An Efficient Deterministic Parallel Algorithm for Adaptive Multidimensional Numerical Integration on GPUs

Kamesh Arumugam
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529
Center for Accelerator Science
Old Dominion University
Norfolk, Virginia 23529

Alexander Godunov
Department of Physics
Old Dominion University
Norfolk, Virginia 23529
Center for Accelerator Science
Old Dominion University
Norfolk, Virginia 23529

Desh Ranjan
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529
Center for Accelerator Science
Old Dominion University
Norfolk, Virginia 23529

Balša Terzić
Center for Advanced Studies of Accelerators
Jefferson Lab
Newport News, Virginia 23606
Center for Accelerator Science
Old Dominion University
Norfolk, Virginia 23529

Mohammad Zubair
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529
Center for Accelerator Science
Old Dominion University
Norfolk, Virginia 23529

***Abstract*—Recent development in Graphics Processing Units (GPUs) has enabled a new possibility for highly efficient parallel computing in science and engineering. Their massively parallel architecture makes GPUs very effective for algorithms where processing of large blocks of data can be executed in parallel. Multidimensional integration has important applications in areas like computational physics, plasma physics, computational fluid dynamics, quantum chemistry, molecular dynamics and signal processing. The computationally intensive nature of multidimensional integration requires a high-performance implementation. In this study, we present an efficient deterministic parallel algorithm for adaptive multidimensional numerical integration on GPUs. Various optimization techniques are applied to maximize the utilization of the GPU. GPU-based implementation outperforms the best known sequential methods and achieves a speed-up of up to 100. It also shows good scalability with the increase in dimensionality.**

I. Introduction and Motivation

Many computational models which involve fast and accurate multidimensional numerical integration of functions require highly efficient adaptive algorithms. A number of such algorithms have been developed and presented in standard numerical libraries such as NAG, IMSL, QUADPACK, CUBA and others [1]–[4]. However, only a few deterministic parallel algorithms have been developed for adaptive multidimensional integration [5]–[7]. Even these parallel algorithms are straightforward extensions of their sequential counterparts, utilizing simply a multithreading nature on the multicore CPU platform and resulting in only modest speed-up. Recent advent of massively parallel GPU platforms presents a great opportunity and a formidable challenge on the adaptive multidimensional integration front. An efficient GPU algorithm must optimize many different components: load balancing, global and local communication, memory management, utilization of registers and cores, etc. This presents a major challenge in developing GPU-optimized algorithms for adaptive numerical integration.

We illustrate the non-trivial nature of developing an efficient parallel algorithm by focusing on load balancing issue which is critical for good performance and scalability. At a first glance, the multidimensional integration problem is embarrassingly parallel. One can divide the region on which the integral is to be computed into P equal subregions, where P is the number of processors available on a parallel machine. Each processor can then independently execute the sequential adaptive integration scheme to estimate the integral for the assigned subregions. The total integral can then be obtained by summing the results of individual computations. This approach could result in satisfactory performance in terms of speed-up for functions that are "well-behaved" over the whole integration region. However, for functions that have different behavior in different regions, this naive way has severe performance bottlenecks due to load balancing. The reason for this is that for these functions different subregions have different computational requirements to estimate the integrals with the desired accuracy. For instance, it is easy to envision a scenario in which most threads finish their assigned work quickly, while only a few threads executing the most poorly-behaved subregions shoulder most of the work and take much longer to execute, resulting in poor performance.

In this paper, we propose a two-phase algorithm that avoids this problem. The first phase filters out subregions where the integral can be calculated with the desired accuracy reasonably quickly. The remaining subregions are passed to the second phase that computes the integral in a simple parallel fashion. The proposed algorithm is implemented and tested on NVIDIA Tesla M2090 on a

set of benchmark functions. The results demonstrate that the first phase balances the load and improves the overall performance. We observed an overall speed-up of up to 100 as compared to the fastest sequential implementation.

The remainder of the paper is organized as follows. In Section II, we briefly overview deterministic methods for adaptive integration. The new parallel algorithm and its implementation for GPU architecture is presented in Section III. In Section IV we apply the new parallel algorithm to a battery of functions and discuss its performance. Finally, in Section V, we discuss our findings and outline the future work.

