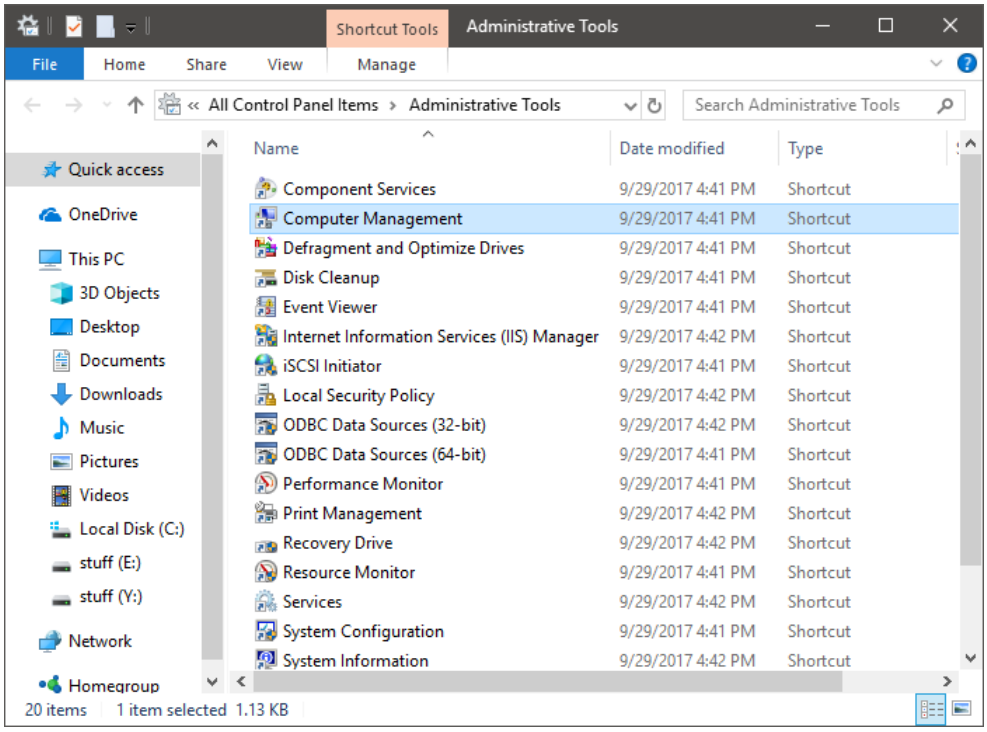
Module: Analysis | Section: CreditMetrics

Search

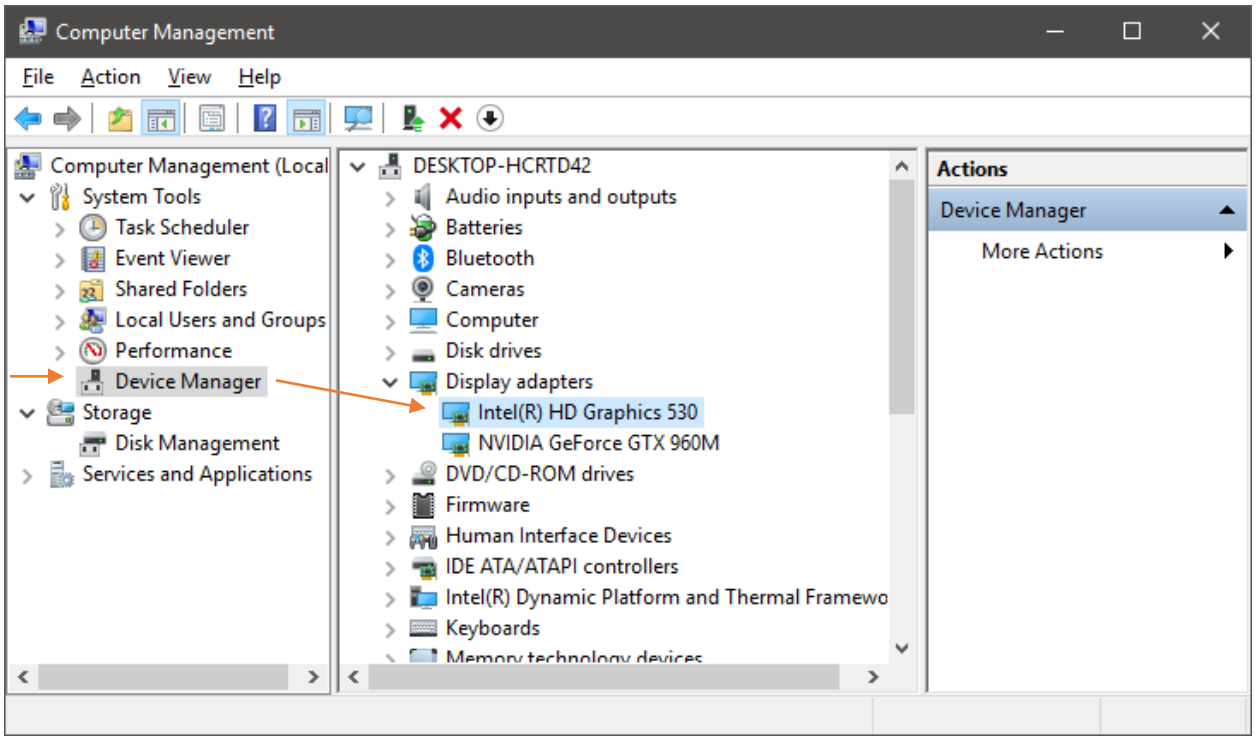
Configuration Item	Configuration Value	Global	User	Configuration Description
CalculateIncrementalVaRandMarginalVaRonIssuerLevel	Yes	☐		
ComponentVaRCalcBasedOnPositionVaR	No	☐		
IncludeIncrementalCVaR	Yes	☑		
IncludeMarginalCVaR	Yes	☑		
IncrementalDelta	0,001	☐		
IncrementalVaRandMarginalVaRnotNegative	Yes	☐		
MAXNumberOfMCRuns	100000000	☑		
OpenCLPlatform	NVIDIA	☑		
UseAdditiveIncrementalVaRMethod	No	☐		
UseCondensedSortforIVaRMVaR	Yes	☐		

OpenCL drivers can be checked by following several steps bellow:

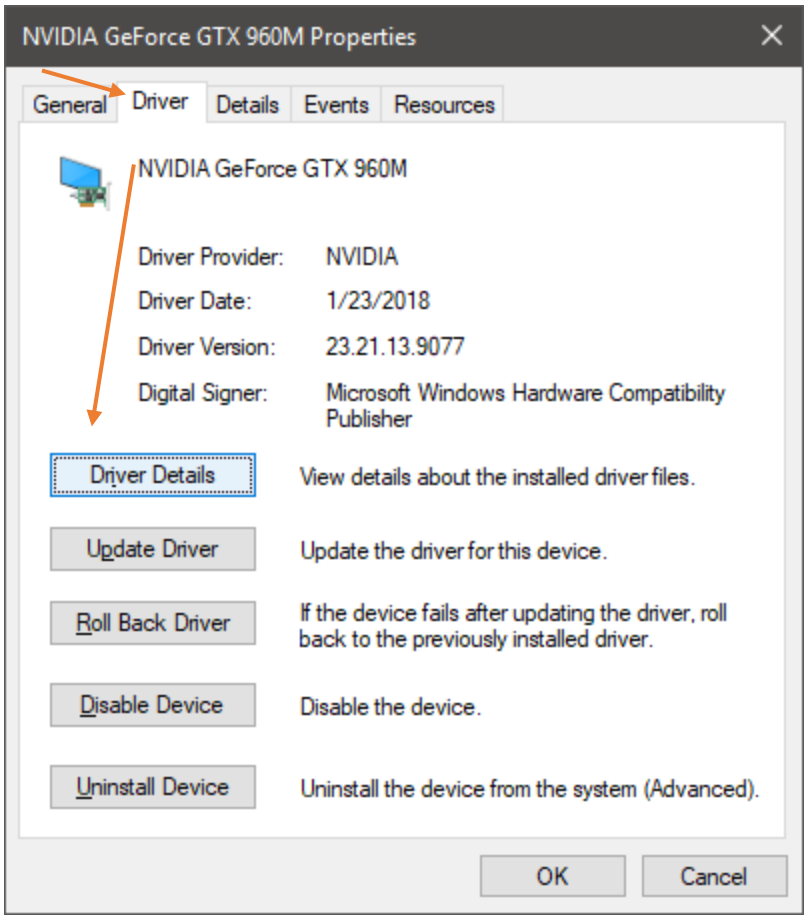
1. Go to Start -> Control Panel -> System & Security - > Administrative Tools
2. Double click on Computer Management



