Stat Comput (2017) 27:1365–1382
DOI 10.1007/s11222-016-9691-9

Fast parallel α-stable distribution function evaluation and parameter estimation using OpenCL in GPGPUs

Guillermo Julián-Moreno[1] · Jorge E. López de Vergara[1] · Iván González[1] · Luis de Pedro[1] · Javier Royuela-del-Val[2] · Federico Simmross-Wattenberg[2]

Received: 19 February 2016 / Accepted: 8 August 2016 / Published online: 23 August 2016

Abstract α-Stable distributions are a family of probability distributions found to be suitable to model many complex processes and phenomena in several research fields, such as medicine, physics, finance and networking, among others. However, the lack of closed expressions makes their evaluation analytically intractable, and alternative approaches are computationally expensive. Existing numerical programs are not fast enough for certain applications and do not make use of the parallel power of general purpose graphic processing units. In this paper, we develop novel parallel algorithms for the probability density function and cumulative distribution function—including a parallel Gauss–Kronrod quadrature—, quantile function, random number generator and maximum likelihood estimation of α-stable distributions using OpenCL, achieving significant speedups and precision in all cases. Thanks to the use of OpenCL, we also evaluate the results of our library with different GPU architectures.

✉ Jorge E. López de Vergara
jorge.lopez_vergara@uam.es

Guillermo Julián-Moreno
guillermo.julian@estudiante.uam.es

Iván González
ivan.gonzalez@uam.es

Luis de Pedro
luis.depedro@uam.es

Javier Royuela-del-Val
jroyval@lpi.tel.uva.es

Federico Simmross-Wattenberg
fedesim@lpi.tel.uva.es

[1] Department of Electronics and Communication Technologies, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Madrid, Spain

[2] Image Processing Lab, E.T.S.I. Telecomunicación, Universidad de Valladolid, Valladolid, Spain

Keywords Gaussian quadrature · α-Stable distribution · Parallel algorithms · Numerical algorithms · OpenCL · GPGPU

1 Introduction

The Central Limit Theorem is a well-known mathematical result, which states that the standardized sum of a sufficiently large number of independent, identically distributed random variables with finite variance and mean will resemble a normal (Gaussian) distribution. This theorem can be generalized for random variables with infinite moments: the resulting distribution is then called an α-stable (or just stable) distribution (Gnedenko and Kolmogorov 1968). Its name comes from another interesting property (Nolan 2015): the fact that, given X_1, X_2 independent copies of a random variable X with stable distribution, then

$$aX_1 + bX_2 \overset{\text{dist.}}{=} cX + d \tag{1}$$

for some constants $a, b, c > 0$ and $d \in \mathbb{R}$.

These properties make stable distributions a suitable model for many events in different fields that naturally exhibit such high variability rates that they cannot be adequately modeled using simple statistical distributions. For example, in medicine they are used for segmentation of brain matter in magnetic resonance imaging (MRI) (Salas-González et al. 2013) and as a model for ultrasound denoising (Achim et al. 2001); in physics they can be used to study and predict atomic behavior (Bardou et al. 2002); in finance they are a common model for asset pricing (Mittnik and Rachev 1993); and their use in networking allows detection and understanding of traffic events and patterns (Simmross-Wattenberg et al. 2011; Li et al. 2015).

However, a problem arises when these distributions are used in environments where quick results are needed, such as in medical imaging, high frequency trading (HFT), or monitoring of multiple network links. It is complex to deal with α-stable distributions since they lack closed expressions for their probability density function (PDF) and cumulative distribution function (CDF) and require complicated, computationally expensive approximations and numerical methods, such as those we will present later, to be applied in solving any of the aforementioned problems. The problem worsens when these functions must be evaluated many times in a time-restricted scenario, such as when estimating parameters in real time, for instance, to monitor network traffic behavior or to render echographic images from a medical probe.

Throughout this paper, we develop a novel approach to compute α-stable distributions using OpenCL (Stone et al. 2010), considerably improving speed and allowing massively parallel computation of the PDF and CDF functions. In turn, these two functions will allow us to parallelize calculations of the quantile function (also called CDF^{-1}) and the estimation of parameters. For completeness of the library, a parallel generator of α-stable distributed random numbers has also been implemented, based on the method proposed by Weron and Weron (1995).

Our purpose is to allow the use of these distributions in time-constrained environments where existing solutions such as John Nolan's STABLE program[1] or *libstable*[2] (Royuela-del-Val 2016) are not fast enough; or where GPU cards are available to offload work from the CPU.

To achieve our goal, we have developed a parallel implementation of the Gauss–Kronrod quadrature rule (Kronrod 1965) for numerical integration, and an implementation of a maximum likelihood estimator based on contracting grid search algorithms (Hesterman et al. 2010) that has been modified using asynchronous OpenCL commands to maximize the use of all the components in the pipeline: given that our algorithm is not memory intensive, the data transfer costs are almost negligible compared with the OpenCL kernel setup costs. The scheduling of the different kernels is left to the OpenCL driver implementation.

We have used the OpenCL framework for the implementation in order to broaden the platforms where our software can run: not only GPUs from different vendors (such as NVIDIA or AMD) but also new parallel platforms such as Alpha Data's FPGA boards (Alpha Data 2013) or Intel's Xeon Phi co-processor (Intel 2013). However, this paper is only centered in the application running on GPUs, leaving tests on other platforms as future work.

In spite of using OpenCL, a device-agnostic language, and maintaining same code for every platform, we have carefully observed memory layout and concurrency issues in our algorithm so that performance in readily available devices, specifically NVIDIA and AMD GPUs, is maximized wherever possible.

Results are very promising, with our software, available at GitHub,[3] being several times faster than *libstable*, the current fastest implementation (Royuela-del-Val 2016), while keeping precision and accuracy.

The rest of the paper is structured as follows: next subsection discusses related work. Section 2 analyzes the equations and mathematical algorithms that will be used for the computation of the distribution. Section 3 shows the translation from those equations to an implementation in OpenCL using parallel algorithms. Finally, we expose our results in Sect. 4, with a corresponding performance analysis, and our final conclusions in Sect. 5.

1.1 Related work

Given their usefulness in several different knowledge areas, several approaches for the computation of α-stable distributions have been developed. Most are centered on giving a complete implementation, such as John Nolan's STABLE based on the numerical equations from the same author (Nolan 1997), a framework for MATLAB (Liang and Chen 2013) or another for the R software (Wuertz and Maechler 2015).

Other implementations have centered in the performance of the methods. A first approach consists of using alternative methods to evaluate the equations: for example, Menn and Rachev (2006) approximate the Fourier inversion integral by means of the Simpson rule for a subset of the parameter space, Robinson (2014) uses interpolation formulae for log-stable distributions—these are α-stable distributions with maximum skewness to the right— and Lombardi (2007) and Koblents et al. (2016) use Monte Carlo methods for parameter estimation. A second approach is the use of parallelism, such as *libstable* (Royuela-del-Val 2016), which uses thread parallelism; or the software proposed by Belovas et al. (2013), which uses OpenMP to improve speed just on the maximum likelihood estimator. However, we have not found in the literature any development that accelerates all computations of this type of distribution using general purpose GPUs and maintaining a high level of accuracy.

Additionally, the implementation of the α-stable algorithms is difficult in parallel environments if we want to go beyond the simple parallelization of point computations, which is the trivial and common approach with other probability distributions where the PDF and CDF have simpler

[1] Available at J.P. Nolan's website: http://academic2.american.edu/~jpnolan.

[2] Available at Javier Royuela-del-Val's website: http://www.lpi.tel.uva.es/~jroyval/.

[3] https://github.com/hpcn-uam/libstable-opencl.

expressions. Only parallel implementations for fast calculation of more complicated functions, such as the inverse Poisson CDF for large numbers (Giles 2015) or for a parameter estimation algorithm via maximum likelihood (Hesterman et al. 2010), can be found in the literature.

In our case, we face not only complex equations, but also the need to calculate an integral by means of numerical methods. Adaptive quadrature methods such as Gauss–Kronrod are not well suited for parallel implementations due to their recursive nature: alternative methods such as (Thuerck et al. 2014) or (Arumugam et al. 2013) have been developed to bypass this issue. However, the use of dynamic, efficient subdivisions of the integration interval does not necessarily result in improved performance due to restrictions on the available resources, workgroup layout and to the increased computations required by these algorithms.

Our implementation takes a simpler approach, making use of the parallel capabilities of the GPUs to compute a high-order quadrature with a fixed number of subintervals. In cases where precision is not good enough, we resort to a quick check that considerably improves the precision, as we will explain in Subsect. 3.1.4.

2 Background

In this section we expose the equations used for the computation of α-stable distributions and the numerical integration algorithm that will be used.

2.1 Equations for α-stable distributions

α-stable distributions are modeled by four parameters. $\alpha \in (0, 2]$ is the stability index, $\beta \in [-1, 1]$ the skewness parameter, $\sigma > 0$ the scale parameter and $\mu \in \mathbb{R}$ the location parameter. σ and μ are explicitly not named standard deviation and mean of the distribution, despite being the common notation for these two concepts, because for α-stable distributions, standard deviation only exists for $\alpha = 2$, and the mean is only defined for $\alpha > 1$.

One of the main problems of α-stable distributions is the lack of closed formulas for the probability density function (PDF) and cumulative distribution function (CDF). However, the equations devised by Nolan (Nolan 1997) allow the numerical computation of the PDF and CDF. These equations use a re-parameterization of the location parameter μ to μ_0, where both values are related by (2):

$$\mu = \begin{cases} \mu_0 - \beta \tan\left(\frac{\alpha\pi}{2}\right)\sigma & \alpha \neq 1 \\ \mu_0 - \beta\frac{2}{\pi}\sigma \ln\sigma & \alpha = 1 \end{cases} \tag{2}$$

