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

Balsa Terzic  
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 involve integration of various functions. Examples include lattice QCD simulations which have to evaluate multidimensional integrals, simulation of coherent synchrotron radiation in charged particle beams via multidimensional space-time integration of retarded potentials, solution of the Navier-Stokes equations using spectral element methods and achieve a speed-up of up to 100. It also shows good scalability with the increase in dimensionality.

as NAG, IMSL, QUADPACK, CUBA and others. However, only a few deterministic parallel algorithms have been developed for adaptive multidimensional integration. 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. The opportunity is to “squeeze” another order of magnitude or more in performance out of the new architecture, at the price of meeting the challenge of constructing a virtually new algorithm to exploit optimally the underlying architecture. The sequential adaptive integration algorithms seek to minimize the number of integrand evaluations because this directly corresponds to minimizing the execution time. In contrast, 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 the 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 suited for GPU architecture is presented in Section III and its implementation described in Section IV. In Section V we apply the new parallel algorithm to a battery of functions and discuss its performance. Finally, in Section VI we discuss our findings and outline the future work.

II. ADAPTIVE INTEGRATION METHODS

An adaptive integration method works in the following way. We start with a region on which integral value needs to be estimated. The method selects a set of points in the interval on which it evaluates the integrand values. From these values, the method computes the integral estimate and the error estimate. Based on the error estimate, the method decides whether to partition the region or not. If it decides to partition the region, the above step is repeated on individual partitions. This process follows a recursive tree structure where recursion on a branch stops when the error estimate of associated region meets the threshold criteria. Once recursion stops on all the branches, the method finishes. The estimate of the integral for any region that should be split if needed. An important feature of C-RULES is that it evaluates the integrand only for \( \frac{1}{5} \) of the points required by a straightforward adaptive integration scheme based on 7/15-point Gauss-Kronrod method.

A. Adaptive Multidimensional Integration

One straightforward way of computing a multidimensional integral is to use nested adaptive integration method. This approach integrates each dimension separately using one of standard 1-D adaptive methods. In this case adaptivity is not truly multidimensional, but confined only one dimension at the time, leading to a suboptimal performance.

A more efficient approach is to use methods that are adaptive on the entire n-D space. A straightforward approach takes a 1-D method that uses \( m \) points to estimate the integral in a 1-D subregion and extends it to a n-D integration method requiring \( m^n \) points for each n-D subregion. This is prohibitive for methods like 7/15 Gauss-Kronrod when the number of dimensions exceeds 4. To avoid this, researchers have developed methods that use fewer points than \( m^n \) points to estimate the integral over an n-D region [5], [6]. The fastest known such open source method is CUHRE [5], [6], which is available as part of CUBA library [4], [12]. Even though the CUHRE method uses much fewer points, in practice it compares fairly well with other adaptive integration methods in terms of accuracy [13].

B. Brief Overview of CUHRE

In this section we describe the sequential CUHRE algorithm for multidimensional integration. The integrals have the form

\[
\int_{a_1}^{b_1} \int_{a_2}^{b_2} \ldots \int_{a_n}^{b_n} f(x) dx,
\]

where \( x \) is an \( n \)-vector, and \( f \) is an integrand. We use \([a, b] \) to denote the hyper rectangle \([a_1, b_1] \times [a_2, b_2] \ldots \times [a_n, b_n] \).

The heart of the CUHRE algorithm is the procedure C-RULES([a, b], f, n) which outputs a triple \((I, \varepsilon, \kappa) \) where \( I \) is an estimate of the integral over \([a, b] \) (Equation 1), \( \varepsilon \) is an error estimate for \( I \), and \( \kappa \) is the axis along which \([a, b] \) should be split if needed. An important feature of C-RULES is that it evaluates the integrand only for \( 2^n + p(n) \) points where \( p(n) = \Theta(n^3) \) [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.

