Parallel Processing of Irregular Workloads on the GPGPU: Adaptive Quadrature

Derek Kern and Gita Alaghband
Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO, USA
{derek.kern, gita.alaghband}@ucdenver.edu

Abstract— *This paper presents a parallel (GPGPU) approach for dealing with the turbid workload of adaptive quadrature, called 'parallel block-cutting adaptive quadrature' (PBCAQ). PBCAQ provides speedups as high as 211 times the performance of its sequential competitors. In addition, it has two intertwined and desirable properties: (1) its speedups increase as the size of the workloads being processed increase; and (2) it performs best over definite integrals requiring larger workloads. These two properties together make PBCAQ a valuable example of computing an inequitable, turbid workload on the GPGPU, devices that require workload simplicity.*

Keywords: Parallel processing, GPU, CUDA, SIMT, adaptive quadrature, numerical integration

1. Introduction

In this paper we explore an efficient implementation of adaptive quadrature (AQ) on GPGPU architectures. This problem is selected as an example of an algorithm that exhibits an unpredictable workload and poses challenges in equitably dividing work. Problems with these characteristics are generally difficult to parallelize effectively for the SIMT (Single Instruction Multiple Thread) parallel model of GPGPUs. This computing model offers significant performance benefits for applications with predictable, regular patterns of parallelism and computation, where a single instruction can be applied to many data items at the same time (within GPGPU *threads*).

On GPGPUs, the instruction sequences, called *kernels*, are launched by the CPU onto the GPGPU forming groups of parallel threads, called *warps*, that will execute concurrently on GPGPU *streaming multiprocessors* [1], [2]. Losses of GPGPU computing efficiency occur when: (1) computing units sit idle, which happens when loads are not properly balanced; (2) threads within warps diverge; and (3) warps sit idle during memory accesses. The additional flexibility that comes from the SIMT model and GPGPU architectures cannot easily be exploited without detailed knowledge of such facets [3].

Some workloads, as that of AQ, are intrinsically difficult to conform with processing workloads suitable for the GPGPUs. Often these workloads are resistant to simple or equitable division. In some instances, it may be because the elements of the workload are not uniform and cannot be further divided into uniform elements. In other instances, it may be because dependences that cannot be discovered statically make workload division difficult; this type of workload is known as an *amorphous* workload [4], [5].[1]

Another reason that a workload may not easily conform to GPGPU processing is because the amount of work left to be done cannot be simply circumscribed. In these cases, the task of deciding upon equitable work divisions, needed for parallel load balancing, is either too costly or not possible in principle. In this paper, we introduce the term *turbid workload* to refer to this type of unpredictable workload. Breadth-first search is a good example of an algorithm with such a workload [6].

Adaptive quadrature (AQ) is another example of an algorithm with a turbid workload. AQ is a divide and conquer process that is used to refine the approximation of definite integrals, the area under the curve of a function over a specific interval [10]. It works by first estimating an approximation of the area under the curve for the given interval; this approximation is checked for accuracy (within a given tolerance); if the integral is not within the tolerance, the interval is divided in half and each subinterval is approximated recursively; accurate integrals calculated for subintervals are accumulated. During this process, integrals are approximated using methods like the trapezoidal rule or Simpson's rule. Each time the interval is divided and work preserved for later processing, it is unclear how much work is left within each division. Thus, throughout the processing of AQ, divisions of workload are unlikely to be equal. These types of problems pose a serious challenge to parallel speedup and efficiency for SIMT-type parallelism.

AQ has many applications. Among them are holographic interferometry [7], multilevel regression models [8], and free-surface motion in liquids [9]. Any of these applications and many others would greatly benefit from a GPGPU accelerated form of adaptive quadrature.

In this paper, a parallel algorithm for GPGPU processing of the turbid workload of adaptive quadrature will be shown. It is called 'parallel block-cutting adaptive quadrature' (PBCAQ). PBCAQ is an effective approach for parallelizing adaptive quadrature. It provides significant speedups over sequential competitors; on some definite integrals, these speedups can reach as much as 211 times.

This paper is organized as follows. In Section 2, the basics

[1]The concept of an amorphous workload is closely tied to the concept of *amorphous data-parallelism* put forward by Kulkarni et al [4], [5]. Whereas amorphous data-parallelism refers to the pattern of parallelism exhibited by an algorithm, an amorphous workload refers to the workload resulting from an algorithm whose pattern of parallelism is amorphous data-parallelism.