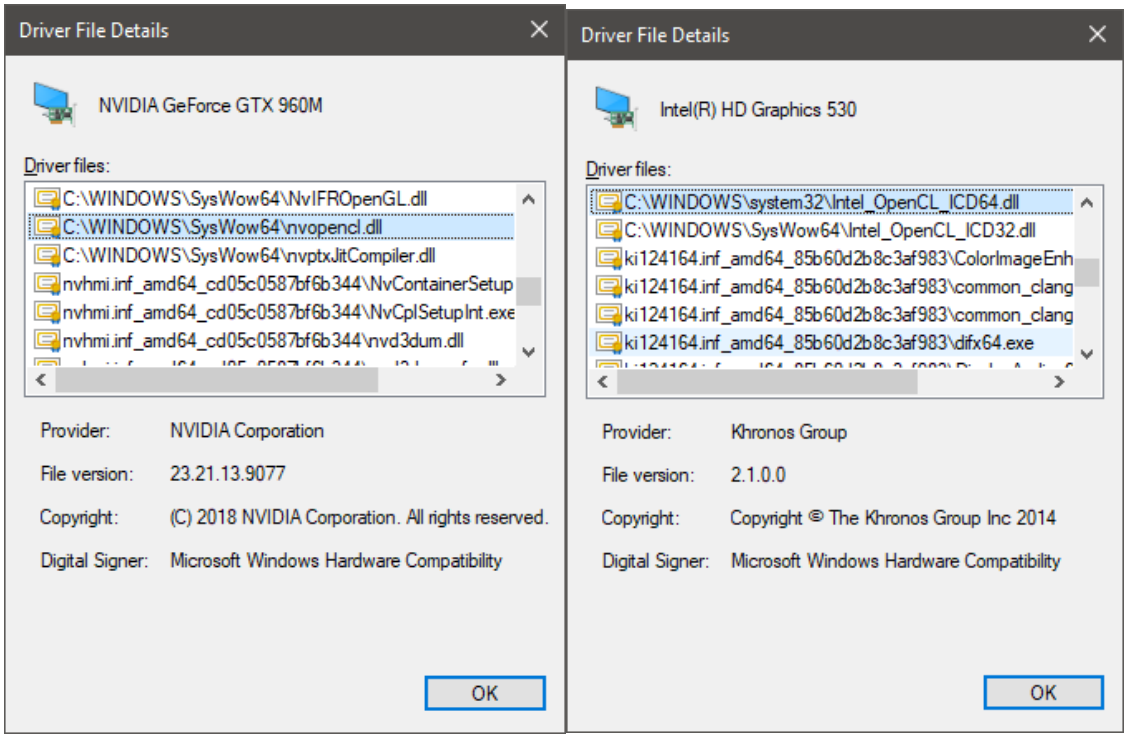
3. Click on Device Manager
4. Click open Display Adapters



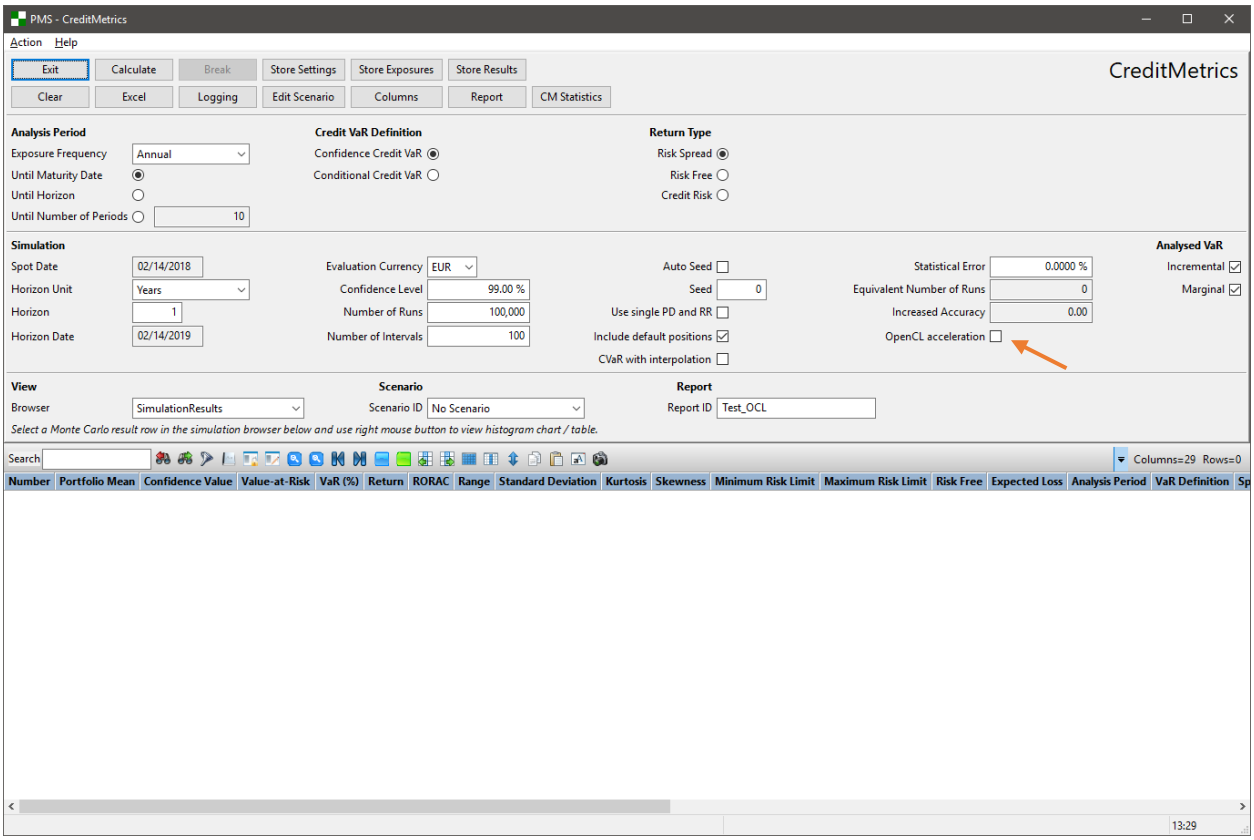
5. Right-click on available adapter and select Properties
6. Click on Driver
7. Go to Driver Details



