A Memory Efficient Algorithm for Adaptive Multidimensional Integration with Multiple 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*—We present a memory-efficient algorithm and its implementation for solving multidimensional numerical integration on a cluster of compute nodes with multiple GPU devices per node. The effective use of shared memory is important for improving the performance on GPUs, because of the bandwidth limitation of the global memory. The best known sequential algorithm for multidimensional numerical integration** CUHRE **uses a large dynamic heap data structure which is accessed frequently. Devising a GPU algorithm that caches a part of this data structure in the shared memory so as to minimizes global memory access is a challenging task. The algorithm presented here addresses this problem. Furthermore we propose a technique to scale this algorithm to multiple GPU devices. The algorithm was implemented on a cluster of Intel® Xeon® CPU X5650 compute nodes with 4 Tesla M2090 GPU devices per node. We observed a speedup of up to 240 on a single GPU device as compared to a speedup of 70 when memory optimization was not used. On a cluster of 6 nodes (24 GPU devices) we were able to obtain a speedup of up to 3250. All speedups here are with reference to the sequential implementation running on the compute node.**

I. Introduction and Motivation

Multidimensional numerical integration is one of the most important and widely used computational problem in various fields of computational science. Examples include lattice QCD simulations, 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 requiring the ability to perform multidimensional integration for billions of points, quantum mechanics calculations and others.

Many numerical algorithms have been developed, and are part of standard numerical libraries such as NAG, IMSL, QUADPACK, CUBA and others [1]–[4]. Providing reliable estimate for the integral at higher dimension requires considerable amount of CPU time, and often this has to be done with efficient parallel algorithms. However, only a few deterministic parallel algorithms have been developed for adaptive multidimensional numerical integration [5]–[8] . Some of the existing parallel algorithms are a simple extensions of their sequential counterparts, utilizing the multithreading nature of the multicore CPU platform and resulting in a moderate speedup.

Recent emergence of accelerator technologies and multicore architectures with CUDA-enabled GPUs, provides the opportunity to significantly improve the performance of adaptive multidimensional integration on commonly available and inexpensive hardware. The advent of multi-core CPUs with a support of multiple GPUs in a cluster has ensured the scalability of the general purpose computing on GPUs. OpenMP and Message Passing Interface (MPI) programming are often used to manage the GPUs in a cluster.

Multidimensional adaptive numerical integration has been implemented on GPU platform [8] using a single Tesla M2090 device [9]. This implementation exploits the power of a single GPU device to accelerate the integral evaluation and obtains a speedup of up to 100 as compared against the fastest sequential implementations. In spite of the computing power of the GPUs, this implementation does not fully exploit the benefit of the hardware. The performance is limited by the frequent access to the global memory. These accesses are the results of storing dynamic heap data structure required by the algorithm in the global memory.

The algorithm presented here addresses this problem by caching a part of the heap data structure in the shared memory. Furthermore we propose a technique to scale this algorithm to multiple GPU devices. The algorithm was implemented on a

978-1-4799-0730-4/13/$31.00 ©2013 IEEE

cluster of Intel® Xeon® CPU X5650 compute nodes with 4 Tesla M2090 GPU devices per node using OpenMP and Message Passing Interface (MPI). We observed a speedup of up to 240 on a single GPU device as compared to a speedup of 70 when memory optimization was not used. On a cluster of 6 nodes (24 GPU devices) we were able to obtain a speedup of up to 3250. All speedups here are with reference to the sequential implementation running on the compute node.

The remainder of the paper is organized as follows. In section II, we briefly overview deterministic methods for adaptive integration and the related work on GPU based algorithms. The memory-efficient algorithm and its implementation is described in section III. In section IV we extend the memory-efficient algorithm to multiple GPU devices. Finally, in section V, we discuss our findings and outline the future work.