II. Adaptive Integration Methods

Researchers have looked at efficient sequential methods for estimating the integral over an n-D region [5], [6], [8]. The fastest known such open source method is CUHRE [5], [6], which is available as part of CUBA library [4], [9].

The heart of the CUHRE algorithm is the procedure C-RULES($[\mathbf{a}, \mathbf{b}], f, n$) which outputs a triplet (I, ε, κ) where I is an estimate of the integral over $[\mathbf{a}, \mathbf{b}]$, ε is an error estimate for I, and κ is the axis along which $[\mathbf{a}, \mathbf{b}]$ should be split if needed. Note that we use $[\mathbf{a}, \mathbf{b}]$ to denote the hyper rectangle $[a_1, b_1] \times [a_2, b_2] \ldots \times [a_n, b_n]$. An important feature of C-RULES is that it evaluates the integrand only for $2^n + p(n)$ points where $p(n)$ is $\Theta(n^3)$ [5]. This is much fewer than 15^n function evaluations required by a straightforward adaptive integration scheme based on $7/15$-point Gauss-Kronrod method.

III. Parallel Adaptive Integration Methods

The sequential adaptive quadrature routine is poorly suited to GPUs because it does not take advantage of the GPU's data parallelism. We propose a parallel algorithm that can utilize the parallel processors of GPU to speed up the computation. The parallel algorithm approximates the integral by adaptively locating the subregions in parallel where the error estimate is greater than some user-specified error tolerance. It then calculates the integral and error estimates on these subregions in parallel. The pseudocode for the algorithm is provided below in the algorithms FIRSTPHASE (Algorithm 1) and SECONDPHASE (Algorithm 2).

A. FirstPhase

In the pseudocode for FIRSTPHASE, L_{max} is a parameter that is based on target GPU architecture. The goal of the algorithm is to create a list of subregions of the whole region $[\mathbf{a}, \mathbf{b}]$, with at least L_{max} elements for which further computation is necessary for estimating the integral to desired accuracy. This list is later passed on to SECONDPHASE. The algorithm maintains an list L of subregions, stored as $[\mathbf{a_j}, \mathbf{b_j}]$. Initially the whole integration region is split into roughly L_{max} equal parts through the procedure INIT-PARTITION. In each iteration of the while loop in FIRSTPHASE, first the CUHRE rules are applied to all subregions in L in parallel to get the integral estimate, error estimate, and the split axis. A list S is created to store the intervals with these values.

Algorithm 1 FIRSTPHASE (n, $\mathbf{a}$, $\mathbf{b}$, f, d, τ_{rel}, τ_{abs}, L_{max})

1: $I^p \leftarrow 0$, $I^g \leftarrow 0$, $\varepsilon^p \leftarrow 0$, $\varepsilon^g \leftarrow \infty$
   ▷ I^p, ε^p keep sum of integral and error estimates for the "good" subregions
   ▷ I^g, ε^g keep sum of integral and error estimates for all subregions
2: $L \leftarrow$ INIT-PARTITION($\mathbf{a}, \mathbf{b}, L_{max}, n$)
3: **while** $(|L| < L_{max})$ and $(|L| \neq 0)$ and $(\varepsilon^g > \max(\tau_{abs}, \tau_{rel}|I^g|)$ **do**
4: $S \leftarrow \emptyset$
5: **for** all j in parallel **do**
6: $(I_j, \varepsilon_j, \kappa_j) \leftarrow$ C-RULES($L[j], f, n$)
7: INSERT($S, (L[j], I_j, \varepsilon_j, \kappa_j)$)
8: **end for**
9: $L \leftarrow$ PARTITION($S, L_{max}, \tau_{rel}, \tau_{abs}$)
10: $(I^p, \varepsilon^p, I^g, \varepsilon^g) \leftarrow$ UPDATE($S, \tau_{rel}, \tau_{abs}, I^p, \varepsilon^p$)
11: **end while**
12: **return** $(L, I^p, \varepsilon^p, I^g, \varepsilon^g)$

Thereafter the algorithm essentially identifies the "good" and the "bad" subregions in S – the good subregions have error estimate that is below a chosen threshold, whereas bad subregions have error estimates exceeding this threshold. The bad subregions need to be further divided, while the integral and error estimates for the good regions can simply be accumulated. This is accomplished through the procedures PARTITION and UPDATE. Pseudocode for these procedures is provided in Listing 1.