8. Scroll down and see if OpenCL is installed (look for %OpenCL%.dll files)



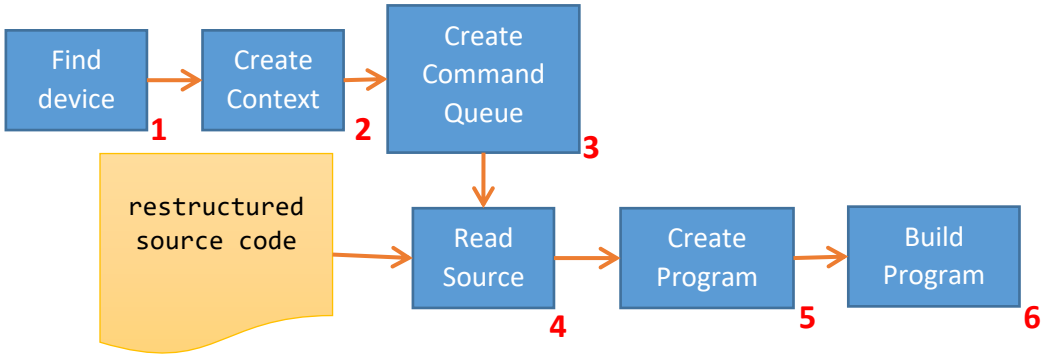
If adapters there are not available adapters or DLL is missing, try with install/update the graphic driver. Note: Do not install OpenCL drivers without first checking for known issues (e.g., some computer manufacturers install modified graphics drivers so replacing these may not be a good idea!). If in doubt, seek advice from an IT professional before proceeding further



OpenCL module initialization

In the lifecycle of one OpenCL module are needed preparations (figure N) before computations. First of all is needed to select platform. If there are error with desired platform (1), simulation will not start. For troubleshooting see section above. For now, module can compute with only one device. Device is chosen automatically by platform. In most cases, most of clients has relation one platform-one GPU device. Device can be represented by collection of compute units. With other words, that is GPU.

ERROR SCREENSHOT



When the system index is found, next step is to create context (2). Context is used by OpenCL runtime for managing command-queues, memory, program and kernel objects, that will be explained later. Also context schedule devices for executing kernels. Next step Is to create

command queue (3). It is object that holds commands that will be executed on specific device. The command-queue is created on a specific device in a context. Command that are proceed are queued in-order in module, but also can be executed out-of-order. Typically, in in-order execution commands are executed in order of submission with each command running to completion before the next one begins. In other case commands may begin and complete execution in any order consistent with constrains imposed by event want list and command-queue barrier. It this module is used only in-order executions. After this step is to read source code (4). Source code of OpenCL module is separated, from PMS and module. It can be in plain text file, binary file or like constant string object in source code of library. For security purpose – source code of OpenCL is placed inline in source code of module. It cannot use the PMS code, because of parallel reconstruction – will be explained bellow in kernel explanation. Final preparation step is creating (5) and building program (6). Program in OpenCL consist set of kernels. Programs may also contain auxiliary functions called by the __kernel functions and constant data. OpenCL allows applications to create a program object using the program source. Goal of this approach is the executable compiled/linked online as the program executable. This can be very useful as it allows applications to load and build program executables online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. Program object encapsulates the following information:

- A reference to an associated context
- A program source of binary
- The latest successfully built program executable. The list of devise for which the program executable is build and also build option, that are used, and logs
- The number of kernel object currently attached

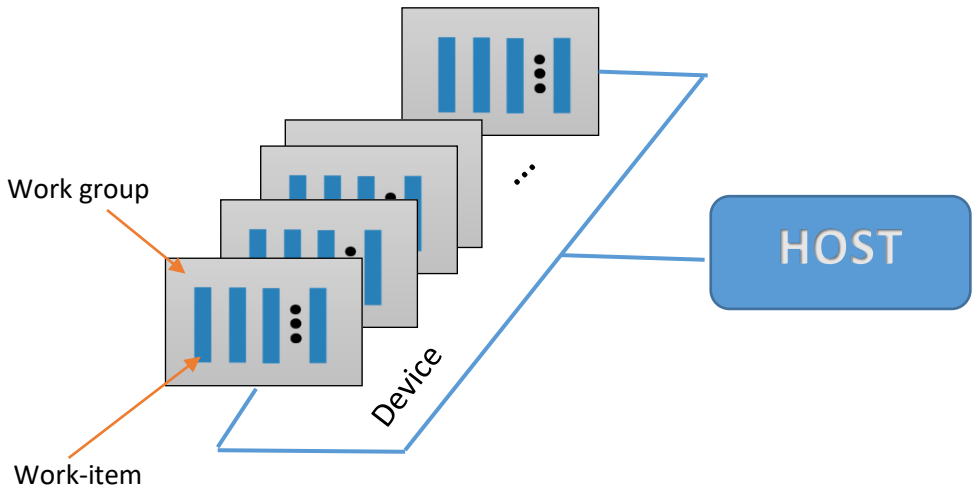
OpenCL Kernel execution

Kernel is a function declared in a program and executed on an OpenCL device. A kernel is identified by the __kernel qualifier applied to any function defined in a program. Kernel object encapsulates a specific __kernel function declared in program and the argument values to be used when executing it. In followed text bellow will be explained in details kernel preparation and execution of “Generation of random numbers” stage. First of all is need to release previous kernel and create a new one from program. Next step is go get information for kernel execution work items – work groups and local groups.

```
	if (m_ckKernel != nullptr) //check if kernel exist
	{
		// release previous kernel
		s_clError = clReleaseKernel(m_ckKernel);
	}
	// create new kernel from program with name and error code for status
	m_ckKernel = clCreateKernel(m_cpProgram, "init", &s_clError);
	//if all is success continue execution else return and log error
	if (check_error("cannot create kernel")) return;
```

An instance of the kernel executes for each point in index space, which provides a global ID for the work item. Each work-item executes the same code but the specific pathway through the code. Work items are organized into work groups. Work groups provide a more coarse-gained decomposition of index space. They are assigned to a unique work-group ID with the same dimensionality as the index space. Work items are assigned to a unique local ID within a work-

group, so that a single work item can be uniquely identified by its global ID or by combination of local ID and work group ID. The work item in a given group execute concurrently on the processing element of a single compute unit. Getting the optimal number of work items are key to gain maximum performance.

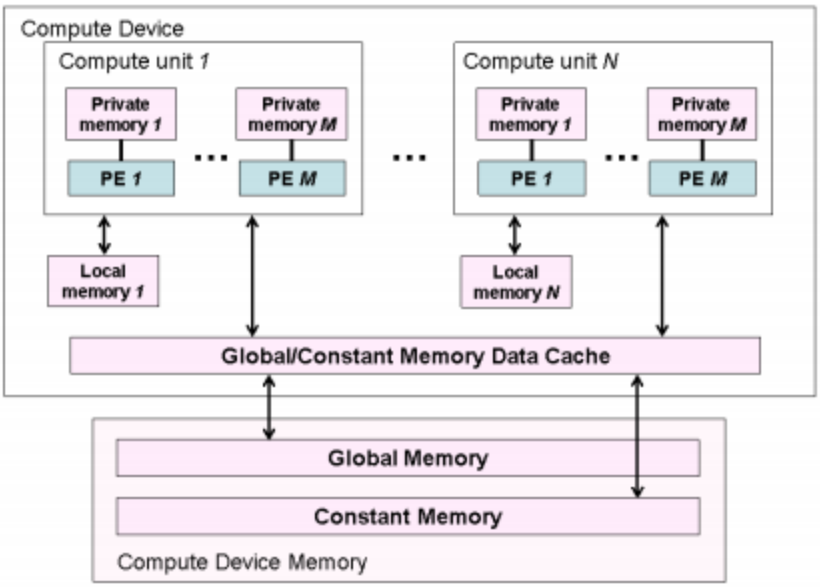


OpenCL provide a method, which gives an optimal workgroup and multiplier. Also module provides one more multiplier for high performance GPUs. Code bellow will create memory object for random number with rights for read and write, with size of variable number of randoms.

```
cl_mem memRes = clCreateBuffer(m_cxContext, CL_MEM_READ_WRITE, (m_uNumberOfRands)
*sizeof(cl_double), NULL, &s_clError);
```