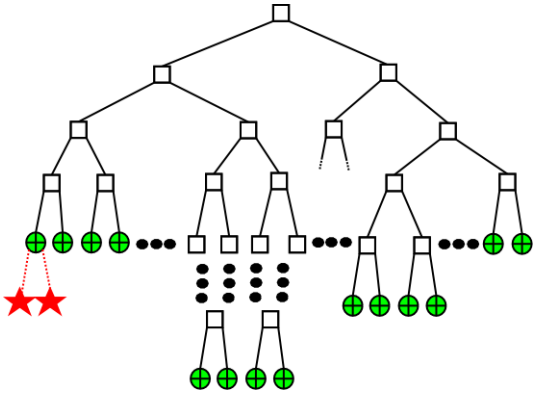


Fig. 1: Computational tree for adaptive quadrature after values of $f(x)$, a, b, and τ are fixed. White boxes represent *mediate computations*; green circles represent *immediate computations*; red stars represent *unnecessarily fine immediate computations*.

of adaptive quadrature are described. Section 3 introduces the continuity assumption, which is the animating principle behind PBCAQ. Section 4 provides a detailed explication of the PBCAQ algorithm. In Section 5, the results for PBCAQ are presented in comparison to sequential AQ versions as well as modified forms of the PBCAQ algorithm. This paper concludes with Section 6.

2. Adaptive quadrature

As stated in Section 1, adaptive quadrature (AQ) is a divide and conquer process that is used to refine the approximation of definite integrals. Let $\int_a^b f(x)\,dx$ be the integral being evaluated using AQ over interval $i = [a, b]$ for a given approximation tolerance τ. During the integration process, subintervals of i, defined as $i_m = [a_m, b_m]$, will be approximated within tolerance τ_m, where τ_m is τ divided as many times as the subinterval of i and $a \leq a_m < b_m \leq b$. For each interval i_m, AQ evaluates the interval using a coarse approximation method and a fine approximation method.[2] Let s_c be the approximation returned by the coarse method and let s_f be the approximation returned by the fine method. If $|s_c - s_f| \leq \tau_m$, then i_m is approximated within tolerance and added to the overall result; if $|s_c - s_f| > \tau_m$, then i_m is divided, typically into two halves, and AQ approximates each of the divisions. Let the divide and conquer approach to AQ be known as 'common adaptive quadrature' (CAQ).

In order to ease discussion, some new terms are needed. Let a *mediate computation* be the computation of subinterval i_m resulting in an approximation not within tolerance; mediate computations are the empty, white boxes in Figure 1. Note that a mediate computation results in its interval, i_m, being further divided into smaller subintervals $i_{m1}, i_{m2}, ...i_{mn}$ for approximation. Let an *immediate computation* be the computation of subinterval i_m resulting in an approximation within tolerance; immediate computations are the green circles in Figure 1. Let an *unnecessarily fine immediate computation* be an immediate computation of a subinterval i_{m1} within tolerance at length ℓ_1 when a subsuming subinterval i_{m2} of length ℓ_2 (where $\ell_2 = \ell_1 * 2^x, x \geq 1$) exists and i_{m2} can be approximated within tolerance; unnecessarily fine immediate computations are the red stars in Figure 1.

[2]All of the algorithms tested in this paper were implemented using the trapezoidal rule for coarse approximation and Simpson's rule for fine approximation.

3. The Continuity Assumption

The design of PBCAQ rests upon a key assumption. This assumption is formally stated below:

Continuity Assumption: Given some continuous function $f(x)$, an integral approximation method M and a tolerance τ, if $\int_a^b f(x)\,dx$ is being approximated using adaptive quadrature and if subinterval i of length l is approximated by M within τ, then the intervals adjacent to i of length l, $i+1$ and $i-1$, will likely also be approximated by M within τ. Similarly, given some continuous function $f(x)$, an integral approximation method M and a tolerance τ, if $\int_a^b f(x)\,dx$ is being approximated using adaptive quadrature and if subinterval i of length l is *not* approximated by M within τ, then the intervals adjacent to i of length l, $i+1$ and $i-1$, will likely *not* be approximated by M within τ.

The continuity assumption is a useful guide to avoiding some of the mediate computations that are normally visited within the AQ computational tree. With the size of adjacent intervals as starting points, much of the mediate work of repeatedly finding correct interval sizes can be skipped. This means that groups of adjacent intervals can be quickly approximated.