It is worth noting that the original CUHRE algorithm always divides selected subregion into two parts along the chosen axis where the integrand has the largest fourth divided difference [5]. The proposed algorithm here uses this strategy of choosing the axis, with the distinction that the selected subregion is divided into d pieces along the chosen axis instead of two. The parameter d is dynamically calculated using a heuristic SPLIT-FACTOR based on the target architecture and on the number of bad intervals. Subdivision of a region refines the resolution of that region along with generating enough subregions to balance the computational load for second phase.

First phase continues until (i) a long enough list of "bad" subregions is created in which case we proceed to the second phase or (ii) there are no more "bad" subregions in which case we can return the integral and error estimates I^g and ε^g as the answer or (iii) I^g, ε^g satisfy the error threshold criteria in which case we also return I^g and ε^g as the answer. Note that, in case (ii) or (iii) second phase of the algorithm is not used.

In our implementation subregions are maintained in GPU global memory. The C-RULE parameters are computed and stored in the shared memory for faster access. The algorithm requires a parameter L_{max} which defines the maximum number of subregions allowed to be processed in parallel. The optimal value for this parameter is estimated at the host based on the target GPU architecture. For our experiments we have used L_{max} to be 32768 for the Fermi architecture [10]. The initial subregions are

Listing 1: **Procedures** in FIRSTPHASE

```
1: function INIT-PARTITION((a, b, L_max, n))
2:     l ← max{j|j^n ≤ L_max}
3:     split [a, b] along each dimension into l equal parts and save these l^n subregions into L
4:     return L
5: end function

6: function UPDATE((S, τ_rel, τ_abs, I^p, ε^p))
7:     t_1 ← I^p, t_2 ← ε^p, t_3 ← 0, t_4 ← 0
   ▷ t_1, t_2 keep the partial sum of integral and error estimates for the "good" subregions
   ▷ t_3, t_4 keep the sum of integral and error estimates for all the subregions
8:     for j = 1 to |S| do
9:         Let ([a_j, b_j], I_j, ε_j, κ_j) be the j^th record in S
10:        if ε_j < max(τ_abs, τ_rel|I_j|) then
11:            t_1 ← t_1 + I_j
12:            t_2 ← t_2 + ε_j
13:        else
14:            t_3 ← t_3 + I_j
15:            t_4 ← t_4 + ε_j
16:        end if
17:    end for
18:    t_3 ← t_3 + t_1
19:    t_4 ← t_4 + t_2
20:    return (t_1, t_2, t_3, t_4)
21: end function

22: function PARTITION((S, L_max, τ_rel, τ_abs))
23:    L_1 ← ∅, L_2 ← ∅
   ▷ L_1 stores the "bad" subregions before subdivision
   ▷ L_2 stores the subregions after subdivision of "bad" subregions
24:    for j = 1 to |S| do
25:        Let ([a_j, b_j], I_j, ε_j, κ_j) be the j^th record in S
26:        if ε_j ≥ max(τ_abs, τ_rel|I_j|) then
27:            insert ([a_j, b_j], κ_j) into L_1
28:        end if
29:    end for
30:    d ← SPLIT-FACTOR(L_max, |L_1|)
31:    for j = 1 to |L_1| do
32:        Let ([a_j, b_j], κ_j) be the j^th record in L_1
33:        split [a_j, b_j] into d equal parts along the axis κ_j and insert all these subregions into L_2
34:    end for
35:    return L_2
36: end function
```

generated by dividing the entire integration region along each dimension into l equal parts. We use one GPU thread to generate a new subregion and thus requiring a total of l^n threads to generate the initial subregion list. Each of the generated subregions are assigned to a GPU thread for the application of C-RULE.

