A study on adaptive algorithms for numerical quadrature on heterogeneous GPU and multicore based systems

Giuliano Laccetti[1], Marco Lapegna[1], Valeria Mele[1], Diego Romano[2]

[1]Deptartment of Mathematics and Applications, University of Naples Federico II, Complesso Universitario Monte S. Angelo, Via Cintia, Naples, Italy
email: {giuliano.laccetti, marco.lapegna, valeria.mele}@unina.it

[2]ICAR-CNR, Via P. Castellino 111, Naples, Italy
email: diego.romano@na.icar.cnr.it

Abstract. In this work, a parallel adaptive algorithm for the computation of a multidimensional integral on heterogeneous GPU and multicore based systems is described. Two different strategies have been combined together in the algorithm: a first procedure is responsible for the load balancing among the threads on the multicore CPU and a second one is responsible for an efficient execution on the GPU of the computational kernel. The performance is analyzed and experimental results on a system with a quad-core CPUs and two GPUs have been achieved.

Key words: Hyerarchical parallelism, hybrid algorithms, adaptive algorithms, multidimensional integration

1 Introduction

Modern HPC systems are today characterized by hybrid computing nodes, where traditional multicore CPUs live together with special purpose hardware such as Graphical Processing Units (GPUs) used as floating point accelerator. These components have very different features and require different algorithmic development methodologies, so that, in order to efficiently use such emerging hybrid hardware, the development of algorithms and scientific software implies a suitable combination of several methodologies to deal with the various forms of parallelism corresponding to each device.

The aim of our work is to study a special class of algorithms for numerical quadrature for such hybrid computing nodes. More precisely we deal with the numerical computation of multidimensional integrals:

$$I(f) = \int_U f(\underline{t})\, d\underline{t} = \int_U f(t_1, ..., t_d)\, dt_1 \cdots dt_d, \tag{1}$$

where $U = [a_1, b_1] \times \cdots \times [a_d, b_d]$ is a d-dimensional hyperrectangular region. In the last thirty years, several efficient routines have been developed for the solution of this problem on traditional CPUs. Most of them (see for example [2, 12, 17]) are based on adaptive algorithms, that allow high accuracy with a reasonable computational cost.

2 Parallelization of adaptive algorithms on hybrid nodes

Given a family of hyperrectangular subdomains $s(k)$ $(k = 1, .., K)$ of a partition $\mathcal{S}$ of U, a basic multidimensional quadrature rule $r(k)$ and an absolute error estimate procedure $e(k)$ defined on $s(k)$, an adaptive algorithm for the computation of (1) is an iterative procedure that, at each iteration j, evaluates an approximation $Q^{(j)}$ of $I(f)$ and an estimate $|E^{(j)}|$ of the error $|Q^{(j)} - I(f)|$:

$$Q^{(j)} = \sum_{s(k)\in\mathcal{S}} r(k) \simeq I(f) \qquad |E^{(j)}| = \sum_{s(k)\in\mathcal{S}} e(k) \simeq |Q^{(j)} - I(f)|$$

To achieve this, the algorithm computes a sequence $Q^{(j)}$ of composite quadrature rules approaching $I(f)$ and a sequence $|E^{(j)}|$ of approximations of the error $|Q^{(j)} - I(f)|$ approaching 0, until a stopping criterion is satisfied. For our purposes we remark that the basic quadrature rules $r(k)$ are based on a summation such as:

$$r(k) = \sum_{i=1}^{n} A_i\, f(\underline{t}_i) \tag{2}$$

For dimension up to dimension $d = 15$ there are several methods to compute the basic rules $r(k)$ and the absolute errors $e(k)$ in standard regions $s(k)$ [1, 4].