II. Background

A. Adaptive Multidimensional Integration

Adaptive integration is a recursive technique in which a quadrature rule is applied on an integration region to compute the integral estimate and the error estimate associated with that region. The region is subdivided if the quadrature rule estimates for the integral has not met the required accuracy. The subdivided regions repeat the above steps recursively until the error estimate of the associated integration region meets the required accuracy. Many different adaptive integration methods have been developed in the past [5], [6], [10]–[13]. Classical methods for 1-D adaptive integration include Simpson's method, Newton-Cotes 8-point method and Gauss-Kronrod 7/15-point and 10/21-point methods. Some of them have been extended to higher dimension [13].

An extension of 1-D quadrature rules for multidimensional integral is characterized by the exponential growth of functional evaluations with increasing dimension of integration region. For example, applying a Gauss-Kronrod 7/15-point along each coordinate axis of a n-dimensional integral requires 15^n evaluations of the integrand. Thus, an efficient integration algorithm for use in higher dimensions should be adaptive in the entire $n-$D space. CUHRE is one such open source algorithm which is available as a part of CUBA library [4], [14]. Even though the CUHRE method uses much fewer points, in practice it compares fairly well with other adaptive integration methods in terms of accuracy [15].

B. Overview of CUHRE

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

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}\ldots\int_{a_n}^{b_n} f(\mathbf{x})d\mathbf{x}, \tag{1}$$

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

The heart of the CUHRE algorithm is the procedure C-RULES$([\mathbf{a},\mathbf{b}],f,n)$ which outputs a triple (I,ε,κ) where I is an estimate of the integral over $[\mathbf{a},\mathbf{b}]$ (Equation 1), ε is an error estimate for I, and κ is the axis along which $[\mathbf{a},\mathbf{b}]$ 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 $p(n)$ 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.

We now give a high-level description of the CUHRE algorithm (Algorithm 1). The algorithm input is n, $\mathbf{a}$, $\mathbf{b}$, f, a relative error tolerance parameter τ_{rel} and an absolute error tolerance parameter τ_{abs}, where $\mathbf{a}=(a_1,a_2,...,a_n)$ and $\mathbf{b}=(b_1,b_2,...,b_n)$. In the description provided below, H is a priority queue of 4-tuples $([\mathbf{x},\mathbf{y}],I,\varepsilon,\kappa)$ where $[\mathbf{x},\mathbf{y}]$ is a subregion, I is an estimate of the integral over this region, ε an estimate of the error and κ the dimension along which the subregion should be split if needed. The parameter ε determines the priority for extraction of elements from the priority queue. The algorithm maintains a global error estimate ε^g and a global integral estimate I^g. It repeatedly splits the region with greatest local error estimate and updates ε^g and I^g. Finally, the algorithm terminates when the $\varepsilon^g \leq \max(\tau_{abs},\tau_{rel}|I^g|)$ and outputs integral estimate I^g and error estimate ε^g.

Algorithm 1 SEQUENTIALCUHRE$(n,\mathbf{a},\mathbf{b},f,\tau_{rel},\tau_{abs})$

1: $(I^g,\varepsilon^g,\kappa)\leftarrow$ C-RULES$([\mathbf{a},\mathbf{b}],f,n)$
2: $H\leftarrow\emptyset$
3: INSERT$(H,([\mathbf{a},\mathbf{b}],I^g,\varepsilon^g,\kappa))$
4: **while** $\varepsilon^g>\max(\tau_{abs},\tau_{rel}|I^g|)$ **do**
5: $([\mathbf{a},\mathbf{b}],I,\varepsilon,\kappa)\leftarrow$ EXTRACT-MAX(H)
6: $a'\leftarrow(a_1,a_2,...,(a_\kappa+b_\kappa)/2,...,a_n)$
7: $b'\leftarrow(b_1,b_2,...,(a_\kappa+b_\kappa)/2,...,b_n)$
8: $(I_{left},\varepsilon_{left},\kappa_{left})\leftarrow$ C- RULES$([\mathbf{a},\mathbf{b}'],f,n)$
9: $(I_{right},\varepsilon_{right},\kappa_{right})\leftarrow$ C-RULES$([\mathbf{a}',\mathbf{b}],f,n)$
10: $I^g\leftarrow I^g-I+I_{left}+I_{right}$
11: $\varepsilon^g\leftarrow\varepsilon^g-\varepsilon+\varepsilon_{left}+\varepsilon_{right}$
12: INSERT$(H,([\mathbf{a},\mathbf{b}'],I_{left},\varepsilon_{left},\kappa_{left}))$
13: INSERT$(H,([\mathbf{a}',\mathbf{b}],I_{right},\varepsilon_{right},\kappa_{right}))$
14: **end while**
15: **return** $I^g$and ε^g