The FIRSTPHASE implements the C-RULE on every GPU thread to estimate the triplet (I, ε, κ) for a subregion assigned to it. This kernel requires at least as many threads as there are subregions in the input list and creating multiple threads hide the latency of global memory by overlapping the execution. The kernel returns a list of triplets computed by each thread along with a identifier which specifies if a subregion has to be further subdivided or not. The intermediate integral estimates are evaluated as the sum of individual estimates for all subregions in the list. We make use of CUDA-based THRUST library [11], [12] to perform such common numerical operations. All the bad subregions are identified and copied to a new list based on the identifier flag. Prefix scan [13] implementation from the CUDA THRUST library is used to identify the position of bad subregions in the subregion list. Identified bad subregions are further partitioned into finer subregions, and the implementation continues with the steps above on these finer subregions. Details of this GPU implementation is described in [14].

B. SECONDPHASE

The algorithm continues with the second phase when the global error estimate is still larger than the required global tolerance. In second phase, on every subregion $[\mathbf{a_j}, \mathbf{b_j}]$ in the list L the algorithm calls sequential CUHRE routine (SEQUENTIALCUHRE) to compute global integral and error estimate for the selected subregion (Line 3). Line 5 and 6 update the global integral and error estimate. Second phase implements a modified version of CUHRE to run in parallel for each of the subregions in the list L returned from first phase. The modified version of CUHRE implemented for GPU take advantage of state-of-the art GPU architectures to speed-up the computations. Our approach combines the original features of CUHRE with the improved algorithm efficiency afforded by massive parallelism on a GPU platform.

Algorithm 2 SECONDPHASE(n, $\mathbf{f}$, τ_{rel}, τ_{abs}, L, I^g, ε^g)

1: **for** $j = 1$ to $|L|$ **parallel do**
2: Let $[\mathbf{a_j}, \mathbf{b_j}]$ be the j^{th} record in L
3: (I_j, ε_j) ←SEQUENTIALCUHRE(n, $\mathbf{a_j}$, $\mathbf{b_j}$, f, τ_{rel}, τ_{abs})
4: **end for**
5: $I^g \leftarrow I^g + \sum_{[\mathbf{a_j},\mathbf{b_j}]\in L} I_j$
6: $\varepsilon^g \leftarrow \varepsilon^g + \sum_{[\mathbf{a_j},\mathbf{b_j}]\in L} \varepsilon_j$
7: **return** $I^g$and ε^g

IV. PERFORMANCE/EXPERIMENTAL RESULTS FOR CUHRE

The CPU-GPU system used in our experiment consists of NVIDIA Tesla M2090 GPU device installed on a host machine with Intel® Xeon® CPU X5650, 2.67GHz. The Tesla M2090 GPU is based on the recent Fermi architecture [10]. A Tesla M2090 offers 6GB of GDDR5 on-board memory and 512 streaming processor cores (1.3 GHz) that delivers a peak performance of 665 Gigaflops in double precision floating point arithmetic. The interconnection

between the host and the device is via a PCI-Express Gen2 interface. We have used CUDA 4.0 programming environment for the parallel code and gcc for the serial one.

We have carried out our evaluation on a set of challenging functions which require many integrand evaluations for attaining the prescribed accuracy. We use the battery of benchmark functions (Table I) which is representative of the type of integration that is often encountered in science: oscillatory, strongly peaked and of varying scales. These kinds of poorly-behaved integrands are computationally costly, which is why they greatly benefit from a parallel implementation.

1. $f_1(\mathbf{x}) = \left[\alpha + \cos^2\left(\sum_{i=1}^{n} x_i^2\right)\right]^{-2}$, where $\alpha = 0.1$

2. $f_2(\mathbf{x}) = \cos\left(\prod_{i=1}^{n} \cos\left(2^{2i} x_i\right)\right)$

3. $f_3(\mathbf{x}) = \sin\left(\prod_{i=1}^{n} i \arcsin(x_i^i)\right)$

4. $f_4(\mathbf{x}) = \sin\left(\prod_{i=1}^{n} \arcsin(x_i)\right)$

5. $f_5(\mathbf{x}) = \frac{1}{2\beta} \sum_{i=1}^{n} \cos(\alpha x_i)$, where $\alpha = 10.0$ and $\beta = -0.054402111088937$

