Real-Time Financial Risk Measurement of Dynamic Complex Portfolios with Python and PyOpenCL

Javier Alejandro Varela
University of Kaiserslautern,
Microelectronic Systems Design Research Group
Kaiserslautern, Germany
varela@eit.uni-kl.de

Norbert Wehn
University of Kaiserslautern,
Microelectronic Systems Design Research Group
Kaiserslautern, Germany
wehn@eit.uni-kl.de

Sascha Desmettre
University of Kaiserslautern,
Department of Mathematics
Kaiserslautern, Germany
desmettre@mathematik.uni-kl.de

Ralf Korn
University of Kaiserslautern,
Department of Mathematics
Kaiserslautern, Germany
korn@mathematik.uni-kl.de

ABSTRACT

Risk measures, such as value-at-risk and expected shortfall, are widely used to keep track of the risk at which a financial portfolio is exposed. This analysis is not only a key part of the daily operation of financial institutions worldwide, but it is also strictly enforced by regulators. While nested Monte Carlo simulations are the most flexible approach that can even deal with portfolios containing complicated derivatives, they traditionally suffer from a high computational complexity. This limits their application at certain intervals of time, mostly daily, by temporarily keeping the composition of the portfolio static.

In this work, we bring together for the first time nested Monte Carlo simulations with the real-time continuous risk measurement of complex portfolios that dynamically change their composition during intraday operation. By combining the development productivity offered by Python, state-of-the-art mathematical optimizations, and the high performance capabilities offered by PyOpenCL targeting heterogeneous computing systems, our new approach reaches a throughput between 16 and 191 trading orders per second per computing node, which corresponds to the worst-case and best-case scenarios respectively.

We have also made use of the Jupyter Notebook, as an interactive interface in an interdisciplinary research environment.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Applied computing** → *Mathematics and statistics*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

KEYWORDS

real-time portfolio risk management, finance, nested Monte Carlo simulations, value-at-risk, expected shortfall, component value-at-risk, python, pyopencl, jupyter notebook

ACM Reference Format:
Javier Alejandro Varela, Norbert Wehn, Sascha Desmettre, and Ralf Korn. 2017. Real-Time Financial Risk Measurement of Dynamic Complex Portfolios with Python and PyOpenCL. In *PyHPC'17: PyHPC'17: 7th Workshop on Python for High-Performance and Scientific Computing, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3149869.3149872

PyHPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5124-9/17/11…$15.00
https://doi.org/10.1145/3149869.3149872

1 INTRODUCTION

Python is a general-purpose and high-level interpreted programming language widely used in academic research and in the financial industry [8, 10]. Python exibits a very large design productivity, and due to its elegant syntax and dynamic typing, it is widely used for scripting, rapid prototyping and for developing sustainable software systems. However, Python lacks in implementation efficiency, i.e. runtime performance, compared e.g. to C and Fortran [3].

But design productivity is highly required in applications that evolve over time, such as the continuous risk measurement of complex financial portfolios. The latter are usually composed of a set of financial products, each with its own associated pricing algorithm. This constitutes a highly heterogeneous application, which also scales as new products are offered, or new algorithms are designed.

The regular monitoring of risk measures, such as the ones highlighted in [22], is not only enforced by regulatory agencies following the financial crisis of 2008 [23], but it is also a key part of the daily operation of any financial institution. Value-at-risk (VaR) and expected shortfall (ES) (also called conditional value-at-risk (cVaR)) are risk measures widely used with financial portfolios [12]. The total risk can also be decomposed into the individual contribution of each product in the portfolio, by means of component value-at-risk (compVaR) and component expected shortfall (compES) [20].

The use of (nested) Monte Carlo (MC) simulations constitutes the most general and flexible approach for portfolio risk measurement [19], although the computational cost is considerably high, since the complete portfolio is revaluated in every run [18]. For this reason,

this approach is traditionally limited to be carried out at fixed time intervals, with a temporarily *static* portfolio composition [6].

In this work, our goal is to continuously measure and update risk measures during intraday operation, on complex portfolios that *dynamically* change their composition with every incoming trading order. And we do so by bringing the compute-intensive nested MC approach together with real-time, continuous risk measurement.

This raises new design challenges, and requires a high performance computing system and design. To this end, Python also offers modules and extensions to other libraries. PyMP brings Open Multi-Processing (OpenMP)-like functionality for multiprocessor-based parallel applications, allowing a readily parallelization of existing serial code. For even higher performance, PyOpenCL provides Python with access to the Open Computing Language (OpenCL) application programming interface (API), in order to create portable parallel code targeting heterogeneous computing systems. Such systems are usually central processing unit (CPU)-based, and include accelerators such as graphics processor unit (GPU), and Intel's many integrated core (MIC) architecture (called Xeon Phi).

By combining the productivity offered by Python, together with the recent state-of-the-art mathematical optimizations on VaR/ES, and the high performance delivered by PyOpenCL on heterogeneous computing systems, we bridge the gap between productivity and efficiency. This way we achieve a speedup of over 6 orders of magnitude compared to the pure Python implementation, and it enables us to reach throughputs in the range of 16 to 191 trading-orders/s on complex financial portfolios and under compute-intensive nested MC simulations.

In summary, we provide detailed analysis and results on:

- the combination of Python's programming productivity and the high-performance offered by PyMP (as an intermediate step) and PyOpenCL (for real-time execution),
- a new approach that brings together compute-intensive nested MC simulations with real-time operation, in order to continuously measure and update widely used risk measures on complex financial portfolios that dynamically change their composition during intraday operation,
- a test case of an interdisciplinary research between mathematics and computer engineering that has profited from the use of the Jupyter Notebook, for development, debugging, and live code testing, as well as the use of a Git repository for remote access, as shown in Fig. 1,
- a flexible design that can be run on multiple platforms, from single multiprocessor CPU to more complex heterogeneous computing systems, with a set of implemented financial products that can be easily extended.

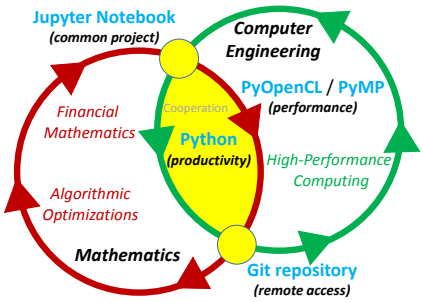


Figure 1: Research cooperation between financial mathematics and engineering, using Python (for productivity), PyOpenCL / PyMP (for performance), Jupyter Notebook (common shared project) and a Git repository (for remote access).

2 BACKGROUND AND RELATED WORK

Python is an easy-to-learn, interpreted programming language. Since it does not require compilation and linking (thanks to the interpreter), it greatly reduces development time. It also has efficient high-level data structures and a simple, yet effective, approach to object-oriented programming. Also due to its elegant syntax and dynamic typing, it is widely used for scripting and rapid prototyping [10]. As mentioned previously in Section 1, this high design productivity comes with an associated lack in implementation efficiency (runtime performance) [3].