C. CUDA and GPU Architecture

Compute Unified Device Architecture (CUDA) [16] 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. The latency of accessing global memory is hundreds of clock cycles. Therefore, handling memory is an important optimization paradigm to exploit the parallel power of the GPU for general-purpose computing [17]. Besides the per-block shared memory and global memory, GPU device offers three other types of memory: per-thread private local memory, texture memory and constant memory.

Texture memory and constant memory can be regarded as fast read-only caches.

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 implement data parallel algorithms where for a given thread, its index is often used to determine 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.

D. Single-GPU Implementation

The two-phase parallel algorithm presented in [8] approximates the integral evaluation by adaptively locating the subregions in parallel where the error estimate is greater than the 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). This two-phase algorithm has been successfully implemented for a single GPU device and the results show a speedup of up to 100 times when compared against its best known sequential counterpart.

Algorithm 2 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 - sum of integral and error estimates for the "good" subregions
▷ I^g, ε^g - 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** $(L, I^p, \varepsilon^p, I^g, \varepsilon^g)$

Algorithm 3 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: $(I_j, \varepsilon_j) \leftarrow$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

III. A MEMORY-EFFICIENT ALGORITHM

We first look at the limitation of Algorithm 3 in terms of access to the global memory. The global memory access issue is relevant in the second phase of the algorithm. The SECONDPHASE kernel implements the modified version of SEQUENTIALCUHRE on every GPU thread to estimate the integral value for the subregion assigned to it. Many threads are created in an attempt to hide the latency of global memory by overlapping the execution. A thread in the SECONDPHASE kernel maintains a list of subregion record in the global memory. Subregion record is a C-language $struct$ containing the subregion $[\mathbf{a}, \mathbf{b}]$, C-RULE output triplet (I, ε, κ) for this subregion and a count on number of times the region was subdivided. With double precision representation, we require a total of $16(n + 2)$ bytes of global memory to store a single subregion record.

The CUHRE algorithm proceeds from one stage to next by always choosing the subregion record with largest estimated error for subdivision. This EXTRACT-MAX procedure requires $\Theta(p)$ accesses to the global memory on every stage, where p is size of the subregion list maintained by a thread. The subregion record with largest estimated error is divided along the chosen axis to generate two new subregion records, and then the algorithm performs C-RULE evaluations on each of these new subregion record thereby updating the subregion record list. Every stage of the algorithm requires $\Theta(16p(n+2))$ bytes of read from the subregion list and $32(n + 2)$ bytes of writes to the subregion list. The number of stages and the size of subregion list associated with each thread are often in the orders of hundreds to few thousands and thus increasing the number of global memory accesses. Besides the EXTRACT-MAX procedure, even the application of C-RULE on a subregion record requires frequent access to a subregion record stored in the global memory. The overall performance is significantly affected by the memory performance, because the SECONDPHASE kernel typically involves frequent access to global memory which results in increase in latency and thereby reducing the overall throughput.

A. Using Caching to Avoid Global Memory Accesses

We observe that the most frequently accessed data from the global memory is the list of subregion records. The entire subregion list cannot fit in the shared memory due to its large size. We propose a caching scheme for storing a partial list of subregion records with the highest error estimates in the shared memory. Since the per-block shared memory is limited, we need to restructure the Algorithm 3 so that a block of threads works on subregion instead of a single thread. This means that for loop in Algorithm 3 is parallelized such that each subregion is mapped to a block of threads. The C-RULE function evaluations in the SEQUENTIALCUHRE procedure are distributed equally among the available threads in a block.