We now give a high-level description of the CUHRE algorithm (Algorithm 1). The algorithm input is \( n, a, b, f, \) a relative error tolerance parameter \( \varepsilon_{rel} \) and an absolute error tolerance parameter \( \varepsilon_{abs} \), where \( a = (a_1, a_2, \ldots, a_n) \) and \( b = (b_1, b_2, \ldots, b_n) \). In the description provided below, \( H \) is a priority queue of 4-tuples \( ([x, y], I, \varepsilon, \kappa) \) where \( [x, y] \) is a subregion, \( I \) is an estimate of the integral over this region, \( \varepsilon \) an estimate of the error and \( \kappa \) the dimension along which the subregion should be split if needed. The parameter \( \varepsilon \) determines the priority for extraction of elements from the priority queue. The algorithm maintains a global error estimate \( \varepsilon^g \) and a global integral estimate \( I^g \). The algorithm repeatedly splits the region with greatest local error estimate and updates \( \varepsilon^g \) and \( I^g \). The algorithm terminates when the \( \varepsilon^g \leq \max(\varepsilon_{abs}, \varepsilon_{rel}(I^g)) \) and outputs integral estimate \( I^g \) and error estimate \( \varepsilon^g \).
Algorithm 1 SEQUENTIAL CUHRE(n, a, b, f, τrel, τabs)
1: (I₀, ε₀, κ) ← C-RULES([a, b], f, n)
2: H ← Ø
3: INSERT(H, ([a, b], I₀, ε₀, κ))
4: while ε₀ > max(τabs, τrel|I₀|) do
5:   ([a, b], I, ε, κ) ← EXTRACT-MAX(H)
6:   a’ ← (a_1, a_2, ..., (a_n + b_n)/2, ..., a_n)
7:   b’ ← (b_1, b_2, ..., (a_n + b_n)/2, ..., b_n)
8:   (I_left, ε_left, κ_left) ← C-RULES([a, b’], I_left, f, n)
9:   (I_right, ε_right, κ_right) ← C-RULES([a’, b], I_right, f, n)
10:  I ← I₀ – I_left + I_right
11:  ε ← ε₀ – ε_left + ε_right
12:  INSERT(H, ([a’, b’], I_left, I_right, ε, κ_left, κ_right))
13: end while
14: return I₀ and ε₀

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 2) and SECONDPHASE (Algorithm 3).

Algorithm 2 FIRSTPHASE (n, a, b, f, d, τrel, τabs, L_max)
1: I₀ ← 0, I₀, ε₀ ← 0, ε₀ ← ∞
2: I₀, ε₀ keep sum of integral and error estimates for the “good” subregions
3: while (|L| < L_max) and (|L| ≠ 0) and (ε₀ > max(τabs, τrel|I₀|)) do
4:   S ← Ø
5:   for all j in parallel do
6:     (I_j, ε_j, κ_j) ← C-RULES(L[j], f, n)
7:     INSERT(S, (L[j], I_j, ε_j, κ_j))
8:   end for
9:   L ← PARTITION(S, L_max, τrel, τabs)
10:  (I₀, ε₀, ε₀) ← UPDATE(S, τrel, τabs, I₀, ε₀)
11: end while
12: return (L, I₀, ε₀, I₀, ε₀)