Programs written in Python tend to be compact and easily readable: high-level data types allow the programmer to express complex operations in a single statement, indentation is used for statement grouping, and no variable or argument declarations are necessary [9]. In our case, the main host code written in Python became more tractable than preliminary work using C/C++/OpenCL.

Python programs can be split into modules that can be later reused in other programs, and it also comes with a large collection of standard modules. Among the latter is *numpy*, a module optimized for numerical computation. Python is also extensible, in particular to tackle performance-critical operations that require maximum throughput, or to link Python programs to other optimized libraries. Two such extensions used in this work are PyMP and PyOpenCL.

In the coming subsections we cover the latter extensions, followed by the Jupyter Notebook. We then introduce the main foundations of our application, and proceed to explain our new approach. Finally, we cover basic principles of option pricing and recent state-of-the-art algorithmic optimizations.

2.1 OpenMP and PyMP

Open Multi-Processing (OpenMP) is an API that enables the creation of shared-memory parallel programs targeting multiprocessor systems, and it is managed by the nonprofit technology consortium OpenMP Architecture Review Board. It provides features that can be added to a sequential program (such as C/C++), describing how the work is to be allocated among the different threads, and to order the accesses to shared data whenever required. In simple terms, the section of code to run in parallel is basically marked with a compiler directive, and each thread is provided an id that can be retrieved. The main advantage of OpenMP is its ease of use and the ability to readily parallelize existing sequential code [4].

PyMP is a Python package that brings OpenMP-like functionality to Python code [17] (see Fig. 2). They share similar syntax, but OpenMP uses light weighted threads, while PyMP forks different Python processes for multicore parallelism. Communication between these processes is accomplished via shared memory (through references), or using a manager to communicate serialized objects in the Python format called Pickle [17]. Parallelism through PyMP has therefore more overhead than OpenMP.

2.2 OpenCL and PyOpenCL

Open Computing Language (OpenCL) is an open, royalty-free standard managed by the nonprofit technology consortium Khronos Group, and it is designed for general purpose parallel programming, targeting heterogeneous computing systems [11]. Code written in OpenCL is in general portable across different platforms, greatly reducing the programming effort.

This framework assumes the presence of a host CPU, and an accelerator (called OpenCL device) connected to it via Peripheral Component Interconnect Express (PCIe). As shown in Fig. 2, an OpenCL device contains *compute units*, each subdivided into *processing elements*. The host code is responsible for the execution of all the OpenCL steps, including the distribution of workload to the device and the transfers of data with it. Fig. 2 also shows the main memory hierarchy.

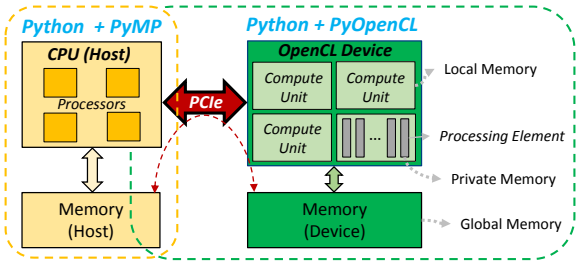


Figure 2: Heterogeneous computing systems using Python (for productivity) and its extensions (for performance).

The device will then execute the OpenCL *kernels*, which are enqueued by the host either as a single-threaded *Task*, or as an *N-Dimensional Range* (NDRange) with a defined number of *work-items* (called *global-size*) grouped into *work-groups* of user-defined size (called *local-size*). Buffers on device global memory are predeclared by the host. These buffers will be used by the *kernel* during computation, and by the host for all data transfers with the device. Our *kernels* are carefully designed for coalesced accesses to device global memory when needed (where consecutive *work-items* access consecutive memory addresses), to avoid any loss in performance.

In this work we mainly consider three OpenCL-enabled accelerators: First, Intel's Xeon processor (CPU) is not only used here as the host, but also as an accelerator [15]. It is a general-purpose platform with good performance per core/thread [25]. Second, Intel's many integrated core (MIC) architecture called Xeon Phi (here we employ the Knights-Corner architecture). Each core includes a 512-bit vector arithmetic unit for wide single instruction multiple data (SIMD) instructions [13]. Third, the Nvidia Tesla K80 GPU card, with a very large set of highly optimized data paths (where parallel threads will be executed). These devices are able to hide the latency of memory accesses [24], increasing the throughput.

PyOpenCL is a package that provides Python with access to the OpenCL parallel computation API [16].

2.3 Jupyter Notebook and Git Repository

Jupyter is an open-source project that supports interactive data science and scientific computing across many programming languages [1]. The Jupyter Notebook is an open-source web application that enables the creation of documents containing explanatory text (including equations in LaTeX), live code execution, and output visualizations (with high-quality rendering of plots). Besides, interactive widgets can be easily added to provide a dynamic interface.

This notebook is particularly suited for interdisciplinary research cooperation, sharing a common project among all participants, where access (as well as automatic backup and version control) can be setup using a Git repository [2]. One such cooperation scenario is presented here, targeting the real-time risk measurement of dynamic complex financial portfolios, using nested MC simulations.

2.4 Application: VaR, ES, compVaR, compES

In the financial sector, an investment is usually diversified into a set of financial instruments, creating a portfolio. This includes securities (such as stocks and bonds), foreign currency, cash (for liquidity purposes), and derivatives (products that derive their price from an underlying asset). At a given horizon time, any portfolio could result in a profit/loss (P&L), exposing the investment to risk.

To measure this risk exposure, value-at-risk (VaR) and expected shortfall (ES) are the most widely employed risk measures in practice [12]. VaR states with probability α that the portfolio is not expected to suffer a loss of more than VaR \$ in the next N days, where α is usually as high as 99%. It is the α-quantile under the probability distribution of the loss, and represents the best-case scenario among the worst cases considered by the quantile. If a loss takes place, ES provides an estimate of the expected loss: it is the expected (mean) value of the loss inside the α-quantile tail.

Mathematically, the MC estimator for VaR is given by

$$VaR_\alpha(\tilde{L}_{t+\tau}) \triangleq \inf_{\ell \in \Re} \{\tilde{F}_{t+\tau}(\ell) \geq \alpha\}, \tag{1}$$

where $\tilde{F}_{t+\tau}(\ell)$ is the empirical distribution of the simulated loss distribution $\tilde{L}_{t+\tau}$ at time $t+\tau$.

In a similar way, the MC estimator for ES is given by

$$ES_\alpha(\tilde{L}_{t+\tau}) \triangleq \frac{1}{1-\alpha} \int_\alpha^1 VaR_\gamma(\tilde{L}_{t+\tau}) d\gamma. \tag{2}$$

In order to keep the overall risk of the portfolio under control, it is necessary to estimate the individual contribution of each financial product included in it. Because VaR and ES are homogeneous [20, 21], they can be rewritten as:

$$VaR_\alpha \equiv \sum_{i=1}^{N} h_i \frac{\partial VaR_\alpha}{\partial h_i}, \tag{3}$$

$$ES_\alpha \equiv \sum_{i=1}^{N} h_i \frac{\partial ES_\alpha}{\partial h_i}, \tag{4}$$

where the partial derivatives in Eqs. (3) and (4) are the marginal contributions per unit of each financial product p_i to the value of the risk measure of the overall portfolio, and h_i is the holding (amount) of such product [21].