Since the convergence rate of this procedure depends on the behaviour of the integrand function (presence of peaks, oscillations, etc), in order to reduce as soon as possible the error, at the iteration j, the subdomain $\hat{s} \in \mathcal{S}$ with maximum error estimate $\hat{e}$ is split in two parts $s(\lambda)$ and $s(\mu)$ that take the place of $\hat{s}$ in the partition $\mathcal{S}$, that is $\mathcal{S} = \mathcal{S} - \{\hat{s}\} \cup \{s(\lambda)\ ,\ s(\mu)\}$. In a similar way the approximations $Q^{(j)}$ and $E^{(j)}$ are updated, evaluating the (2) in the new subdomains.

```
Algorithm 1:
Initialize $\mathcal{H}$, $Q^{(0)}$ and $E^{(0)}$
**while** (stopping criterion not satisfied) **do** iteration j
        1) select $\hat{s} \in \mathcal{H}$ such that $\hat{e} = \max_{k=1,..,K} e(k)$
        2) divide $\hat{s}$ in two parts $s(\lambda)$ and $s(\mu)$
        3) compute $r(\lambda)$ , $e(\lambda)$ , $r(\mu)$ and $e(\mu)$
        4) sort the subdomains according to their errors
        5) update $\mathcal{H}$, $Q^{(j)}$ and $E^{(j)}$
**endwhile**
```

To implement an adaptive algorithm for numerical quadrature, it is necessary to store all the subdomains $s(k)$ of the partition $\mathcal{S}$ in a suitable data structure, where the subdomain with maximum error estimate $\hat{e}$ can be found with a small computational cost. This can be achieved by storing the data related to the subdomains $s(k)$ in a partially ordered binary tree $\mathcal{H}$ called *heap*, where the subdomain with the largest error estimate is in the root. The computational cost to sort a heap is $\log_2 K$, where K is the number of subdomains in $\mathcal{H}$.

A framework for a sequential global adaptive algorithm for the computation of multidimensional integrals is therefore the Algorithm 1 [11]:

There are several approaches to introduce parallelism in adaptive algorithms [11]. The main strategies are the following:

- Integrand Level Parallelism: the degree of parallelism is given by the number of integrand functions that have to be be eventually computed at the same time. Because the integrals are distinct, this is an embarassingly form of parallelism, and is well suited to computer systems that do not require frequent communications and / or synchronizations between tasks, such as geographically distributed systems;
- Subdivision Level Parallelism: the degree of parallelism is given by the number of subdomains that are subdivided at the same time, so that it is possible to process several subdomains at each iteration. This is a high form of parallelism suited for a SPMD programming model such as that one used for clusters or MPP systems;
- Subregion Level Parallelism: in this case only one subdomain is divided in several parts concurrently processed , and the degree of parallelism is given by the number of these parts. This is a more tight form of parallelism with respect to the previous level.
- Integration Formula Level Parallelism: the degree of parallelismi is given by the number of integrand functions required by the integration rule (2). This is a low level form of parallelism that does not require MIMD based computing systems, because the function evaluations all have the same expression. So it is well suited to SIMD or GPU accelerated systems.
- Integrand Function Level: the degree of parallelism is given by the simultaneous calculation of different tasks of the integrand function, so it depends strongly by its analytical form.

For our aims, consider then an environment represented by a computing node (e.g. a cluster node or a blade in a server) with a node main mamory, one o more host multicore CPUs and one or more floating point accelerator devices such as the NVIDIA's GPUs or the Intel Xeon Phy. Furthermore the acceleration device has a private memory and cannot access directly the node main memory, so that the data have to be moved from the host memory to the device memory and viceversa. From the above, the best strategy to develop a hybrid algorithm for this environment is then to use a combination of the Subdomain Level Parallelism for the subdomains management on the host multicore CPU, and an Integration Formula Level Parallelism to evaluate the basic rule (2) on the GPU device.

2.1 The host algorithm

To introduce a Subdomain Level Parallelism in Algorithm 1, consider a multicore based computing environment, where N threads T_i $(i = 0,..,N-1)$ are in execution, one on each core, sharing the node main memory.