The equation for a standard[4] α-stable distribution (i.e., with location parameter $\mu = 0$ and scale parameter $\sigma = 1$) denoted by X are the following:

$$f_X(x; \alpha, \beta) = \begin{cases} \frac{\alpha}{\pi(x-\zeta)|\alpha-1|} \\ \quad \cdot \int_{-\theta_0}^{\frac{\pi}{2}} h_{\alpha,\beta}(\theta; x)\mathrm{d}\theta & x > \zeta \\ \frac{\Gamma(1+\frac{1}{\alpha})\cos\theta_0}{\pi(1+\zeta^2)^{\frac{1}{2\alpha}}} & x = \zeta \\ f_X(-x; \alpha, -\beta) & x < \zeta \end{cases} \tag{3}$$

$$F_X(x; \alpha, \beta) = \begin{cases} c_1(\alpha, \beta) + \frac{\operatorname{sign}(1-\alpha)}{\pi} \\ \quad \cdot \int_{-\theta_0}^{\frac{\pi}{2}} e^{-g_{\alpha,\beta}(\theta; x)}\mathrm{d}\theta & x > \zeta \\ \frac{1}{\pi}\left(\frac{\pi}{2} - \theta_0\right) & x = \zeta \\ 1 - F(-x; \alpha, -\beta) & x < \zeta \end{cases} \tag{4}$$

where

$$\zeta(\alpha, \beta) = -\beta \tan\left(\frac{\pi\alpha}{2}\right) \tag{5}$$

$$\theta_0(\alpha, \beta) = \frac{1}{\alpha}\arctan\left(\beta\tan\left(\frac{\pi\alpha}{2}\right)\right) \tag{6}$$

$$V_{\alpha,\beta}(\theta) = (\cos\alpha\theta_0)^{\frac{1}{\alpha-1}}\left(\frac{\cos\theta}{\sin(\alpha(\theta_0+\theta))}\right)^{\frac{\alpha}{\alpha-1}} \cdot \tag{7}$$

$$\cdot \frac{\cos(\alpha\theta_0 + (\alpha-1)\theta)}{\cos\theta} \tag{8}$$

$$g_{\alpha,\beta}(\theta; x) = V_{\alpha,\beta}(\theta) \cdot (x - \zeta)^{\frac{\alpha}{\alpha-1}} \tag{9}$$

$$h_{\alpha,\beta}(\theta; x) = g_{\alpha,\beta}(\theta; x)e^{-g_{\alpha,\beta}(\theta; x)} \tag{10}$$

$$c_1(\alpha, \beta) = \begin{cases} \frac{1}{\pi}\left(\frac{\pi}{2} - \theta_0\right) & \alpha < 1 \\ 1 & \alpha > 1 \end{cases} \tag{11}$$

However, the equations have a discontinuity when $\alpha = 1$. This is solved by changing the expressions to

$$f_X(x; 1, \beta) = \begin{cases} \frac{1}{2|\beta|}e^{-\frac{\pi x}{2\beta}}\int_{\frac{-\pi}{2}}^{\frac{\pi}{2}} h_{1,\beta}(\theta; x)\mathrm{d}\theta & \beta \neq 0 \\ \frac{1}{\pi(1+x^2)} & \beta = 0 \end{cases} \tag{12}$$

$$F_X(x; 1, \beta) = \begin{cases} \int_{\frac{-\pi}{2}}^{\frac{\pi}{2}} e^{-g_{1,\beta}(\theta; x)}\mathrm{d}\theta & \beta > 0 \\ \frac{1}{2} + \frac{1}{\pi}\arctan x & \beta = 0 \\ 1 - F_X(x; \alpha, -\beta) & \beta < 0 \end{cases} \tag{13}$$

with

$$g_{1,\beta}(\theta; x) = e^{-\frac{\pi x}{2\beta}}V_{1,\beta}(\theta) \tag{14}$$

$$h_{1,\beta}(\theta; x) = V_{1,\beta}(\theta) \cdot e^{-g_{1,\beta}(\theta; x)} \tag{15}$$

$$V_{1,\beta}(\theta) = \frac{2}{\pi}\left(\frac{\frac{\pi}{2} + \beta\theta}{\cos\theta}\right)e^{\frac{1}{\beta}(\frac{\pi}{2}+\beta\theta)\tan\theta} \tag{16}$$

[4] Evaluations for general distributions are calculated shifting and scaling the parameter x as usual.

This change does not imply a discontinuity: it has already been demonstrated (Nolan 1997) that this piecewise definition of the PDF and CDF is continuous.

From these expressions, it is clear that there is room for acceleration using parallel algorithms to integrate the expressions in (3), (4), (12) and (13): numerical integration algorithms rely on the evaluation of the integral at different points, with the evaluations being independent. Instead of relying only on one thread to do the evaluations serially, multiple threads can be scheduled so each one evaluates one point.

2.1.1 Gauss–Kronrod quadrature

As said before, Eqs. (3) and (12) can be calculated using numerical integration algorithms. However, not all algorithms are appropriate for this problem. Adaptive algorithms that rely on several iterations to achieve a precise value suffer from warp divergence issues, and the cost of the multiple serial evaluations of the integrand will also affect performance. An alternative approach is (Thuerck et al. 2014), where the authors devise the ∂^2 heuristic algorithm to generate integration intervals, avoiding recursion in the GPU while achieving the desired precision by the user.

This approach is, however, not appropriate for this situation. As exposed in the previous section, the function to be integrated changes with x, the point in which the PDF or CDF is evaluated at. Thus, in order to calculate the PDF or CDF at n points, it would be needed to apply n times the ∂^2 heuristic, which brings in a significant performance loss, given the cost of the function evaluation.

Our solution requires the use of static quadrature rules without iterations, avoiding thus the cost of several serialized calculations of the integrand. It might use more intervals than the ∂^2 heuristic algorithm, but as the GPU workgroups can only be of certain sizes, the reduction in the used number of intervals would not necessarily imply a reduction in the resources used. Given that the integral to calculate the probability distributions is one-dimensional, the number of required intervals is not high and thus, specialized algorithms that dynamically create the necessary subdivisions, such as (Arumugam et al. 2013), are not needed and their extra performance cost can be avoided.

For the integration of each subinterval we have chosen Gauss–Kronrod quadrature, as it yields accurate results and error estimates which are quick to compute and do not rely on the different results between iterations.

Gaussian quadrature states that

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=1}^{n} w_i \cdot f(x_i) \tag{17}$$

for a certain set of points $x_i \in \mathcal{X}_n$ and corresponding weights $w_i \in \mathcal{W}_n$. The Gauss–Kronrod quadrature rules (Kronrod 1965) are a common variant where a first set of n nodes is chosen and then extended with $n + 1$ additional points.

```
function GK_INTEGRATE($f$, $a$, $b$)
    gauss ← 0
    kronrod ← 0
    center ← $\frac{b-a}{2}$
    for $i = 0$; $i < \frac{2n+1}{2}$; $i++$ do
        $x \leftarrow \mathcal{X}[i]$
        $v \leftarrow f(\text{center} + x)$
        if $x \neq 0$ then        ▷ Avoid evaluating twice at the 0 node
            $v \leftarrow v + f(\text{center} - x)$
        end if
        gauss ← gauss + $v \cdot \mathcal{W}_G[i]$
        kronrod ← kronrod + $v \cdot \mathcal{W}_K[i]$
    end for
    result ← kronrod
    error ← |kronrod − gauss|
    return result, error
end function
```

Fig. 1 Pseudo-code for a serial implementation of the Gauss–Kronrod quadrature rule, where $\mathcal{X}$ is the set of nodes and $\mathcal{W}_G, \mathcal{W}_K$ the corresponding weights for the Gauss and Kronrod rules. $\mathcal{W}_G[i]$ is zero if i is only a Gauss–Kronrod node

Thus, an evaluation of the function in the set of $2n + 1$ Gauss–Kronrod nodes yields both a high order estimate for the integral and an error estimate. Figure 1 shows pseudo-code for a traditional implementation of the Gauss–Kronrod quadrature.

3 Proposed algorithms and implementation

In this section, we will show the algorithms used to parallelize the PDF and CDF evaluations (Subsects. 3.1, 3.2), including an approach for simultaneous calculation of both in Subsect. 3.3, and also the algorithms for the quantile function and parameter estimation (Subsects. 3.4, 3.5). A discussion of limitations imposed by the hardware and the framework is presented in Subsect. 3.6.

For the sake of brevity, we do not we do not describe the implementation of our parallel random number generator, based on the work developed by Weron and Weron (1995), as it is a straightforward parallelization.

3.1 PDF evaluation

The evaluation of the α-stable PDF requires to ascertain the value of an integral by numerical methods. The approach presented in *libstable* (Royuela-del-Val 2016), using the Gauss–Kronrod quadrature, is very well suited for general purpose GPUs if the iterative subdivision algorithm is replaced by a fixed partitioning that returns precise enough results: given the cost of the integrand evaluation and the parallelization capabilities of GPGPUs, it is more efficient to calculate a large number of points at the same time than calculating less points but iteratively.

In order to use the full capabilities of the GPU processor, the workload has been scheduled in the best possible way to reduce memory access times and improve parallelism. The memory layout avoids any memory bottlenecks: all the threads access memory positions either sequentially, when storing the partial results, or by broadcasting when retrieving constants from the global memory space. The algorithm does not suffer from divergent threads, as all of them execute the same algorithm and go through the same branching paths.

Our algorithm is divided in three sections: Subsect. 3.1.1 explains the workgroup layout chosen to take advantage of the GPU capabilities to calculate the integral, the procedure for the evaluation of the function at the necessary points is detailed in Subsects. 3.1.2 and 3.1.3 shows how the final calculations are done, avoiding large performance hits due to memory sharing and synchronization issues. Finally, some precision issues found during the development and the solutions implemented are discussed in Subsect. 3.1.4.

3.1.1 Workgroup layout

First, the global arguments (parameters of the distribution, pre-calculated values and indicators of the equations that should be used) are transferred to the GPU constant memory space (NVIDIA 2009a). The points to be evaluated are sent to global memory space. The two buffers to hold the results (Gauss and Kronrod sums) are created and their addresses passed as arguments to the kernel.

Once the memory layout is ready, the kernel is enqueued to compute the numerical integration in the GPU. The integration interval in (3) is divided in a fixed number of subintervals. The Gauss–Kronrod quadrature algorithm is then applied to each one of these intervals.

This approach allows a natural two-dimensional, static workgroup layout for the GPU (see Fig. 2): each local workgroup is responsible for the evaluation of a single point, and in that workgroup the (i, j) item will calculate the jth Gauss–Kronrod point of the ith subinterval. With this notation, there will be I subintervals, J Gauss–Kronrod points and a total of $I \cdot J$ threads per workgroup.