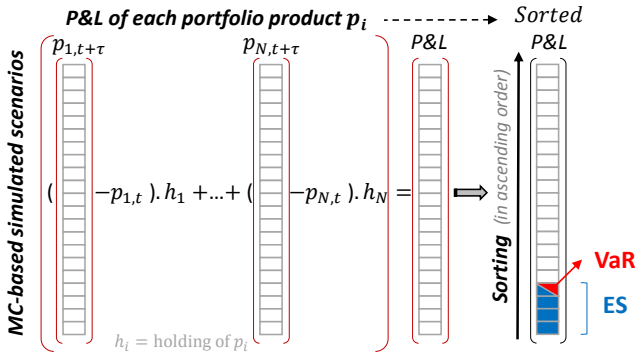
Thus, component value-at-risk ($compVaR_\alpha^{(i)}$) and component expected shortfall ($compES_\alpha^{(i)}$) can be estimated by:

$$compVaR_\alpha^{(i)} \equiv h_i \frac{\partial VaR_\alpha}{\partial h_i} \approx -k\, h_i\, S_i, \tag{5}$$

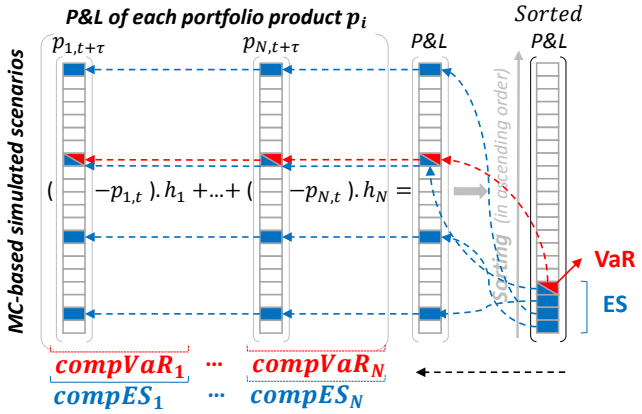
$$compES_\alpha^{(i)} \equiv h_i \frac{\partial ES_\alpha}{\partial h_i} \approx -q\, h_i\, S_i, \tag{6}$$

where S_i is the vector of simulated P&L scenarios for product p_i that contribute to the sorted P&L vector, k is a Gaussian smoothing kernel peaked around the VaR-index, and q is a step function from 0 (zero) to the mentioned index [21].

From a simulation point of view, the portfolio P&L is basically a vector containing the outcome (profit/loss) of each MC scenario [5], as shown on the right-hand side in Fig. 3a. After sorting the vector in ascending order, the VaR is the (negative) value in the vector indicated by the index $(1-\alpha)\cdot length$ (Eq. (1)), and ES is the mean of all vector elements up to that index (Eq. (2)).



(a) Nested MC simulation, profit/loss (P&L) and VaR/ES computation.



(b) $compVaR_i$ and $compES_i$ are the components of VaR and ES.

Figure 3: Aggregation process, VaR/ES, compVaR/compES.

Then the individual contribution of each product can be estimated using Eq. (5) and Eq. (6). To do so, it is necessary to retrieve the corresponding P&L values from the original vectors of simulated scenarios, as shown in Fig. 3b. For each $compVaR_i$ we approximate the Gaussian kernel with a Dirac delta centered on the position corresponding to VaR (red color), and each $compES_i$ is obtained by averaging the values that contributed to the overall ES (blue color).

To obtain the portfolio P&L in Fig. 3a, the different financial instruments in the portfolio have to be aggregated across all scenarios using their corresponding weights (holdings).

The complexity lies in the MC simulations required to price the different products in the portfolio, as shown in Fig. 4. In the *external* simulation we find stocks, bonds, and foreign currency, whereas in the *internal* simulation we find derivatives (such as financial options, Section 2.6), which is the most compute-intensive part [6].

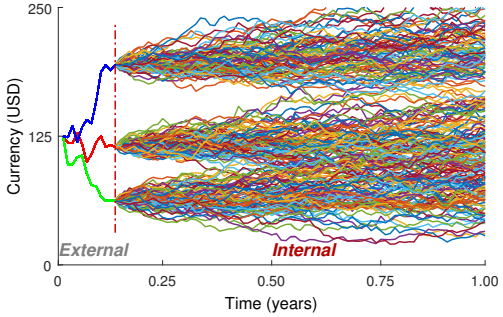


Figure 4: Nested MC-based simulations.

2.5 Risk Measurement of Dynamic Complex Portfolios

Although the (nested) MC approach is considered to be the most general and flexible one to the analysis of financial risk, the size of the simulations becomes a computational bottleneck for any bank, especially if combined with the revaluation of the full portfolio [18]. In this regard, a report presented by McKinsey & Company in 2012 showed that, at the time, only 15% of the surveyed banks would use this approach (with a downward tendency), whereas the remaining institutions would resort to easier but less accurate approaches [19], such as historical simulation or first order approximations [18, 19].

To grasp the complexity of the nested MC simulations, we assume a single, well-diversified, medium-size portfolio composition of 1000 financial products. An extrapolation of the standard implementation used as a reference in [6] yields on average ~ 19.3hs on a single compute node with one hardware accelerator. Applying the state-of-the-art optimizations (covered later in Section 2.7), the average runtime becomes ~ 11.5s in the same hardware configuration, and ~ 2.3s in a much more heterogeneous setup, per portfolio. Since banks handle a large pool of portfolios simultaneously, this approach would require a large (therefore expensive) compute cluster.

In this work we bring, for the first time, the nested MC simulations together with real-time operation, in such a way that we can use the cluster nodes to track the individual contribution of each trading desk to the risk of the different portfolios in a typical trading floor, as illustrated in Fig. 5.

Instead of running the simulation at fixed intervals of time while keeping the composition of the portfolio temporarily static (see e.g. [6, 26]), here we compute the risk measures dynamically, during the normal trading operation (intraday), and with every new incoming trading order. This raises new design challenges, since the update process (see Fig. 4), as well as the aggregation and risk measurement process (see Fig. 3), have to be kept at a minimum.

Trading orders are requests to buy or sell financial products, and they differ depending on the conditions which trigger their execution. Here we directly consider *market orders*, which are requests

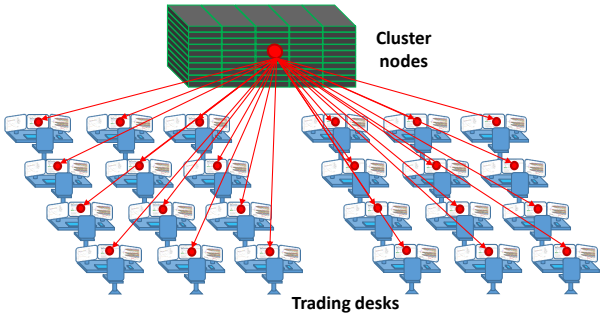


Figure 5: Real-time portf. risk analysis at trading floor level.

to trade immediately at the best available price in the market [12]. Nevertheless, our analysis here is independent of the conditions that triggered the execution.