In a such environment it is then possible to process N subregions concurrently by different threads. This can be achieved by storing the data related to the subdomains $s(k)$ in a shared heap $\mathcal{H}$ stored in the node main memory. But, in this centralized approach, where all threads access a single shared heap with a global synchronization, all the basic operations on the heap must be carried out in a critical section, so that the synchronization cost depends on the number of threads N, with a strong scalability degradation [7].

In order to avoid global critical sections, we give up the idea of a single centralized heap, and we split the heap $\mathcal{H}$ in N separate heaps $\mathcal{H}_i$, one for each thread, each of them accessing its private data structure without synchronizations with other threads. However also this approach has a side effect: because of the N items $\hat{s}_i$ with the largest error, resident in the heap roots of $\mathcal{H}_i$ are not those that globally have the highest priority, some threads can process unimportant items with a slow numerical convergence. At this regard note that the sequence of items with large error is unpredictable, so that it is impossible to distribute the subdomains $\hat{s}_i$ with large errors uniformly among the heaps $\mathcal{H}_i$ before the computation.

In other words we have to deal with the contraposition between a parallel algorithm with a centralized data structure requiring several global synchronizations, with fast numerical convergence and low efficiency, and a parallel algorithm with independent data structures that do not require synchronizations, with a slower numerical convergence and a higher efficiency.

In order to combine fast convergence with high efficiency, in our approach, at each iteration j, the threads compare the maximum error $\hat{e}_i$ in the roots of $\mathcal{H}_i$ and, if the critical items are not equally distributed among the heaps, they attempt to reorganize the subdomains in a more suitable way.

To this aim we propose a loosely coordinated approach, where the N threads are logically organized according to a 2-dimensional periodical mesh $\mathcal{M}_2$. This structure is a grid of $\Lambda_0 \times \Lambda_1 = N$ threads, arranged along the points of a 2-dimensional space with integer non negative coordinates in which a shared memory between each couple of connected nodes is established. The shared memories are used as buffer to exchange data between two threads according to a producer-consumer protocol. In addition, the corresponding threads on the opposite faces of the mesh are connected too, so that the mesh is periodical.

In a 2-dimensional periodical mesh, each thread T_i has 4 neighbors: 2 for each direction. In the horizontal direction $(dir = 0)$, we define $T_{i-}^{(0)}$ and $T_{i+}^{(0)}$ respectively the leftmost and the rightmost thread of T_i in $\mathcal{M}_2$. Analogously in the vertical direction $(dir = 1)$ we define $T_{i-}^{(1)}$ and $T_{i+}^{(1)}$ the lowermost and the uppermost threads of T_i.

We then define $\mathcal{H}^*$ a *loosely coordinated heap* as a collection of heap $\mathcal{H}_i \;\; i = 0,..,N-1$, where the roots are connected among them according to the $\mathcal{M}_2$ topology.

With the described threads organization, at the iteration j, each thread T_i attempts to share its item $\hat{s}_i \in \mathcal{H}_i$, with largest error $\hat{e}_i$, only with the neighbor thread $T_{i+}^{(dir)}$ alternatively in the two directions horizontal and vertical. More precisely, in a fixed direction dir, let $\hat{e}_i$ $\hat{e}_{i+}$ and $\hat{e}_{i-}$ be respectively the errors of the subdomains in the heap root of $\mathcal{H}_i$, $\mathcal{H}_{i+}^{(dir)}$ and $\mathcal{H}_{i-}^{(dir)}$. If $\hat{e}_i > \hat{e}_{i+}$ then the item $\hat{s}_i \in \mathcal{H}_i$ with largest error $\hat{e}_i$ is moved forward to the heap $\mathcal{H}_{i+}$ along the direction dir, using a producer-consumer protocol on the shared space. In the same way if $\hat{e}_{i-} > \hat{e}_i$ the item $\hat{s}_{i-} \in \mathcal{H}_{i-}$ with largest error $\hat{e}_{i-}$ is moved to the heap $\mathcal{H}_i$. In this way, the critical items with large error are shared among the heaps with a faster cenvergence.