In OpenCL work items executing a kernel have access to four distinct memory regions:

- Global memory – this memory region permits read/write access to all work groups. Work items can read or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device;
- Constant memory – this memory region of global memory, which remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory;
- Local memory – this memory region is local to a work group. It can be used to allocate variables that are shared by all work items in that work group. Mau be implemented as dedicated regions of memory on the device. With other words, the memory regions may be mapped onto sections of the global memory;
- Private memory - a region of memory per work item. Variables defined in one work items private memory are not visible to another work item



After getting the optimal work size, next step is to create memory objects for input and output parameters. In OpenCL kernel parameter can be only pointers or complex objects. Also all types of data must be in CL notation. If memory allocation, input buffer should be initialized. Initialization is by coping the data from HOST device to Device. That is needed, because of memory difference between CPU and GPU. In some cases, buffers are so big and it is needed synchronization of command queue. After writing and filling the parameters they should be assigned to kernel.

```
s_clError = clSetKernelArg(m_ckKernel, 0, sizeof(cl_mem), (void *)&memRes);
```

Last step is to enqueue task or with other word invoke the function. Here generation is accomplished apart. It is developed that way, because of TDR.

```
//calculatin limit per execution and optimal steps
m_confData.itersPerCore = l_limit / m_workgroup_size;
size_t ls_globalIters = ( ceilf((float)(m_uNumberOfRands)/l_limit));
for (size_t i = 0; i < ls_globalIters; i++)
{
	//moving the start cursor
	m_confData.startIter = i*l_limit;
	//write configuration stucture
	s_clError = clEnqueueWriteBuffer(m_cqCommandQueue, m_memConfData,
		CL_TRUE,0, sizeof(confDataOCL), &m_confData, 0, NULL, NULL);
	//enqueue task
	s_clError = clEnqueueNDRangeKernel(m_cqCommandQueue, m_ckKernel, 1,
		NULL, &m_workgroup_size, &m_local_item_size, 0, NULL, NULL);
	//synchronize
	s_clError = clFinish(m_cqCommandQueue);
}
```

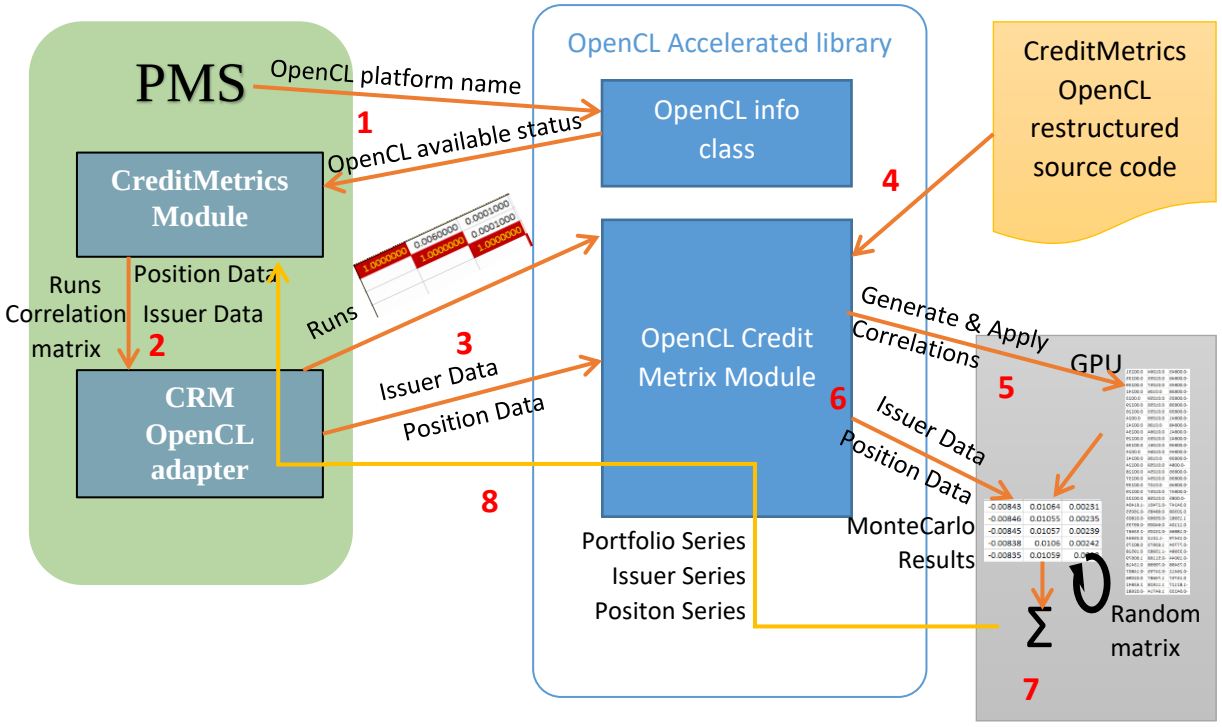
Then if necessary copy the needed data to Host memory and release memory objects.

```
s_clError = clEnqueueReadBuffer(m_cqCommandQueue, memRes, CL_TRUE, 0, (m_uNumberOfRands)
* sizeof(cl_double), l_result, 0, NULL, NULL);
```

At the end it need to release unnecessary data.

```
s_clError = clReleaseMemObject(memStates);
```

Module data flow



Basic concept for OpenCL modules is to get minimal data input result and return result. Before calculation PMS gives name of platform, which is stored in PMS configuration. In OpenCL library, there is a class for information. It has basic role to find desired platform and device, by the given name. If there are not available platforms or configuration name is incorrect, library will return error status, analysis will not be proceeded. Data flow between PMS CreditMetrics and OpenCL module is accomplished via adapter, which main role is to get and pass number of runs, correlation matrix, positon data and issuer data to DLL. Then library create buffers, copy needed data and build kernels. Just like was explained above, OpenCL module build code in runtime. That means source code is read and compiled, and then calculations will be started. Advantage of GPU is that it can take big data flows, because of good bandwidth. The less heavy part of algorithm is to generate random number and apply correlations. To get better results it needed this data to be corrected for zero correlation and perfect normal distributed form – standard deviation should be around 1.0 and mean 0.0. Then random numbers are applied by correlation matrix. Because of timers of some operation systems, e.g. Timeout Detection and Recovery (TDR) – which is problem of stability in graphic occurs when a computer “hangs” or appears completely “frozen”, while, in reality, it processing command or operations. The frozen appearance of the computer typically occurs because the GPU is busy processing intensive graphical operations, typically during game play. The GPU does not update the display screen, and the computer appears frozen. It is busy because of computations. Because of that problem OpenCL generates random number apart. Random number matrix stays in GPU memory until

analysis is finished. It most cases it takes the biggest part in memory distribution. MonteCarlo series are computed apart too.