When an order is executed, the portfolio risk measures change. As the portfolio changes over time, so does the contribution of the individual products to the overall risk.

The key observation to make here is the following: assume that a new trading order O_i is accepted at time t_i, right after determining that VaR/ES will remain under predefined limits, and that its contribution (compVaR/compES) to the portfolio is acceptable. At a later time $t + \tau$, N additional trading orders $O_{i+1}, ..., O_{i+N}$ would have also been accepted, and the porfolio would have changed accordingly, including VaR/ES. However, the new contribution of O_i (compVaR/compES) at time $t + \tau$ would certainly change. In other words: trading orders that the financial institution was willing to accept at time t, might no longer be desired at $t + \tau$.

In this regard, our goal is to setup a realistic trading-orders processing simulation flow where we can test the changes in the mentioned risk measures in real time, with every new incoming trading order, and to measure the maximum system performance. This flow is summarized in Fig. 6.

Several decision points can be overriden in Fig. 6, in order to find best- and worst-case scenarios in terms of runtime.

Random trading orders are pre-generated the following way:

- a uniform distribution determines which product to select, from a pool of all implemented products (in this work: Bond, Stock, Foreign Currency, Options: European Asian, European Barrier, American Vanilla),
- a second uniform distribution determines whether it is a *sell* or *buy* order,
- a third uniform distribution is used to alter several parameters in each product (in a predefined range), including the underlying asset for options (only stocks here, but any of them), and the amount required in the transaction.

Finally, we can also override the product selection process, fixing all trading orders to a certain product (used for testing).

The portfolio may have any initial composition at time t_0 of the current trading day, and the risk measures and simulated scenarios (vectors) can be initialized following the approach in [26]. The same could be done once the market closes and the trading operations cease, but this is only necessary if the parameters of the products in the portfolio have changed.

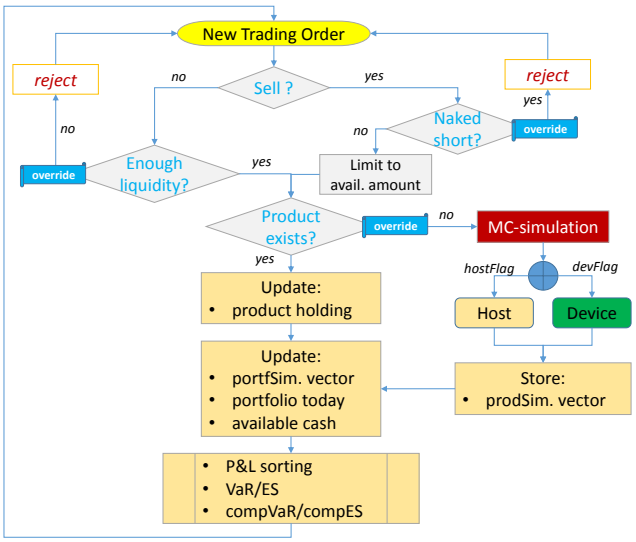


Figure 6: Flow diagram to process incoming trading orders.

Here we focus on the acceleration of the flow shown in Fig. 6, on a single portfolio. To this end, we assume that each trading order is already in the memory accessible by the host at the time of processing, excluding external communication overhead. We also concentrate on optimizing one incoming queue of orders requests or, in other words, the performance of a single node in a cluster.

In the coming sections we offer additional details on other technical foundations regarding this application.

2.6 Foundations of Options and Option Pricing

Options are contracts that (typically) provide a non-negative future payment to their holder. The height of this payment is derived from the behaviour of the price of an underyling security (such as e.g. a stock, an interest rate, or a currency) [12]. The so-called *option pricing* process consists in assigning a fair price to the option. When no complex features are added to an option, it is called a *vanilla* option. However, exotic options are widely traded. Representative examples used here are *asian* options, whose price depend on the average evolution of the underlying asset until *maturity*, and *barrier* options, which are validated/invalidated depending on the underlying asset crossing a predefined barrier price.

The so-called *european* options can only be exercised (by the holder) at *maturity*, and their pricing process only requires the stochastic information of the simulation, making a MC approach an ideal candidate. On the other hand, so-called *american* options can be exercised at any time, which requires the determination of an exercising strategy that largely increases the computational complexity, usually requiring backward induction. For one-dimensional *american* options (that depend on one underlying asset) we resort to the binomial tree under the Cox-Ross-Rubinstein (CRR) model [12], due to its computational efficiency. For *american* options on multiple of stocks only computationally intensive MC algorithms such as the Longstaff-Schwartz-method "are tractable", as binomial tree methods suffer from the curse of dimensionality [12].

To simulate the evolution of an asset in a MC simulation, we employ two widely used models in practice: 1) Black-Scholes (BS), with one input parameter, but quite robust; 2) Heston, with five input parameters to better match the real evolution of the asset, but requires a larger runtime in general. For more details please refer to [12]. The simulation of these two models relies on normally distributed random numbers (RNs), which we will later generate with the *numpy.random.randn()* routine in Python's *numpy* package.

2.7 State-of-the-art Algorithmic Optimizations

In terms of computing risk measures of complex portfolios via nested MC simulations, the state-of-the-art approach exploits the so-called *common numbers technique* [6]. It is based on the observation that the RN sequences should be reused among the internal simulations corresponding to the same financial product. Therefore, the RN sequences can be externally generated once, and then reused appropriately inside each product in the internal simulation. This *RNs-reuse* approach has achieved speedups in the order of $8x$.

In the case of *european*-style derivatives (options), recent research has shown that the *stochastic information* at maturity (such as paths values, average, max/min values) can be extracted from a normalized simulation, and then denormalized based on the external simulations [26]. This optimization, called *Info-reuse*, was originally designed for OpenCL implementations, delivering over $1000x$ acceleration compared to the *RNs-reuse* approach. In this work we show that this optimization is also valid when using Python.

In the case of *american* options, the pricing process is more complex. The idea proposed by [14] implies the pre-generation of a grid of option prices, and the subsequent interpolation between them. The interpolation scheme is valid since the generated grid of options is a smooth curve, provided a fine grid granularity is used [14]. This *Interpolation* scheme has achieved over $1000x$ acceleration compared to the *RNs-reuse* approach. In this work, we extend this approach by dynamically generating the grid to span between the maximum and minimum values of the external simulation (which corresponds to the underlying asset), and then to generate the curve on-the-fly, before the interpolation can take place.

2.8 Related Work

Python has been successfully employed in high performance and scientific computing, e.g. in [27]. A case study of performance vs productivity of parallel code with Python can be found e.g. in [3]. For the computation of portfolio VaR, a hybrid MC approach with delta-gamma approximations was studied in [7]. Beyond the references given throughout Section 2, to the best of our knowledge this is the first time that the nested MC-based simulation of VaR/ES and compVaR/compES risk measures, applied in real-time to dynamic complex financial portfolios, is carried out and published.