The Gauss–Kronrod quadrature method involves two identical sets of operations on each point: calculation of the value where the function should be evaluated, evaluation of the function to integrate and then multiplication by the cor-

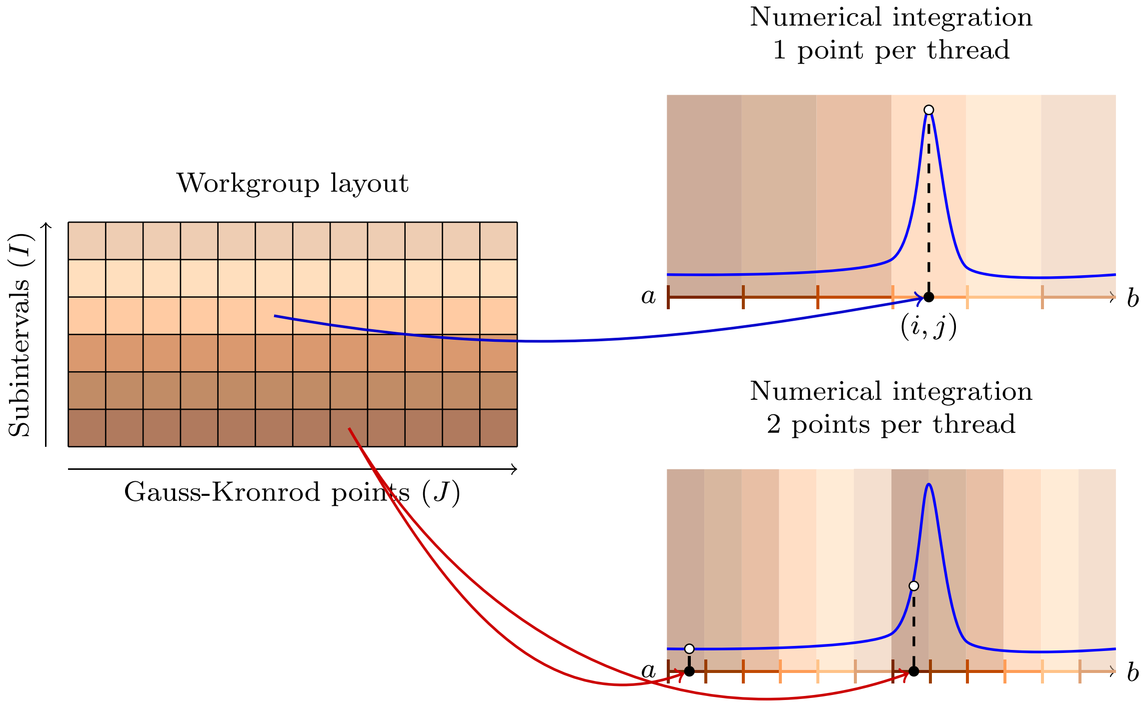


Fig. 2 Workgroup layout and integration strategy. Each one of the I rows in the workgroup maps to a subdivision of the integration interval. The jth column integrates evaluates the integral at the jth Gauss–Kronrod point of the corresponding subinterval. When using multiple points per thread, one thread evaluates the function at more subintervals. Not pictured for simplicity: each point is actually two points located the same distance from the subinterval center, as the Gauss–Kronrod rule is symmetric

responding weight of the point. The only difference between the Gauss and Kronrod quadratures is the weight assigned to each point (which will be 0 in points not pertaining to the Gauss quadrature).

This, together with the fact that Gauss–Kronrod quadrature is symmetrical, allows the extensive use of vector operations to improve performance in the kernel and reduce the number of instructions.

3.1.2 Point evaluation procedure

The two values where the function will be evaluated are obtained as a vector named $\mathbf{x}$

$$\mathbf{x} = \begin{pmatrix} a \\ a \end{pmatrix} + l \cdot \left[\begin{pmatrix} i + 0.5 \\ i + 0.5 \end{pmatrix} + \begin{pmatrix} \text{abscissa}[j] \\ -\text{abscissa}[j] \end{pmatrix} \right] \tag{18}$$

where i is the subinterval index, j is the index of the point to be calculated, abscissa is an array holding the offsets for the Gauss–Kronrod quadrature points (stored in constant memory space), a is the beginning of the whole integration interval and l is the length of the subinterval, calculated as $l = \frac{L}{I}$ with L the length of the entire integration interval.

Equation (18) first calculates the center of the subinterval and then adds the corresponding Gauss–Kronrod abscissas, which are symmetric with respect to the origin (in this case, the origin is the subinterval center). Finally, the result is properly scaled and translated to fit with the integration interval.

The integrand is then evaluated at those points using OpenCL's vector operations, thus obtaining two results with just one call to the function to be integrated. Both results are added and the resulting sum is multiplied by the vector of weights, of which the first coordinate is the Kronrod quadrature weight and the second one is the Gauss weight. The final result of the evaluation is a vector with the values of both Gauss and Kronrod quadrature rules at the given point.

Our software allows the use of larger vectors to bypass limitations on the size of workgroups (see Subsect. 3.6 for an explanation) and to increase the number of subdivisions of the integration interval. We can substitute two-dimensional vectors by four or eight-dimensional vectors to evaluate, respectively, two or four subintervals per thread, thus doubling or quadrupling the subdivisions of the integration interval and obtaining more precise results. In this case, the vector $\mathbf{x}$ from (18) would be calculated instead as

$$\mathbf{x} = \begin{pmatrix} a \\ \vdots \\ a \end{pmatrix} + l \cdot \left[\begin{pmatrix} I \cdot 0 + i + \frac{1}{2} \\ I \cdot 0 + i + \frac{1}{2} \\ \vdots \\ I(n-1) + i + \frac{1}{2} \\ I(n-1) + i + \frac{1}{2} \end{pmatrix} + \begin{pmatrix} \text{abscissa}[j] \\ -\text{abscissa}[j] \\ \vdots \\ \text{abscissa}[j] \\ -\text{abscissa}[j] \end{pmatrix} \right] \tag{19}$$

with n the number of points being evaluated per thread (1, 2 or 4) and I the second dimension of the workgroup size.

The main change in (19) is that, apart from using wider vectors (size $2n$), we assign to each thread n different subintervals. For example, using 16 subdivisions with 2 points per thread, threads with subinterval index $i = 1$ would integrate points in the subintervals 1 and 9. It is also easy to see that, when $n = 1$, (19) is equivalent to (18).

A downside of this implementation is the fact that some hardware may serialize the vector operations if the vectors are too wide. In this case, instead of executing one instruction per operation with vectors of size 8, it may execute two instructions, each one computing 4 components of the vector. This issue may result in noticeable performance impact depending on the hardware and the points per thread used.

In our tests, we have found that a total of 16 subdivisions with two points per thread achieves the best balance between performance and precision. Increasing the number of points per thread affects performance without meaningful precision improvements. The results discussed in Sect. 4 use this setting by default.

3.1.3 Final result calculations

Once all the points have been evaluated, a local memory barrier command is issued to synchronize all the threads in the local workgroup, and the sum of the values of every point is calculated using a reduction in $O(\log_2 n)$ operations that maximizes thread usage and memory coalescing. Local memory barriers are used as the synchronization mechanism between threads, as it is the most efficient way to complete the sums.

This reduction is first applied to the points in each subinterval and then to the partial sums of each subinterval. A detailed pseudo-code algorithm is presented and explained in Fig. 3.

3.1.4 Precision issues when $x \to \infty$ or $x \to \zeta$

The integrand function $h_{\alpha,\beta}$ comes closer to a singular peak when $x \to \infty$ or when $x \to \zeta$ (see Fig. 4). This behavior reduces the precision of the numerical method considerably.

In the special case of $x \to \zeta$, the specific formula for $x = \zeta$ from (3) can be used when x is in a small interval around ζ. However, this is not enough as the intervals where the approximation is valid are not big enough, and neither solves the precision problem when $x \to \infty$.

In previous implementations (Royuela-del-Val 2016), the proposed solution to the problem is the determination of that peak using numerical methods and the usage of different integration methods around that peak. However, we have found a more suitable approach for our implementation.

```
Require: subinterval_index $i$, GKpoint $p$
  for $o$ = points_count/2; $o > 0$; $o >>= 1$ do
    if $p < o$ then
      $\text{sums}[i][p] \leftarrow \text{sums}[i][p] + \text{sums}[i][p+o]$
    end if
    BARRIER(local)
  end for
  for $o$ = subintervals_count/2; $o > 0$; $o >>= 1$ do
    if $i < o$ then
      $\text{sums}[i][p] \leftarrow \text{sums}[i][p] + \text{sums}[i+o][p]$
    end if
    BARRIER(local)
  end for
```

Fig. 3 Pseudo-code for the reduction that returns the final results, where »= is the shift and assignment operator. First, the threads for each interval perform a parallel sum of all the results of each point, stored in a two-dimensional array sums of dimension subintervals_count × points_count. Partial results are also stored there. At the end of the loop, the values sums[i][0] will contain the Gauss–Kronrod results for each subinterval. The procedure is then repeated with those results

As GPGPUs are highly parallelizable, the cost in cycles of increasing the number of nodes of the Gauss–Kronrod quadrature is almost negligible and only affects the cycles invested in the summation of all the results. But as the summation is done in $O(\log_2 n)$ time, we can effectively double the degree or the number of subintervals with minimal performance impact.

Thus, high-order Gauss–Kronrod quadrature formulas are used to reduce considerably the impact of the peaks without needing additional measures.

However, when $\alpha > 1$, increasing the points used in Gauss–Kronrod may not be enough to achieve the desired precision or is not even possible: GPGPUs limit the size of local workgroups, so there is a ceiling in the number of points that can be used (this issue is further discussed in Subsect. 3.6).

This could be considered as a failure of the single-iteration approach to the numerical integration stated at the beginning of Subsect. 3.1, with a possible solution being the use of adaptive algorithms such as (Thuerck et al. 2014) in these difficult parameter cases. However, we have solved this problem without resorting to such complex and costly approaches: we use simple checks based on knowledge of the specific integrand that do not affect performance significantly and achieve the desired precision.

After obtaining the Gauss–Kronrod result for each interval, the first thread of each subinterval (i.e., the $(i, 0)$ thread) checks if its contribution is greater than a certain threshold (experimentally chosen in order to achieve enough precision). This way, the interval in which the integrand $h_{\alpha,\beta}$ has significant values is detected.

If this contributing interval is too small (again, the threshold has been determined experimentally in order to achieve the desired precision), it will be an indicator of the presence of a sharp peak. A reevaluation is then triggered and the local workgroup reevaluates the integrand in that contributing interval, increasing precision. This process can be repeated again if the contributing interval is still too small.

This method avoids large performance hits in evaluations in which the single pass evaluation is good enough (there is only an additional local memory barrier, the rest is achieved using atomic operations), and returns precise results when the integrand comes close to a single point; all of this while maximizing GPGPUs parallel capabilities.

Another precision issue happened with small values of the stability index ($\alpha < 0.3$). As shown in Fig. 5, the integrand increases abruptly at the beginning of the interval and decreases the precision of the numerical integration. The method of contributing subintervals exposed above does not detect this issue. In order to improve the accuracy in these cases, we force instead a reevaluation in the beginning of the integration interval.