As it was mention before, in OpenCL cannot exist object oriented principles. To accomplish credit metrics analysis first its needed to rewrite code for objects – analysis controller objects, as known as “configuration object”, model for issuer and position. In configuration object are common data for analysis, including:

- Runs;
- Dimensions;
- Issuer number;
- Start iterator – for kernels, that are executed apart;
- Include interpolation flag;
- Using single PD flag;
- Iterators per core (work item);

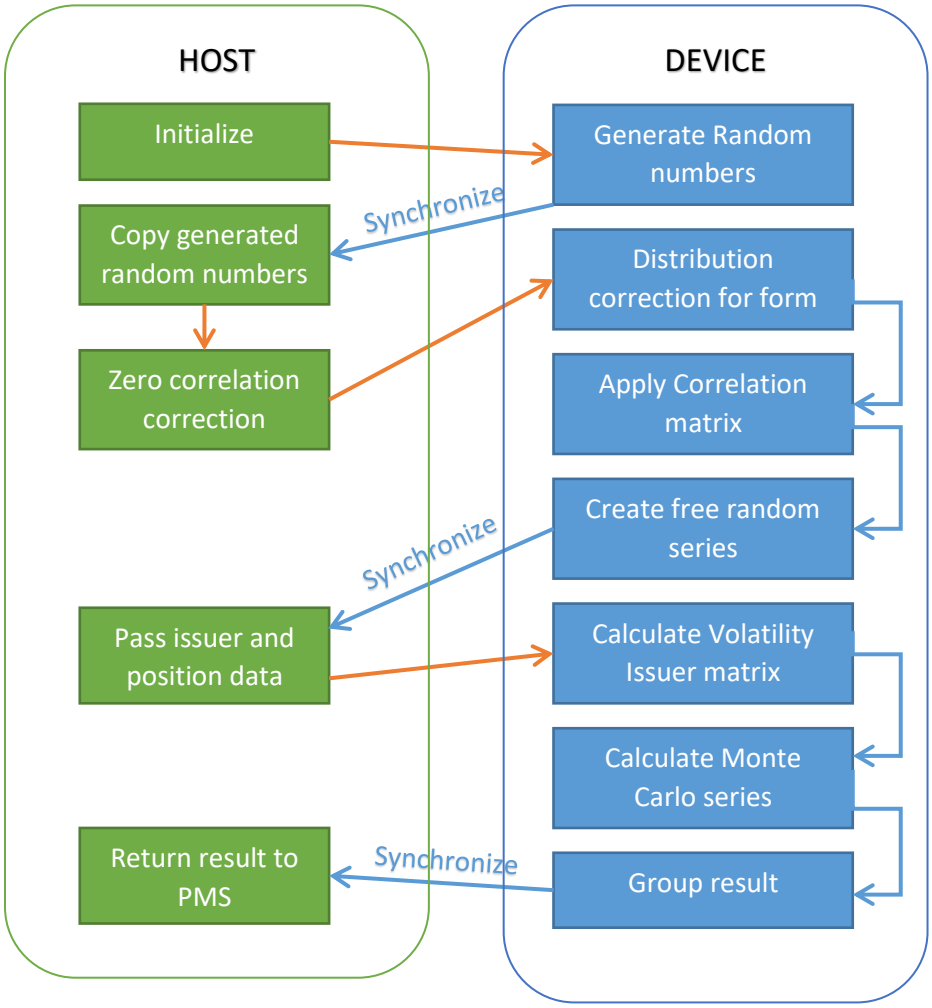
Members of structure for issuer are:

- Synthetic value;
- Synthetic volatility;
- No indices flag;
- Number of rating per issuer
- Used RR
- Beta distribution values
- Constant beta value
- Theshold values;

Members of structure for position are:

- Valid position;
- The first exposure;
- Rating values table;
- Collateral Absolute Value
- Collateral Coverage

Algorithm workflow



The algorithm schedule is separated in 2 main part generation of random series and calculations of Monte carlo simulation. In the start is initialization of OpenCL module. Then generation of random number module is enquired. Generation of random number is based on “XOR Shift“ Algorithm. This is pseudorandom number generators that were discovered by George Marsaglia. They are a subset of linear-feedback shift registers (LFSRs) which allow a particularly efficient implementation without using excessively sparse polynomials. They generate the next number in their sequence by repeatedly taking the exclusive or of a number with a bit-shifted version of itself. This makes them extremely fast on modern computer architectures. Like all LFSRs, the parameters have to be chosen very carefully in order to achieve a long period. Xorshift generators are among the fastest non-cryptographically-secure random number generators, requiring very small code and state. Although they do not pass every statistical test without further refinement, this weakness is well-known and easily amended (as pointed out by Marsaglia in the original paper) by combining them with a non-linear function. A xorshift* generator takes a xorshift generator and applies an invertible multiplication (modulo the word size) to its output as a non-linear transformation, as suggested by Marsaglia. The following 64-bit generator with 64 bits of state has a maximal period of $2^{64} - 1$

```
__private double xorshift64star(__global uint* state64) {
```

```
    size_t x = state64[0];  /* The state must be seeded with a nonzero value. */
    x ^= x >> 12; // a
    x ^= x << 25; // b
    x ^= x >> 27; // c
    state64[0] = x;
    return convert_double(x * 0x2545F4914F6CDD1D) / 0xffffffffffffffff;
}
```

For states each work group has own states. By default, seeds are based to be equally every simulation.

```
//all kernel functions must return void
__kernel void initLight(__global double *res, //matrix cannot be presented by 2 dimensons
    __global uint* stateUint,
    __global confDataOCL* pConfData)
{
    int idx = get_global_id(0); //get index of current core
    int g_size = convert_int(get_global_size(0)); //get number of all cores
    int j = 0;
    //private variables keyword __private is by default, like other languages
    double v1, v2,s, m;
    // iterations per core are set by Host
    for (j = 0; j <convert_int(pConfData[0].itersPerCore); j++){
        if(
            (j*g_size +idx + pConfData[0].startIter)>=
                ((pConfData[0].dimensions) * pConfData[0].runs))
        {
            break;
        }
        do {
            v1 = 2.0*xorshift64star(stateUint+5*idx)-1;
            v2 = 2.0*xorshift64star(stateUint+5*idx)-1;
            s=v1*v1+v2*v2;
        }
        while (s>=1.0);
        m=sqrt(-2.0*log(s)/s);
       //each result is placed by core index
        res[idx + j*g_size + pConfData[0].startIter] = v1*m;
    }
    return;
}
```

Next step of simulation is distribution correction. Distribution correction is separated in 3 parts – pre-distribution correction, bitonic sort and postdistribution correction. In the section bellow will be presented code reconstruction in details of distribution correction algorithm.

Because of thread concurrency, some fragments in module are reconstructed. It is not fully code duplication of source code. In PMS distribution correction of normal distribution is accomplished by one single function. OpenCL approach is different. The idea of correction is to apply transformation of series to commutative distributed series. Then sort it, but keeping the original order, adjust series by order number and return it back to normal distribution and original order.

```
distributionCorrection(double* p_pSeries,
                       size_type p_Runs,
```