Furthermore, it should be noted that in this proposed data redistribution, at each iteration j, there are not global synchronizations among threads T_i and each of them exchanges data only with the two threads $T_{i+}^{(dir)}$ and $T_{i-}^{(dir)}$ with $dir = mod(j, 2)$, so that the cost of threads synchronization is constant and it does not depend on the number of threads N, so that the resulting algorithm can be considered scalable [7].

2.2 The device algorithm

Modern GPUs are designed to efficiently deal with problems in the field of computer graphics. In this field, it is typically necessary to perform the exact same operations on all pixels in the image where you want to recreate the same effect. For this reason, modern GPUs provide a SIMD type parallelism where hundreds of single computing elements work synchronously on different data, under the control of a single Control Unit. On the other hand, each computing element is designed as simple as possible in order to keep its production cost low, so that the power of the single elements is much lower in comparison to those of the traditional CPUs. These characteristics mean that only some algorithms are suitable for an efficient implementation on these devices. More precisely only a fine grained parallelism on many data is able to unleash the computing power of these devices.

From this point of view, the computation of (2) is well suited for an execution on a GPU because of the large value of the number of nodes n, so that the n products $A_i f(\underline{t}_i)$ are evaluated concurrently by the GPU computing elements according to the Integration Formula Level Parallelism.

It should be noted, however, that the use of these environments involves a heavy overhead. For example in CUDA (the computing platform and programming model created by NVIDIA for its GPUs), the computing elements cannot directly access the data stored in the node memory, so that it is necessary to allocate space on the memory graphics card and to transfer data in it. This transfer is a tremendous bottleneck for the computation: just think that the NVIDIA Tesla C1060 has a peak performance $p^* = 933$ Gflops (single precision) and a memory bandwidth of only $m^* = 102$ GByte / sec (i.e. 25.5 Gwords/sec, about 3% of the peak performance) . For such a reason, a key parameter for the development of efficient algorithms for such computing device is the ratio $\Theta = p^*/m^*$, which gives a measure of the number of floating point operations required for each data transferred, in order to support the peak performance. For the NVIDIA Tesla C1060 we have $\Theta \simeq 35$.

To this aim we observe that the integrand formula (2) requires the transfer from the host memory to the device memory of 2 d-dimensional array (the center of the region and the length of its edges) and it is based on n independent function evaluations where $d^3 < n < d^4$ (see for example [8]), large enough to support the parameter Θ.

In any case it is important to observe that in a sum-based formula (2), after the parallel evaluation of the$n$ products $A_i\, f(\underline{t}_i)$, it is necessary to collect these values together, by summing pairs of partial sums in parallel. Each step of this pair-wise summation cuts the number of partial sums in half and ultimately produces the final sum after $\log_2 n$ steps. This procedure that computes a single value from a set of data by using an associative operation (e.g. sum or maximum) is called *reduction*, and its optimization is a key problem in the development of algorithms for the GPUs, due to a decreasing number of active threads in the cascade scheme required to calculate a single value from data produced by several processing units. For such a reason we use the optimization strategies described in [10] to compute (2).

```
Algorithm 2:
initialize $\mathcal{H}_i$, $Q_i^{(0)}$ and $E_i^{(0)}$
**while** (local stopping criterion not satisfied) **do** iteration $j$
  define $dir = mod(j, 2)$
  **if** $\hat{e}_i > \hat{e}_{i+}$ **then**
    **remove** $(\hat{s}_i)$ from $\mathcal{H}_i$
    **produce** $(\hat{s}_i)$ for $T_{i+}^{(dir)}$
  **endif**
  **if** $\hat{e}_{i-} > \hat{e}_i$
    **consume** $(\hat{s}_{i-})$ produced by $T_{i-}^{(dir)}$
    **insert** $(\hat{s}_{i-})$ in $\mathcal{H}_i$
  **endif**
      1) select $\hat{s}_i \in \mathcal{H}_i$ such that $\hat{e}_i = \max_{k=1,..,K} e(k)$
      2) divide $\hat{s}_i$ in two parts $s_i(\lambda)$ and $s_i(\mu)$
      3) compute $r_i(\lambda)$ , $e_i(\lambda)$ , $r_i(\mu)$ and $e_i(\mu)$ on the GPU device
      4) sort the subdomains according to their errors
      5) update $Q_i^{(j)}$ and $E_i^{(j)}$
**endwhile**
```