A block in the new SECONDPHASE kernel works independent of the other blocks in estimating the integral value of the subregion assigned to them. Each block maintains a list of subregion record in the memory, which is split between per-block shared memory and global memory. The subregion records stored in shared memory is composed of subregions with higher error estimates than the records stored in global

memory. The maximum number of records that can fit in the shared memory often depends on the dimensionality of the problem.

The EXTRACT-MAX procedure for the new algorithm differs from the one used in Algorithm 3. The pseudocode for the procedure EXTRACT-MAX is provided in Listing 1, where H is the subregion list stored in shared memory and G is the subregion list stored in global memory and H_{max} is the maximum number of subregion records that can fit in the shared memory. The SEQUENTIALCUHRE iterates through each stage of the algorithm by processing the subregion records in the shared memory until the size of H exceeds H_{max}. At this point, all the newly-generated H_{max} subregions records are inserted to the subregions list G in global memory. All insertions to list G are *coalesced*. Shared memory is then reset and the top $H_{max}/2$ records with maximum error are inserted to the list H from G using the GM-EXTRACT-MAX routine. At every stage, the subregions with largest estimated errors are assured to be stored in the list H until the list size exceeds H_{max}. SM-EXTRACT-MAX(H) returns the subregion with largest estimated error from list H. The subregion list G is accessed once in every $H_{max}/2$ iterations of the algorithm, which reduce the frequency of global memory access by a factor of $H_{max}/2$.

The benefit of using shared memory will be more prominent when the integrand associated with a region requires less than H_{max} subregions to converge, then no access is made to global memory because shared memory can store all the subregions required in the computation. When the integrand requires more than H_{max} subregions to converge then the last set of subregions in the shared memory are never written back to global memory. The performance benefit achieved from these factors potentially reduce the data transfer between global memory and shared memory thereby contributing to the reduction of overall execution time.

In addition to these improvements to reduce access to global memory we make two more improvements to the algorithm.

Efficient retrieval of subregions: GM-EXTRACT-MAX procedure on the list G is implemented using parallel reduce algorithm. The parallel reduce algorithm works by copying the error estimates of the subregion records in G to a new memory location along with the index of each record. This array of error estimates is reduced in parallel to obtain the top $H_{max}/2$ error estimates. At this stage, the shared memory has been reset and has no valuable information. This allows the reuse of entire shared memory during the parallel reduction operation. The parallel reduction is implemented in shared memory until the size of error estimates array exceeds the available shared memory. When the size of this array grows beyond the available shared memory, then the implementation is moved to global memory. This approach offers a substantial performance benefit especially when the integrand requires fewer subregions to converge.

Use of constant memory: We use the constant memory to store the C-RULE parameters that do not change during the algorithm execution such as evaluation points on a unit hypercube, and the corresponding weights. This provides a significant performance improvement as compared to storing them in global memory. Before invoking the GPU kernels on a set of subregions, all these C-RULE parameters are loaded into constant memory. The 64KB memory capability of the constant memory limits the number of parameters that can be stored in the constant space. Structure and representation of C-RULE parameters are optimized to best fit in the available constant memory.

```
1: function EXTRACT-MAX(H)
2:     if |H| = H_max then
3:         for j = 1 to H_max parallel do
4:             ([a, b], I, ε, κ) ← H[j]
5:             INSERT(G, ([a, b], I, ε, κ)
6:         end for
7:         H ← ∅
8:         for j = 1 to H_max/2 parallel do
9:             ([a, b], I, ε, κ) ← GM-EXTRACT-MAX(G)
10:            INSERT(H, ([a, b], I, ε, κ)
11:        end for
12:    end if
13:    ([a, b], I, ε, κ) ← SM-EXTRACT-MAX(H)
14:    return ([a, b], I, ε, κ)
15: end function
```