3.2 CDF evaluation

The evaluation of the CDF uses the same algorithms presented in the previous section. Because of the similarity of

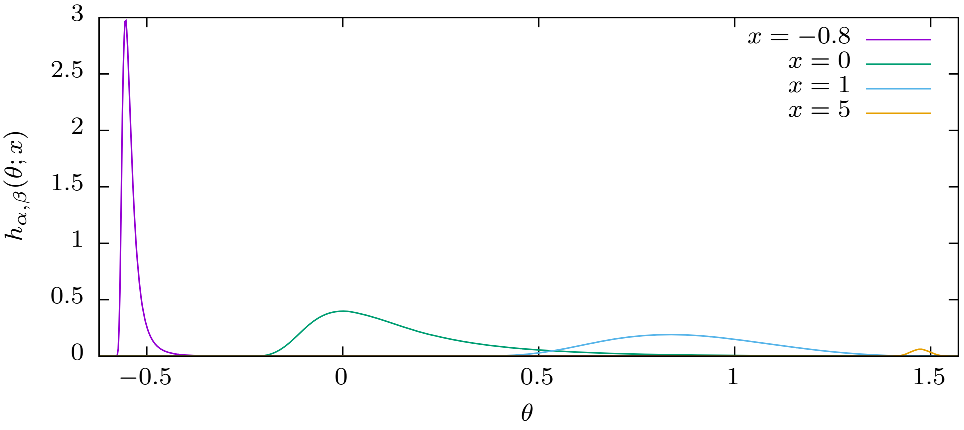


Fig. 4 Behavior of the function $h_{\alpha,\beta}(\theta; x)$ from (10), for $\alpha = 1.2$ and $\beta = -0.3$. For these values, $\zeta \approx -0.923$: it can be seen that when x tends to that value, h behaves like a singular peak

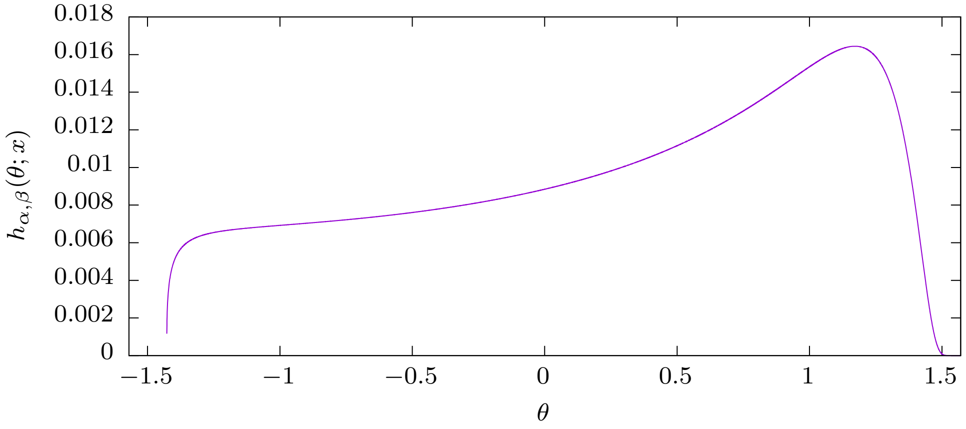


Fig. 5 Behavior of the function $h_{\alpha,\beta}(\theta; x)$ from (10), for $x = 22$, $\alpha = 0.3$ and $\beta = 0.9$. In this case, there is a sharp increase at the beginning of the interval that must be integrated carefully to reduce the error

the equations for the PDF and the CDF (see (3), (4) and (12), (13)), the code used is exactly the same: a flag decides whether the function to compute will be the PDF of the CDF. Depending on the flag, the code will compute accordingly the multiplication factors of the integral and, in the case of the CDF, the number to be added after the integration is completed ($c_1(\alpha, \beta)$ or $1 - c_1(\alpha, \beta)$ depending on whether x is greater or less than ζ).

The integrand of the CDF evaluation is better behaved than the PDF one, so no additional measures had to be taken to achieve significant precision.

3.3 Combined PDF and CDF evaluation

As explained above, an advantage of the equations used for the evaluation of the PDF and CDF ((3), (4) and (12), (13)) is that they are similar. The integrand of the PDF is the same as the one of the CDF except for one additional operation, for all values of α.

This allows computing simultaneously the PDF and CDF values, a feature that will become especially useful when computing the quantile function (Subsect. 3.4). However, in our implementation of this dual mode (called *PCDF* throughout the code) error estimates are not generated, as those would increase the complexity of the code, and would subsequently affect performance.

The evaluation code is exactly the same as the one for the PDF and CDF evaluations. The difference is that, when calculating the integrand at the necessary abscissas as explained in Subsect. 3.1.2, the code does not return a pair with the values of the Gauss and Kronrod quadrature nodes (that is, the integrand multiplied by the corresponding Gauss or Kronrod weights), but instead returns a pair with the values of the Kronrod quadratures for the PDF and CDF. The rest of the procedure is the same, with the kernel returning two arrays. The host code will detect that a PCDF evaluation has been issued and will return the two arrays to the client code.

3.4 Quantile function evaluation

Given a probability p, the quantile function $Q(p)$ specifies the value for which the probability of the random variable being less than or equal to this value is equal to the given probability. Formally, that is expressed as

$$Q(p) = \inf_{x \in \mathbb{R}} \{p \leq F_X(x)\} \tag{20}$$

where F_X is the CDF. When F_X is continuous, as it is in the α-stable case, the quantile function can be simplified as the inverse of the CDF: $Q = F_X^{-1}$.

There is not a closed analytic formula for a general quantile function. Thus, it requires a numerical inversion of the CDF function: given a probability p for an α-stable distribution with parameters α, β, μ, σ, the equation

$$\phi(x) = F_X(x; \alpha, \beta, \mu, \sigma) - p = 0 \tag{21}$$

has to be solved for x to obtain the result.

To find that root and obtain the desired results we have used the Newton method. This algorithm uses knowledge of the derivative of the function, achieving fast rates of convergence. The successive points are calculated using the following equation, beginning with an initial guess x_0:

$$x_{n+1} = x_n - \frac{\phi(x_n)}{\phi'(x_n)} \tag{22}$$

with the error estimation calculated as

$$\epsilon = \left| \frac{x_{n+1} - x_n}{x_{n+1}} \right| \tag{23}$$

The algorithm iterates until the desired accuracy is achieved. The fast convergence of the Newton method allows us to move completely the algorithm to the GPU.

The implementation of the algorithm uses the same code that would be used in a regular implementation for com-

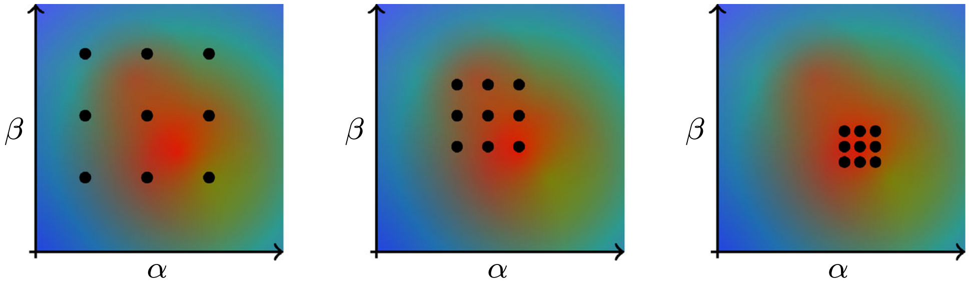


Fig. 6 An illustration of the contracting grid algorithm to find the maximum of the log-likelihood function

mon programming languages. Each workgroup calculates one quantile value to allow parallel calculation of multiple quantiles. The first thread of each workgroup determines the initial guess using interpolation on precalculated values, obtains its corresponding PDF and CDF values and calculates the next guess and error estimate. These steps are repeated until the error estimate is lower than the desired accuracy. When the iterations stop, the thread saves the result and error estimate in a global array: once all the workgroups finish their evaluations, the kernel will finish and control will return to the host code.

The differences with a regular Newton algorithm implementation come from the fact that the calculation of PDF and CDF values requires multiple threads with the workgroup layout from Subsect. 3.1.1. Thus, the point where the PDF and CDF are going to be evaluated is distributed to all the threads of the workgroup using local memory and a single barrier. The error estimate is transmitted in the same way, so all the threads of the workgroup finish at the same time: when the guess has reached enough precision.

3.5 Parameter estimation

Once the algorithm for the parallel evaluation α-stable PDF is implemented, an immediate application is parameter estimation. Given the cost of the PDF evaluation, maximum likelihood estimators have not been a practical option, and alternative approaches based on other estimators have been proposed (Koutrouvelis 1981; McCulloch 1986).

However, these methods are iterative so their parallel implementation is not straightforward. On the other hand, the presence of a PDF evaluation algorithm in GPGPUs facilitates the implementation of a parallel maximum likelihood estimator for the four parameters of the distribution. As the likelihood function of α-stable distributions has a single maximum and evolves smoothly (DuMouchel 1973), the estimator is consistent. Thus, this has been the parameter estimation method finally implemented.

The search algorithm has been chosen to maximize the use of the parallel processor. A contracting grid search algorithm (Hesterman et al. 2010) evaluates multiple points per iteration (see Fig. 6), so it is well suited for GPGPUs.

Before using the contracting grid search algorithm a first rough estimate is calculated using McCulloch's estimators. They calculate quickly a first estimate that can be used to reduce the search space.

This first estimate is used as the center of the grid. Our software sets the grid width and calculates the set of points in the parameter space where the likelihood should be evaluated. These evaluations are also done with OpenCL in the GPU in order to improve performance.

The point with the maximum likelihood is set as the new center of the grid and a new set of points is calculated with a narrower grid. This continues until the grid is smaller than the error tolerance or when the maximum difference of likelihood between the points of the grid is small enough.

The use of McCulloch's as initial estimations impose a limitation on our MLE, which is the impossibility to fit stable data with $\alpha < 0.6$: McCulloch's estimators are not valid in that region. Without initial estimations, a search in the whole space of parameters (specifically, location and scale parameters) is not feasible and our ML estimator will not return meaningful results.

To further improve performance with regards to the work of Hesterman et al. (2010), OpenCL kernel transfer and setup costs are reduced using asynchronous commands and multiple queues. Consequently, all the components in the pipeline (host CPU, PCI memory transfer bus and GPU processor) are used simultaneously, reducing waiting times and speeding up execution.

Another possibility that our software can use to improve performance is to rely on McCulloch's estimators for the parameters μ and σ, and setting the grid estimation only for α and β. On each iteration, the estimation of μ and σ is recalculated with the new α, β values to further improve precision.

3.6 Hardware limitations

During the development, we have faced hardware limitations that forced us to change the initial approach. The main inconvenience has been the limitation on the size of workgroups.

OpenCL workgroups are not of unlimited size: the maximum number of local work items depends on the hardware

Table 1 Specifications for the test devices

	Global bandwidth (GBps)	Local bandwidth (GBps)	Core clock (GHz)	Core count	Compute units	PCIe bus
Tesla M	177	2662	1.300	512	16	PCIe 2 ×16 (8 GB/s)
Tesla K	288	17,165	0.745	2880	90	PCIe 3 ×16 (15.75 GB/s)
AMD	352	2816	1.000	2816	44	PCIe 2 ×16 (8 GB/s)

and the kernel complexity. In order to work with multiple points and multiple dimensions, we had to reduce the number of subdivisions, a solution that caused some precision issues, as discussed in Subsect. 3.1.4. Even with this fix implemented, the limit on the number of subdivisions is still present so, depending on the used GPU model, the precision can decrease significantly if enough workgroups are not available. Dynamic parallelism (i.e., spawning new kernels) was not possible because it is not supported in the OpenCL versions we have used.

