Parallel Interval Newton Method on CUDA

Philip-Daniel Beck and Marco Nehmeier

Institute of Computer Science, University of Würzburg
Am Hubland, D 97074 Würzburg, Germany
nehmeier@informatik.uni-wuerzburg.de

Abstract. In this paper we discuss a parallel variant of the interval Newton method for root finding of non linear continuously differentiable functions on the CUDA architecture. For this purpose we have investigated different dynamic load balancing methods to get an evenly balanced workload during the parallel computation. We tested the functionality, correctness and performance of our implementation in different case studies and compared it with other implementations.

Keywords: Interval arithmetic, Interval Newton method, Parallel computing, Load balancing, CUDA, GPGPU

1 Introduction

In the last years the GPU has come into focus for general purpose computing by the introduction of CUDA (Compute Unified Device Architecture) [12] as well as the open standard OpenCL (Open Computing Language) [9] to exploit the tremendous performance of highly parallel graphic devices.

Both technologies, CUDA as well as OpenCL, have a huge impact onto the world of scientific computing and therefore it is a matter of importance for the interval community to offer their algorithms and methods on these systems. One of the famous algorithms using interval arithmetic is the interval Newton method for which a parallel implementation on CUDA is discussed in this paper.

2 Interval Arithmetic

Interval arithmetic is set arithmetic working on intervals defined as connected, closed and not necessarily bounded subsets of the reals

$$X = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \tag{1}$$

where $\underline{x} = -\infty$ and $\overline{x} = +\infty$ are allowed. The set of all possible intervals together with the empty set is denoted $\overline{\mathbb{IR}}$. The basic arithmetic operations on intervals

Preprint. The final publication is available at link.springer.com
http://dx.doi.org/10.1007/978-3-642-36803-5_34

are based on powerset operations:

$$X \circ Y = [\min_{x \in X, y \in Y}(\nabla(x \circ y)), \max_{x \in X, y \in Y}(\triangle(x \circ y))] \tag{2}$$

With floating point numbers the value of the lower bound is rounded toward $-\infty$ (symbol ∇) and the upper bound is rounded toward $+\infty$ (symbol $\triangle$) to include all possible results of the powerset operation on the real numbers[1]. Continuous functions could be defined in a similar manner [8].

The enclosure of all real results of a basic operation or a function is the fundamental property of interval arithmetic and is called *inclusion property.*

Definition 1 (Inclusion property). *If the corresponding interval extension* $F : \overline{\mathbb{IR}} \to \overline{\mathbb{IR}}$ *to a real (continuous) function* $f : \mathbb{R} \to \mathbb{R}$ *is defined on an interval* X *it follows:*

$$f(X) \subseteq F(X)$$

2.1 Interval Newton Method

The interval Newton method in Algorithm 1 is one of the famous applications based upon interval arithmetic. Likewise the well known Newton method, it is an iterative method to compute the roots of a function. But it has the benefit that it can — for some functions — compute *all* zeros of a non linear continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ in the start interval X_0 with guaranteed error bounds and it can provide information about the existence and uniqueness of the roots[2] [5].

The iterative computation of the enclosing interval of a root is defined as

$$X_{n+1} := X_n \cap N(X_n), \qquad n = 0, 1, 2, \ldots \tag{3}$$

$$N(X_n) := \mathrm{mid}(X_n) - \frac{F(\mathrm{mid}(X_n))}{F'(X_n)} \tag{4}$$

where F and F' are the corresponding interval extension and derivative to the function f. In this paper we use an extended form of the interval Newton method [5] which will return two distinct intervals for the case $0 \in F'(X_n)$ in (4) for which the computation is performed recursively, see Algorithm 1 line 7 et seq. This has the advantage that we can use the algorithm for non monotonic functions.

With the mean value theorem it can be easily shown that each root of the function f in X_n also lies in X_{n+1} [5] and therefore we have a sequence of nested intervals[3] which means that the algorithm will always converge.

[1] Note that monotonicity properties could be used to define the result of an interval operation or function only using the bounds of the input intervals.

[2] Note that for the reason of readability the check for the uniqueness of the roots is not included in Algorithm 1 but is included in our implementation. Basically it is a test if $N(X_n)$ is an inner inclusion of X_n, see [5] for more details.

[3] In the case of $X_{n+1} = X_n$ the interval is bisected and the computation is performed on both distinct intervals recursively, see Algorithm 1 line 13 et seq.