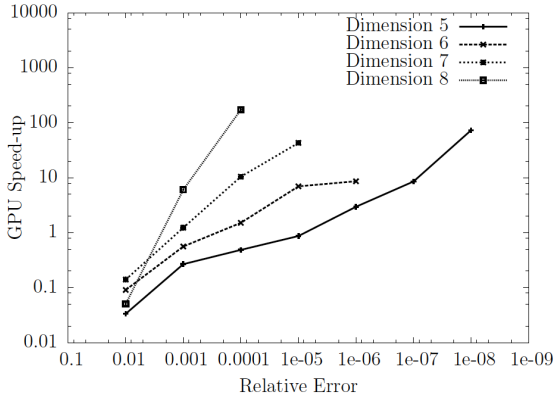
Table I: n-D benchmark functions

In our evaluation, the region of integration for all the benchmark functions is a unit hypercube $[0, 1]^n$. In order to provide a fair comparison, we use the serial C-implementation of CUHRE from the CUBA package [4], [9] executed on the host machine of the GPU.

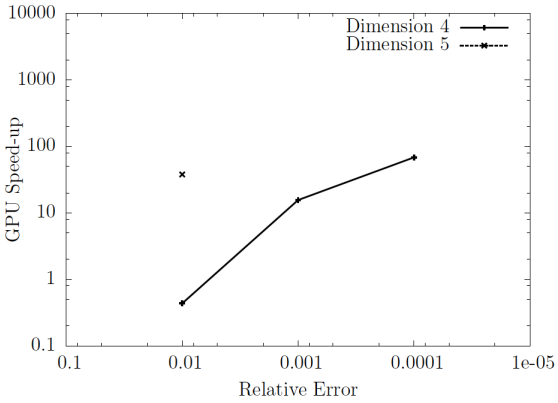
In Figure 1 we plot the test results for all the benchmark functions. For each of these functions we plot the GPU speed-up against the relative error τ_{rel} for different dimension n. The speed-up here is computed by comparing the total execution time for the parallel code on GPU against the time taken by serial code on the host machine. The points shown are only those for which both CPU and GPU were able to compute the answers before reaching the limit for total function evaluation of 10^8. The proposed method for GPU is up to 100 times faster than the serial code.

In Figure 1a to 1e, we observe that the speed-up considerably increases with the dimension. The execution time here greatly depends on the number of function evaluations and complexity of the integrand. At higher dimension, the GPU implementation clearly benefits from the massive parallelism provided by the GPU. Lower-dimensional integration, on the other hand, is not as efficient on the GPU due to fewer number of function evaluations. At lower dimension the execution time is dominated by the GPU initialization and the memory allocation time.

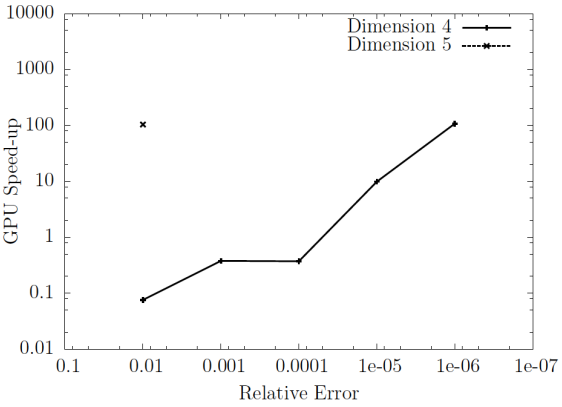
Table II shows a breakdown of performance metrics for each of the two phase in our GPU implementation and compares it with the performance of serial code in



(a) Speed-up for function $f_1(\mathbf{x})$.



(b) Speed-up for function $f_2(\mathbf{x})$.

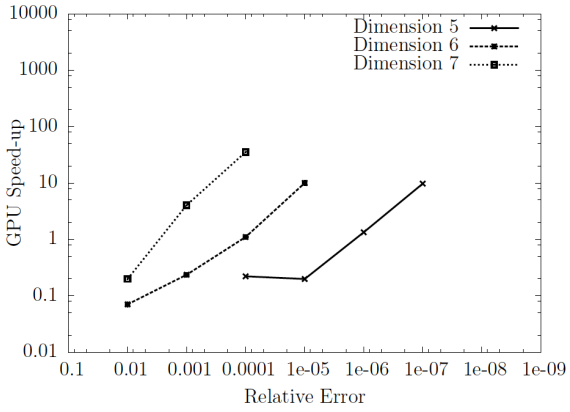


(c) Speed-up for function $f_3(\mathbf{x})$.