These hardware limitations also discarded a single-kernel, all-GPGPU approach for parameter estimation, having to resort to multiple command queues. The former approach would have required even bigger workgroups that would not have been supported by our test hardware.

Although OpenCL is presented as a framework to develop parallel algorithms independently of the underlying hardware, we have found that the hardware actually matters. For example, our software can't compile the kernel when used on OpenCL platforms without support for double-precision numbers, or does not work on some CPUs due to hard limits on the size of workgroups, and requires capabilities not available in some platforms, such as atomic operations or vector operations.

4 Results

In this section, we expose the results obtained by our developed software and compare them with *libstable* as the current fastest serial implementation (Royuela-del-Val 2016). We describe our testing devices in Subsect. 4.1, show the results for the PDF evaluation, CDF evaluation, quantile function and parameter estimators in Subsects. 4.2, 4.3, 4.5 and 4.6 respectively. Additionally, in Subsect. 4.7 we analyze the performance parameters of our code to find the bottlenecks.

As we explained at the beginning of Sect. 3, for brevity we do not provide detailed results of the straightforward parallelization of the random number generator algorithm proposed by Weron and Weron (1995). As expected, we checked that it that performs better than the serial counterpart [the GNU Scientific Library implementation (Gough 2009)] for enough random numbers generated (in our tests, 1000 or more).

4.1 Testing devices and environment

We have tested our application in three different GPUs, named as follows:

- *TeslaM* A NVIDIA Tesla M2090 GPU, Fermi architecture, with 6GB of GDDR5 memory and 512 cores at 1.3 GHz. It offers a memory bandwidth of 177 GB/s.
- *TeslaK* The most advanced card in our test setup, an NVIDIA Tesla K40 card, Kepler architecture, targeted for high performance servers and workstations. This card has 12 GB of GDDR5 memory and 2880 cores at 745 MHz. The memory bandwidth is 288 GB/s.
- *AMD* An AMD/ATI Radeon 290X GPU, GCN architecture, with 4GB of GDDR5 memory and 2816 cores running at 1 GHz. The offered memory bandwidth is 352 GB/s.

Table 1 shows the relevant performance details for each GPU card. The data has been retrieved from the specifications of the vendors (NVIDIA 2012, 2013; AMD 2013). The local memory bandwidth (referred to as "shared memory" in NVIDIA documentation) for the whole device has been calculated from those specs and from the corresponding computing architectures (NVIDIA 2009b, 2014; AMD 2012) as follows:

$$\text{local bw} = \text{bank bw} \cdot \text{banks / SU} \cdot \frac{\text{cores}}{\text{cores / SU}} \cdot \text{core clock} \tag{24}$$

where the bank bandwidth is expressed as the number of bits that can be read/written per processor. The scheduling unit (SU, referred to as "wavefronts" in AMD's architectures and "warps" in NVIDIA's) are groups of 32 and 64 cores in NVIDIA and AMD architectures, respectively.

All the devices ran the same code, which is the main advantage of using OpenCL. The OpenCL version used has been 1.1 in the NVIDIA cards, as it does not offer drivers for newer versions of the framework. The AMD card used OpenCL version 2.0. This has to be taken into account when evaluating the results, as the newer improved versions offer better performance and precision. The only activated option for the compilation of the OpenCL kernel is `-cl-no-signed-zeros`. The compilers used are the

Table 2 Precision results when $x \in (-100, 100)$ for the PDF of a standard ($\mu = 0, \sigma = 1$) stable distribution

α	β	Abs. error	Rel. error	Precision
0.25	0	5.11×10^{-14}	8.64×10^{-11}	1.14×10^{-9}
0.25	0.5	5.96×10^{-14}	1.05×10^{-10}	1.31×10^{-9}
0.25	1	5.29×10^{-18}	2.52×10^{-16}	3.26×10^{-17}
0.5	0	1.36×10^{-19}	2.32×10^{-16}	4.2×10^{-17}
0.5	0.5	1.08×10^{-19}	2.06×10^{-16}	6.43×10^{-17}
0.75	0	2.71×10^{-19}	9.22×10^{-16}	3.77×10^{-10}
0.75	0.5	2.71×10^{-19}	8.71×10^{-16}	6.72×10^{-10}
0.75	1	4.13×10^{-18}	4.05×10^{-16}	1.24×10^{-10}
1.25	0	5.58×10^{-16}	1.26×10^{-11}	1.07×10^{-16}
1.25	0.5	4.48×10^{-16}	1.23×10^{-11}	1.43×10^{-11}
1.25	1	2.78×10^{-17}	1.81×10^{-15}	1.81×10^{-11}
1.5	0	2.37×10^{-16}	2.96×10^{-11}	1.05×10^{-16}
1.5	0.5	2.13×10^{-16}	2.93×10^{-11}	1.09×10^{-16}
1.5	1	2.17×10^{-19}	4.11×10^{-16}	7.59×10^{-12}

ones included in the respective SDKs: CUDA 6.0 in the NVIDIA cards and AMD APP SDK 2.9 in the AMD case.

The comparisons have been made with *libstable* running on an Intel Core Xeon E5-2630 v2 with 12 cores running at 2.60 GHz, compiled with GCC 4.7.2 with options `-O3 -march=native` and linked against Fedora's official build of the GSL library, version 1.15.

4.2 PDF evaluation

Despite certain limitations imposed by the hardware (see Subsect. 3.6), Table 2 shows that in the interval (−100, 100) our software achieves reasonable precision in comparison with the software *libstable* (Royuela-del-Val 2016): absolute error is small, both calculated as the difference with *libstable* and as the difference between Gauss and Kronrod quadrature rules, nearing machine precision in some instances. Relative error committed is also small, below 1.05×10^{-10} in every instance.

The error is measured as the median of the absolute differences between the result of our software and the one of the *libstable* software (Royuela-del-Val 2016) taken as reference. The precision is the estimated relative error committed, calculated from the difference between Gauss and Kronrod quadrature rules.

Regarding performance, the parallel PDF evaluation considerably improves performance when the number of points to be evaluated is significant enough (e.g., a *libstable* execution can be faster when evaluating just one point). In our tests, 1000 point batches showed a significant speedup: from 0.031055 ms per point (32,200.93 point evaluations per second) with *libstable* to 0.003 ms per point or 333,333.33 point evaluations per second on a NVIDIA Tesla K40 card, which makes it 10.35 times faster.

Our solution is even faster than *libstable* using 12 threads on an Intel Core Xeon CPU, that despite showing significant performance improvements with regards to the single threaded tests (0.007202 ms per point or 138,850.32 points per second with 1000 points) is still slower than our solution running both in the Tesla K40 and in the Tesla M2090, as the last one achieves 0.0042 ms per point (238,095.24 points per second).

Figure 7 shows the evolution of performance depending of the number of points being evaluated in different hardware, and Table 3 shows the exact performance measures for 1000 points. It can be observed that there are not further performance gains after a certain number of points: this is caused by the fact that GPUs cannot absorb an unlimited number of

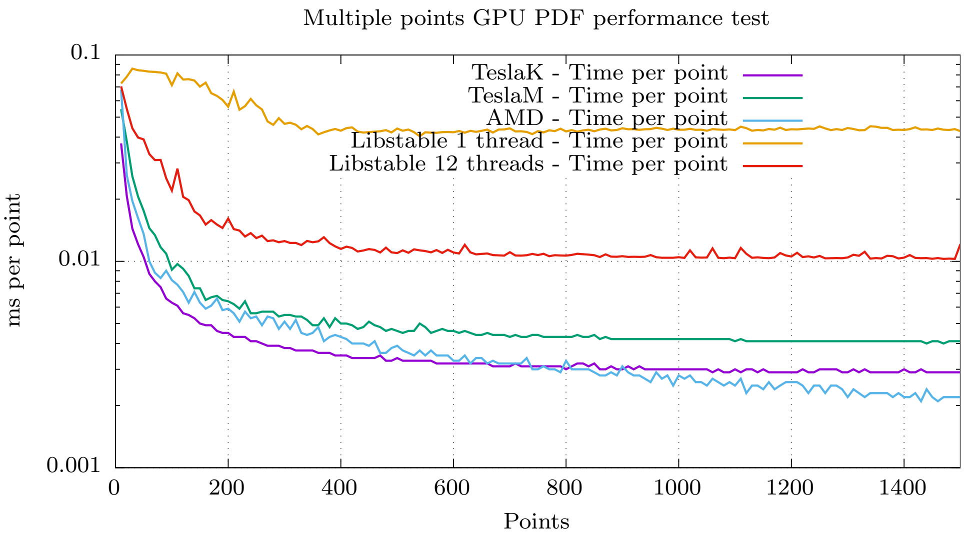


Fig. 7 Performance of the PDF calculation in different GPU cards in comparison with the results obtained with *libstable* on an Intel Core Xeon CPU, depending on the number of points evaluated. Hardware details are exposed at the beginning of Sect. 4

Table 3 Summary of the different PDF performance measures for 1000 points

	Milliseconds / point	Points per seconds
Tesla K	0.003	333,333.33
Tesla M	0.0042	238,095.24
AMD	0.0028	357,142.86
Libstable 1 thread	0.0311	32,200.93
Libstable 12 threads	0.0072	138,850.32

workgroups; instead, the workgroups are separated in batches and their execution is serialized.

An interesting result is the comparison between the NVIDIA cards and the AMD Radeon R9 290X GPU. First, we have to take into account that the AMD card does not support workgroups as big as the NVIDIA ones, so we had to halve the number of integration subintervals. To account for the decrease in precision, we doubled the number of points per thread, so our software used vectors of 8 doubles in each thread, a change that decreases performance.

Even with this handicap, for 1000 points, the AMD GPU achieves significant performance speedups (0.0028 ms per point and 0.0028 points per second), taking only 93.33% of the time of the high-end NVIDIA Tesla K40 computing card, and running faster than the multi-threaded *libstable* on an Intel Core Xeon CPU.

Possible reasons for the comparable speeds on such different cards (the AMD is a gaming card, while the Tesla are specialized for high-performance computing) could be the high memory bandwidth in the AMD and the upgraded OpenCL version (AMD supports OpenCL 2.0, while NVIDIA only has OpenCL 1.1 drivers), which includes improved memory coalescing features and better overall performance. It has also been shown (Fang et al. 2011) that NVIDIA cards show better performance with CUDA than with OpenCL. We explore further this issue in Subsect. 4.7.

Figure 8 shows how the performance varies depending on the parameters used. Our software is faster when $\alpha \in (0.4, 1)$ as that is the region where integration becomes easier and does not require further passes to improve precision. When α comes closer to 2, the integrand behaves more like a singular peak and multiple passes are required, thus slowing down integration. The skewness parameter β does not affect significantly the execution time.