Within Figure 2, the mediate computations that this assumption eliminates can be seen. PBCAQ finds an initial interval size (depth first) and then, by assuming continuity, traverses the leaves of the computational tree (horizontally) until the interval size no longer applies; at which point, it either slightly enlarges or shrinks the interval size (and tolerance); it then continues traverse the leaves of the computational tree at the new interval size.[3] While mediate computations are not eliminated, their number can be mitigated by the continuity assumption.

4. Parallel, block-cutting adaptive quadrature for the GPGPU

4.1 Basic algorithm

PBCAQ is implemented in NVIDIA's Compute Unit Device Architecture (CUDA). As such, CUDA nomenclature will be used throughout. Physically, an NVIDIA GPGPU consists of some number of *streaming multiprocessors* (SM), each of which has some number of *cores*, usually 32.[4] Thus, if an NVIDIA GPGPU has 14 SMs, then it has 448 cores. Logically, the primary unit of computation in CUDA is the

[3]As will be discussed, PBCAQ works on the level of regions (of intervals). However, the continuity assumption applies just the same.

[4]At times, cores will also be referred to as *compute units*.

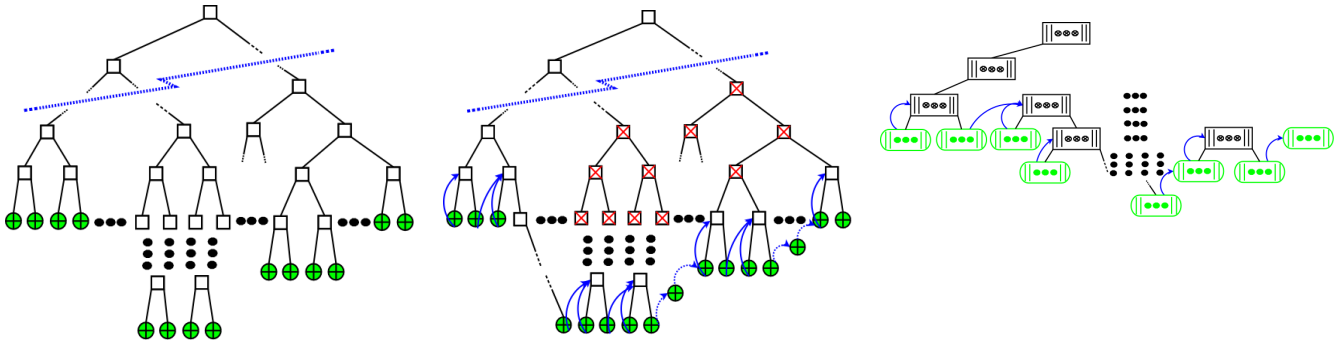


Fig. 2: Comparison of computational trees for CAQ (left), SCAQ* and PBCAQ (right). In CAQ and SCAQ, squares and circles, respectively, represent mediate and immediate computations of intervals; squares with X's represent mediate computations not performed by SCAQ. In PBCAQ, rectangles and ovals, respectively, represent mediate and immediate computations of (intervals within) regions. * Note that SCAQ is described below in Section 4.3

thread such that a single thread will be run on a single core of a single SM. Threads are organized into *blocks* and blocks are organized into *grids*. While any thread can utilize global memory on the device, threads within the same block will execute on the same SM and, thus, are able to utilize shared local memory. Finally, since the number of threads that can be launched on a GPGPU vastly outnumber the number of available cores, threads that are part of the same block are organized into *warps*, which are groups of threads that execute simultaneously on a SM.

AQ algorithms, like CAQ, often work on the level of individual intervals (and subintervals). PBCAQ, on the other hand, works by simultaneously approximating huge numbers of intervals. Let the intervals simultaneously processed on the GPGPU by PBCAQ be known as a *region*. When CAQ approximates an interval i, it compares the error for that approximation to the tolerance to determine whether it is within tolerance. When PBCAQ approximates a region r, it simultaneously approximates all intervals within r on the GPGPU; it then compares the sum of the errors for all intervals in r to the tolerance to determine whether the approximation of r is within tolerance. Thus, PBCAQ never considers whether the approximation of an individual interval of a region is within tolerance; instead, PBCAQ is only concerned with whether a region, as a whole, has been approximated within tolerance. The distinction between the individual intervals processed by CAQ and regions (of intervals) processed by PBCAQ can be seen in Figure 2.