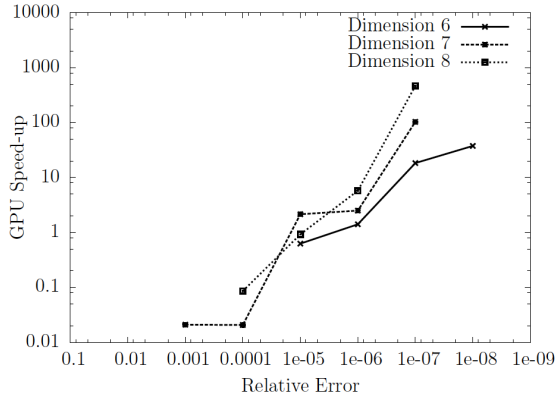
Figure 1: Simulation Results

Table III for a set of functions from the benchmark. The dimensionality and accuracy of computation depicted in Table II and Table III is chosen to be a representative sample of all of the simulations executed. We observe that algorithm spends most of the time in SECONDPHASE after a brief stay in FIRSTPHASE. This suggest us that the algorithm starts to focus on "bad regions" by quickly eliminating the "good" regions. In the 8-D function $f_5(\mathbf{x})$ with $\tau_{rel} = 10^{-5}$, the integral estimate computed by FIRSTPHASE satisfied the global error requirement and the algorithm terminates without executing the SECONDPHASE.

In Figure 2, we show the effectiveness of having two



(d) Speed-up for function $f_4(\mathbf{x})$.



(e) Speed-up for function $f_5(\mathbf{x})$.

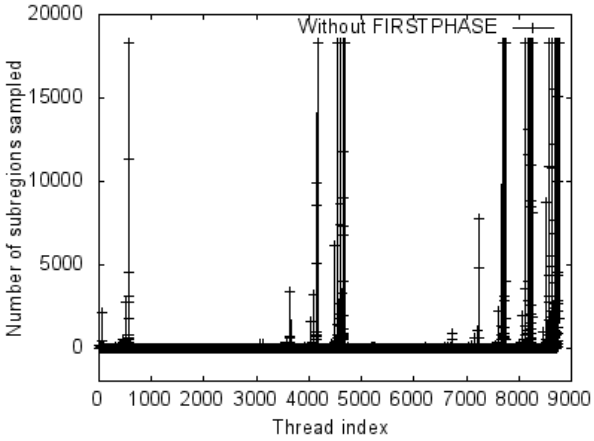
Figure 1: Simulation results.

Function	n	τ_{rel}	FIRSTPHASE		SECONDPHASE time(sec)	
			Number of Iterations	GPU Time (sec)	With FIRSTPHASE	Without FIRSTPHASE
$f_1(\mathbf{x})$	7	10^{-5}	4	2.52	51.59	196.97
$f_2(\mathbf{x})$	5	10^{-2}	2	1.60	55.19	89.95
$f_3(\mathbf{x})$	5	10^{-2}	6	1.91	51.10	86.31
$f_4(\mathbf{x})$	6	10^{-5}	4	1.60	219.56	748.08
$f_5(\mathbf{x})$	8	10^{-7}	1	3.38	-	14.99

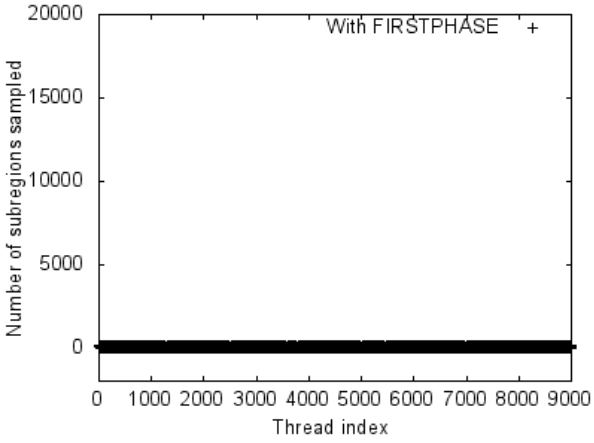
Table II: Breakdown of GPU execution time.