Algorithm 1: INewton

```
   Input:
   F : function
   X : interval
   ϵ : accuracy of enclosing intervals
   zeros : list of enclosing intervals
   Output: zeros
1  begin
2      /* Use Definition 1 to check possible existence of a root      */
3      if 0 ∉ F(X) then
4          return zeros;
5      c ← mid(X);
6      /* Newton step with extended division; formula (3) and (4)     */
7      (N_1, N_2) ← F(c)/F'(X);
8      N_1 ← c − N_1;
9      N_2 ← c − N_2;
10     X_1 ← X ∩ N_1;
11     X_2 ← X ∩ N_2;
12     /* Bisection in case of no improvement                          */
13     if X_1 = X then
14         X_1 ← [x_lower, c];
15         X_2 ← [c, x_upper];
16     foreach i = 1, 2 do
17         /* No root                                                  */
18         if X_i = ∅ then
19             continue;
20         /* Suitable enclosure of a root                             */
21         if width(X_i) < ϵ then
22             zeros.append(X_i);
23         /* Recursive call                                           */
24         else
25             INewton(F, X_i, ϵ, zeros);
26     return zeros;
```

Additionally a verification step could be performed after the computation of Algorithm 1 to check the uniqueness of the enclosed roots, see [5] for details.

3 Dynamic Load Balancing

The main challenge in the parallelization of the interval Newton method is a good utilization of all parallel processes during the computation. As described in Sec. 2, we can have a bisection of the workload for the cases $X_{n+1} = X_n$ in (3) or $0 \in F'(X_n)$ in (4). On the other hand, a thread will become idle in the case $X_{n+1} = \emptyset$ which means that no root exists in X_n. Hence, static load balancing is probably not the best way to precompute a good distribution of the workload.

Therefore we have investigated an implementation of the parallel interval Newton method on CUDA with four different dynamic load balancing methods to get an evenly balanced workload during the computation.

Blocking Queue [3] is a dynamic load balancing method which uses one queue in the global memory for the distribution of the tasks. The access to the queue is organized by mutual exclusion using the atomic operations `atomicCAS` and `atomicExch` on an `int`-value to realize the lock and unlock functionality, see [6] for more details.

```
__device__ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
}

__device__ void unlock( void ) {
    atomicExch( mutex, 0 );
}
```

Listing 1. Lock and unlock functionality

Task Stealing [1, 4] is a lock-free dynamic load balancing method which uses a unique global queue for each CUDA thread block [12]. In the case of an empty queue, the thread block will steal tasks from other thread blocks to avoid idleness. To ensure a consistent dequeue functionality with atomic operations, the CUDA function `__threadfence_system` is used during the enqueue.

```
__device__ void pushBottom(T const & v) {
    int localBot = bot;
    buf[localBot] = v;
    __threadfence_system();
    localBot += 1;
    bot = localBot;
    return;
}
```

Listing 2. Enqueue functionality

Distributed Stacks is a lock-free load balancing method using local stacks in shared memory for each thread block and is almost similar to distributed queuing [13]. Dynamic load balancing is only realized between threads of a thread block. In the case of storing an element onto the stack, the atomic operation `atomicAdd` is used to increase the stack pointer. Reading from the stack is realized simultaneously without atomic operations using the thread id `threadIdx.x` to access the elements of the stack. The workload for the thread blocks is statically distributed at the beginning of the parallel computation.

Static Task List [3] is a lock-free method which uses two arrays, the in-array and the out-array, for the dynamic load balancing. In an iteration the in-array is a read-only data-structure containing the tasks. For each task in the in-array a thread is started writing their results in the out-array. After each iteration the in-array and the out-array are swapped[4], see Fig. 1 for more details.