3 IMPLEMENTATION

The use of Python greatly simplifies the host code generation. Object-oriented programming is used for the financial products, where each class contains all relevant data and parameters, as well as the following methods:

- parameters initialization (including random initialization),
- the computation of today's value (whenever required),
- the implementation of the simulation kernel to run on host (including the use of PyMP if appropriate),
- the initialization of PyOpenCL buffers and kernels,
- enqueueing the kernel(s) and a separate interface to wait for their completion,
- and a print function (mainly used for debugging).

Input parameters to these methods are grouped using Python lists as appropriate, in such a way that all classes (products) offer the same interface. This helps automate the invocation of these methods from the host code, and it facilitates the future inclusion of new products (this application easily scales).

From the host (CPU) side, the portfolio, the incoming trading orders, all external buffers, as well as the RN sequences are all grouped in (multidimensional) lists, which simplifies the searching process. The normally distributed RN sequences required for the mathematical models used in the simulation are generated via *numpy.random.randn()*, carefully controlling seeds and initialization in order to allow for reproducibility of results. In the case of heterogeneous computing systems, a command queue (and all related PyOpenCL steps) is established for every active device, which can be enabled either manually (in an interactive way) or automatically.

A simplified snapshot of the Jupyter Notebook used is presented in Fig. 7, where the entire simulation is set up via interactive widgets. Upon execution, the code cell automatically queries all available platforms and devices, from which the user can then chose via *pid-platform* and *did-device*.

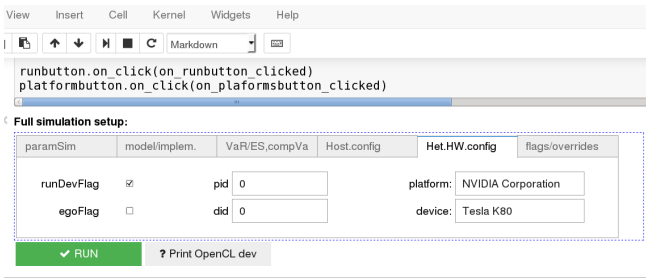


Figure 7: Jupyter Notebook (representative snapshot) using interactive widgets for dynamic simulation configuration.

Hidden under the Jupyter Notebook is the entire design in Python, as well as the accelerated code via PyMP and/or PyOpenCL. Listing 1 shows the main part of the code used for processing the trading orders flow (see Fig. 6).

Listing 1: Python: trading-orders processing

```
# flags and transaction values are computed before
okFlag1=sellFlag&((not nakedSellFlag)|override...)
okFlag2=buyFlag&((not liquidityFlag)|override...)
if(okFlag1 | okFlag2): # accepted order
 if(existingFlag & (not overrideExistingFlag)):
  #--------------------------------------
  existing.holding += order.holding
  if(args.runHostFlag): # run on host
   portfSim += existing.sim_host * order.holding
  if(args.runDevFlag): # run on device
   portfSim += existing.sim_dev * order.holding
  #--------------------------------------
```

```
 else: # new product
  if(args.runHostFlag): # run on host
   [portfSim,...] = aux.run_prod_host(...)
  if(args.runDevFlag): # run on device
   if(args.egoFlag): # ego kernels used
    ego=[x for x in egos if x.prod==order.prod]
    [portfSim,...] = aux.run_prod_dev_ego(...)
   else: # individual kernel per order
    [portfSim,...] = aux.run_prod_dev(...)
  portfolio.append( [..., order] )
  #---------------------------------------
 portfToday += prod_today * order.holding
 cash.amount -= cash_required
# Computing risk measures
[risks,...] =  aux.run_var_es_compvar_compes(paramSim,
     portfolio, portfSim, portfToday, cash,...)
```

Listing 2: Python: computing risk measures

```
def run_var_es_compvar_compes(..., paramSim, portfolio,
      portfSim, portfToday):
 profitloss = portfSim - portfToday
 #'quicksort','mergesort','heapsort'
 sortedPL = np.sort(profitloss,kind='mergesort')
 sortId = np.argsort(profitloss,kind='mergesort')
 #---------------------------------------
 varIndex = np.int32(len(profitloss)*(1.0-paramSim.
      varProb))
 VaR = -sortedPL[varIndex]
 ES  = -np.mean(sortedPL[0:varIndex+1])
 VaRp = VaR / portfToday * np.float32(100.0)
 ESp  = ES  / portfToday * np.float32(100.0)
 #---------------------------------------
 compVaR = []
 if compvarFlag:
  pos_var = sortedIndex[varIndex]
  rpi = []
  if(args.runHostFlag):
   rpi = [(p.sim_host[pos_var]-p.today)*p.holding for p
         in portfolio]
  if(args.runDevFlag):
   rpi = [(p.sim_dev[pos_var] -p.today)*p.holding for p
         in portfolio]
  compVaR =[-x/VaR*np.float32(100.0) for x in rpi]
 #---------------------------------------
 compES = []
 if compesFlag:
  pos_es_list = sortedIndex[0:varIndex+1]
  rpi = []
  if(args.runHostFlag):
   rpi = [np.mean(p.sim_host[sortedIndex[0:varIndex+1]]
         -p.today)*p.holding
        for p in portfolio ]
  if(args.runDevFlag):
   rpi = [np.mean(p.sim_dev [sortedIndex[0:varIndex+1]]
         -p.today)*p.holding
        for p in portfolio ]
  compES =[-x/ES*np.float32(100.0) for x in rpi]
 #---------------------------------------
 return [profitloss,VaR,ES,compVaR,compES,...]
```

Listing 2 shows how to compute the portfolio P&L vector, its sorting process, the computation of VaR and ES, as well as compVaR and compES. It is easy to notice that compES is computationally more expensive than compVaR, a fact that will be briefly shown later in the results (Section 4.4).

Example code with PyMP and PyOpenCL is excluded here due to space limitations, but they can be generated from the references [16, 17, 26]. For details on the parallelization schemes used in the internal MC simulation, see [6]. How to optimally handle the instances of the trading orders and the products in the portfolio, is later covered in Section 4.6.

4 RESULTS

HW and simulation setups are provided, followed by numerical results on runtime, throughput and energy efficiency.

4.1 Hardware Setup

It consists of a SuperMicro Superserver 7048GR-TR with: dual-socket X10DRG-Q motherboard, 2000W high-efficiency redundant power supplies (Titanium level), cooling kit MCP-320-74701-0N-KIT (for GPU and Xeon Phi); *CPUs*: (2x) Intel Xeon Processor E5-2670 v3 (Haswell architecture, technology node 22 nm, 12 cores (24 threads), base frequency 2.3 GHz), 64GB DDR4; *Xeon Phi*: (1x) Intel Xeon Phi Coprocessor 7120P (Many Integrated Core (MIC) Knights Corner architecture, technology node 22 nm, 61 cores, base frequency 1.238 GHz, 16 GB DRAM, passive cooling); *GPU*: (x1) Nvidia Tesla K80 GPU Accelerator (dual GK210 sharing a single PCIe link, Kepler architecture, technology node 28 nm (2x) 2596 CUDA Cores, base frequency 560 MHz, 12 GB per GK210, passive cooling). The operating system is Linux RedHat 6.6 (Linux 2.6.32-504.el6.x86_64), including all OpenCL drivers. The room temperature is controlled at $\sim 23°C$.