We conclude this section reporting, in Algorithm 2, the description of the hybrid algorithm obtained by integrating the two described methods. More precisely, using the programming model SPMD, we describe the subdomains management based on the parallelization at Subdivision Level between the threads T_i, and at the same time we remark the section of the algorithm with the evaluation of the quadrature formula in step 3) executed in SIMD mode on the GPU using a Formula Level Parallelism.

3 Test results

In this section we present the experimental results achieved on a system composed by a quad-core CPU, an Intel Core I7 950 operating at 3.07 Ghz, and two NVIDIA's C1060 GPUs (Tesla). Each NVIDIA C1060 GPU has 240 streaming processor cores operating at 1.3 Ghz with a peak performance of 933 Gflops in single precision arithmetic (78 Gflops in double precision arithmetic). The host main memory is 12 GBytes large and the bandwidht between the host memory and the device memory is 102 GByte/sec.

In this computational environment we implemented our hybrid Algorithm 2 in double precision using C language, with the CUDA library for the implementation of the step 3) on the GPU, and POSIX thread library and semaphores for the redistribution of the subdomains among the threads in the host algorithm. For the experiments we used a standard procedure based on the well known Genz package [9]. This package is composed by six different families of functions, each of them characterized by some issues making the problem **??** hard to integrate numerically (peaks, oscillations, singularities..). Each family is composed by 10 different functions where the parameters α_i and β_i change and average test results are computed (execution time, error,). Here we report the results for the following three families:

$$\begin{array}{ll} f^{(1)}(\underline{x}) = cos(2\pi\beta_1 + \sum_{i=1}^{d} \alpha_i x_i) & \text{Oscillating functions} \\ f^{(2)}(\underline{x}) = (1 + \sum_{i=1}^{d} \alpha_i x_i)^{-d-1} & \text{Corner peak functions} \\ f^{(3)}(\underline{x}) = \exp(-\sum_{i=1}^{d} \alpha_i |x_i - \beta_i|) & C^{(0)} \text{ functions} \end{array} \quad (3)$$

on the domain $U = [0, 1]^d$ with dimension $d = 10$. We selected these functions because their different analytical features. However, for other functions in the Genzs package we achieved similar results. We remark that in our algorithm we use the Genz and Malik quadrature rule with $\phi = 1,245$ function evaluations so that at each iteration $2\phi = 2,490$ function evaluations are computed in the two new subdomains s_λ and s_μ.

A first set of experiments is aimed to study the parallelization at the subdivison level implementing only the host algorithm. In these experiments we measured

- the Scaled Speed-up SS_N and the Scaled Efficiency SE_N [7] with N=1, 2, 3 and 4 threads.
- The minimum (MinErr) and the maximum (MaxErr) relative error $|I(f)Q(f)|/|I(f)$ on the 10 functions of each family

To compute SS_N we set $F = 10 \times 10^6$ function evaluations in each threads, so that the total number of function evaluations is $FVAL = N \times 10 \times 10^6$ when the number of threads increases. The local stopping criterion is based on a maximum allowed number of iterations in each thread $Maxit = F/2\phi = 4016$.

Table 1 refers to the experiments executed only on the CPU and it reports the Scaled Speed-up for the three families of functions by using 1, 2, 3 and 4 threads . We observe a good scalability when the number of threads increases. As already remarked, the evaluation of the multidimensional integration rules are tasks with a favorable ratio of floating point computation on data movement so that the data can be easily stored in the core caches and reused in the next iterations with an extensive use of cached data.