Listing 1: EXTRACT-MAX procedure with shared memory

IV. A MULTI-GPU APPROACH

The general idea is to extend the memory-efficient algorithm across a cluster of compute nodes with multiple GPU devices per node. This involves dividing the subregions generated by the FIRSTPHASE kernel equally among the available GPU devices and implementing the SECONDPHASE kernel on each of these device. The pseudocode for FIRSTPHASE is provided in Algorithm 2. The algorithm here creates a list of subregions for 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. The optimal value of L_{max} is estimated based on the target architecture and the number of available GPU devices. For our implementation we have used $L_{max} = 2048d$, where d is the number GPU devices. The generated list of subregions are equally partitioned among the available GPU devices and each partition is assigned to a GPU device implementing the SECONDPHASE kernel.

Communication between GPU devices attached to a compute node are handled using OpenMP, whereas the communication between the compute nodes are handled using MPI programming. All the memory transfers between GPU devices at a node are done using the host (compute node) as an intermediary. The algorithm starts by creating an MPI process for each compute node. One of the MPI process initializes the C-RULE parameters and implements the FIRSTPHASE on a single GPU device to generate a list of subregions. The generated list is transferred to the host memory where it is partitioned equally among the available compute nodes. Each of these partitions are distributed to the compute nodes

using MPI routines. Compute nodes in the cluster receives a set of subregions from the node implementing FIRSTPHASE. These subregions are further partitioned among the available GPU devices at the compute node. Using OpenMP routines, each node creates a thread for every GPU device attached to it. A thread running at the compute node initializes the assigned GPU device and transfers the subregion list to the GPU device memory. SECONDPHASE is executed is parallel by all the threads at the compute node. After completion of SECONDPHASE, the results are transferred back to the node implementing FIRSTPHASE using MPI routines. In our implementation we make use of CUDA-based THRUST library [18], [19] to perform common numerical operations such as summation and prefix scan [20]. These operations are often used in reducing the results obtained from each device.

The scalability of multi-GPU approach often depends on the cluster size and nature of integrand. When the integrand converge to the required accuracy during the FIRSTPHASE, then the SECONDPHASE is never used. The overall performance for such integrands is not affected by the cluster size beyond a threshold. However, this scenario is not common with the poorly behaved integrands that is often encountered in science.

V. PERFORMANCE/EXPERIMENTAL RESULTS

Our experiments for single GPU versions were carried out on a NVIDIA Tesla M2090 GPU device installed on a compute node (host) with Intel® Xeon® CPU X5650, 2.67GHz. 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. The experiments for multiple GPU approach were carried out on a cluster of 6 Intel® Xeon® CPU X5650 compute nodes with 4 NVIDIA Tesla M2090 GPU devices on each compute node. The GPU code was implemented using CUDA 4.0 programming environment. The source code of our multi-GPU implementation is made available at *https://github.com/akkamesh/GPUComputing*.

We 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. The region of integration for all the benchmark functions in our experiments is a unit hypercube $[0,1]^n$. For comparison, we use the sequential C-implementation of CUHRE from the CUBA package [4], [14] that was executed on the host machine. All the GPU kernels in our experiments are executed with a block size of 256 threads and H_{max} is chosen to be 128 records.

Table II compares the performance results for sequential implementation, previous implementation (without memory optimization) on one device, and an memory optimized implementation with one and 24 devices for a subset of functions from the benchmark suite. The dimension and accuracy for

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

TABLE I: n-D benchmark functions.

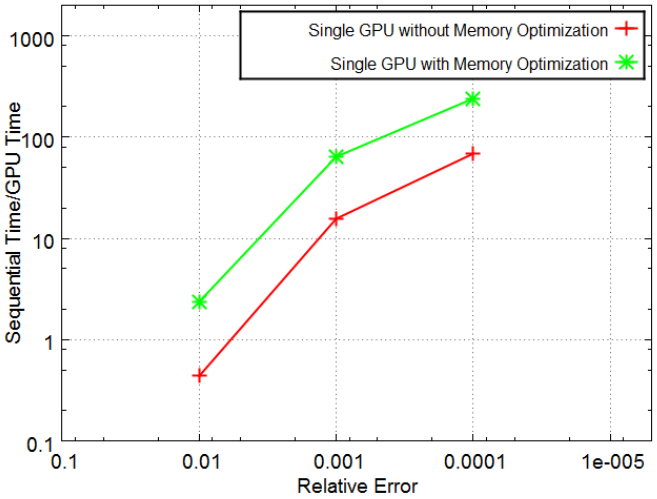

Fig. 1: Speedup results for previous implementation (without memory optimization) and the implementation with memory optimization on one GPU device for the function $f_2(\mathbf{x})$ in 4-D space.