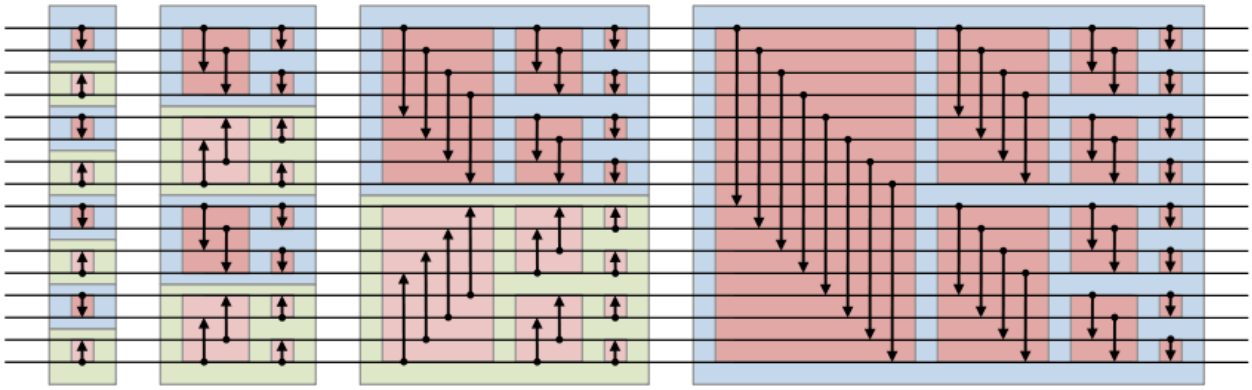
```
                        size_type p_Dim)
{
    const size_type br = p_Runs;
    // array for storing the order of series
    int* l_vOrd = new int[br];
    double l_dOff = (double)1/(double)br;
    double l_dST  = (double)1/(double)br/2.0;
    int i, j;
    for(j = 0; j < p_Dim; j++)
    {
#pragma omp parallel for shared(i, j, l_vOrd)
            for(i = 0; i < br; i++)
            {
                p_pSeries[j + i * p_Dim] =
                    Normal::cdf(p_pSeries[j + i * p_Dim]);
            }

#pragma omp parallel for shared(i, j, l_vOrd)
            for(i = 0; i < br; i++)
            {
                l_vOrd[i] = i;
            }
```

To gain better performance in OpenCL first stage of distribution correction is merged for all dimensions. Action between all series values are independent. This is perfectly suitable for massive parallel work. Each work item will apply cumulative distribution function on several series values for all dimensions. At the final of this stage will be created orders. Orders are from zero to number of runs. Because of sorting algorithm, which is sorting behavior is better, when numbers are grade of 2, is added fake transformation of needless number. With other words, used technology is state-full. As opposed to OpenMP, OpenCL paralelism is strictly defined. Each compute unit will compute several values, there are not available concurrency. This is one more reason for rewriting and reconstruction code. CDF function is and probability distribution function are private, with other words each work group will execute own instance of function.

```
__kernel void preDistributionCorrection(__global double* p_pSeries, __global confDataOCL*
pConfData, __global int* orders)
{
    uint localId = get_local_id(0);
    int idx = get_global_id(0);
    int globalSize = convert_int(get_global_size(0));
    int currIndex;
    for (int d = 0; d<( pConfData[0].dimensions); d++)
    {
        for ( int i = 0 ; i <pConfData[0].itersPerCore; i++)
        {
            if((idx*pConfData[0].itersPerCore+i)>=pConfData[0].runs)
            {
                p_pSeries[currIndex] = 2.0;
                continue;
            }
```

```
            currIndex  =d+(idx*pConfData[0].itersPerCore+i)*(pConfData[0].dimensions);
            p_pSeries[currIndex]=cdf(p_pSeries[currIndex]);
            orders[idx*pConfData[0].itersPerCore+i] = idx*pConfData[0].itersPerCore+i;
        }
    }
}
```

After applying CDF and creating order array, next part is sorting. In PMS is used quick sort with sorting values in pair with order. Quick sort cannot be applied in parallel. Possible approach is to sort apart, but it will be needed more time for merging sub arrays. Another problem is that quicksort is recursion and stack based, which cannot fit in OpenCL library. To sort series in OpenCL kernels, we found a suitable algorithm for massive parallel work and transform it to work with pairs. Sort algorithm is known as Bitonic Sort. Bitonic mergesort is a parallel algorithm for sorting. It is also used as a construction method for building a sorting network. The algorithm was devised by Ken Batcher. The following is a bitonic sorting network with 16 inputs:



The 16 numbers enter at the inputs at the left end, slide along each of the 16 horizontal wires, and exit at the outputs at the right end. The network is designed to sort the elements, with the largest number at the bottom.

The arrows are comparators. Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number. If they are out of order, they are swapped. The colored boxes are just for illustration and have no effect on the algorithm. If the inputs happen to form a bitonic sequence, then the output will form two bitonic sequences. The top half of the output will be bitonic, and the bottom half will be bitonic, with every element of the top half less than or equal to every element of the bottom half (for dark red) or vice versa (for light red). This theorem is not obvious, but can be verified by carefully considering all the cases of how the various inputs might compare, using the zero-one principle. The red boxes combine to form blue and green boxes. Every such box has the same structure: a red box is applied to the entire input sequence, then to each half of the result, then to each half of each of those results, and so on. All arrows point down (blue) or all point up (green). This structure is known as a butterfly network. If the input to this box happens to be bitonic, then the output will be completely sorted in increasing order (blue) or decreasing order (green). If a number enters the blue or green box, then the first red box will sort it into the correct half of the list. It will then pass through a smaller red box that sorts it into the correct quarter of the list within that half. This continues until it is sorted into exactly the correct position. Therefore, the output of the green or blue box will be completely sorted.

Bitonic Sort host code:

```
for (indecies[1] = 2; indecies[1] <= lRuns; indecies[1] <<= 1)
{
    /* Minor step */
    for (indecies[0] = indecies[1] >> 1; indecies[0]>0; indecies[0] = indecies[0] >> 1) {
        //bitonic_sort_step << <blocks, threads >> >(dev_values, j, k);
        s_clError += clEnqueueNDRangeKernel(m_cqCommandQueue, m_ckBitonicSortKernel, 1, NULL,
&m_workgroup_size, &localGroups, 0, NULL, NULL);
        s_clError += clFinish(m_cqCommandQueue);
        s_clError += clEnqueueWriteBuffer(m_cqCommandQueue, cl_inc, CL_TRUE, 0,
sizeof(cl_uint) *INDECIES_ARR_SIZE, indecies, 0, NULL, NULL);
    }
}
```

Bitonic Sort device code:

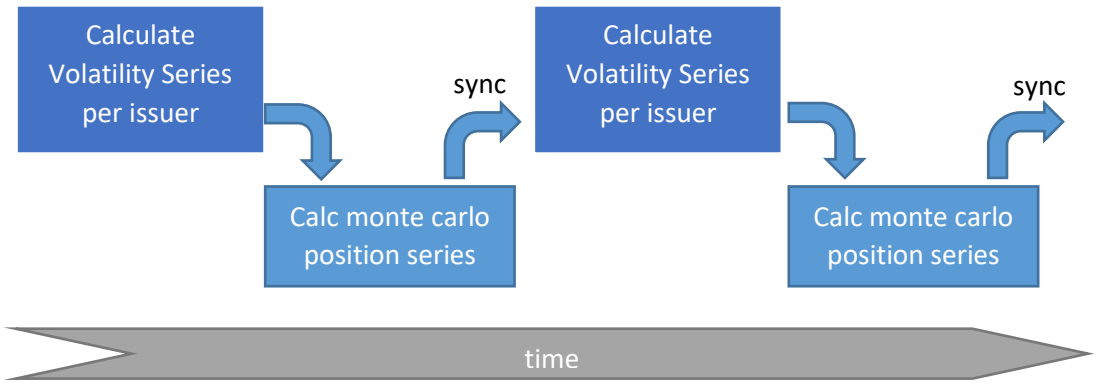
```
__kernel void bitonicSortStep(__global double* dev_values, __global int* indecies, __global
int* orders)
{
    uint localId = get_local_id(0);
    uint globalId = get_global_id(0);
    uint groupSize = get_global_size(0);
    //int core = (localId + globalId*groupSize);
    for (uint  idx = 0 ; idx < indecies[2]; idx++)
    {
        uint i, ixj,o,oxj; /* Sorting partners: i and ixj */
        i =globalId + idx*groupSize;
        //i = o*indecies[5] + indecies[4];

        ixj = i^indecies[0];
        //oxj = o^indecies[0];
        if (oxj > indecies[3] ){
            continue;
        }
        if ((ixj) > i) {
            if ((i&indecies[1]) == 0) {
                /* Sort ascending */
                if (dev_values[i*indecies[5] + indecies[4]] > dev_values[ixj*indecies[5] +
indecies[4]]) {
                    /* exchange(i,ixj); */
                    double temp = dev_values[i*indecies[5] + indecies[4]];
                    dev_values[i*indecies[5] + indecies[4]] = dev_values[ixj*indecies[5] +
indecies[4]];
                    dev_values[ixj*indecies[5] + indecies[4]] =temp;
                    //XOR swap
                    orders[i]^= orders[ixj];
                    orders[ixj]^=orders[i];
                    orders[i]^= orders[ixj];
                }
            }
            if ((i&indecies[1]) != 0) {
                /* Sort descending */
```

```
                if (dev_values[i*indecies[5] + indecies[4]] < dev_values[ixj*indecies[5] +
indecies[4]]) {
                    /* exchange(i,ixj); */
                    double temp = dev_values[i*indecies[5] + indecies[4]];
                    dev_values[i*indecies[5] + indecies[4]] = dev_values[ixj*indecies[5] +
indecies[4]];
                    dev_values[ixj*indecies[5] + indecies[4]] = temp;
                    //XOR swap
                    orders[i]^= orders[ixj];
                    orders[ixj]^=orders[i];
                    orders[i]^= orders[ixj];
                }
            }
        }
    }
}
```