In this work, the host (CPU) as well as three combinations *host+accelerator* are evaluated:

- **Host**: *CPU* only,
- **CPU**: *CPU* (host)+*CPU* (dual-socket motherboard),
- **PHI** : *CPU* (host)+*Xeon Phi*,
- **GPU**: *CPU* (host)+*GPU*,

Although the Nvidia's K80 PCIe card contains two GK210 GPUs, here we will use only one of them at any given time.

4.2 Simulation Setup

Our representative initial portfolio is composed of: (1) corporate bond, (2,3) two correlated stocks, (4) foreign currency (EUR), (5,6,7) three options: european asian, european barrier, and american vanilla (all on $stock_2$), (8) cash (providing liquidity). This setup covers complex portfolios usually found in practical situations.

Nested MC simulations setup: pathsMCext=32768, stepsMCext=10, pathsMCint=32768, stepsMCint=stepsTree=252, EuroBarrier kernels require $4x$ steps, interpolateN=128. (Py) OpenCL NDRange: global-size=32768, local-size=(16/16/512) for (CPU/PHI/GPU). The

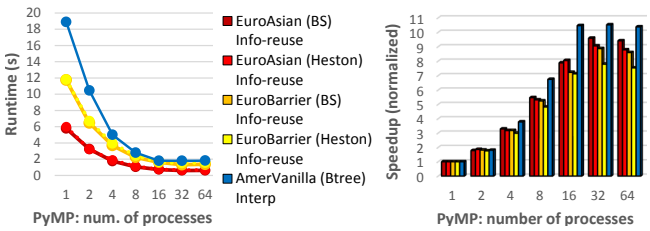


Figure 8: PyMP performance, scaling number of processes.

optimal values for global-size and local-size are derived from [26], whereas the optimal number of PyMP parallel processes is 32, following Fig. 8. All computations are carried out in single-precision floating point, which provides enough accuracy here.

The OpenCL kernels used in this work can be ported to any device supporting OpenCL version 1.2. Python version 3.4.2, all runtime measurements were carried out on the host side (with proper synchronization points whenever required) using *perf_counter()* from Python's *time* module; PyMP version 0.2; PyOpenCL version 2017.1.1; Jupyter Notebook version 5.0.0.

4.3 Results Validation

For the given setup and portfolio composition[1]: $\text{Portfolio}_{\text{today}}$= 7,037,073.71 $; 10-day(VaR,ES) with α=99%: BS model (6.38%, 7.05%), Heston model (6.65%, 7.37%). All hardware configurations deliver the same results (the kernels access the memory buffers in the same order, irrespective of the parallelization granularity).

4.4 Dynamic Variations of compVaR/compES and VaR/ES

As mentioned in Section 2.5, whenever a buy/sell trading order is accepted, the portfolio VaR and ES metrics change accordingly, and so does the contribution of the individual products to the overall risk. Fig. 9 shows a snapshot of such evolution, starting from the initial portfolio composition, and during the first 10 new orders.

In line with theoretical expectations, ES is always larger than VaR (recall Eqs. (1) and (2)). Negative values seen in compVaR imply that the product actually makes a profit in the scenario leading to the VaR/ES estimation. The key message to take from Fig. 9 is that all risk measures, including the individual contributions of each product in the portfolio, dynamically change over time. This justifies the need for real-time risk measurement systems and continuous monitoring.

CompES follows the same pattern as compVaR in Fig. 9, although it is computationally far more complex (refer to Listing 2). Measured on 1k accepted new trading orders that are all included in the portfolio (meaning that the portfolio size grows linearly), the measured runtimes are: $t_{VaR+ES} = 5.2s$ (including sorting of the P&L vector, refer to Fig. 3a); $t_{compVaR} = 0.6s$; $t_{compES} = 11.5s$. This makes the use of compES impractical for real-time continuous measurements, and we should use compVaR instead.

4.5 Nested MC Simulation Runtime

The main results on the compute intensive part of the nested MC simulation (which is the one corresponding to the internal simulation in Fig. 4) are shown in Fig. 10 for our given setup.

[1] Portfolio parameters required for results validation (Section 4.3): A) Initial composition: bond=28.17%, stock_0=13.93%, stock_1=14.21%, fcurrency=12.67%, euroAsian=4.24%, euroBarrier=8.21%, amerVanilla=4.36%, cash=14.21%. B) Additional parameters: Bond: par=1000.0 $, r=0.07, coupon_i=70.00 $, σ=0.50, yields={5.00, 5.69, 6.09, 6.38, 6.61, 6.79, 6.94, 7.07, 7.19, 7.30}, n_{coupons}=10, hold=$2k$; Stocks: $S_{0,0}$=140.00 $, $S_{0,1}$=50.00 $, ρ_{stocks}=0.25, $\text{hold}_{S0}=7k$, $\text{hold}_{S1}=20k$; under BS model: r_1=r_2=2.5%, σ_1=0.20, σ_2=0.25; under Heston model: r_1=r_2=2.5%, $v_{0,1}$=0.04, $v_{0,2}$=0.0625, κ_1=κ_2=3.0, θ_1=0.04, θ_2=0.0625, σ_1=0.20, σ_2=0.25, ρ_1=−0.70, ρ_2=−0.80; Foreign Currency: rate=1.1144, σ=0.1318, amount=800k EUR; European Asian: underlying=S_0, put, hold=(550 · 100); European Barrier: underlying=S_0, S_{barrier}=182.00 $, call, up, in, hold=(750 · 100); American Vanilla: underlying=S_0, call, hold=(245 · 100); Cash: amount=1M $.

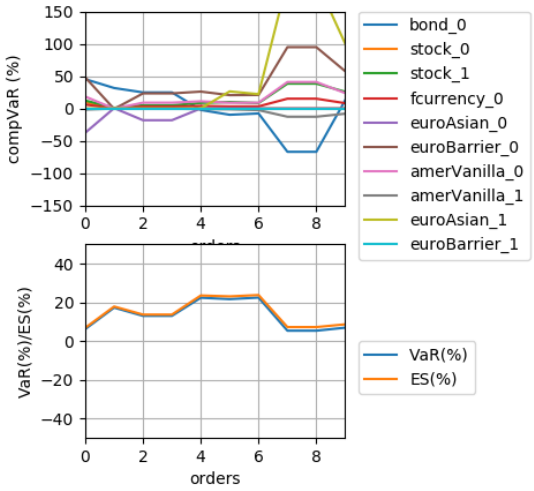


Figure 9: Evolution of compVaR, VaR/ES in the first 10 random trading orders, starting from the initial portfolio. Their dynamic evolution over time justifies the need for real-time risk measurement systems and continuous monitoring.

