Efficient heuristic adaptive quadrature on GPUs: Design and Evaluation*

Daniel Thuerck[1], Sven Widmer[2], Arjan Kuijper[1,3] and Michael Goesele[2]

[1]TU Darmstadt, Germany
[2]Graduate School of Computational Engineering, TU Darmstadt, Germany
[3]Fraunhofer IGD, Germany

Abstract. Numerical integration is a common sub-problem in many applications. It can be solved easily in CPU-based applications using adaptive quadrature such as the adaptive Simpson's rule. These algorithms rely, however, on error estimation yielding a significant computational overhead. In addition, they require recursive function evaluations, which are not well suited for parallel computation on graphics processing units (GPUs) due to warp divergence issues. In this paper, we introduce heuristic forward quadrature as an alternative that is not only more efficient than traditional methods, but also better suited for accelerated massively-parallel calculation on GPUs. Additionally, we will give an error estimate for our method and demonstrate performance results for 1D and 2D integral applications which show that the algorithm leverages quadrature for the efficient implementation on GPUs.

Key words: Numerical integration, GPGPU, Numerical algorithms, Heuristics, Interval estimation

1 Introduction

General purpose programming on graphics processing units (GPGPU) has been popularized with the advent of CUDA [1], OpenCL and related techniques and is currently one of the state-of-the-art approaches both inside and outside the computer science domain. GPGPU is often used to numerically solve ordinary or partial differential equations (ODEs, PDEs), e.g. in flow simulations, image processing [3], or the economic sciences (option pricing via the Black-Scholes equation) [4]. Especially when solving PDEs in financial mathematics, integration is required at some point. If there is no analytical solution available, we need to rely on numerical integration (also known as *quadrature*). Adaptive methods such as the adaptive Simpson's method or the Gauss-Kronrod algorithm are used with a given error tolerance to ensure exact values.

The principle of standard adaptive quadrature algorithms is shown in Algorithm 1. The integration is first performed on the whole interval. Then, the error is estimated by a given heuristic (in Simpson's case, the result is compared with a second integration

* The final publication is available at www.springerlink.com, DOI 10.1007/978-3-642-55224-3_61

Algorithm 1 Principle of adaptive quadrature algorithms.

function ADAPTIVEQUADRATURE(f, a, b, ε)
  $q \overset{\approx}{\leftarrow} \int_a^b f(x)dx$
  $\delta \leftarrow |q - \int_a^b f(x)dx|$ ▷ Using some given error estimator.
  if $\delta > \varepsilon$ **then**
    $q \leftarrow$ ADAPTIVEQUADRATURE($f, a, a + (b - a)/2, \varepsilon$) + ADAPTIVEQUADRATURE($f, a + (b - a)/2, b, \varepsilon$)

using standard Simpson's rule to check the difference against a user-defined error tolerance). If the threshold is exceeded, the interval is subdivided in at least two parts and the method is called recursively on the subintervals.

While the method is able to guarantee a given error threshold, it requires a significant computational overhead: Each time the interval is subdivided, the last result is discarded. Additionally, the error estimates are complex and computationally intensive as they need to compute a better approximation than the current algorithm's level. Given those two characteristics, the algorithm is not well suited for GPUs. First, interval subdivision often yields branching which severely affects the performance of threads in the same warp. Second, recursive kernels are not yet possible in consumer cards (but will become available with Nvidia HyperQ). In CUDA, recursion is limited to device functions while it is not available at all in OpenGL (and OpenGL ES). Although transformation in a non-recursive algorithms is possible, it is quite complex [5]. In this paper, we propose an alternative method that results in a smaller number of function evaluations (and thus in a significant improvement of performance) and is especially well suited for GPUs.

Our contributions are as follows: We propose *heuristic adaptive forward* quadrature as an alternative quadrature method. We motivate and investigate a special heuristic, give an error estimate and its proof, and show that the number of function evaluations is significantly smaller than with today's standard routine. Furthermore, we show a GPU implementation and analyze its performance using an application from the image processing domain.