PBCAQ begins by setting $length$, which is the length of the region being approximated, to be the entire length from $lower$ to $upper$. It approximates the first region at level $\log_2(\alpha * \beta)$ (where $\alpha * \beta = 2^x, x \geq 1$) such that α is the number of blocks executed on the GPGPU and β is the number of threads per block. It continues (depth first) to shrink $length$ by half, summing all of the errors from all of the blocks (in the region) approximated on the GPGPU, until it achieves an approximation within tolerance for the entire region. After such an approximation, it adds the region approximation to the total approximation, doubles $length$ and then, moving from $upper$ to $lower$, proceeds to approximate the adjacent region, if there is one, which will continue to be the case until it breaches $lower$.

function PBCAQ($f, lower, upper, \tau$)
  Let α be the number of thread blocks;
  Let β be the number of threads per block;
  $I \leftarrow 0.0$; $length \leftarrow upper - lower$;
  Let S and E be apprxs and errs;
  while $upper > lower$ **do**
    ▷ Apprxs and errs are generated by thrds in next step;
    Approximate $length$ on GPGPU using $(\alpha * \beta)$ thrds; barrier;
    Reduce apprxs and errs, S and E, on GPGPU; barrier;
    Transfer apprxs and errs, S and E, to host mem;
    ▷ Compare error for region to tolerance
    if $E \leq \tau$ **then**
      $I \leftarrow I + S$;
      $upper \leftarrow lower$;
      $length \leftarrow 2 * length$; $\tau \leftarrow \tau * 2$; } ▷ Expansion step
    else
      $length \leftarrow length/2$;
      $\tau \leftarrow \tau/2$;
    end if
    $lower \leftarrow max(upper - length, lower)$;
  end while
  return I;
end function

Fig. 3: Parallel, block-cutting adaptive quadrature (PBCAQ)

Considering the approximation of regions by PBCAQ, let the region being approximated be r_m. PBCAQ (displayed in Figure 3) breaks r_m into $\alpha*\beta$ intervals, where α is the number of thread blocks and β is the number of threads per block. It submits the region to the GPGPU where each thread is assigned its own interval to approximate. Each thread will find both an approximation and an error for its assigned interval.

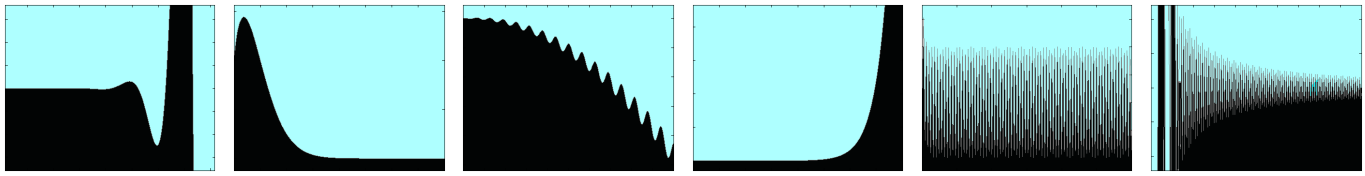
After each of the $\alpha * \beta$ intervals are approximated within the region, the approximations and the corresponding errors

Table 1: Adaptive quadrature versions tested

Name	Description
CAQ	Sequential, queue-based, divide and conquer
SCAQ	Sequential, side-cutting
BCAQ	Sequential version of PBCAQ
PSCAQ	Parallel, side-cutting
PBCAQ	Parallel, block-cutting
PBCAQr	PBCAQ with coin-flip interval doubling
PBCAQs	PBCAQ with error-slope-based interval doubling
PBCAQg	PBCAQ with log of tolerance over error-based interval doubling

Table 2: Functions and intervals integrated over for testing

#	f(x)	Intervals
1	$e^{2x}sin(3x)$	**0.0–5.0**, **0.0–6.0**, **0.0–7.0**
2	$(x^x)^{-1}$	**0.0–1000.0**, **0.0–2000.0**, **0.0–3000.0**
3	$(4x)cos(2x) - (x-2)^2$	**0.0–30.0**, **0.0–50.0**, **0.0–70.0**
4	$e^{4x}(\sqrt{1+e^{4x}})^r - 1$	**0.0–5.0**, **0.0–6.0**, **0.0–6.75**
5	$e^{-3x}cos(5\pi x)$	**0.0–10.0**, **0.0–20.0**, **0.0–30.0**
6	$sin(10\pi x)(\pi x)^{-1}$	**1e-6–10.0**, **1e-6–15.0**, **1e-6–20.0**



Function #1 Function #2 Function #3 Function #4 Function #5 Function #6

Fig. 4: Graphs of functions #1–#6

for these intervals must be reduced (summed) into a single approximation and error bound for the region. This reduction is done primarily on the GPGPU. After it is complete, two highly reduced arrays, one containing errors and the other containing approximations, are transferred back into main memory for final reduction by the CPU.