Function	n	τ_{rel}	Execution Time (s)		Function Evaluations	
			CPU	GPU	CPU	GPU
$f_1(\mathbf{x})$	7	10^{-5}	2349.2	54.8	$1.05\text{x}10^9$	$6.92\text{x}10^8$
$f_2(\mathbf{x})$	5	10^{-2}	2082.9	55.0	$4.09\text{x}10^8$	$2.56\text{x}10^8$
$f_3(\mathbf{x})$	5	10^{-2}	5300.3	51.0	$6.48\text{x}10^8$	$1.13\text{x}10^9$
$f_4(\mathbf{x})$	6	10^{-5}	2316.1	231.3	$6.52\text{x}10^8$	$6.57\text{x}10^8$
$f_5(\mathbf{x})$	8	10^{-7}	1275.3	3.4	$1.25\text{x}10^9$	$7.24\text{x}10^7$

Table III: Function evaluations in CPU and GPU.



(a) Without FIRSTPHASE for $f_3(\mathbf{x})$ with $\tau_{rel} = 10^{-2}$ and $n = 5$.

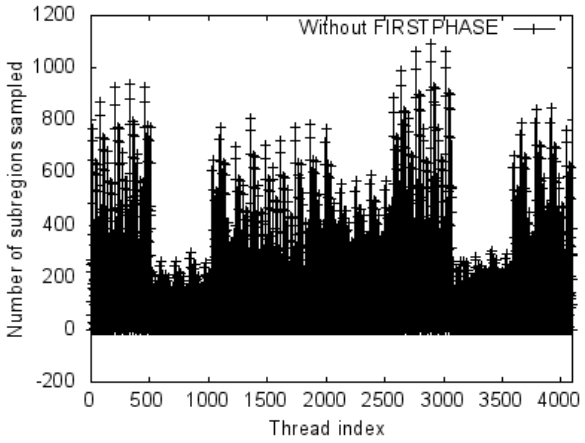


(b) With FIRSTPHASE for $f_3(\mathbf{x})$ with $\tau_{rel} = 10^{-2}$ and $n = 5$.

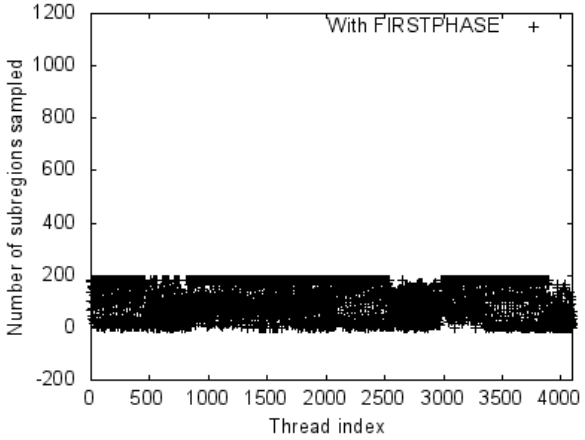
Figure 2: GPU results for execution with FIRSTPHASE and without FIRSTPHASE.

phases in our algorithm by comparing the results of the implementation with FIRSTPHASE against the one without FIRSTPHASE. Figure 2a and Figure 2c show the result of executing two-phase GPU algorithm without FIRSTPHASE and Figure 2b and Figure 2d show the normal execution with FIRSTPHASE. Both these evaluations were performed on a 5-D function $f_3(\mathbf{x})$ chosen from the benchmark with a relative error requirement of 10^{-2} and 10^{-3}.

In each of these figures we plot the number of subregions sampled by a thread in SECONDPHASE against the thread index. Computational load of a thread here is directly related to the number of subregions sampled by that thread. GPUs that are built on SIMD architecture require every thread to share approximately equal load to gain maximum performance. In Figure 2a and Figure 2c, we observe a wide variance of subregions sampled by the threads. Some of these threads have longer execution time than others, which results in an unbalanced computational load. The overall execution time greatly depends on these threads which have longer execution times. This brings out the importance of FIRSTPHASE to share the load across the threads. Figure 2b and Figure 2d show the execution of SECONDPHASE with the FIRSTPHASE behaving as a load balancer. We notice that the number of subregions sampled by the threads are approximately same, reflecting a efficient load balancing. The total execution time in both cases – with or without the FIRSTPHASE – depends on the execution time of the most highly loaded thread, which in the case when FIRSTPHASE serves as a load balancer is considerably shorter (Figure 2a and Figure 2c). Table II provides the execution time for SECONDPHASE under



(c) Without FIRSTPHASE for $f_3(\mathbf{x})$ with $\tau_{rel} = 10^{-3}$ and $n = 5$.