2 Related Work

Adaptive quadrature is a well-investigated topic. Research has, however, been discontinued in recent years. A good overview can be found in Gander and Gautschi [6]. The precision of the traditional, recursive algorithm depends largely on the error metric. Typically, an integrand is integrated with two different methods, one more precise than the other, and it is tested whether a given error threshold is exceeded [6]. Further investigation on those estimators has been conducted by Shapiro [7] and Berntsen et al. [8,9]. Both methods rely on the traditional method and improve the performance by clever interval subdivision and error estimators that combine global and local precision.

As stated above, these traditional approaches use recursion intensively and are not well suited for GPUs. Quadrature implementations on GPUs rely mostly on non-adaptive integration [10], which is embarrassingly parallel. Another approach for multidimen-

Algorithm 2 Principle of heuristic adaptive forward quadrature.

function HEURISTICQUADRATURE(f, a, b, h_*, h^*)
  $p \leftarrow a$
  $q \leftarrow 0$
  while $p < b$ **do**
    $h \leftarrow$ ESTIMATEINTERVALLENGTH(f, p, h_*, h^*)
    $q \leftarrow q + h\frac{f(p)+f(p+h)}{2}$
    $p \leftarrow p + h$

sional quadrature was proposed by Arumugam et al. [11]. Integration is done in two phases – interval division and integration – and relies on recursion. The recursive subdivision part is, however, implemented on the CPU using a hybrid CPU/GPU architecture. Anson et al. [12] presented a similar method on a reconfigurable FPGA architecture. Existing CPU libraries as QUADPACK [13] implement only the Simpson and Gauss-Kronrod methods, which are badly suited for the GPU. Currently, there is no published method for adaptive integration on GPUs available. In contrast to these libraries, we leverage adaptive quadrature for efficient use on commodity GPUs by introducing a new algorithm and a suitable implementation.

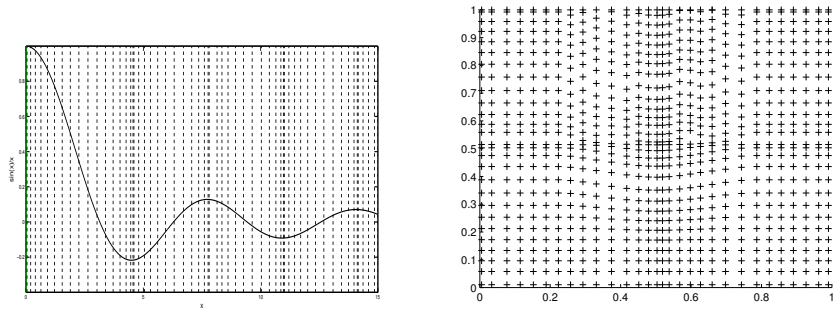
3 Non-recursive adaptive quadrature

As mentioned in the introduction, there are two ways of adaptive quadrature: One can either estimate the quadrature error *a posteriori* and subdivide intervals thereafter or estimate the interval length *a priori*. Our method concentrates on the latter. An overview over this algorithm is given in Algorithm 2. In essence, we apply the trapezoid rule for every interval. In usual quadrature, the result is discarded if it exceeds the error threshold and the interval is subdivided, effectively returning to its begin. This method could be called *forward-backward*, while our method is of the *forward* kind: One after another, the heuristic selects intervals and arbitrary algorithms can be applied to them, never discarding the result. Of course the success depends largely on the interval selection heuristic. Before we propose a particular heuristic, let us shortly review advantages and disadvantages of this approach.

The clear advantage is, as mentioned, that all intervals contribute to the final result and the number of function evaluations is minimized. As the evaluation of an error estimator is unnecessary, we save computational power and do need to evaluate conditionals. On the other hand, the error of our method is clearly dependent on the heuristic. Additionally, there is no possibility to implement a hard error threshold: the heuristic defines the interval lengths and such (indirectly) the error. Using careful design, the error can, however, be bounded and practical results are promising as shown in Section 4.