4.3 CDF evaluation

As explained in Subsect. 3.2, the function to integrate is better behaved in the CDF than in the CDF. This translates to higher precision (see Tables 4, 5 for a comparison with *libstable* taken as reference) without the need for additional measures. Most of the relative error is near machine precision, and the lowest precision (4.99×10^{-11}) occurs in extreme regions of the parameter space.

The behavior of the integrand also affects performance: as Figs. 9 and 10 show, the CDF is slightly faster than the PDF as it does not need as much reevaluations to achieve significant precision. It is not unexpected given the fact that the majority of the code is shared between the two calculations.

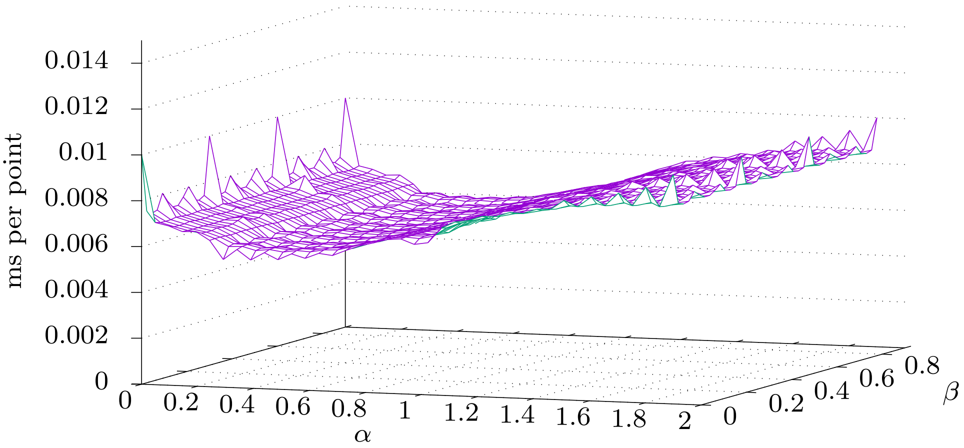
As with the PDF, the CDF performance is not especially affected by the skewness parameter β, being the stability index α the one determining the evaluation time.

4.4 Combined PDF and CDF calculation

As explained in Subsect. 3.3, the similarity of the CDF and PDF integrand functions allows our software to calculate simultaneously both functions without significant performance decreases.

The comparison with *libstable* shown in Fig. 11 and Table 6 has been made calling its CDF and PDF functions separately, while using the simultaneous calculation in the GPU. It is not a fair comparison but shows how the GPU capabil-

Fig. 8 Variation of the performance depending on the parameters. The software was tested with 500 points in a NVIDIA Tesla M2090



ities can be used and demonstrates an important advantage for applications that require the calculation of the PDF and CDF values simultaneously.

Table 4 Precision results when $x \in (-100, 100)$ for the CDF of a standard ($\mu = 0, \sigma = 1$) stable distribution

α	β	Abs. error	Rel. error	Precision
0.25	0	7.65×10^{-12}	4.99×10^{-11}	4.12×10^{-10}
0.25	0.5	1.24×10^{-11}	4.47×10^{-11}	4.72×10^{-10}
0.25	1	5.55×10^{-17}	1.49×10^{-16}	1.92×10^{-17}
0.5	0	6.66×10^{-16}	3.83×10^{-15}	1.45×10^{-16}
0.5	0.5	7.39×10^{-16}	7.5×10^{-15}	1.42×10^{-16}
0.75	0	6.66×10^{-16}	2.58×10^{-15}	3.04×10^{-12}
0.75	0.5	7.1×10^{-16}	1.82×10^{-15}	2.89×10^{-13}
0.75	1	3.33×10^{-16}	3.62×10^{-16}	5.98×10^{-14}
1.25	0	7.55×10^{-16}	1.51×10^{-15}	1.51×10^{-16}
1.25	0.5	7.36×10^{-16}	2.17×10^{-15}	2.05×10^{-9}
1.25	1	5.55×10^{-16}	6.67×10^{-16}	3.2×10^{-9}
1.5	0	7.46×10^{-16}	1.43×10^{-15}	4.89×10^{-15}
1.5	0.5	7.07×10^{-16}	1.64×10^{-15}	1.39×10^{-14}
1.5	1	5.55×10^{-16}	6.67×10^{-16}	6.62×10^{-16}

Table 5 Summary of the different CDF performance measures for 1000 points

	Milliseconds / point	Points per seconds
Tesla K	0.0029	344,827.59
Tesla M	0.0041	243,902.44
AMD	0.0024	416,666.67
Libstable 1 thread	0.0525	19,042.18
Libstable 12 threads	0.0115	87,244.81

The time consumed by the simultaneous calculation is almost the same as the required by the PDF and CDF evaluations separately. The only disadvantage of this simultaneous calculation approach is that error estimates are not calculated as explained in Subect. 3.3. Precision-wise, the results are the same than the ones returned by the standalone CDF or PDF functions.

4.5 Quantile function

To evaluate the quantile function results, we have generated a set of equally spaced points in the real line, then obtained their CDF values and used our quantile function, comparing its output with the original points to validate precision. We have filtered out quantiles below 0.1 or above 0.9: given the characteristic heavy tails of the α-stable distribution, results in those regions will not be meaningful as the CDF grows too slowly.

Table 7 shows the precision achieved by our software with a tolerance setting of just 10^{-4}. The precision can be set as high as desired, but we have found this tolerance setting returns precise values with very good performance. Figure 12 shows the evolution of the quantile function performance depending on the number of evaluated points, with exact numbers for 1000 points in Table 8.

Our solution is considerably faster than *libstable*: for 1000 points, the Tesla K is 5.8 times faster than the quantile function from *libstable* running with 12 parallel threads on the Intel Xeon CPU.

4.6 Parameter estimation

To validate the maximum likelihood estimation algorithm presented in Subsect. 3.5, we have generated synthetic stable data and then estimated the distributions using our library.

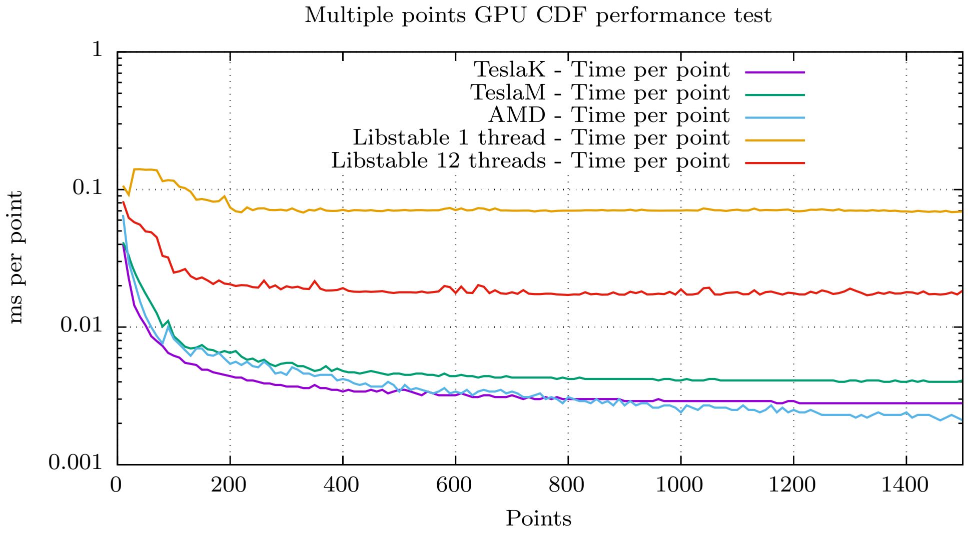


Fig. 9 Performance of the CDF calculation in different GPU cards in comparison with the results obtained with *libstable* on an Intel Core Xeon CPU, depending on the number of points evaluated

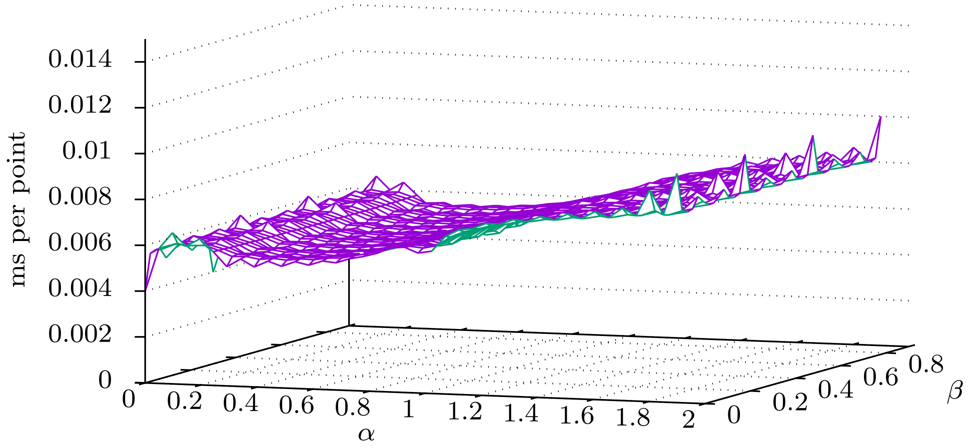


Fig. 10 Variation of the performance of the CDF depending on the parameters. The software was tested with 500 points in a NVIDIA Tesla M2090

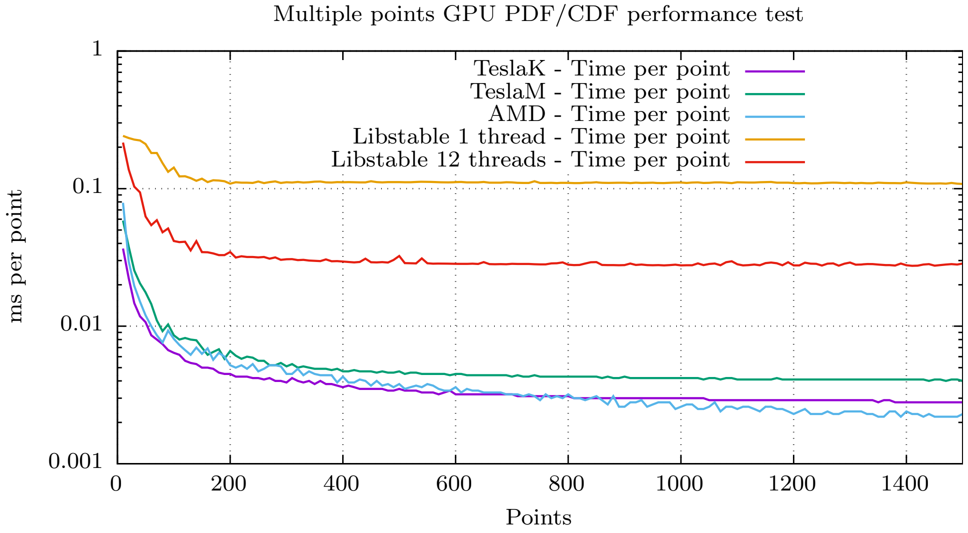


Fig. 11 Performance of the PDF and CDF calculation in different GPU cards in comparison with the results obtained with *libstable* on an Intel Core Xeon CPU, depending on the number of points evaluated

Table 6 Summary of the different PCDF performance measures for 1000 points

	Milliseconds / point	Points per seconds
Tesla K	0.003	333,333.33
Tesla M	0.0042	238,095.24
AMD	0.0026	384,615.38
Libstable 1 thread	0.0829	12,065.78
Libstable 12 threads	0.0158	63,219.12

We generated 20 sets of 1000 stable-distributed values for each one of the 17,400 sample points in the α-β-σ-μ parameter space (that is, $\alpha \in [0.6, 2]$, $\beta \in [0, 1]$, $\mu \in [-1, 1]$ and $\sigma \in [0.5, 3]$). The bias is calculated as the difference between the estimated parameters and the ones the data was generated with. We only show the bias depending on the α and β parameters because we have not found significant changes in bias when changing μ and σ.