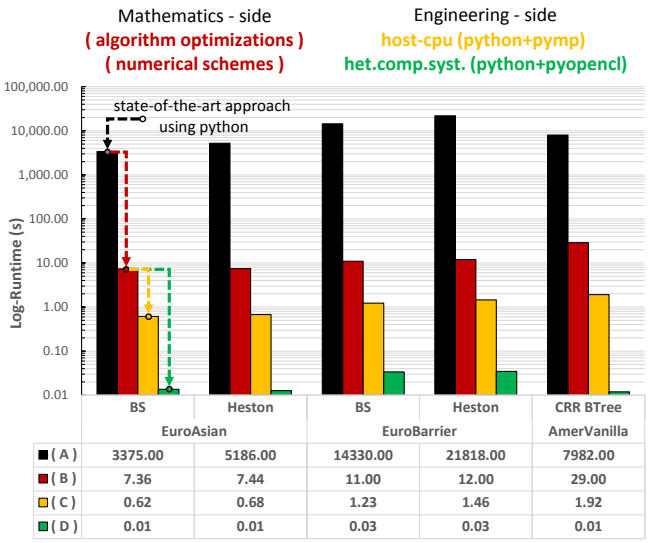


Figure 10: Kernels runtime (measured from the host): (A) RN-reuse (European options) / standard approach (American option), in Python running on the host; (B) employing Info-reuse/Interpolation-dynamic schemes; (C) introducing PyMP; (D) introducing PyOpenCL (heterogeneous systems).

Following Section 2 on the interdisciplinary research cooperation between mathematics and engineering (see Fig. 1), Fig. 10 first shows how algorithmic optimizations and numerical schemes originally designed for, and used with, OpenCL are also valid on single-threaded Python code, without any parallelization scheme (refer to Section 2.7). On top of these optimizations, PyMP is used to readily provide a substantial speedup on multiprocessor CPUs, by parallelizing the sequential code using a few directives and features.

Table 1: Trading-Orders Processing Throughput (measured at 1k orders, all processed and accepted)

Best-case[a] / Worst-case[b] :	**Best-case** [a]	**Worst-case** [b]						
Platform (HW configuration) :	(*platform-*	**Host** [c]	**CPU** [c]		**PHI** [c]		**GPU** [c]	
Implementation style of the kernels :	*independent*)	**Indiv.**	Indiv.	**Ego**	Indiv.	**Ego**	Indiv.	**Ego**
Throughput (orders/s)	**191.1**	**0.7**	13.9	**16.8**	9.9	**30.4**	21.8	**26.8**
Runtime for 1k processed orders (s)	5.23	1435.09	72.04	59.67	100.82	32.91	45.77	37.25
		Contribution to the measured runtime (%) :						
– check if the product instance has been prev. traded	0.41	0.01	0.26	0.35	0.28	0.74	0.39	0.43
– if a new product instance is required :								
– OpenCL initialization (pyopencl)	0.00	0.00	20.51	1.31	63.48	3.76	21.19	1.72
– Kernel runtime	0.00	99.02	69.70	85.25	23.14	58.84	63.79	79.95
– Buffers (PCIe) transfers (1 write, 2 reads)	0.00	0.00	0.56	0.67	1.54	2.99	0.71	0.86
– VaR/ES computation	97.60	0.93	7.31	10.37	10.15	29.59	11.62	14.31
– compVaR computation	0.44	0.02	1.21	1.52	0.97	2.84	1.42	1.69
– miscellaneous calculations	1.55	0.02	0.46	0.54	0.44	1.25	0.88	1.03

[a] Best-case scenario: all trading orders correspond to existing product instances in the portfolio (the MC simulation can be skipped).
[b] Worst-case scenario: all trading orders are new product instances of the most compute-intensive product, each requiring its MC simulation.
[a,b] Every new trading order forces an update of the simulated portfolio scenarios and all risk metrics (compES is skipped in this setup).
[c] Host employs Python+PyMP; CPU/PHI/GPU employ Python+PyOpenCL.

This first-stage speedup helps in making the mentioned cooperation more fluid, reducing the time required for theoretical vs numerical analysis and debugging. The second stage of acceleration requires the use of an accelerator (on an heterogeneous computing system) and the use of PyOpenCL, to obtain a far larger speedup. We emphasize though, that the design and programming effort of this second stage is far larger than the previous one, since it is necessary to code all the OpenCL kernels that will run on the device (accelerator), as well as the PyOpenCL code to be executed on the host.

When we compare the extremes in Fig. 10, we observe a speedup of $\sim$ 6 orders of magnitude. This acceleration makes it feasible to employ, for the first-time, nested MC simulations for real-time financial risk measurement with Python.

4.6 Throughput and Energy Efficiency

When it comes to dynamically changing portfolios during intraday operation, the processing flow presented in Section 2.5 and Fig. 6, together with the Python implementation in Section 3, achieves the throughput detailed in Table 1. In this regard, the following remarks are worth highlighting.

First, the best- and worst-case throughput is measured over 1k trading orders that are all accepted (rejecting orders misleadingly increases the throughput).

Second, the best-case scenario takes place when all trading orders correspond to products that already exist in the portfolio. Nevertheless, the following updates are required: holdings, portfolio today, portfolio simulated values, and all risk measures (including P&L sorting). Since all these calculations and updates are carried out on the host, the best-case scenario is platform independent (no accelerator is involved here).

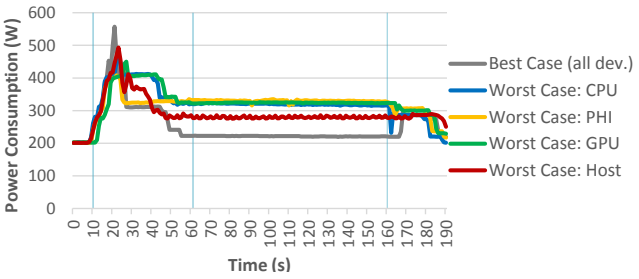
Third, the worst-case scenario is an extreme situation when all trading orders are treated as new products, and they all correspond to the most compute-intensive kernel available (in this case the European Barrier option under the Heston model).

Fourth, each trading order, as well as each product in the portfolio, is an instance of a certain (Python) class, which will have its own buffers (both on host- and device-side), as well as instantiated OpenCL kernels. Because the incoming trading orders are unkown at design time, the initialization process in PyOpenCL (buffers and kernels) needs to take place upon requesting the execution of the kernel. These instances can be treated independently, which we call *Indiv.* in Table 1. However, we observed that the OpenCL initialization process was taking a large percentage of the runtime, this being more critical when employing PHI. To overcome this situation, we offer here an alternative approach (which we call *Ego* in Table 1): the basic idea is to have a single instance of each product in each active device (accelerator). *Ego* instances only keep the OpenCL-related buffers and instantiations, whereas the new orders (as well as the existing products in the portfolio) only keep the host-side data and parameters. Both approaches are valid, but *Ego* achieves higher throughput, since now the runtime is mostly determined by the kernels and the computation of VaR/ES/compVaR risk measures.

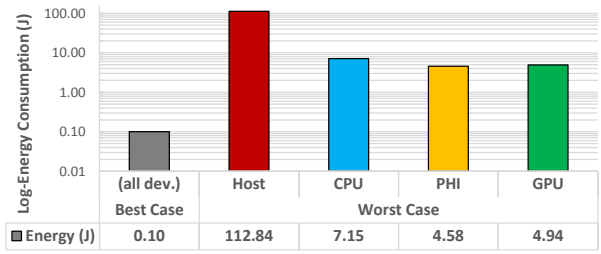
When targeting heterogeneous computing systems with Python+PyOpenCL, we achieve 16$\sim$30 trading-orders/s in the worst-case, and 191 trading-orders/s in the best-case scenario. Excluding automated high frequency trading, this throughput is high enough to be considered real-time in any human-based trading desk in finance.