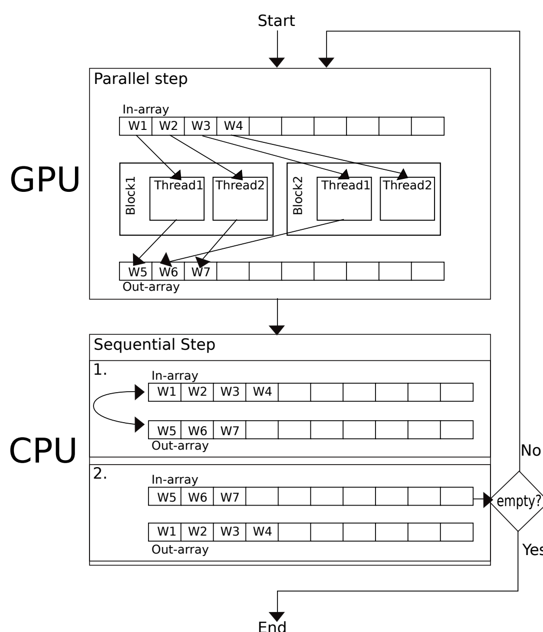


Fig. 1. Static task list

[4] This means that the kernel is called with swapped pointers for the in-array and the out-array.

4 Implementation

In our parallel implementation of the interval Newton method on CUDA it was first of all necessary to have interval arithmetic on CUDA. For this purpose we have implemented the required data structures, interval operations, and standard functions in CUDA C.

For the CUDA C implementation of the interval Newton method we simulated the recursive Algoritm 1 by an iterative CUDA kernel which uses two different concepts depending on the different load balancing methods.

Static task list is the only one of our four used load balancing methods which almost meets the common concept of lightweight threads in CUDA. In our case, this means that the threads are created at the start of the kernel and then only compute one iteration of the parallel interval Newton method. After this iteration all threads are terminated and a new kernel with new threads is started for the next iteration, see Fig. 1 for more details.

For the other three load balancing methods we use so called persistent threads [13] which means that all required threads are started at the beginning of the interval Newton method and keep alive until the algorithm terminates.

The initialization and execution of the CUDA kernels is wrapped in a C++ function which handle the data transfer between the host and the GPU. The specification of the function, which should be analyzed, and their corresponding derivative is done by functors.

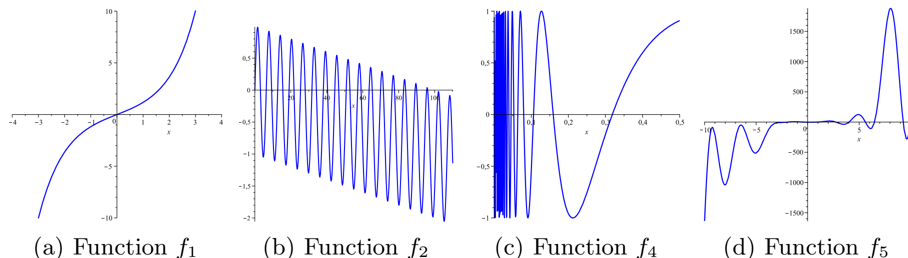
5 Performance Tests

We tested our implementation on a Nvidia Tesla C2070 GPU with CUDA compute capability 2.0 hosted on a Linux Debian 64 Bit System with an Intel Xeon E5504 2.0 GHz CPU and 8 GB Memory.

For all four different implementations we have analyzed the performance for the following functions

$$
\begin{aligned}
f_1(x) &= \sinh x \\
f_2(x) &= \sin x - \frac{x}{100} \\
f_3(x) &= \sin x - \frac{x}{10000} \\
f_4(x) &= \sin \frac{1}{x} \\
f_5(x) &= (3x^3 - 5x + 2) \cdot \sin^2 x + (x^3 + 5 \cdot x) \cdot \sin x - 2x^2 - x - 2 \\
f_6(x) &= x^{14} - 539.25 \cdot x^{12} + 60033.8 \cdot x^{10} - 1.77574e^6 \cdot x^8 \\
&\quad + 1.70316e^7 \cdot x^6 - 5.50378e^7 \cdot x^4 + 4.87225e^7 \cdot x^2 - 9.0e^6
\end{aligned}
$$

with different thread and block configurations. Additionally we have compared our implementations with a parallel bisection algorithm on CUDA as well as with filib++ [10] and Boost [2] implementations on the CPU.



(a) Function f_1 (b) Function f_2 (c) Function f_4 (d) Function f_5

Fig. 2. Sketches of some used functions for the performance tests

Prior to the performance tests it was necessary to analyze the best block-grid-ratio of the four different implementations on a GPU to ensure the best possible performance of each implementation. This means that we have measured the runtime with a variable number of threads per block as well as a variable number of blocks per grid on the GPU, see [7].