Table 7 Precision results when $q \in (0.1, 0.9)$ for the quantile function of a standard ($\mu = 0$, $\sigma = 1$) stable distribution

α	β	Abs. error	Rel. error
0.25	0	6.18×10^{-5}	1.52×10^{-7}
0.25	0.5	6.72×10^{-5}	3.87×10^{-6}
0.25	1	2.79×10^{-5}	6.48×10^{-6}
0.5	0	4.37×10^{-5}	7.89×10^{-7}
0.5	0.5	2.94×10^{-5}	4.19×10^{-6}
0.75	0	1.09×10^{-5}	2.21×10^{-6}
0.75	0.5	2.34×10^{-5}	2.88×10^{-6}
0.75	1	2.71×10^{-5}	6.92×10^{-6}
1.25	0	1.46×10^{-5}	4.74×10^{-6}
1.25	0.5	4.7×10^{-6}	1.92×10^{-6}
1.25	1	1.57×10^{-5}	6.97×10^{-6}
1.5	0	7.5×10^{-6}	2.56×10^{-6}
1.5	0.5	5.42×10^{-6}	1.09×10^{-7}
1.5	1	1.4×10^{-5}	2.58×10^{-6}

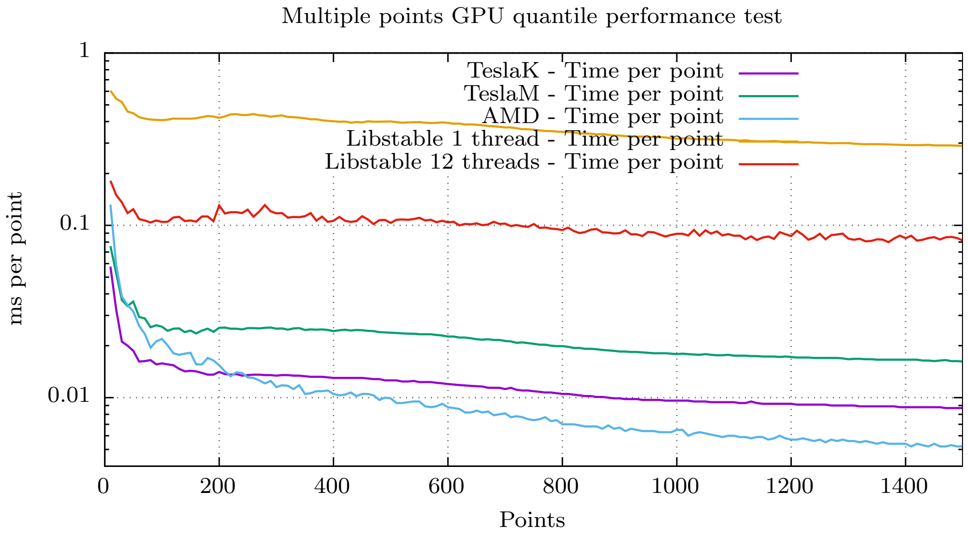


Fig. 12 Performance of the quantile function calculation in different GPU cards in comparison with the results obtained with *libstable* on an Intel Core Xeon CPU, depending on the number of points evaluated

Table 8 Summary of the different quantile performance measures for 1000 points

	Milliseconds / point	Points per seconds
Tesla K	0.0096	104,166.67
Tesla M	0.0179	55,865.92
AMD	0.0065	153,846.15
Libstable 1 thread	0.2631	3800.432
Libstable 12 threads	0.0557	17,965.26

The bias is low in most cases, as Fig. 14 shows. Also, most of the estimation errors happen in the extremes of the parameter space (see Fig. 13) and with the β parameter, which comes not as a surprise given the fact that, when α tends to 2, the distribution resembles a Gaussian one and the symmetry parameter β does not affect its shape.

Regarding performance, our ML estimation algorithm improves by orders of magnitude the time per estimation required by a previous ML estimator developed in *libstable* (Royuela-del-Val 2016) (11291.476 milliseconds on average compared with 82.73002 milliseconds on the Tesla M2090). Table 9 shows the detailed time results. We have included the results from the ML estimator from *libstable* with the PDF being evaluated in the GPU (named *Libstable - TeslaM*). The GPU evaluation gives a significant speedup but the grid algorithm shows that there was still room for improvement.

We have also included the results from a MATLAB maximum likelihood estimator based on off-line precomputed PDF values[5] (Simmross-Wattenberg et al. 2015). The comparison is not fair (MATLAB code is interpreted and probably slower than C code) but it shows that our algorithm performs better than other approaches to fast maximum likelihood estimation.

4.7 Kernel performance analysis

To understand the performance results in the different GPU cards shown in the previous sections, we must explore how their specifications (see Table 1) affect our code. The usual measure for performance in GPU cards is the memory bandwidth usage, as it is the common bottleneck in parallel applications. We have studied the PDF evaluation as a sample, although the results are similar in other cases.

Table 10 shows the bandwidth usage depending on the GPU. The kernel time is the time measured by the OpenCL driver profiler, so it only takes into account the kernel execution time in the GPU and not operations in the host computer, including the setup time.

The results found in the table reflect the expected features of our kernel code. It is not memory intensive, especially in the global memory space. Each thread retrieves only four values from global memory: the point to integrate (which is the same for all the threads of a workgroups), the abscissa of the Gauss–Kronrod node and the two corresponding weights (these last three accesses are to constant memory, which has a higher bandwidth than regular global memory). Finally, there are only two writes to global memory per workgroup (the two Gauss–Kronrod integration results): it is not a surprise that most of the kernel execution time is spent on tasks other than global memory accesses.

Local memory is used more extensively than global memory in our code, but still it is not the bottleneck. The increased usage corresponds to use of local matrices to hold the partial integration results and the reduction algorithm exposed in Subsect. 3.1.3: although it is faster than a simple *for* loop, it

[5] Code is available online at http://es.mathworks.com/matlabcentral/fileexchange/44576-fast-calculation-of-stable-density-functions-based-on-off-line-precomputations.

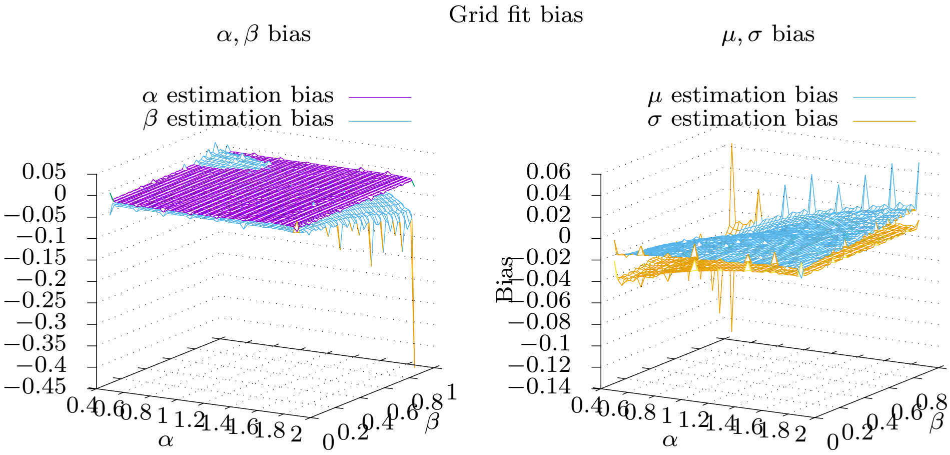


Fig. 13 Bias in the estimation of synthetic stable data. The dataset consists of 1000 points and 20 experiments for each possible value of $\alpha, \beta, \mu, \sigma$ for $\alpha \in [0.6, 2]$, $\beta \in [0, 1]$, $\mu \in [-1, 1]$ and $\sigma \in [0.5, 3]$

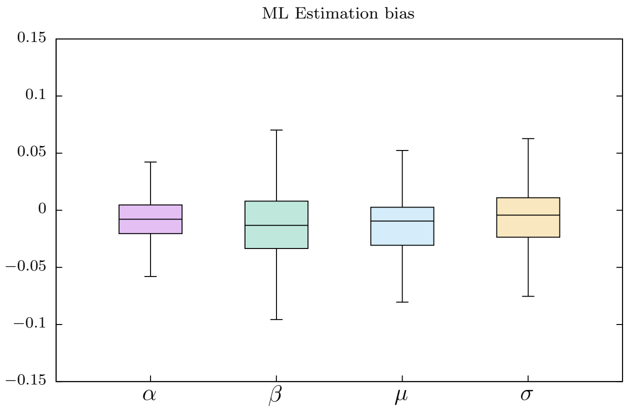


Fig. 14 Box plot of the distribution of the bias in the estimation of synthetic stable data. The dataset consists of 1000 points and 20 experiments for each possible value of $\alpha, \beta, \mu, \sigma$

Table 9 Detail of the performance results of the maximum likelihood estimator: average time for a fit and the 95th percentile of the time distribution

	Avg. time (ms)	95th percentile (ms)
Libstable	11,291.476	18,244.981
Libstable—TeslaM	628.2777	1134.724
Grid—TeslaM	82.73002	168.3666
Grid—TeslaK	20.07744	27.97425
Grid—AMD	120.3731	180.3383
ML offline	257.0785	585.6634

is more memory intensive ($2N$ reads and $\frac{N}{2}$ writes versus N reads and 1 write of the *for* loop).

The results show that our bottleneck is not the memory, as usually happens with GPU applications, but the processor. The NVIDIA Tesla K card has the lowest processor clock of our test setup, 745 MHz. However, this is not the only factor that dominates performance. The number of cores is important: more compute units mean more threads can run in parallel. The PCIe bus speeds also influence the performance when few points are computed and the cost of transferring memory and instructions to and from the GPU are significant relative to the computation.

To fairly compare the performance results, we have to take into account what we explained in the previous sections: as the AMD card does not support workgroups of size 512, we had to reduce their size and double the number of points per thread to maintain precision. However, when using 2 points per thread as in the NVIDIA GPUs, the AMD performs better than the Tesla K in most cases and not only for a large number of points.