Not only high throughput, but also energy efficiency plays a significant role nowadays (which is related to the cost of running such high-performance computing systems). To do so we have measured the power consumption of the entire system at the power plug, sampling once per second. The spikes on the left hand side of Fig. 11a correspond to the simulation setup, including the creation of the full sequence of random trading orders, as well as the

ramp-up of the cooling system set to optimal mode. The trading orders are actually processed between the last two vertical markers in Fig. 11a, and these are the values considered (100 points in total). The system-level dynamic energy consumption (which is the one actually consumed to process the trading orders) for the best- and worst-cases is shown in Fig. 11b. We observe that the use of heterogeneous computing systems increases the system-level dynamic energy efficiency by 15 ∼ 24x, compared to a pure host configuration (for the worst-case scenario).



(a) Power consumption measurements.



(b) System-level dynamic energy consumption.

Figure 11: Power and dynamic energy consumption.

5 CONCLUSIONS

While continuously monitoring portfolio risk measures (VaR and ES) widely used in practice, our simulation results show how the individual contributions (compVaR) of the products in the portfolio to the overall risk change over time. Trading orders that have previously been accepted might no longer be desirable at a later time, which highlights the importance of real-time risk analysis. In this regard, we have achieved a minimum throughput of 16 ∼ 30 trading-orders/s, high enough for any human-based trading operation. Moreover, the runtime of compVaR scales linearly with the number of products in the porfolio, whereas the achieved throughput is independent of the size of the portfolio when measuring VaR and ES. This makes our approach easily scalable.

ACKNOWLEDGMENTS

We gratefully acknowledge financial support from the Deutsche Forschungsgemeinschaft (DFG) within the research training group 1932 "Stochastic Models for Innovations in the Engineering Sciences", and from the Center of Mathematical and Computational Modelling $(CM)^2$ of the University of Kaiserslautern. The authors alone are responsible for the content of this paper.

REFERENCES

[1] 2014. Project Jupyter. Online. (2014). Retrieved October 10, 2017 from http://jupyter.org/
[2] 2017. Git. Online. (2017). Retrieved October 10, 2017 from https://git-scm.com/
[3] Achim Basermann, Melven Röhrig-Zöllner, and Joachim Illner. 2015. Performance and Productivity of Parallel Python Programming: A Study with a CFD Test Case. In *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing (PyHPC '15)*. ACM, 2:1–2:10.
[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.
[5] Jon Danielsson. 2011. *Financial Risk Forecasting*. Wiley.
[6] Sascha Desmettre, Ralf Korn, Javier Alejandro Varela, and Norbert Wehn. 2016. Nested MC-Based Risk Measurement of Complex Portfolios: Acceleration and Energy Efficiency. *Risks* 4, 4 (2016). http://www.mdpi.com/2227-9091/4/4/36
[7] Matthew F. Dixon, Thomas Bradley, Jike Chong, and Kurt Keutzer. 2012. Monte Carlo-Based Financial Market Value-at-Risk Estimation on GPUs. In *GPU Computing Gems* (jade ed.), Wen-Mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann, Chapter 25, 337–353.
[8] Shayne Fletcher and Christopher Gardner. 2009. *Financial Modelling in Python*. Wiley Publishing.
[9] Python Software Foundation. 2017. The Python Tutorial. Online. (2017). Retrieved October 10, 2017 from https://docs.python.org/3/tutorial/
[10] Yves Hilpisch. 2014. *Python for Finance: Analyze Big Financial Data* (1st ed.). O'Reilly Media, Inc.
[11] Lee Howes and Aaftab Munshi. 2015. The OpenCL Specification. online. (Jan 2015). https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf
[12] John C. Hull. 2012. *Options, Futures, And Other Derivatives* (8th ed.). Pearson.
[13] Intel. 2014. OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor. Online. (2014). Retrieved October 10, 2017 from https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor?language=ru
[14] Adriaan Joubert and L.C.G. Rogers. 1997. Fast, accurate and inelegant valuation of American options. In *Proceedings of the Numerical Methods Workshop at the Isaac Newton Institute, April 1995*, L.C.G. Rogers and D. Talay (Eds.). Cambridge University Press, 88–92. http://www.statslab.cam.ac.uk/~chris/papers/aj2.pdf
[15] Vadim Kartoshkin and Timothy Mattson. 2011. Tutorial: OpenCL - Introduction for HPC Programmers. (Mar 2011). https://software.intel.com/en-us/articles/tutorial-opencl-introduction-for-hpc-programmers last access: 31 Aug 2017.
[16] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (mar 2012), 157–174.
[17] Christoph Lassner. 2016. PyMP. Online. (2016). Retrieved October 10, 2017 from https://github.com/classner/pymp
[18] Alexander J. McNeil, RÃijdiger Frey, and Paul Embrechts. 2005. *Quantitative Risk Management: Concepts, Techniques, and Tools*. Princeton University Press, Princeton and Oxford.
[19] Amit Mehta, Max Neukirchen, Sonja Pfetsch, and Thomas Poppensieker. 2012. *Managing market risk: Today and tomorrow*. McKinsey Working Papers on Risk 32. McKinsey & Company. http://www.mckinsey.com/business-functions/risk/our-insights/managing-market-risk-today-and-tomorrow
[20] Attilio Meucci. 2005. *Risk and Asset Allocation*. Springer, Berlin Heidelberg.
[21] Attilio Meucci. 2010. Factors on Demand: Building a Platform for Portfolio Managers, Risk Managers and Traders. *Risk* 23, 7 (2010), 84–89. https://ssrn.com/abstract=1565134
[22] J. P. Morgan and Reuters. 1996. *RiskMetrics: RiskMetrics Technical Document*. Technical Report 4th ed. New York.
[23] Basel Committee on Banking Supervision. 2010. *Basel III: A global regulatory framework for more resilient banks and banking systems*. Technical Report. Bank for International Settlements. http://www.bis.org/publ/bcbs189.pdf (rev June 2011).
[24] David A. Patterson and John Hennessy. 2009. *Computer Organization and Design* (4 ed.). Morgan Kaufmann.
[25] James Reinders. 2012. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors. Intel, Online. (2012). Retrieved October 10, 2017 from http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf
[26] Javier Alejandro Varela and Norbert Wehn. 2017. Near Real-Time Risk Simulation of Complex Portfolios on Heterogeneous Computing Systems with OpenCL. In *Proceedings of the 5th International Workshop on OpenCL (IWOCL 2017)*. ACM, New York, NY, USA, 2:1–2:10. http://doi.acm.org/10.1145/3078155.3078161
[27] Peter Vincent, Freddie Witherden, Brian Vermeire, Jin Seok Park, and Arvind Iyer. 2016. Towards Green Aviation with Python at Petascale. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2016.1