(d) With FIRSTPHASE for $f_3(\mathbf{x})$ with $\tau_{rel} = 10^{-3}$ and $n = 5$.

Figure 2: GPU results for execution with FIRSTPHASE and without FIRSTPHASE.

both these scenarios for the set of functions chosen from the benchmark. We notice that due to the nature of GPUs, we obtain higher performance by having two phases.

V. DISCUSSION AND CONCLUSION

From a survey of earlier studies on adaptive and multidimensional integration, as well as our own experience, it is evident that there is no single optimal algorithm for all numerical integration needs. In our present study, we focus on a set of challenging cases which require many integrand evaluations for attaining the prescribed accuracy. We use a battery of test functions which is representative of the type of integration that is often encountered in science: oscillatory, strongly peaked and of varying scales. These kinds of poorly-behaved integrands are computationally costly, which is why they greatly benefit from a parallel implementation.

The new parallel algorithm for numerical integration we developed here is up to two orders of magnitude more efficient than the leading sequential method. This improvement is demonstrated on a battery of multidimensional functions, which serve as a template on how this new parallel approach can improve simulations involving numerical integration of similar complexity. Computing the n-D integral with the new parallel approach is at least as efficient as computing the $(n–1)$-D integral with a sequential method at the same accuracy. This essentially means that the new GPU-based algorithm "earns" at least one dimension in multidimensional integration.

B. T. would like to acknowledge the support of the U.S. Department of Energy (DOE) Contract No. DE-AC05-06OR23177.

REFERENCES

[1] NAG, "Fortran 90 Library," Numerical Algorithms Group Inc., Oxford, U.K., 2000.

[2] IMSL, "International mathematical and statistical libraries," Rogue Wave Software, 2009.

[3] R. Piessens, E. de Doncker-Kapenga, C. Überhuber, and D. Kahaner, *QUADPACK:A Subroutine Package for Automatic Integration*. Springer-Verlag, Berlin, 1983.

[4] T. Hahn, "CUBA a library for multidimensional numerical integration," *Computer Physics Communications*, vol. 176, pp. 712–713, June 2007.

[5] T. E. J. Bernsten and A. Genz, "An adaptive algorithm for the approximate calculation of multiple integrals," *ACM Transactions on Mathematical Software (TOMS)*, vol. 17, no. 4, pp. 437–451, December 1991.

[6] J. Bernsten, T. Espelid and A. Genz, "DCUHRE: an adaptive multidemensional integration routine for a vector of integrals," *ACM Transactions on Mathematical Software (TOMS)*, vol. 17, no. 4, pp. 452–456, December 1991.

[7] J. Bernsten, "Adaptive-multidimensional quadrature routines on shared memory parallel computers," *Reports in Informatics 29, Dept. of Informatics, Univ. of Bergen*, 1987.

[8] A. Genz and A. Malik, "An adaptive algorithm for numerical integration over an n-dimensional rectangular region," *Journal of Computational and Applied Mathematics*, vol. 6, pp. 295–302, December 1980.

[9] T. Hahn, "CUBA The CUBA library," *Nuclear Instruments and Methods in Physics Research*, vol. 559, pp. 273–277, 2006.

[10] NVIDIA, "NVIDIAs Next Generation CUDA Compute Architecture: Fermi ." [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[11] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," *GPU Computing Gems Jade Edition*, 2011.

[12] N. Bell and J. Hoberock, "Thrust library for GPUs." [Online]. Available: http://thrust.github.com/

[13] H. Nguyen, "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems 3*, 2007.

[14] K. Arumugam, A. Godunov, D. Ranjan, B. Terzić, and M. Zubair, "An Efficient Deterministic Parallel Algorithm for Adaptive Multidimensional Numerical Integration on GPUs." [Online]. Available: http://www.cs.odu.edu/~akamesh/publications/paper/agrtz2012.pdf