these functions are chosen based on the highest value of these parameters at which the sequential implementation was able to compute the results before reaching the limit for total function evaluations of 10^9.

A. Implementation on a Single GPU

Figure 1 illustrates speedup plot for the previous implementation (without memory optimization) and the implementation with memory optimization for the function $f_2(\mathbf{x})$ in 4-D space against the relative error τ_{rel}. This speedup behavior shown in Figure 1 is similar for other functions in the benchmark suite. Speedup here is computed with reference to the sequential implementation running on the compute node. We observe that the memory optimized GPU implementation improves the speedup by a factor of up to 240 as compared to a speedup of 70 when memory optimization was not used.

We have mainly optimized the SECONDPHASE on GPU, and for FIRSTPHASE the implementation is similar for both the approaches except for some minor modifications. Table III compares the split time for the two components (FIRSTPHASE and SECONDPHASE) in both the approach. We observe that considerable amount of time is spent in SECONDPHASE as compared to FIRSTPHASE. Thus all improvement in the GPU

Function	n	τ_{rel}	Sequential Time (sec.)	Without Memory Optimization One GPU Time (sec.)	Without Memory Optimization One GPU Speedup	With Memory Optimization One GPU Time (sec.)	With Memory Optimization One GPU Speedup	With Memory Optimization 24 GPUs Time (sec.)	With Memory Optimization 24 GPUs Speedup
$f_1(\mathbf{x})$	7	10^{-5}	2349	54.76	42.89	18.12	129.63	3.87	607.18
$f_2(\mathbf{x})$	4	10^{-4}	3621	52.85	68.52	15.10	239.92	1.11	3252.6
$f_3(\mathbf{x})$	4	10^{-5}	286	28.98	9.87	11.36	25.19	4.55	62.89
$f_4(\mathbf{x})$	7	10^{-4}	9876	279.39	35.35	71.75	137.65	19.60	503.90

TABLE II: Performance results for sequential implementation on CPU, previous implementation (without memory optimization) on one GPU device, memory optimized implementation with one and 24 GPU devices for benchmark functions in Table I.

τ_{rel}	Sequential time (sec.)	GPU time without memory optimization (sec.) FIRSTPHASE	GPU time without memory optimization (sec.) SECONDPHASE	GPU time with memory optimization (sec.) FIRSTPHASE	GPU time with memory optimization (sec.) SECONDPHASE
10^{-2}	2.4	1.3	4.21	0.019	1.0
10^{-3}	315.3	1.1	19.15	0.022	4.9
10^{-4}	3621.3	1.0	51.83	0.023	15.1

TABLE III: Breakdown of time for the two components FIRSTPHASE and SECONDPHASE for the implementation without memory optimization and with memory optimization on one GPU for the function $f_2(\mathbf{x})$ in 4-D space.

implementation is mainly due to the optimization of the SECONDPHASE.

B. Implementation on Multiple GPUs

We performed experiments to see the impact on the speedup with the number of GPU devices. Figure 2 illustrate speedup plots for two different functions: $f_2(\mathbf{x})$ in 4-D space with a relative error of $\tau = 10^{-4}$ and $f_1(\mathbf{x})$ in 7-D space with relative error of $\tau = 10^{-4}$. These two functions are chosen to illustrate different behaviors of all the simulations executed. The speedup here is with reference to the memory optimized implementation on one GPU device. We observe a speedup of up to 14 on 24 GPU devices compared against one GPU device. This translates to a overall speedup of up to 3250 with 24 GPUs compared to a sequential implementation.