For our measurements we used configurations with 1 up to 56 blocks using 32, 64, or 128 threads. Thereby the number of 56 blocks is an upper bound which could be computed out of the used memory of our implementations. Table 1 shows the used memory information provided by the `nvcc` compiler using option `-ptxas-options=-v`. Note that for static task list there is no shared memory used due to the fact that there is no communication between the threads.

Table 1. Used memory

Method	Register per thread	Shared memory per Block
BlockingQueue	63	8240 Byte
TaskStealing	63	8240 Byte
DistributedStacks	63	32784 Byte
StaticTaskList	59	0 Byte

For our runtime tests we used the maximum of 128 threads per block. Additionally, a multiprocessor of a NVIDIA GPU with CUDA compute capability 2.0 is specified with 32768 registers and 48 KB shared memory [12]. Hence we can easily compute

$$\left\lfloor \frac{32768[registers/multiprocessor]}{128[threads/block] * 63[registers/thread]} \right\rfloor = 4[blocks/multiprocessor]$$

which leads to the upper bound of 56 blocks for a Nvidia Tesla C2070 GPU with 14 multiprocessors.

Figure 3 shows some sketches of the performed runtime tests with a variable number of blocks and Tab. 2 shows the best configurations for our test cases.

Note that for static task list we have not measured any difference between 32, 64, or 128 threads per block. Hence, we used 32 threads per block for the other performance tests. Furthermore, the number of blocks per grid is not specified for static task list in Tab. 2 due to the fact that the number of blocks depend on the current workload of each iteration and is adjusted automatically.

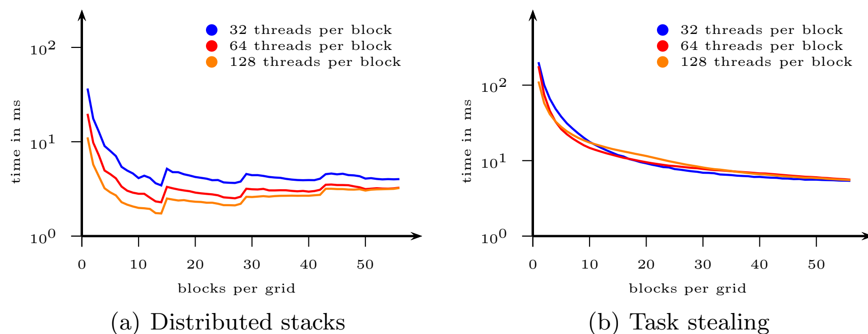


Fig. 3. Sketches of the runtime measurements for function f_3 with a variable number of blocks

Table 2. Block-grid-ratio

Method	Blocks per grid	Threads per block
BlockingQueue	14	64
TaskStealing	28	64
DistributedStacks	14	128
StaticTaskList	-	32

Figure 4 shows the average runtime of 1000 runs for our test cases with double precision and an accuracy of $\epsilon = 10^{-12}$ for the enclosing intervals. It is easily visible that the additional expenses for the computation on the GPU are not worth it for simple functions like f_1 or f_2. In these cases the GPU is outperformed by filib++ or Boost on a CPU. But for harder problems like f_3 or f_4 the GPU, especially with static task list or distributed stacks, dominates filib++ and Boost.

Additional performance tests have shown that there is no significant difference between the runtime with single or double precision, see Fig. 5. This