3.1 The ∂^2 heuristic.

Every heuristic has the goal of providing small interval sizes in regions where the integrand is curved while using larger intervals on near-linear parts. Ideally (in terms of



(a) (b)

Fig. 1: Interval selection for ∂^2-heuristic and integrand $\frac{sin(x)}{x}$ visualized (a) and in 3D for a Gaussian Kernel $e^{-|\mathbf{x}-0.5|_1/0.25^2}$ view from above (b) with $h_* = 0.01$, $h^* = 0.05$.

error), every interval would only contain a linear subset of the integrand's graph. To predict how a function develops in a given interval, we can use its first and second derivatives. For the error estimation (see Section 4) we assume that the integrand is C^2-continuous on $[a, b]$. The second derivative yields information about the curvature of the integrand and thus about the development of the rate of value change. If the second derivative is small or even zero, the curve's steepness will remain almost constant and a greater interval length can be used. Alternatively, if the second derivative grows, the curvature of the integrand increases and we need to consider smaller intervals.

A first approximation of this heuristic (given a minimal interval size h_* and maximal interval size h^*) is

$$h_i = h_* + (\alpha - f''(p_i))h_{i-1}(h^* - h_*). \tag{1}$$

p_i is the last integration point (with interval length h_i). α is a given constant. Remember that we use a forward method: each interval length depends on the length of the last interval to model the integrand's change. Unfortunately, Equation 1 shows bad behavior when the second derivative of the integrand is small, e.g. in the case of the sine function. To improve the estimation here, we replace $\alpha - f''(p_i)$ by $\beta = \max(|f''(p_i)|, |f'(p_i) - f'(p_{i-1})|)$. The second $\max$ term captures the behavior of functions where the curvature is small but nonetheless, the rate of change (hence, the first derivative) is huge such as piecewise linear functions. Note that although the later error proof needs the continuity assumption, in practice the algorithm can be applied to non-continuous functions, too.

A linear involvement of the curvature is, however, not very useful since for only small changes, no interval length change is necessary. Following this intuition, we introduce weighting by $e^{-\beta}$. This yields the final heuristic

$$h_i = h_* + e^{-\max(|f''(p_i)|, |f'(p_i) - f'(p_{i-1})|)}(h^* - h_*). \tag{2}$$

Algorithm 3 ∂^2 heuristic algorithm implemented using simple central difference approximations.

function FAQ(f, a, b, h_*, h^*)
  $q \leftarrow 0$
  $h \leftarrow h_*$
  $p \leftarrow a$
  lastVal $\leftarrow f(p)$
  lastHill $\leftarrow (f(p + 0.1h) - \text{lastVal})/(0.1h)$ ▷ Capture extreme behaviour at graph start.
  while $p < b$ **do**
    $p \leftarrow p + h$
    thisVal $\leftarrow f(p)$
    thisHill $\leftarrow |\text{thisVal} - \text{lastVal}|/h$ ▷ Approximate f'.
    $q \leftarrow q + h \cdot (\text{thisVal} + \text{lastVal})/2$
    $d \leftarrow \max(|(\text{thisHill} - \text{lastHill})/\text{thisHill}|, |\text{thisHill} - \text{lastHill}|)$ ▷ Approximate f''.
    $h \leftarrow h_* + e^{-d}(h^* - h_*)$
    if $p + h > b$ **then**
      $h \leftarrow b - p$
    lastVal $\leftarrow$ thisVal
    lastHill $\leftarrow$ thisHill

function	**relative error**	**absolute error**	**percentage of function evaluations compared to Matlab** `quad`
$\sin(x)$	0.01	-0.01	48%
$\sinh(x)$	0.01	0.02	48%
$x^8 + x^4 + x^3 + x + 1$	0.0	0.18	37%
$\exp(-x)$	0.01	0.01	65%
$\sin(x)/x$	0.0	0.0	64%
$\log(x)$	-0.15	-0.09	21%
$\sqrt(x)$	0.0	-0.01	27%

Table 1: Error and performance results for our forward quadrature method in comparison to adaptive Simpson quadrature.

A visualization on how this heuristics works in an application is given in Figure 1.