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 [a, 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 [a_j, 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. 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, whereas 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].
used. That, in case (ii) or (iii) second phase of the algorithm is not
I case we can return the integral and error estimates
subregions is created in which case we proceed to the second
end for
32: if \( \delta \) \( L \)
12: \( t \)
11: if \( \epsilon \) \( S \)
8: for \( j = 1 \) to \( |S| \) do
9: \( \left[ a_j, b_j \right] \in \left[ L_{max} \right] \}
3: split \( [a, b] \) along each dimension into \( l \) equal parts
and save these \( l^2 \) subregions into \( L \)
4: return \( L \)
5: end function
6: function UPDATE(\( S, \tau_{rel}, \tau_{abs}, P^P, \varepsilon_P \))
7: \( t_1 \leftarrow P^P, t_2 \leftarrow \varepsilon_P, t_3 \leftarrow 0, t_4 \leftarrow 0 \)
▷ \( t_1, t_2 \) keep the partial sum of integral and error estimates
▷ \( t_3, t_4 \) keep the sum of integral and error estimates for
all the subregions
8: for \( j = 1 \) to \( |S| \) do
9: \( \left( [a_j, b_j], I_j, \varepsilon_j, \kappa_j \right) \) be the \( j \)th record in \( S \)
10: if \( \varepsilon_j \leq \max(\tau_{abs}, \tau_{rel}|I_j|) \) then
11: \( t_1 \leftarrow t_1 + I_j \)
12: \( t_2 \leftarrow t_2 + \varepsilon_j \)
13: else
14: \( t_3 \leftarrow t_3 + I_j \)
15: \( t_4 \leftarrow t_4 + \varepsilon_j \)
16: end if
17: end for
18: \( t_3 \leftarrow t_3 + t_1 \)
19: \( t_4 \leftarrow t_4 + t_2 \)
20: return \( (t_1, t_2, t_3, t_4) \)
21: end function
22: function PARTITION(\( S, \left[ L_{max}, \tau_{rel}, \tau_{abs} \right] \))
23: \( L_1 \leftarrow \emptyset, L_2 \leftarrow \emptyset \)
▷ \( L_1 \) stores the “bad” subregions before subdivision
▷ \( L_2 \) stores the subregions after subdivision of “bad”
subregions
24: for \( j = 1 \) to \( |S| \) do
25: \( \left( [a_j, b_j], I_j, \varepsilon_j, \kappa_j \right) \) be the \( j \)th record in \( S \)
26: if \( \varepsilon_j \geq \max(\tau_{abs}, \tau_{rel}|I_j|) \) then
27: insert \( (\left[ a_j, b_j \right], \kappa_j) \) into \( L_1 \)
28: end if
29: end for
30: \( \delta \leftarrow \text{SPLIT-FACTO}\left( \left[ L_{max}, \left[ L_1 \right] \right] \right) \)
31: for \( j = 1 \) to \( |L_1| \) do
32: \( \left( [a_j, b_j], \kappa_j \right) \) be the \( j \)th record in \( L_1 \)
33: split \( [a_j, b_j] \) into \( \delta \) equal parts along the axis \( \kappa_j \)
34: and insert all these subregions into \( L_2 \)
35: end for
36: return \( L_2 \)
37: end function

Listing 1: Procedures in FirstPhase

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 \( \varepsilon^g \)
as the answer or (iii) \( I^g, \varepsilon^g \) satisfy the error threshold criteria
in which case we also return \( I^g \) and \( \varepsilon^g \) as the answer. Note
that, in case (ii) or (iii) second phase of the algorithm is not
used.

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 \( [a_j, b_j] \) in the
list \( L \) the algorithm calls sequential C Uhre routine (Algorithm
1) 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 3 SecondPhase(\( n, f, \tau_{rel}, \tau_{abs}, \left[ L \right], I^g, \varepsilon^g \))
1: for \( j = 1 \) to \( |L| \) parallel do
2: let \( \left[ a_j, b_j \right] \) be the \( j \)th record in \( L \)
3: \( (I_j, \varepsilon_j) \leftarrow \text{SequentialCuhre}(n, a_j, b_j, f, \tau_{rel}, \tau_{abs}) \)
4: end for
5: \( I^g \leftarrow I^g + \sum \left[ a_j, b_j \right] \in L \) \( I_j \)
6: \( \varepsilon^g \leftarrow \varepsilon^g + \sum \left[ a_j, b_j \right] \in L \) \( \varepsilon_j \)
7: return \( I^g \) and \( \varepsilon^g \)