	$N = 1$	$N = 2$	$N = 3$	$N = 4$
Family $f^{(1)}$				
SS_N	1	1.9	2.8	3.4
SE_N	1	0.95	0.93	0.85
Family $f^{(2)}$				
SS_N	1	1.9	2.8	3.6
SE_N	1	0.95	0.93	0.90
Family $f^{(3)}$				
SS_N	1	1.9	2.7	3.4
SE_N	1	0.95	0.90	0.85

Table 1 Scaled Speed-up and Scaled Efficiency for the three families of functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ with 1, 2, 3 and 4 cores. The workload in each processing unit is $F = 10 \times 10^6$ when the number of core increases. The average execution times with 1 core for the three families of functions are: $Time(f^{(1)}) = 0.27$ sec, $Time(f^{(2)}) = 0.22$ sec and $Time(f^{(3)}) = 0.28$ sec.

A second set of experiments is aimed to investigate the performance gain using a GPU device as a floating point accelerator. In this case the quadrature formula (2) has been implemented in the CUDA programming environment for a scheduling on the GPU, as described in previous section. More precisely, for our experiments, we have been used several quadrature formulas belonging to the family of Genz and Malik [8] with number of nodes $n = 1245, 2585, 9385, 37389$ respectively. This is because the utilization of the GPU involves a high overhead due to the data transfer between the host memory and device memory, which is balanced only by an intensive use of its computational capabilities.

	$n = 1245$	$n = 2585$	$n = 9385$	$n = 37384$
Family $f^{(1)}$				
exec. time	0.079	0.071	0.064	0.055
$FVAL$/time	506×10^6	563×10^6	625×10^6	727×10^6
Family $f^{(2)}$				
exec. time	0.059	0.054	0.048	0.041
$FVAL$/time	677×10^6	740×10^6	833×10^6	975×10^6
Family $f^{(3)}$				
exec. time	0.082	0.074	0.066	0.057
$FVAL$/time	487×10^6	540×10^6	606×10^6	701×10^6

Table 2 Execution time and number of function evaluations per second with $N = 4$ threads without the use of GPU, when the number of node n in the basic rule changes. The total number of function evaluations is $F = 4 \times 10 \times 10^6$.

	$n = 1245$	$n = 2585$	$n = 9385$	$n = 37384$
Family $f^{(1)}$				
exec. time	0.080	0.053	0.031	0.018
$FVAL$/time	500×10^6	754×10^6	1290×10^6	2222×10^6
Family $f^{(2)}$				
exec. time	0.065	0.049	0.028	0.015
$FVAL$/time	615×10^6	816×10^6	1428×10^6	2666×10^6
Family $f^{(3)}$				
exec. time	0.085	0.056	0.033	0.020
$FVAL$/time	470×10^6	714×10^6	1212×10^6	2000×10^6

Table 3 Execution time and number of function evaluations per second with $N = 4$ threads with the use of GPU, when the number of node n in the basic rule changes. The total number of function evaluations is $F = 4 \times 10 \times 10^6$.

In Tables 2 and 3 are reported the performance results of the hybrid algorithm by using only the quad-core CPU and by using also the GPU devices as a floating point accelerator respectively. As a performance measure, we used the number of function evaluations per second. Also in this case the local stopping criterion is based on the maximum function evaluations in each thread $F = 10 \times 10^6$, so that the total number of function evaluations is $FVAL = N \times 10 \times 10^6$ ($N = 4$ is the number of threads) for all test.

From these Tables it is evident that a basic rule with a small number of function evaluations ($n = 1245$ and $n = 2585$) is unable to exploit the computing power of the GPU used in these experiments. More precisely, we can observe that the performance gain obtained with the use of the GPU is wasted because of the overhead related to the memory device allocation and the data transfer, without significant benefit for the performance. Only with a large number of nodes in the basic rule ($n = 9385$ and $n = 37384$) we report a significant performance gain. Compared with the value in Tables 3, the performance gain reported in Table 4 is about $3\times$.