Final step of distribution correction is to apply correction to commutative values and inverse it to normal distributed values and return it to initial order to restore original correlation between dimensions. Code is modeled just like first stage of correction. Each work item compute inverse CDF and restore the order. For inverse commutative is used alternative approach. Main goal is performance. It uses approximation based on polynomial coefficients.

```
double icdf(double p)
{
    const double a1 = -39.69683028665376;
    const double a2 = 220.9460984245205;
    const double a3 = -275.9285104469687;
    const double a4 = 138.3577518672690;
    const double a5 = -30.66479806614716;
    const double a6 = 2.506628277459239;

    const double b1 = -54.47609879822406;
    const double b2 = 161.5858368580409;
    const double b3 = -155.6989798598866;
    const double b4 = 66.80131188771972;
    const double b5 = -13.28068155288572;

    const double c1 = -0.007784894002430293;
    const double c2 = -0.3223964580411365;
    const double c3 = -2.400758277161838;
    const double c4 = -2.549732539343734;
    const double c5 = 4.374664141464968;
    const double c6 = 2.938163982698783;

    const double d1 = 0.007784695709041462;
    const double d2 = 0.3224671290700398;
    const double d3 = 2.445134137142996;
    const double d4 = 3.754408661907416;

    //Define break-points.
```

```
    const double p_low = 0.02425;
    double p_high = 1 - p_low;
    double  q, r, e, u;
    double x = 0.0;

    //Rational approximation for lower region.

    if (0 < p && p < p_low) {
        q = sqrt(-2 * log(p));
        x = (((((c1*q + c2)*q + c3)*q + c4)*q + c5)*q + c6) / ((((d1*q + d2)*q + d3)*q + d4)*q
+ 1);
    }
    if (p_low <= p && p <= p_high) {
        q = p - 0.5;
        r = q*q;
        x = (((((a1*r + a2)*r + a3)*r + a4)*r + a5)*r + a6)*q / (((((b1*r + b2)*r + b3)*r +
b4)*r + b5)*r + 1);
    }

    if (p_high < p && p < 1) {
        q = sqrt(-2 * log(1 - p));
        x = -(((((c1*q + c2)*q + c3)*q + c4)*q + c5)*q + c6) / ((((d1*q + d2)*q + d3)*q +
d4)*q + 1);
    }
    //Pseudo-code algorithm for refinement

    if ((0 < p) && (p < 1)) {
        e = 0.5 * erfc(-x / 1.41421356237) - p;
        u = e * sqrt(2 * 3.14159265359) * exp(x*x / 2);
        x = x - u / (1 + x*u / 2);
    }
    return x;
}
```



The main reason for this separation, is lack of memory. Performance is slower, memory for GPU is limited. High class GPU has around 8-12 GB. First algorithm calculates the volatility series for issuer and runs, then value for position. This is repeated until is not reached end of input series matrix. After each calculation of position series, work items are synchronized.

Monte carlo simulation Host code:

```
for (int i = 0; i <l_iters;i++)
{
    //reset result aggregation buffers to 0 values
    s_clError = clEnqueueFillBuffer(m_cqCommandQueue, l_memResult, &initValue,
sizeof(cl_double), 0, sizeGrid * sizeof(cl_double), 0, NULL, NULL);
    s_clError = clEnqueueFillBuffer(m_cqCommandQueue, l_memIssuerResult, &initValue,
sizeof(cl_double), 0, (m_confData.issuerNumber*sizeGrid) * sizeof(cl_double), 0, NULL, NULL);
    s_clError = clEnqueueFillBuffer(m_cqCommandQueue, l_memPositionResult, &initValue,
sizeof(cl_double), 0, (m_confData.positionNumber*sizeGrid) * sizeof(cl_double), 0, NULL,
NULL);

    //invoke function for vola series
    s_clError = clEnqueueNDRangeKernel(m_cqCommandQueue, m_ckKernel, 1, NULL, &sizeGrid,
&m_local_item_size, 0, NULL, NULL);
    if (check_error("cannot finish kernel in calctrue mc vola issuers")) return;

    //invoke function for MC resutls
    s_clError = clEnqueueNDRangeKernel(m_cqCommandQueue, l_ckMcKernel, 1, NULL, &sizeGrid,
&m_local_item_size, 0, NULL, NULL);
    s_clError = clFinish(m_cqCommandQueue);
    if (check_error("cannot finish kernel in MC calc")) return;

    //copy resut to host memory
    s_clError = clEnqueueReadBuffer(m_cqCommandQueue, l_memResult, CL_TRUE, 0, (finalRuns)*
sizeof(cl_double), p_dbResultPtr + sizeGrid*i, 0, NULL, NULL);
    s_clError = clEnqueueReadBuffer(m_cqCommandQueue, l_memIssuerResult, CL_TRUE, 0,
(m_confData.issuerNumber*finalRuns) * sizeof(cl_double), p_dbIssuerResultPtr + sizeGrid*i, 0,
NULL, NULL);
    s_clError = clEnqueueReadBuffer(m_cqCommandQueue, l_memPositionResult, CL_TRUE, 0,
(m_confData.positionNumber*finalRuns) * sizeof(cl_double), p_dbPositionResultPtr + sizeGrid*i,
0, NULL, NULL);
    s_clError = clFinish(m_cqCommandQueue);
}
```

Reconstruction in code for Monte Carlo run simulation is based on changed structures and results. Another point is linear presentation of 2 dimensional array.