IV. IMPLEMENTATION

A. CUDA and GPU Architecture

Compute Unified Device Architecture (CUDA) \( 14 \) is a
parallel computing platform and programming model for
designing computations on the GPU. At the hardware level,
a CUDA-enabled GPU device is a set of Single Instruction
Multiple Data (SIMD) stream multi-processors (SM) with
several stream processors (SP) each. Each SP has a limited
number of registers and a private local memory. Each SM
contains a global/device memory shared among the SPs within
the same SM. Thread synchronization through shared memory
is only supported between threads running on the same SM.
Shared memory is managed explicitly by the programmers.
The access to shared memory and register is much faster than
access to global memory. Therefore, handling memory is an
important optimization paradigm to exploit the parallel power
of the GPU for general-purpose computing.

CUDA programming model is a collection of threads
running in parallel. A set of threads are organized as thread
blocks and then, blocks are organized into grids. A grid issued
by the host computer to GPU is called kernel. The maximum
number of threads per block and number of blocks per grid
are hardware-dependent. CUDA computation is often used
to implemented data parallel algorithms where for a given
thread, its index is often used to determined the portion of
data to be processed. Threads in common block communicate
through shared memory. CUDA consists of a set of C language
extensions and a runtime library that provides API to control
the GPU device.

In our implementation we make use of CUDA-based
Figure 1: Flow-chart for GPU-implementation of adaptive multidimensional numerical integration.

THRUST library \([15],\ [16]\) to perform common numerical operations such as summation and prefix scan \([17]\).

B. CUDA Implementation for Multidimensional Numerical Integration

Figure 1 presents a flow-chart of our GPU-implementation. We describe the details of implementation below.

1) Initialization: C-RULE elements are computed and transferred to GPU global memory as a part of GPU initialization. These data are later transferred to shared memory for faster access.

For all the kernels involving a subregions list, the thread and block number are so specified that the kernel consist of at least as many threads as the number of subregions in the input list. It is possible to have at most 1024 threads within each kernel block. However, due to the limited capacity of shared memory and registers, we use a minimum of 128 threads within each thread block.

2) INIT-PARTITION Kernel: The algorithm uses a parameter \(L_{max}\) which defines the maximum number of subregions allowed to be processed in parallel. This optimal value of 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 \([18]\).

INIT-PARTITION Kernel generates \(l^n\) subregions from the entire integration region. The choice of \(l\) is such that \(l^n \leq L_{max}\) but \((l+1)^n > L_{max}\). We use one thread for every new subregion. The thread splits the region along each dimension into \(l\) equal parts and stores them in a list. With each thread generating one new subregion the INIT-PARTITION kernel requires at least \(l^n\) threads.

At the beginning of computation, the region \([a, b]\) is transferred from CPU memory to GPU device memory. In our implementation every subregion \([a, b]\) is stored in global memory as an array of integration starting limits and the length along every dimension.

3) C-RULE Kernel: The goal of this kernel is to maximize the concurrency between threads in locating the subregions with individual error estimates larger than the required tolerance. In addition, due to the high latency of global memory access (400 to 600 cycles), global memory access should be minimized.

We use one CUDA thread to compute the C-RULES on one subregion. At the beginning of the computation, each thread loads a subregion entry from the global memory. The C-RULE elements are stored in the global memory. Since each access to the global memory takes 400-600 clock cycles, it is necessary to store the C-RULE elements in the faster texture memory or shared memory. Moreover, the data are only readable by kernels through texture fetching. It is more costly than loading the data into the shared memory and to compute the C-RULES there. Therefore, the C-RULE elements are stored in the shared memory. Every thread in a block loads a part of the C-RULE elements from global memory into the shared memory. So, all threads must be synchronized before the integral computation begins.

A thread with thread index as \(j\) computes the triplet \((I_j, e_j, \kappa_j)\) for the subregion \([a_j, b_j]\) it is responsible for, where each thread work independently of the others. The kernel returns the list of triplets computed by each thread along with an identifier flag which specifies if the regions has to be further subdivided or not. The identifier flag in our implementation is set to 1 if the subregion has to be further subdivided and 0 otherwise.