This makes the Tesla M the slowest device: with just 512 cores, the advantage of the high processor clock disappears as the card serializes the execution of a large number of threads. Meanwhile, the AMD card performs extremely well thanks to the processor speed and high number of cores, although the fact that it uses PCIe Gen2 penalizes its performance when it computes a low number of points. In those situations, the Tesla K is the best performer despite the low processor speeds, as it has a really high local memory bandwidth, and the largest core count and PCIe bus speeds.

These predictions fit with what we found experimentally modifying the code: we found that some instructions took an unusual amount of time. For example, the exponentiation in (10) takes the 54% of the kernel execution time on the GPU, which translates to an approximate 30% of the time required to evaluate a set of points. This could be caused by our code hitting the worst-case of the NVIDIA processor's exponentiation function.

However, the different OpenCL versions and different compilers used (each vendor distributes its own OpenCL compiler) can also affect the results. The quantile function is only slightly more computationally intensive than the other computations: it involves the PDF and CDF calculation and the root finding algorithm, but the latter is not complex as it only involves one check and a small calculation as spec-

Table 10 Bandwidth usage per GPU. The kernel time refers to the average kernel execution time for a PDF calculation of 500 points

	Tesla M	Tesla K	AMD
Kernel time (ms)	2.11	1.03	1.44
Kernel global memory b.w. (GBps)	13.59	27.98	10.00
Global memory b.w. usage (%)	7.68	9.72	2.84
Kernel local memory b.w. (GBps)	980.32	2018.03	1384.36
Local memory b.w. usage (%)	36.82	11.76	49.16

ified in Subsect. 3.4. In fact, as Fig. 12 shows, the AMD card performs clearly better than the Tesla K after just 250 points evaluating four points per thread instead of two as the NVIDIA card does. In this case, optimizations performed by the AMD compiler could improve the performance, explaining the significant performance differences in this case with respect to the PDF and CDF results.

5 Conclusions

Throughout this paper, we have shown that significant performance improvements (up to 10.35 times better in the PDF and CDF, 27.41 times in the quantile function and 562.4 times in the maximum likelihood estimations compared with single-threaded solutions) can be achieved by the use of GPUs, taking advantage of their parallel capabilities to implement algorithms such as the Gauss–Kronrod quadrature (Subsect. 3.1) and maximum likelihood estimation (Subsects. 3.5) with enough precision.

Our solution shows that the use of GPUs, even consumer-level ones, can be useful to accelerate α-stable computations to a level where new applications on real-time environments can be developed. We have also analyzed the performance of our code (Subsect. 4.7), finding that the GPUs where our code should perform better are those with high processor speeds and a large number of cores, as our application bottleneck is not on the memory but on the processor.

There are further work areas regarding α-stable computations, especially regarding parameter estimation. In our paper, we have used initial estimators (McCulloch 1986) that do not work in the full parameter space. Our estimation algorithm could be improved by finding estimators that work where McCulloch's does not work (that is, when $\alpha < 0.6$) to allow consistent and precise fit of every kind of stable data.

Another area of work would be the parallelization of the estimation at a higher level: using the parallel capabilities of the GPU to estimate multiple sets of data at the same time. However, our ML estimators would not be suitable for this task: they already consume a considerable number of GPU cores, so there would be no room for an additional parallel level with current hardware. A possible solution to this problem could be the replacement of the maximum likelihood estimator by another estimator that does not require a high number of GPU cores. Simpler estimators, such as those proposed by Koutrouvelis (1981) or McCulloch (1986) could be used to estimate simultaneously multiple sets of data in the GPU.

Finally, our approach could also be extended to a multi-GPU environment: given the independence of the computations for each point, the evaluations for the PDF, CDF, quantile function, random number generation and parameter estimations can be easily distributed across GPUs to improve performance.

Acknowledgements This work was partially supported by the Spanish Ministry of Economy and Competitiveness under the projects PackTrack (TEC2012-33754), TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R) and kt-WiSE-MR (TEC2014-57428R), and by the Universidad Autónoma de Madrid under the Project "Implementación de Modelos Computacionales Masivamente Paralelos" (CEMU-2013-14). The authors also thank the Spanish Junta de Castilla y León for grant VA136U13 and the Universidad de Valladolid–Banco de Santander Grant Program 2012.

References

Achim, A., Bezerianos, A., Tsakalides, P.: Wavelet-based ultrasound image denoising using an alpha-stable prior probability model. In: Proceedings 2001 International Conference on Image Processing, vol. 2, pp. 221–224. IEEE (2001)

Alpha Data. ADM-PCIE-7V3 datasheet, Nov 2013. URL http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf

AMD. AMD graphics cores next (GCN) architecture. http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf (2012)

AMD. AMD Radeon R9 series graphics cards. http://www.amd.com/en-us/products/graphics/desktop/r9#, 2013

Arumugam, K., Godunov, A., Ranjan, D., Terzic, B., Zubair, M.: An efficient deterministic parallel algorithm for adaptive multidimensional numerical integration on GPUs. In: International Conference on Parallel Processing—The 42nd Annual Conference (ICPP 2013), pp. 486–491. IEEE, October 2013

Bardou, F.: Lévy Statistics and Laser Cooling: how Rare Events Bring Atoms to Rest. Cambridge University Press, Cambridge (2002)

Belovas, I., et al.: Mixed-stable modeling of high-frequencyfinancial data: parallel computing approach. Technical report. Public center of BSU, Minsk (2013)

DuMouchel, W.H.: On the asymptotic normality of the maximum-likelihood estimate when sampling from a stable distribution. Ann. Stat. **1**(5), 948–957 (1973)

Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing (ICPP), pp. 216–225. IEEE (2011)

Giles, M.B.: Approximation of the inverse poisson cumulative distribution function. To appear in ACM Transactions of Mathematical Software (2015)

Gnedenko, B.V., Kolmogorov, A.N.: Limit Distributions for Sums of Independent Random Variables. Series in Statistics, revised edn. Addison-Wesley, Cambridge, MA (1968)

Gough, B.: GNU Scientific Library Reference Manual, 3rd edn. Network Theory Ltd., Bristol (2009)

Hesterman, J.Y., Caucci, L., Kupinski, M.A., Barrett, H.H., Furenlid, L.R.: Maximum-likelihood estimation with a contracting-grid search algorithm. IEEE Trans. Nucl. Sci. **57**(3), 1077–1084 (2010)

Intel. The Intel Xeon Phi product family. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html (2013)

Koblents, E., Mguez, J., Rodrguez, M.A., Schmidt, A.M.: A nonlinear population monte carlo scheme for the bayesian estimation of parameters of α-stable distributions. Comput. Stat. & Data Anal. **95**, 57–74 (2016)

Koutrouvelis, I.A.: An iterative procedure for the estimation of the parameters of stable laws: an iterative procedure for the estimation. Commun. Stat.-Simul. Comput. **10**(1), 17–28 (1981)

Kronrod, A.S.: Nodes and weights of quadrature formulas: sixteen-place tables. Consultants Bureau (1965)

Li, R., Zhao, Z., Qi, C., Zhou, X., Zhou, Y., Zhang, H.: Understanding the traffic nature of mobileinstantaneous messaging in cellular networks: a revisiting to α-stable models. Access, IEEE **3**, 1416–1422 (2015)

Liang, Y., Chen, W.: A survey on computing lévy stable distributions and a new matlab toolbox. Signal Process. **93**(1), 242–251 (2013)

Lombardi, M.J.: Bayesian inference for α-stable distributions: a random walk MCMC approach. Comput. Stat. & Data Anal. **51**(5), 2688–2700 (2007)

McCulloch, J.H.: Simple consistent estimators of stable distribution parameters. Commun. Stat.-Simul. Comput. **15**(4), 1109–1136 (1986)

Menn, C., Rachev, S.T.: Calibrated fft-based density approximations for α-stable distributions. Comput. Stat. & Data Anal. **50**(8), 1891–1904 (2006)

Mittnik, S., Rachev, S.T.: Modeling asset returns with alternative stable distributions. Econom. Rev. **12**(3), 261–330 (1993)

Nolan, J.P.: Numerical calculation of stable densities and distribution functions. Commun. Stat. Stoch. Models **13**(4), 759–774 (1997)

Nolan, J.P.: Stable Distributions—models for Heavy Tailed Data. Birkhauser, Boston, MA (2015). In Progress, Chap 1. http://academic2.american.edu/~jpnolan

NVIDIA. NVIDIA OpenCL Best Practices Guide, August 2009a

NVIDIA. Whitepaper: NVIDIA's next generation CUDA compute architecture. Fermi (2009b). http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA. Tesla M2090 dual-slot computing processor module board spec, June 2012

NVIDIA. Tesla K40 GPU active accelerator board spec, Nov 2013

NVIDIA. Whitepaper: NVIDIA's next generation CUDA compute architecture. Kepler GK110/210 (2014). http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Robinson, G.K.: Practical computing for finite moment log-stable distributions to model financial risk. Stat. Comput. **25**(6), 1233–1246 (2014)

Royuela-del-Val, J., Simmross-Wattenberg, F., Alberola-López, C.: Libstable: fast, parallel and high-precision computation of α-stable distributions in C/C++ and MATLAB. J. Stat. Softw. .https://uvadoc.uva.es/bitstream/10324/15155/1/RoyuelaJSSoft.pdf (2016)

Salas-González, D., Górriz, J.M., Ramírez, J., Schloegl, M., Lang, E.W., Ortiz, A.: Parameterization of the distribution of white and grey matter in MRI using the α-stable distribution. Comput. Biol. Med. **43**(5), 559–567 (2013)

Simmross-Wattenberg, F., Asensio-Pérez, J.I., Casaseca-de-la Higuera, P., Martín-Fernández, M., Dimitriadis, I., Alberola-López, C.: Anomaly detection in network traffic based on statistical inference and alpha-stable modeling. IEEE Trans. Dependable Secur. Comput. **8**(4), 494–509 (2011)

Simmross-Wattenberg, F., Martín-Fernández, M., Higuera, P.C.D., Alberola-López, C.: Fast calculation of α-stable density functions based on off-line precomputations. Application to ml parameter estimation. Digit. Signal Process. **38**, 1–12 (2015)

Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. & Eng. **12**(1–3), 66–73 (2010)

Thuerck, D., Widmer, S., Kuijper, A., Goesele, M.: Efficient heuristic adaptive quadrature on GPUs: design and evaluation. In: Parallel Processing and Applied Mathematics, pp. 652–662. Springer, Berlin (2014)

Weron, A., Weron, R.: Computer Simulation of Lévy α-stable Variables and Processes. Springer, New York (1995)

Wuertz, D., Maechler, M.: CRAN Package 'stabledist'. https://cran.r-project.org/web/packages/stabledist/stabledist.pdf (2015)