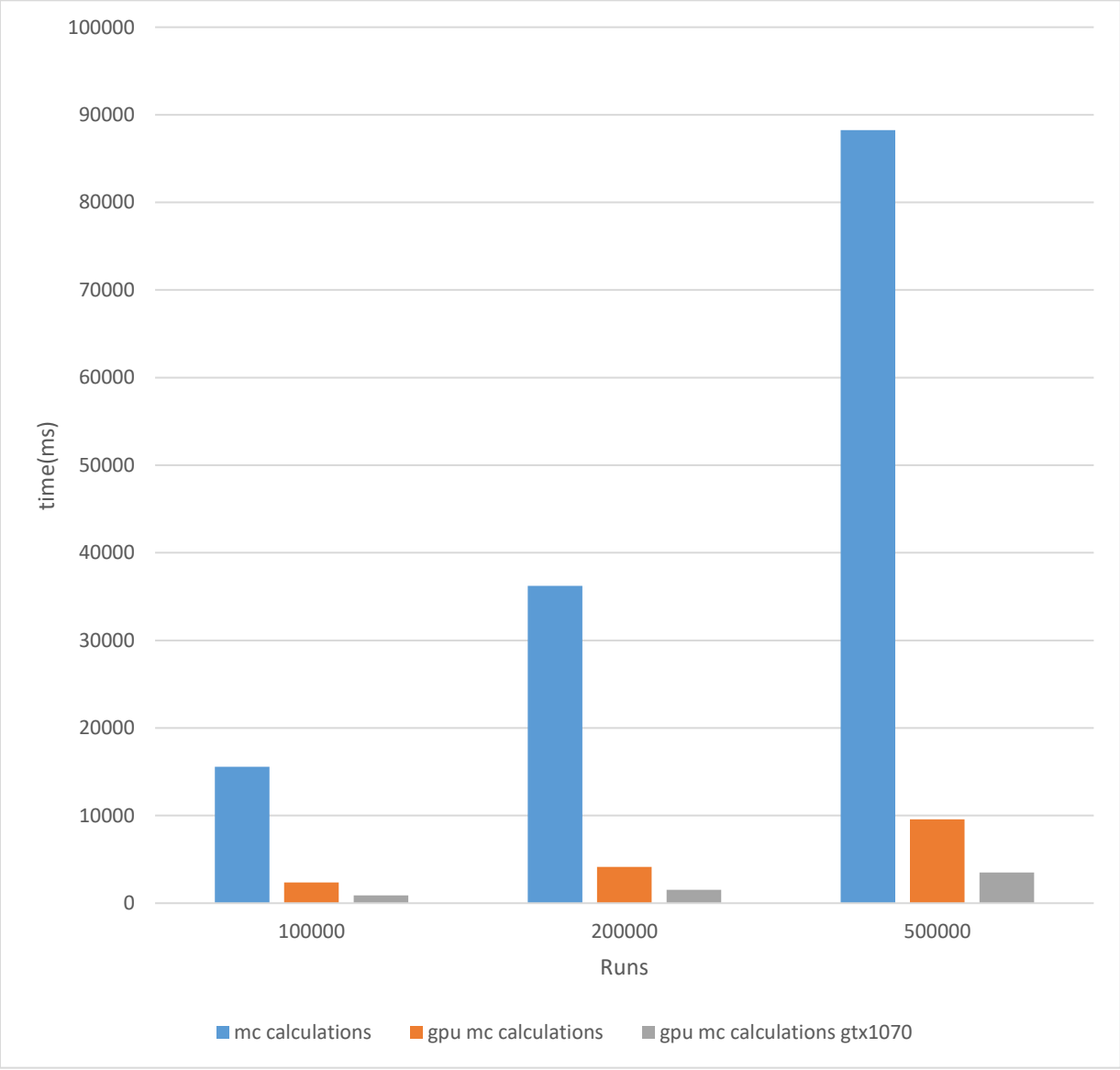
Performance & benchmarks

Performance aspects is accomplished as expected. On single position OpenCL perform up to 270 times faster than CPU multithreaded approach. Machines CPU is intel i7 6700 – 4 threads, 8 cores. Result are shown below.

runs	mc calculations	gpu mc calculations	mc calculation speed up
100000	2816	19	148
200000	5007	24	209
500000	14651	53	276

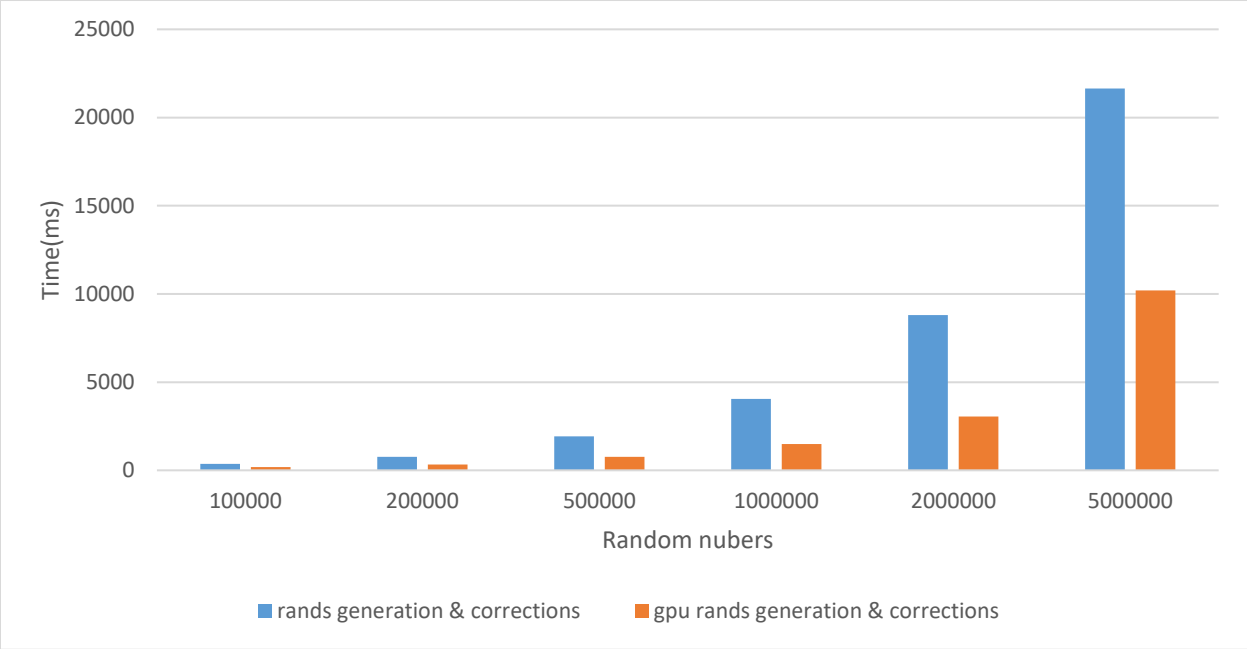
1000000	27699	101	274
2000000	53391	206	259
5000000	131958	513	257
10000000	280155	1023	274
20000000	530500	2032	261

Performance is getting slower by increasing positions and dimensions. At 275 positions and 55 dimension we get improved time. We try this test with medium range GPU - NVidia GTX 1070 and laptop GPU – NVidia 960m. From hardware specification GTX 1070 has 1920 cores and 960 has 640. This means approximates 3 times faster calculations. Because of software and driver issues we expect GTX 1070 get 2.5 times faster than 960m. Results are shown below.



runs	mc calculations	GTX960M mc calculations	GTX1070 mc calculations	Speed up CPU/GTX960m	Speed up CPU/GTX1070
100000	15582	2379	881	6.549810845	17.68671964
200000	36234	4160	1519	8.710096154	23.85385122
500000	88262	9563	3498	9.229530482	25.23213265

Algorithm is also faster in generating random series ND(0,1) and applying correlation. In the diagram bellow are shown benchmarks.



Future development

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

Future suitable development algorithms:

- Longstaff-Schwartz method on GPU
- Monte Carlo VaR
- Stochastic Volatility Modeling
- Large-Scale Interest-Rate Swaps Risk
- Derivatives simulation, ABS/SPV simulation
- Complex EOT simulation