4 Conclusions

We presented a hybrid multicore CPU/GPU approach that can exceed $3\times$ the performance of traditional quadrature adaptive algorithms running just on current homogeneous multicore CPUs. In any case we report a significant performance gain only with a large nymber of function evaluations of the basic rule ($n > 10^4$), because the overhead introduced by the memory device management. In any case our approach demonstrates the utility of graphics accelerators for multidimensional quadrature problems in a large number of dimensions. Furthermore we remark that our approach can be combined with other hybrid strategies for multidimensional quadrature, such as that described in [14] or [13], as well as for other on going works [3][5][6][15][16].

References

1. J. Berntsen, Practical error estimation in adaptive multidimensional quadrature routines, *Journal of Computational and Applied Mathematics*, 25 (1989), pp. 327-340.
2. J. Berntsen, T. Espelid and A. Genz , Algorithm 698: DCUHRE - An adaptive multidimensional integration routine for a vector of integrals, *ACM Transaction on mathematical software*, 17 (1991), pp. 452-456.
3. L. Carracciuolo, L. D'Amore, A. Murli, Towards a parallel component for imaging in PETSc programming environment: A case study in 3-D echocardiography, *Parallel Computing*, Vol. 32 (2006), pp. 67-83
4. R. Cools and P. Rabinowitz, Monomial cubature rules since "Stroud": a compilation, *Journal of Computational and Applied Mathematics*, 48 (1993), pp. 309-326.
5. L. D'Amore, D. Casaburi, A. Galletti, L. Marcellino, A. Murli, Integration of emerging computer technologies for an efficient image sequences analysis *Integrated Computer-Aided Engineering*, vol. 18 (2011), pp 365-378
6. L. D'Amore, A. Murli, Image sequence inpainting: Towards numerical software for detection and removal of local missing data via motion estimation *Journal of Computational and Applied Mathematics*, vol. 198 (2007), pp. 396-413
7. J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, *Sourcebook of parallel computing*, Morgan Kaufmann, 2003.
8. A. Genz and A. Malik, An embedded family of fully symmetric numerical integration rules, *SIAM Journal on Numerical Analysis*, 20 (1983), pp. 580-588.
9. A. Genz, Testing multiple integration software, in B. Ford, J.C. Rault and F. Thommaset eds., *Tools, methods and language for scientific and engineering computation*, (North Holland, New York, 1984).
10. M. Harris, Optimizing parallel reduction in CUDA, presentation packaged with CUDA Toolkit, NVIDIA Corporation, 2007
11. A. Krommer and C. Ueberhuber, Numerical integration on advanced computer systems, *Lecture Notes in Computer Science* vol. 848, Springer-Verlag, 1994.
12. G Laccetti, M Lapegna, PAMIHR. A parallel FORTRAN program for multidimensional quadrature on distributed memory architectures, *Lecture Notes in Computer Science*, vol. 1685 (1999), pp. 1144-1148
13. A. D'Alessio, M. Lapegna , A Scalable Parallel Algorithm for the Adaptive Multidimensional Quadrature in Parallel Processing for the Scientific Computing, R. Sincovec et al eds, SIAM 1993, pp. 933 - 936
14. G. Laccetti, M. Lapegna, V. Mele, D. Romano, A.Murli, A Double Adaptive Algorithm for Multidimensional Integration on Multicore Based HPC Systems, *International Journal on Parallel Programming*, vol. 40 (2012) , pp. 397-409
15. L. Maddalena, A. Petrosino, G. Laccetti, A fusion-based approach to digital movie restoration, *Pattern Recognition*, vol. 43 (2009), pp. 1485-1495
16. V. Boccia, L' D'Amore, M.R. Guarracino, G. Laccetti, A grid enabled PSE for medical imaging: experiences on MediGrid, *Proc. IEEE Symposium on comupter based medical systems*, pp. 529-536
17. P. Van Dooren and L. De Ridder, An adaptive algorithm for numerical integration over an n-dimensional cube, *J. Comput. Appl. Math.*, 2 (1976), pp. 207-217.