With the increase in number of GPUs, FIRSTPHASE kernel generates more balanced computational load and thus improving the performance. The load balancing nature of FIRSTPHASE has been proved to be an important strategy in [8]. In our multi-GPU approach the overall execution time is a combination of FIRSTPHASE kernel, SECONDPHASE kernel and other overheads. The overheads includes MPI communication between the compute nodes, GPU device initialization and so on.

In Figure 2, we observe a near-linear scaling of up to 24 GPUs for the function $f_2(\mathbf{x})$. Figure 3 shows the breakdown of the execution time for different components of the implementation. We observe a near linear increase in execution time for FIRSTPHASE. With every new device in the cluster, the FIRSTPHASE performs more work to generate a relatively larger list of subregions and thus resulting in a linear increase in execution time. SECONDPHASE is the computational intensive component of the algorithm which is distributed across multiple GPU devices. Thus SECONDPHASE time decreases with increase in number of GPU devices and thereby improving the overall performance. We notice that the overhead involved in the implementation grows with the number of GPU devices which can potentially bring down the performance. However,

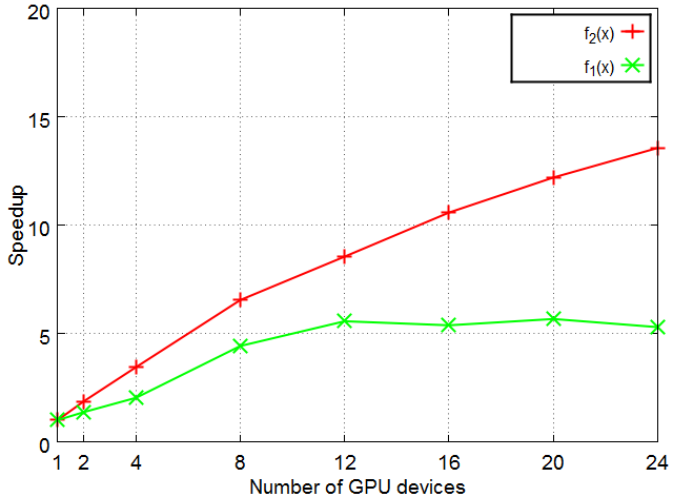

Fig. 2: Speedup results for the memory optimized implementation on GPU with varying number of GPUs for two functions: $f_1(\mathbf{x})$ in 7-D and $f_2(\mathbf{x})$ in 4-D (speedup is with reference to the memory optimized implementation on one GPU).

for the function $f_2(\mathbf{x})$ the performance gain is dominated by the SECONDPHASE and thereby suppressing the effects of overhead.

In Figure 2, we see a clear deviation from the linear scaling as the number of GPUs are increased for the function $f_1(\mathbf{x})$. Figure 4 shows the split time for the different components of implementation. We notice that with increase in number of devices the execution time is dominated by the FIRSTPHASE and other additional overheads. This function attains its maximum performance benefit with 12 GPU devices and beyond which the FIRSTPHASE time grows linearly with the number of devices. This behavior of function $f_1(\mathbf{x})$ limits the speedup in multi-GPU environment. Note that in general the speedup

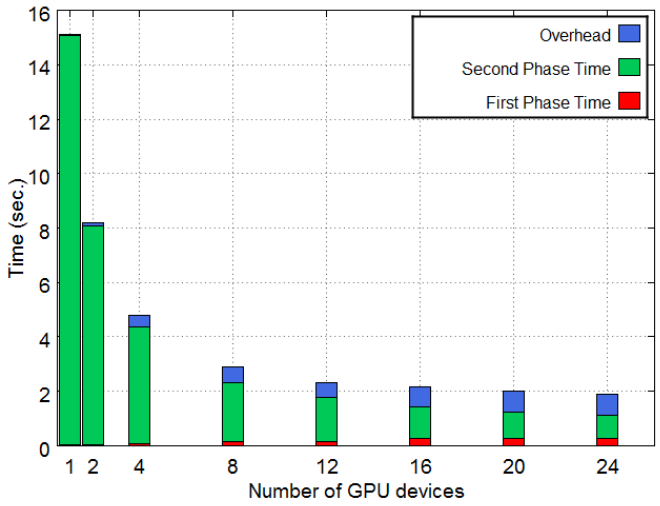

Fig. 3: Graph showing the split computation time with different number of GPUs for the function $f_2(\mathbf{x})$ in 4-D space.