The performance benefits of PBCAQ are due to several factors. First, no mediate computation takes place for the top part of the computational tree up to level $\log_2(\alpha * \beta)$. This means that approximation of the integral will start for the first region at approximately level 19.[5] Second, using the continuity assumption, we evaluate adjacent equally divided intervals within regions. If a region is approximated accurately, the integral of the entire region is achieved and no further evaluation is needed for this region. Adjacent regions are then approximated accordingly and only divided further if the desired accuracy is not achieved for the region. Third, the abundance of dedicated parallelism in the GPGPU results in small enough intervals within each region to achieve the result within the desired accuracy very fast. In fact, this method would work very well for many core MIMD platforms. Finally, PBCAQ is able to simultaneously approximate entire regions, made up of hundreds of thousands or more intervals. It can do this efficiently because it can spread the approximation of a region across the many threads available on the GPGPU.

[5]This assumes $\alpha = 2048$ and $\beta = 256$. These block and thread allocations values were used for most of the test functions (see Table 2) in this paper.

4.2 Improving the algorithm

Aside from the speedups that come from the simultaneous region approximation of PBCAQ, there is a means of further improving its performance. In Figure 3, note the two bracketed lines colored blue and labeled "Expansion step". The expansion step executes each time a region is approximated within tolerance. If the next, adjacent region of $length * 2$ can be approximated within tolerance, then the expansion step diminishes (unnecessarily fine) immediate computations. However, if this is not the case, then it increases the number of mediate computations.

What is needed is a way to execute the expansion step when it diminishes the number of immediate computations and skip it when it increases the number of mediate computations. This need gives rise to three further versions of PBCAQ: (1) PBCAQr, which executes the expansion step based upon a random coin-flip; (2) PBCAQs, which executes the expansion step based upon the slope of the errors returned from the GPGPU, i.e. when errors decreased over the region; and (3) PBCAQg, which executes the expansion step based upon the (natural) log of the tolerance over the sum of the errors returned from the GPGPU.

While PBCAQs and PBCAQg have obvious value in that they provide an inductive prediction upon whether the next region of $length$ will be approximated within tolerance, PBCAQr, on the surface seems to be useful only as a point of comparison. However, it has value independent of its comparative value. PBCAQr makes no prediction. Yet, it does provide the means to skip the expansion step half of the time,

Parallel Processing of Irregular Workloads on the GPGPU: Adaptive Quadrature

Derek Kern and Gita Alaghband
Department of Computer Science and Engineering, University of Colorado Denver, Denver, CO, USA
{derek.kern, gita.alaghband}@ucdenver.edu

Abstract— *This paper presents a parallel (GPGPU) approach for dealing with the turbid workload of adaptive quadrature, called 'parallel block-cutting adaptive quadrature' (PBCAQ). PBCAQ provides speedups as high as 211 times the performance of its sequential competitors. In addition, it has two intertwined and desirable properties: (1) its speedups increase as the size of the workloads being processed increase; and (2) it performs best over definite integrals requiring larger workloads. These two properties together make PBCAQ a valuable example of computing an inequitable, turbid workload on the GPGPU, devices that require workload simplicity.*

Keywords: Parallel processing, GPU, CUDA, SIMT, adaptive quadrature, numerical integration

1. Introduction

In this paper we explore an efficient implementation of adaptive quadrature (AQ) on GPGPU architectures. This problem is selected as an example of an algorithm that exhibits an unpredictable workload and poses challenges in equitably dividing work. Problems with these characteristics are generally difficult to parallelize effectively for the SIMT (Single Instruction Multiple Thread) parallel model of GPGPUs. This computing model offers significant performance benefits for applications with predictable, regular patterns of parallelism and computation, where a single instruction can be applied to many data items at the same time (within GPGPU *threads*).

On GPGPUs, the instruction sequences, called *kernels*, are launched by the CPU onto the GPGPU forming groups of parallel threads, called *warps*, that will execute concurrently on GPGPU *streaming multiprocessors* [1], [2]. Losses of GPGPU computing efficiency occur when: (1) computing units sit idle, which happens when loads are not properly balanced; (2) threads within warps diverge; and (3) warps sit idle during memory accesses. The additional flexibility that comes from the SIMT model and GPGPU architectures cannot easily be exploited without detailed knowledge of such facets [3].