Often, derivatives are not directly given and thus not available for calculation. In this case, we need to approximate first and second derivative using simple central differences 3.

With this heuristic, we tested several functions on the unit interval and compared the error and the number of integrand evaluations required to MATLAB's `quad` function, an improved adaptive Simpson's method with $\epsilon = 10e - 6$. As Table 1 shows, this approach is much more efficient for most functions while loosing only very little precision.

4 Error estimation

After motivating and introducing our heuristic in the previous sections, we still need to discuss how the method's error can be estimated. Note that while other quadrature methods offer the possibility to define a hard error threshold, we are unable to do so. Instead, we need to estimate the methods's error a priori. As for non-adaptive methods, a theorem on how the error behaves can be derived. Using the trapezoidal rule error estimation and the fact that our integrand f is C^2-continuous we arrive at the following error bound theorem:

Theorem 1. *Let I be the result of adaptive quadrature with heuristic (2) applied to the function f and bounds a, b with minimum and maximum interval sizes h_*, h^* and let further f be a twice continuously differentiable function on $\mathbb{R}$. Then the error $\Delta I = |I - \int_a^b f(x)dx|$ is bounded by*

$$\Delta I \leq \sup_{\eta \in [a,b]} \frac{\tilde{h_i}^2 (b-a)}{24} f''(\eta)$$

where $\tilde{h_i}$ is defined as given in the proof.

Proof. By the given algorithm, it is obvious that we can express the interval lengths as a sequence $(h_i)_{i \in \mathbb{R}^+}$ with

$$h_i = h_* + e^{-\frac{|f'(a+\sum_{j=0}^{i-1} h_j) - f'(a+\sum_{j=0}^{i-2} h_j)|}{f'(a+\sum_{j=0}^{i-2} h_j)}} (h^* - h_*) \tag{3}$$

where $a + \sum_{j=0}^{i-2} h_j$ is the second-to-previous quadrature anchor point and $a + \sum_{j=0}^{i-1} h_j$ the previous. The quotient

$$\frac{|f'(a+\sum_{j=0}^{i-1} h_j) - f'(a+\sum_{j=0}^{i-2} h_j)|}{f'(a+\sum_{j=0}^{i-2} h_j)} \tag{4}$$

in the algorithm is essentially an approximation of the second derivative f'', which we can use for error estimation. As f'' is continuous, we can use the mean value theorem, such that there is an $\xi \in [a, b]$ that

$$f''(\xi) = \frac{f'(b) - f'(a)}{b - a} \tag{5}$$

or

$$f''(\xi) \frac{b-a}{f'(a)} = \frac{f'(b) - f'(a)}{f'(a)} \tag{6}$$

whose last part is – when applied to each interval h_i (so that $h_i = b - a$) – exactly the approximated quotient mentioned earlier. Hence we can provide an upper bound for the interval length with a being the starting point of interval h_i

$$h_i \leq \tilde{h_i} = \sup_{\xi \in [a - h_{i-1}, a]} \left(h_* + e^{-f''(\xi) \frac{h_{i-1}}{f'(a-h_{i-1})}} (h^* - h_*) \right). \tag{7}$$

Let now $|\mathcal{H}| = \frac{b-a}{\tilde{h}_i}$, then we can estimate the quadrature error by using the estimate for the rectangle rule (as integration in a given interval is done by rectangle rule in the algorithm), being x_i the calculation point in interval h_i:

$$\Delta I \leq \sum_{i=1}^{|\mathcal{H}|} \frac{\tilde{h}_i^3}{24} f''(x_i) \leq \sup_{\eta \in [a,b]} |\mathcal{H}| \frac{\tilde{h}_i^3}{24} f''(\eta). \tag{8}$$