4) UPDATE Kernel: After the computation of C-RULES, the algorithm has to compute the partial estimates for the integral and error for all the subregions in the list. The estimates are evaluated as the sum of individual estimates for all subregion in the list. To perform this step in CUDA, we make use of the reduce operation from THRUST library \([15],\ [16]\).

5) PARTITION Kernel: The next step in first phase is to partition the subregions in the list for which the identifier flag is set to be 1. To perform this step in CUDA, we first copy the subregions with identifier flag set as 1 to a new list and then apply the PARTITION kernel on this new subregion list in parallel. Both the original and the new subregion list are stored in GPU device memory. The data copy between the two lists can be implemented efficiently in parallel if the index of the subregions to be copied is known in advance. To perform
this operation, we perform a prefix scan \([17]\) on the identifier flag array that generates the index for all the subregions to be copied. Prefix scan operation from CUDA thrust library \([15], [16]\) is used in our implementation. Once we have the result from prefix scan, we call a CUDA kernel where each thread copies a data element from the original list and stores it in the new list at the index specified by the prefix scan.

6) SECOND PHASE Kernel: This kernel implements the sequential CUHRE (Algorithm \([4]\) on every thread. The only difference from the original implementation is the use of shared memory for storing C-RULE elements. Threads in a block load a part of C-RULE elements into the shared memory. These elements are stored in the shared memory. All the threads must be synchronized before the C-RULE computation begins. A thread works independently of the others in estimating the integral value of the subregion assigned to it using the CUHRE approach \([5]\).

SECOND PHASE kernel outputs the integral and error estimates for every subregion. The global estimate for the entire integration region is estimated as the sum of these individual estimates from SECOND PHASE and the partial estimates from the FIRST PHASE. We use reduce operation from thrust library to perform this step to get our global integral results.

V. PERFORMANCE/EXPERIMENTAL RESULTS FOR CUHRE

The CPU-GPU system used in our experiment consists of Intel\(^{\circledR}\) Xeon\(^{\circledR}\) CPU X5650, 2.67GHz and NVIDIA Tesla M2090 GPU. The Tesla M2090 GPU is based on the recent Fermi architecture \([18]\). 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 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.

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 \([4], [12]\) executed on the host machine of the GPU.

In Figure \(2\) 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 \(\tau_{rel}\) 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 \(10^9\). The proposed method for GPU is up to 100 times faster than the serial code.

<table>
<thead>
<tr>
<th>(f_1(x))</th>
<th>(f_2(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension 5</td>
<td>Dimension 6</td>
</tr>
<tr>
<td>Dimension 7</td>
<td>Dimension 8</td>
</tr>
<tr>
<td>Dimension 9</td>
<td>Dimension 10</td>
</tr>
</tbody>
</table>

**Fig. 2: Simulation Results**
In Figure 2, 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 I 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 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 \( f_5(x) \) with \( \tau_{rel} = 10^{-5} \), the integral estimate computed by FirstPhase satisfied the global error requirement and the algorithm terminates without executing the SecondPhase.

Table II shows a breakdown of GPU execution time.

Table III: Function evaluations in CPU and GPU.

Table III: Function evaluations in CPU and GPU.

In Figure 3, we show the effectiveness of having two phases in our algorithm by comparing the results of the implementation with FirstPhase against the one without FirstPhase. Figure 3a and Figure 3c shows the result of executing two-phase GPU algorithm without FirstPhase and Figure 3b and Figure 3d shows the normal execution with FirstPhase. Both these evaluations were performed on a 5-D function \( f_5(x) \) 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 3a and Figure 3c 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 3b and Figure 3d shows 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 3a and Figure 3c). Table II provides the execution time for SECONDPHASE under 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.

VI. 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