Some workloads, as that of AQ, are intrinsically difficult to conform with processing workloads suitable for the GPGPUs. Often these workloads are resistant to simple or equitable division. In some instances, it may be because the elements of the workload are not uniform and cannot be further divided into uniform elements. In other instances, it may be because dependences that cannot be discovered statically make workload division difficult; this type of workload is known as an *amorphous* workload [4], [5].[1]

Another reason that a workload may not easily conform to GPGPU processing is because the amount of work left to be done cannot be simply circumscribed. In these cases, the task of deciding upon equitable work divisions, needed for parallel load balancing, is either too costly or not possible in principle. In this paper, we introduce the term *turbid workload* to refer to this type of unpredictable workload. Breadth-first search is a good example of an algorithm with such a workload [6].

Adaptive quadrature (AQ) is another example of an algorithm with a turbid workload. AQ is a divide and conquer process that is used to refine the approximation of definite integrals, the area under the curve of a function over a specific interval [10]. It works by first estimating an approximation of the area under the curve for the given interval; this approximation is checked for accuracy (within a given tolerance); if the integral is not within the tolerance, the interval is divided in half and each subinterval is approximated recursively; accurate integrals calculated for subintervals are accumulated. During this process, integrals are approximated using methods like the trapezoidal rule or Simpson's rule. Each time the interval is divided and work preserved for later processing, it is unclear how much work is left within each division. Thus, throughout the processing of AQ, divisions of workload are unlikely to be equal. These types of problems pose a serious challenge to parallel speedup and efficiency for SIMT-type parallelism.

AQ has many applications. Among them are holographic interferometry [7], multilevel regression models [8], and free-surface motion in liquids [9]. Any of these applications and many others would greatly benefit from a GPGPU accelerated form of adaptive quadrature.

In this paper, a parallel algorithm for GPGPU processing of the turbid workload of adaptive quadrature will be shown. It is called 'parallel block-cutting adaptive quadrature' (PBCAQ). PBCAQ is an effective approach for parallelizing adaptive quadrature. It provides significant speedups over sequential competitors; on some definite integrals, these speedups can reach as much as 211 times.

This paper is organized as follows. In Section 2, the basics

[1]The concept of an amorphous workload is closely tied to the concept of *amorphous data-parallelism* put forward by Kulkarni et al [4], [5]. Whereas amorphous data-parallelism refers to the pattern of parallelism exhibited by an algorithm, an amorphous workload refers to the workload resulting from an algorithm whose pattern of parallelism is amorphous data-parallelism.

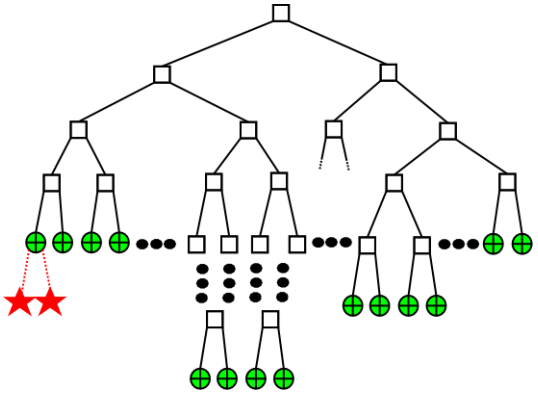


Fig. 1: Computational tree for adaptive quadrature after values of $f(x)$, a, b, and τ are fixed. White boxes represent *mediate computations*; green circles represent *immediate computations*; red stars represent *unnecessarily fine immediate computations*.

of adaptive quadrature are described. Section 3 introduces the continuity assumption, which is the animating principle behind PBCAQ. Section 4 provides a detailed explication of the PBCAQ algorithm. In Section 5, the results for PBCAQ are presented in comparison to sequential AQ versions as well as modified forms of the PBCAQ algorithm. This paper concludes with Section 6.

2. Adaptive quadrature

As stated in Section 1, adaptive quadrature (AQ) is a divide and conquer process that is used to refine the approximation of definite integrals. Let $\int_a^b f(x)\,dx$ be the integral being evaluated using AQ over interval $i = [a, b]$ for a given approximation tolerance τ. During the integration process, subintervals of i, defined as $i_m = [a_m, b_m]$, will be approximated within tolerance τ_m, where τ_m is τ divided as many times as the subinterval of i and $a \leq a_m < b_m \leq b$. For each interval i_m, AQ evaluates the interval using a coarse approximation method and a fine approximation method.[2] Let s_c be the approximation returned by the coarse method and let s_f be the approximation returned by the fine method. If $|s_c - s_f| \leq \tau_m$, then i_m is approximated within tolerance and added to the overall result; if $|s_c - s_f| > \tau_m$, then i_m is divided, typically into two halves, and AQ approximates each of the divisions. Let the divide and conquer approach to AQ be known as 'common adaptive quadrature' (CAQ).