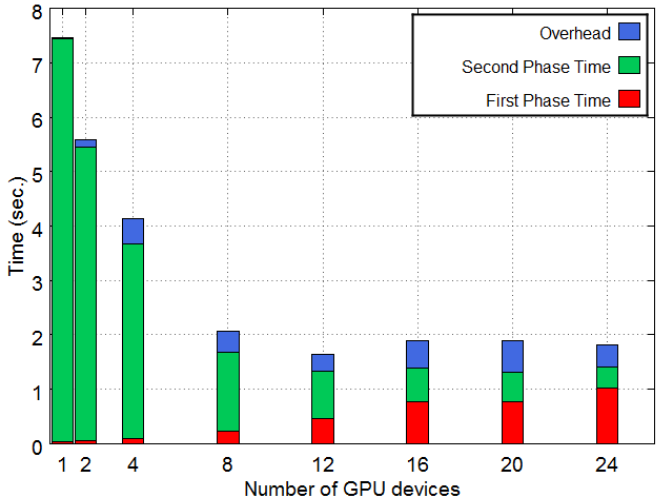

Fig. 4: Graph showing the split computation time with different number of GPUs for the function $f_1(\mathbf{x})$ in 7-D space.

scales near linearly with the increase in number of devices until a threshold number of device beyond which the performance would degrade due to additional overheads involved.

VI. Conclusion

In this paper, we presented a memory-efficient algorithm for solving multidimensional numerical integration on a cluster of compute nodes with multiple GPU devices per node. We demonstrated that it is possible to significantly improve the performance by using shared memory in GPUs. We obtained a speedup of up to 240 on a single GPU device as compared to a speedup of 70 when memory optimization was not used. We demonstrated that a cluster with 6 compute nodes with 4 GPU devices per node can speedup the computation by a factor of 3250 compared to a leading sequential method. To enhance the performance further it is necessary to reduce the additional overhead involved in the multi-GPU implementation and improve the global memory efficiency by coalescing the memory access.

Acknowledgment

We would like to thank the support of the Jefferson Science Associates Project 712336 and the U.S. Department of Energy 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 and E. de Doncker-Kapenga and 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] J. Bernsten, T. Espelid, 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 and 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] K. Arumugam, A. Godunov, D. Ranjan, B. Terzić, and M. Zubair, "An Efficient Deterministic Parallel Algorithm for Adaptive Multidimensional Numerical Integration on GPUs," *International Conference on Parallel Processing (ICPP)*, October 2013.

[9] NVIDIA, "Tesla M2090." [Online]. Available: http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf

[10] A. Genz, "An adaptive multidimensional quadrature procedure," *Computer Physics Communications*, vol. 4, pp. 11–15, October 1972.

[11] A. Genz and R. Cools, "An adaptive numerical cubature algorithm for simplices," *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 3, pp. 297–308, September 2003.

[12] 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.

[13] G. Dalquist and A. Björck, *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics, 2008, vol. 1.

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

[15] P. Gonnet, "A review of error estimation in adaptive quadrature," *ACM Computing Surveys (CSUR)*, vol. 44, no. 22, December 2012.

[16] NVIDIA, "CUDA C Programming Guide." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[17] Y. Kim and A. Shrivastava, "CuMAPz: A tool to analyze memory access patterns in CUDA," *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 128 – 133, June 2011.

[18] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems Jade Edition*, 2011.

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

[20] H. Nguyen, "Parallel prefix sum (scan) with CUDA," *GPU Gems 3*, 2007.