4.1 Extension on two-dimensional integrands

So far, the integrand was implicitly a function $f : [a, b] \to \mathbb{R}$. This is, however, not always the case. Often, an integral needs to be evaluated on an area $\Omega \subset \mathbb{R}^2$. As our integrand f is differentiable, it is also continuous. Let now $[a, b]$ be compact, then we can use Fubini's theorem to formulate the approximation for $f : \mathbb{R}^2 \to \mathbb{R}$ on a rectangle $\Omega = I \times J$:

$$\int_\Omega f(x,y)\, dxdy = \int_J \left(\int_I f(x,y)dx \right) dy \tag{9}$$

$$\approx \int_J \left(\sum_{i=1}^{|\mathcal{H}_x|} |h_{x,i}| f(p_i, y) \right) dy \tag{10}$$

$$\approx \sum_{i=1}^{|\mathcal{H}_x|} |h_{x,i}| \left(\sum_{k=1}^{|\mathcal{H}_{y,i}|} |h_{y,k}| f(p_i, p_{i,k}) \right) \tag{11}$$

where $\mathcal{H}_d$ is the set of intervals chosen by the heuristic in direction d. Effectively, we get a grid on Ω for quadrature. An example grid is shown in Figure 1 (b). By repeating this method, we can extend the algorithm to n dimensions.

5 Implementation and performance on GPUs

As the proposed quadrature algorithm is well suited for GPUs, we implemented it using CUDA Version 5.0 [1]. The performance is evaluated using two different applications, one for the one dimensional and the other for the two dimensional case. An extension to n dimensions is straightforward. We compare the performance against a multi-threaded CPU version with our heuristic as well as a GPU and multi-threaded CPU implementation of the quadrature by the standard adaptive Simpson's rule. To the author's best knowledge, there are no implementations for quadrature on CUDA that can be considered as industry standard and baseline. The well-known QUADPACK [**?**] is limited to one-dimensional functions and despite its popularity, there is no massively-parallel implementation available. Hence, programmers usually create their own implementations of popular methods such as the Simpson rule or the Gauss-Kronrod method. Although the last one is considered state-of-the-art, it is computationally more expensive than using the Simson rule which is why we compare our performance to the Simpson rule.

All experiments were performed on a PC running Ubuntu 12.04 with the latest Nvidia drivers (version 304.88). The system is equipped with an Intel Core i7-3930K hexacore CPU 64 GB of RAM, a Nvidia Geforce GTX 620 (primary device) and a Nvidia Geforce GTX 680 card with two gigabyte of RAM as a headless compute device.

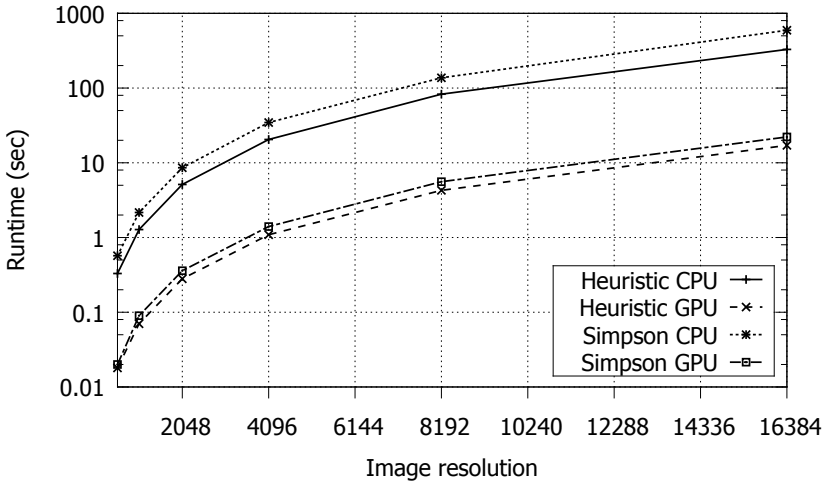


Fig. 2: Performance of the modified Perona and Malik diffusion by Thuerck and Kuijper [14] using the adaptive Simpson's rule.

5.1 One-dimensional case

As an application for the one-dimensional case, we use the Perona and Malik [3] diffusion. This filter models the physical process of diffusion described by the PDE

$$I_t = \nabla \cdot (c(|\nabla I|)\nabla I) \tag{12}$$

where I is the image intensity function of a given grayscale image on a region Ω. The function c is called the *diffusivity*. In contrast to linear diffusion which is equivalent to convolving the image with a Gaussian the diffusion strength varies in nonlinear diffusion over the image domain. For the diffusivities proposed by Perona and Malik, quadrature is not necessary. However, Thuerck and Kuijper [14] presented a diffusivity which leads to a well-posed process but has no analytical integral. To implement this model, we need numerical integration and can apply our algorithm.