In order to ease discussion, some new terms are needed. Let a *mediate computation* be the computation of subinterval i_m resulting in an approximation not within tolerance; mediate computations are the empty, white boxes in Figure 1. Note that a mediate computation results in its interval, i_m, being further divided into smaller subintervals $i_{m1}, i_{m2}, ...i_{mn}$ for approximation. Let an *immediate computation* be the computation of subinterval i_m resulting in an approximation within tolerance; immediate computations are the green circles in Figure 1. Let an *unnecessarily fine immediate computation* be an immediate computation of a subinterval i_{m1} within tolerance at length ℓ_1 when a subsuming subinterval i_{m2} of length ℓ_2 (where $\ell_2 = \ell_1 * 2^x, x \geq 1$) exists and i_{m2} can be approximated within tolerance; unnecessarily fine immediate computations are the red stars in Figure 1.

[2]All of the algorithms tested in this paper were implemented using the trapezoidal rule for coarse approximation and Simpson's rule for fine approximation.

3. The Continuity Assumption

The design of PBCAQ rests upon a key assumption. This assumption is formally stated below:

Continuity Assumption: Given some continuous function $f(x)$, an integral approximation method M and a tolerance τ, if $\int_a^b f(x)\,dx$ is being approximated using adaptive quadrature and if subinterval i of length l is approximated by M within τ, then the intervals adjacent to i of length l, $i+1$ and $i-1$, will likely also be approximated by M within τ. Similarly, given some continuous function $f(x)$, an integral approximation method M and a tolerance τ, if $\int_a^b f(x)\,dx$ is being approximated using adaptive quadrature and if subinterval i of length l is *not* approximated by M within τ, then the intervals adjacent to i of length l, $i+1$ and $i-1$, will likely *not* be approximated by M within τ.

The continuity assumption is a useful guide to avoiding some of the mediate computations that are normally visited within the AQ computational tree. With the size of adjacent intervals as starting points, much of the mediate work of repeatedly finding correct interval sizes can be skipped. This means that groups of adjacent intervals can be quickly approximated.

Within Figure 2, the mediate computations that this assumption eliminates can be seen. PBCAQ finds an initial interval size (depth first) and then, by assuming continuity, traverses the leaves of the computational tree (horizontally) until the interval size no longer applies; at which point, it either slightly enlarges or shrinks the interval size (and tolerance); it then continues traverse the leaves of the computational tree at the new interval size.[3] While mediate computations are not eliminated, their number can be mitigated by the continuity assumption.

4. Parallel, block-cutting adaptive quadrature for the GPGPU

4.1 Basic algorithm

PBCAQ is implemented in NVIDIA's Compute Unit Device Architecture (CUDA). As such, CUDA nomenclature will be used throughout. Physically, an NVIDIA GPGPU consists of some number of *streaming multiprocessors* (SM), each of which has some number of *cores*, usually 32.[4] Thus, if an NVIDIA GPGPU has 14 SMs, then it has 448 cores. Logically, the primary unit of computation in CUDA is the

[3]As will be discussed, PBCAQ works on the level of regions (of intervals). However, the continuity assumption applies just the same.

[4]At times, cores will also be referred to as *compute units*.