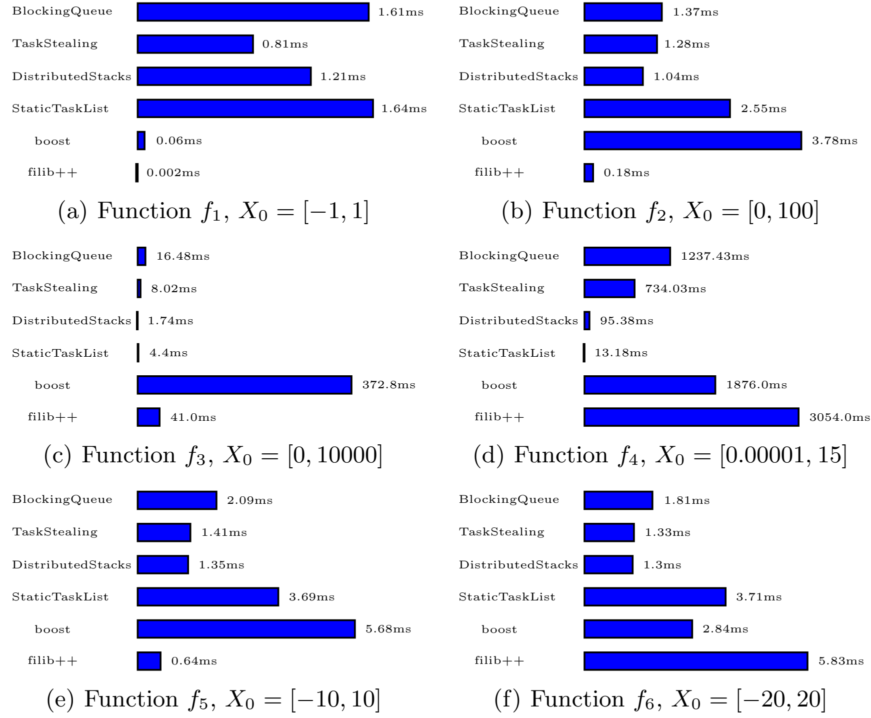
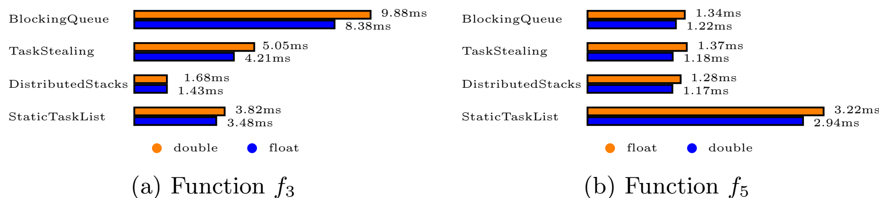


Fig. 4. Average runtime with double precision

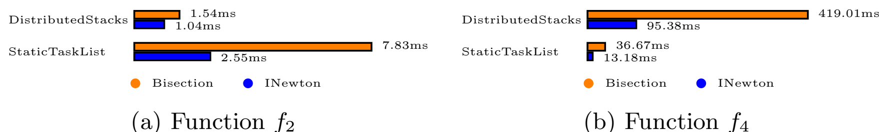
results in the assumption that our implementation is mainly limited by the load balancing and not by the interval arithmetic on the GPU.



(a) Function f_3 (b) Function f_5

Fig. 5. Runtime float vs. double

Finally we compared a bisection algorithm [5] on a GPU using the same load balancing methods with our interval Newton method. Figure 6 shows some measurements which reflect our observation that the bisection algorithm is outperformed by the interval Newton method for all our test cases.



(a) Function f_2 (b) Function f_4

Fig. 6. Runtime Bisection vs. INewton

6 Related Work

In [3] dynamic load balancing on a GPU is discussed for the task of creating an octree partitioning of a set of particles.

Furthermore, in [4] dynamic load balancing for an interval Newton method is analyzed for an implementation on a cluster of workstations using message passing [11].

7 Conclusion

In this paper we have discussed a parallel implementation of an interval Newton method on CUDA and especially different load balancing concepts to utilize the highly parallel CUDA architecture.

[5] Simply it is a branch-and-prune algortihm [8] which only uses the function value and bisection.

Performance analyzations of our approach showed promising results for some hard problems. Especially the two load balancing methods — static task list and distributed stacks — are well suited for complicated functions. Thereby distributed stacks should be preferred for functions with "evenly" distributed roots whereas static task list is more preferable for functions with "unevenly" distributed roots.

Further investigations in the area of parallel interval arithmetic on the GPU as well as on multicore CPU's are planned.

References

1. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.
2. Boost Interval Arithmetic Library, November 2012. `http://www.boost.org/doc/libs/1_52_0/libs/numeric/interval/doc/interval.htm`.
3. D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.
4. C.-Y. Gau and M. A. Stadtherr. Parallel interval-newton using message passing: dynamic load balancing strategies. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 23–23, New York, NY, USA, 2001. ACM.
5. R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for Verified Computing I: Basic Numerical Problems*. Springer. Berlin, 1995.
6. E. K. Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Longman, Amsterdam, 2010.
7. E. K. Jason Sanders. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Longman, Amsterdam, 2010.
8. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 1 edition, Sept. 2001.
9. Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1.44*, June 2011.
10. M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster, and W. Krämer. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw.*, 32(2):299–324, 2006.
11. Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.
12. NVIDIA. *NVIDIA CUDA reference manual, version 3.2 Beta*, August 2010.
13. S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In M. Doggett, S. Laine, and W. Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.