The CUDA implementation of the proposed quadrature algorithm is straightforward. For each pixel, finite differences with its neighbors are calculated and used to calculate the diffusivity in this point by quadrature. Essentially, Algorithm 3 can be implemented in CUDA as a device function directly as there is no further inherent parallelism. Figure 2 shows that both GPU implementations outperform the CPU equivalents by more than one order of magnitude. We can observe a performance increase of about 20 percent over the GPU based Simpson algorithm as well.

5.2 Two-dimensional case

Application cases of 2D integration are quite prominent in fluid simulations and financial mathematics. Especially in the field of option pricing using the Black-Scholes

[4] equation, quadrature of a Gaussian kernel is required when the influence of other options is included in the calculation.

To evaluate the key aspects we developed a simplified prototype for performance evaluation. As input data, we take an image of a given size. The CUDA kernel then reads the intensity of each pixel in the whole image and performs a quadrature of an Gaussian bell in the region $[0, 0]$ to $[i, i]$ where i is the intensity to ensure that the threads operate on different data. The difference to a Black-Scholes implementation is thus only a constant for runtime purposes.

The algorithm is split up into three phases. First, one CUDA thread calculates the samples for the first dimension as needed by Fubini's theorem. Afterwards, each thread can use the 1D implementation concurrently to execute integration in the second dimension, which results in a grid as shown in Figure 1 (b). Upon finish, each thread writes its result to shared memory and, after parallel reduction, the final result is written to global memory.

By executing the integration in each of the dimensions in sequence, we reduce the complexity to $\mathcal{O}$(dimensions) rather than $\mathcal{O}$(x-samples). Hence, we observe a much higher speed up than in the 1D case. While in this case, the speed up is only a result of the reduced number of integration steps, the 2D case generates enough workload to satisfy the GPU and a better performance is achieved by doing a large number of integration steps in parallel in addition to each 1D integration being less complex. However, there is room for improvement: A carefully designed 2D heuristic could improve thread utilization and enable us to execute integration in two/dimensions at least partly concurrent. Nevertheless, the performance evaluation shows that the given heuristic algorithm effectively enables us to use the GPU for quadrature, which usual methods cannot do.

In the two dimensional case the adaptive Simpson's rule's performance is similar to both CPU implementations, as seen in Figure 3. Our presented algorithm outperforms all three comparison implementations with one order of magnitude.

Naturally, the speed-up in the 2D case is dramatically higher than in the 1D case. This is due to the quadratic number of integrations in the 2D case compared to the linear number of integrations in the 1D case.

6 Conclusion and future work

In this paper, we showed that heuristic adaptive integration can speed up CPU as well as GPU implementations while keeping the accuracy constant when choosing suitable bounds h_*, h^*. In the GPU case, the performance evaluation resulted in a strong recommendation to use the heuristic on the GPU as sensible speed ups cannot be achieved with traditional Simpson quadrature. The prototypical implementations confirm this fact. Furthermore, the presented algorithm can be extended to n dimensions using Fubini's theorem.

Most of our future plans are already mentioned above. Currently, when integrating in n dimensions, we first create the sampling coordinates and interval sizes for $n - 1$ dimensions and use the nth dimension for the actual integral calculation. The sampling process in each dimension can be parrallelized. However, as the result of every dimensions depend on the previous dimensions, the algorithm scales linear with the number