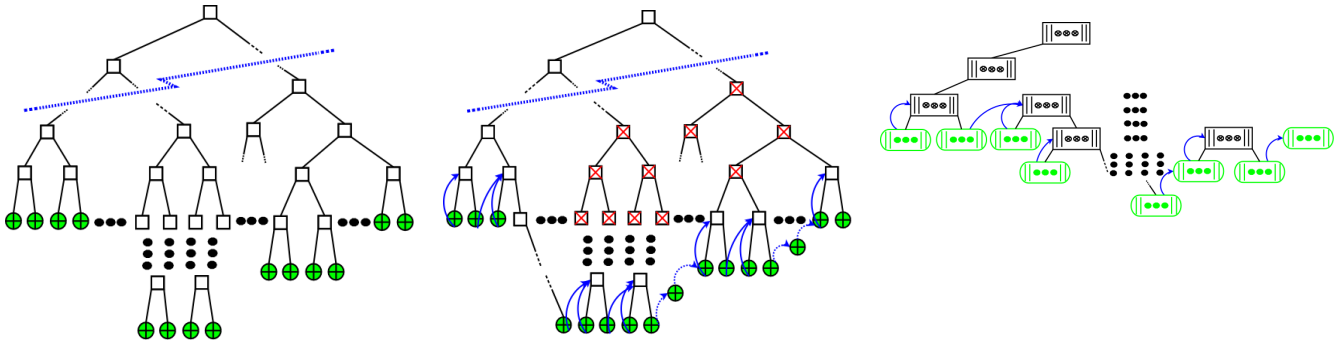


Fig. 2: Comparison of computational trees for CAQ (left), SCAQ* and PBCAQ (right). In CAQ and SCAQ, squares and circles, respectively, represent mediate and immediate computations of intervals; squares with X's represent mediate computations not performed by SCAQ. In PBCAQ, rectangles and ovals, respectively, represent mediate and immediate computations of (intervals within) regions. * Note that SCAQ is described below in Section 4.3

thread such that a single thread will be run on a single core of a single SM. Threads are organized into *blocks* and blocks are organized into *grids*. While any thread can utilize global memory on the device, threads within the same block will execute on the same SM and, thus, are able to utilize shared local memory. Finally, since the number of threads that can be launched on a GPGPU vastly outnumber the number of available cores, threads that are part of the same block are organized into *warps*, which are groups of threads that execute simultaneously on a SM.

AQ algorithms, like CAQ, often work on the level of individual intervals (and subintervals). PBCAQ, on the other hand, works by simultaneously approximating huge numbers of intervals. Let the intervals simultaneously processed on the GPGPU by PBCAQ be known as a *region*. When CAQ approximates an interval i, it compares the error for that approximation to the tolerance to determine whether it is within tolerance. When PBCAQ approximates a region r, it simultaneously approximates all intervals within r on the GPGPU; it then compares the sum of the errors for all intervals in r to the tolerance to determine whether the approximation of r is within tolerance. Thus, PBCAQ never considers whether the approximation of an individual interval of a region is within tolerance; instead, PBCAQ is only concerned with whether a region, as a whole, has been approximated within tolerance. The distinction between the individual intervals processed by CAQ and regions (of intervals) processed by PBCAQ can be seen in Figure 2.

PBCAQ begins by setting $length$, which is the length of the region being approximated, to be the entire length from $lower$ to $upper$. It approximates the first region at level $\log_2(\alpha * \beta)$ (where $\alpha * \beta = 2^x, x \geq 1$) such that α is the number of blocks executed on the GPGPU and β is the number of threads per block. It continues (depth first) to shrink $length$ by half, summing all of the errors from all of the blocks (in the region) approximated on the GPGPU, until it achieves an approximation within tolerance for the entire region. After such an approximation, it adds the region approximation to the total approximation, doubles $length$ and then, moving from $upper$ to $lower$, proceeds to approximate the adjacent region, if there is one, which will continue to be the case until it breaches $lower$.

function PBCAQ($f, lower, upper, \tau$)
  Let α be the number of thread blocks;
  Let β be the number of threads per block;
  $I \leftarrow 0.0$; $length \leftarrow upper - lower$;
  Let S and E be apprxs and errs;
  while $upper > lower$ **do**
    ▷ Apprxs and errs are generated by thrds in next step;
    Approximate $length$ on GPGPU using $(\alpha * \beta)$ thrds; barrier;
    Reduce apprxs and errs, S and E, on GPGPU; barrier;
    Transfer apprxs and errs, S and E, to host mem;
    ▷ Compare error for region to tolerance
    if $E \leq \tau$ **then**
      $I \leftarrow I + S$;
      $upper \leftarrow lower$;
      $length \leftarrow 2 * length$; $\tau \leftarrow \tau * 2$; } ▷ Expansion step
    else
      $length \leftarrow length/2$;
      $\tau \leftarrow \tau/2$;
    end if
    $lower \leftarrow max(upper - length, lower)$;
  end while
  return I;
end function

Fig. 3: Parallel, block-cutting adaptive quadrature (PBCAQ)

Considering the approximation of regions by PBCAQ, let the region being approximated be r_m. PBCAQ (displayed in Figure 3) breaks r_m into $\alpha*\beta$ intervals, where α is the number of thread blocks and β is the number of threads per block. It submits the region to the GPGPU where each thread is assigned its own interval to approximate. Each thread will find both an approximation and an error for its assigned interval.

After each of the $\alpha * \beta$ intervals are approximated within the region, the approximations and the corresponding errors