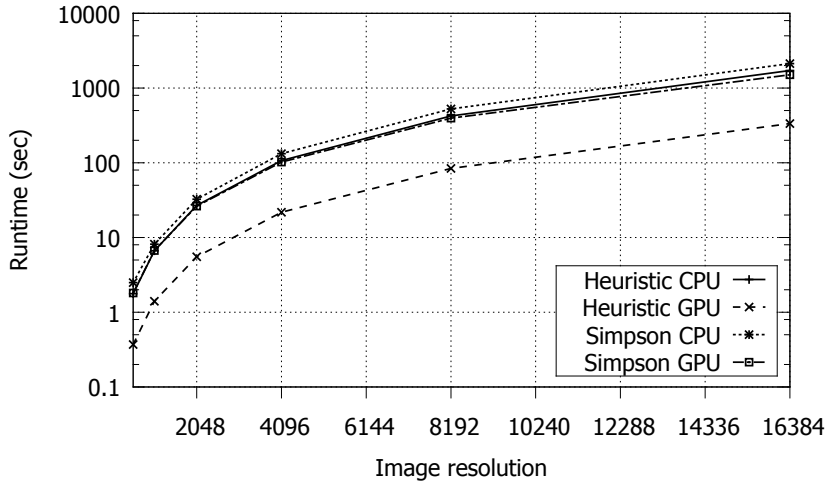


Fig. 3: Performance of the two dimensional case.

of dimensions for fixed grid sizes. Therefore, we wish to improve this behaviour in the future. Second, we would like to fully implement a Black-Scholes option pricing kernel to determine the speed-up in a non-artificial application. Although a reference implementation from NVIDIA exists [15], it is restricted to 1D quadrature. Lastly, we consider developing suited heuristics for the multidimensional case so there is no need to revert to the one-dimensional heuristic via Fubini's theorem.

7 Acknowledgements

The work of Sven Widmer is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Tech- nische Universität Darmstadt.

References

1. NVIDIA: CUDA Compute Unified Device Architecture. `www.nvidia.com/object/cuda_home_new.html`
2. dos Santos, F.P., Lage, P.L., Senocak, I.: Parallel programming on cpu-gpu for solving population balance equation
3. Perona, P., Malik, J.: Scale-space and edge detection using anisotropic diffusion. Pattern Analysis and Machine Intelligence, IEEE Transactions on **12**(7) (1990) 629–639
4. Black, F., Scholes, M.: Taxes and the pricing of options. The Journal of Finance **31**(2) (1976) 319–332
5. McKeeman, W.M., Tesler, L.: Algorithm 182: nonrecursive adaptive integration. Commun. ACM **6**(6) (June 1963) 315–
6. Gander, W., Gautschi, W.: Adaptive quadrature revisited. BIT Numerical Mathematics **40**(1) (2000) 84–101

7. Shapiro, H.D.: Increasing robustness in global adaptive quadrature through interval selection heuristics. ACM Transactions on Mathematical Software (TOMS) **10**(2) (1984) 117–139
8. Berntsen, J., Espelid, T.O., Sørevik, T.: On the subdivision strategy in adaptive quadrature algorithms. Journal of Computational and Applied Mathematics **35**(1) (1991) 119–132
9. Berntsen, J.: Practical error estimation in adaptive multidimensional quadrature routines. Journal of Computational and Applied Mathematics **25**(3) (1989) 327–340
10. Windisch, A., Alkofer, R., Haase, G., Liebmann, M.: Examining the analytic structure of greens functions: Massive parallel complex integration using gpus. Computer Physics Communications (2012)
11. Arumugam, K., Godunov, A., Ranjan, D., Terzic, B., Zubair, M.: An efficient deterministic parallel algorithm for adaptive multidimensional numerical integration on gpus (2013)
12. Anson, H., Chow, G.C., Jin, Q., Thomas, D.B., Luk, W.: Optimising performance of quadrature methods with reduced precision. In: Reconfigurable Computing: Architectures, Tools and Applications. Springer (2012) 251–263
13. Piessens, R., Doncker-Kapenga, D., Überhuber, C., Kahaner, D., et al.: Quadpack, a subroutine package for automatic integration. status: published (1983) 301p
14. Thuerck, D., Kuijper, A.: Cosine-driven non-linear denoising. In: Image Analysis and Recognition. Springer (2013) 245–254
15. Podlozhnyuk, V.: Black-scholes option pricing. Part of CUDA SDK documentation (2007)