Global Extrema Locator Using Interval Arithmetic

CS6235 Intermediate Project Report

Arnab Das
Sandesh Borgaonkar

Plan To Evolve The Project:

The goal of the project is to locate a global extrema for a given function in a specific interval using Interval Analysis and Branch & Bound technique (IBBA). In addition to IBBA, it uses the GPU compute capability to intermediately modify the worklist distribution (priority distribution of the intervals in queue) for the CPU, thus effectively helping the CPU thread to work on intervals with higher priorities of reaching maxima.

Note: We have taken-in the provided feedback and presenting below the proposed operation of the gpuKernel(the manner in which the work-list distribution is performed)

Basic Idea of work-list distribution: The cpu works through the interval branch and bound and keeps pushing newer intervals to the queue. However, convergence is very slow for a sequential operation of this type. We leverage here the gpu compute capability to reprioritizing the intervals in the queue thus aiding the cpu to reach better intervals earlier in the stage. The gpu kernel is invoked when the number of intervals in the queue has exceeded a threshold(this is tunable). Suppose the gpu receives a list of N intervals. For multivariate functions of M variables, each interval will have M sub-intervals along each dimension.
A single dimension(i) is chosen and it is broken into k subintervals. Now the function is computed from a random sample from rest of the intervals and one sample from each of these k sub-sub-intervals of the i'th sub-interval. The max of these computations forms the priority of the ith sub-intervals. These are done in parallel for each of the sub-intervals along the M dimensions for all the N intervals in received work-list. Thus we get M priority values for each interval. The max of these M priority values becomes the priority of the corresponding interval. The revised priorityList and intervalList is reverseSorted and the kernel terminates. The cpu thread can use this updated priorityList to get its next interval for computation.

We use Gaol interval library for cpu thread interval computations and the thrust library for operations on the gpu side(passing vectors of interval_gpu types).

Below is a CPU thread for IBBA(Interval branch and Bound)
Input : function 'f' of M dimensions(variables)
Interval Xo = [of M intervals along each of the m dimensions]
k = knob to control number of samples for gpu evaluation per interval
CPU_THRESHOLD = Knob to control Minimum number of intervals in queue to trigger gpu computation
ε_o = output epsilon(precison)
ε = input epsilon(precion)

Algorithm:

```
  temp_Queue  = Xo ; int addIntervalSize = 0;
  temp_priority_queue.push(f(midpoint(Xo));
  f_best ← Max_float;
  while Queue or Temp_Queue not empty do
        If (gpuHandleThread exists)    //-- Synchronize with the gpu thread updates
        {
              Join gpuHandleThread();  //-- Wait for the gpuThread to terminate
               Queue.clear() ; // Clear the Queue before receving the new work-list distribution
              Queue = CopyUpdated(IntervalList) from GPU
               Priority_Queue = CopyUpdated(Interval_priority_list) from GPU
        }
        Queue.push(Temp_Queue)
          Priority_Queue.push(Temp_priority_queue);
        If (temp_Q_size() = 0)   addIntervalSize += Temp_Queue.size();
        If (addIntevalSize > CPU_THRESHOLD)
                    Thread(gpuHandleThread); // Fork a gpuThread if new added intervals exceed threshold
        temp_Queue.clear();
        f_besttag = getFromSharedMemory (update from GPU kernel its best value)
        f_best  = Max(f_best, f_besttag)
        X ← Q.pop();
        [L,U] = F(X)   //--Evaluation of the interval X by the inclusion function


        If (U < f_best or Width(X) ≤ Ɛ or Width(F(X)) ≤ Ɛ_o )
              get_next_element from Queue    //-- Discard current interval and get new interval
        else
              [X_1, X_2] = split(X)   //-- split along the largest dimension
        for i ϵ [1,2] do
              e_i  ← f(midpoint(X_i)
              if e_i > f_best then
                    f_best ← e_i
                    X_best ← X_i
                end if
              temp_Queue.push(X_i)    //-- Fill the queue with newly generated sub-intervals
        end
    end
end
```

<u>PseudoCode for the gpu thread handler</u>

Input - IntervalList (list of N unsorted intervals with each interval having M subintervals along its M dimensions)

PriorityList (list of current priority values of the intervals in the corresponding index of IntervalList)

Dimension = M

Division of threads and blocks: Each interval is assigned to a block. Thus there will be N blocks. Each block gets K*M threads for the evaluation across k samples points across each of the M sub-intervals.

Algorithm:

```
   gpuHandleThread( intervalList, PriorityList, dimension) {
            if( K*M > 512)  exit(-1) ;
            else {
                dim3  dimBlock(K,M);
                dim3  dimGrid(N);
                gpuKernel<<<dimGrid, dimBlock>>>( intervalList, PriorityList, dimension)
                cudaDeviceSynchronize();
             }
}
```

<u>GPU-Kernel</u>

```
__global__ gpuKernel( intervalList, priorityList, dimension) {
     BLOCK_SM2[M];  //--shared memory
    BLOCK_SM1[K][M]; //--shared memory
    BLOCK_SM2[tiy] = intervalList[blockIdx.x*blockDim.x + tiy] ;
       __syncthreads() ;
    BLOCK_SM1[tix][tiy] = (BLOCK_SM2[tiy]/K) * (tix + 1) ;
       __syncthreads() ;
    BLOCK_SM2[tiy] = randomSample(BLOCK_SM2[tiy]) ;
       __syncthreads();

    BLOCK_SM[tix][tiy] = GPU_Function_Compute( BLOCK_SM2.replace(tiy,
BLOCK_SM1[tix][tiy])) ;
//---- The returned computed function value is stored in the the same location in the shared memory
BLOCK_SM2[tiy] = PrefixMax(BLOCK_SM1[tix = 0 → K][tiy] ;
priorityList[blockIdx.x] = PrefixMax(BLOCK_SM2) ;

__syncthreads();

ParallelReverseSort(intervalList, PriorityList);
//--- This sorts the updated priorityList in reverse order(highest priority at the front of the queue) and also rearranges the intervals
in the intervalList corresponding to their priority distribution.
}
```

Show At Least One Thing Working:

1. The IBBA thread has been coded and tested with interval analysis using the Gaol interval library.
2. The translation between the Gaol library data structures and interval_gpu data structures has been implemented. We are using the thrust library's device_vector type to allocate memory on the gpu. The reason being able to pass along vector type intervals to the gpu . Every point of synchronization has the overhead of data-structure translation due to non-availability of compatible gpu-interval libraries with Gaol(most suited for our case of high precision)

3. The synchronization of the GPU thread with the cpu thread is mostly complete.
4. Next, we need to start implementation of the gpuKernel and testing of the overall system.

How The Work Is Being Divided Among Team Members:

Description	Member Responsible
1.Code Development :	
1. Pseudo Code	Arnab, Sandesh
2. Implementation	Arnab
3. Testing	Sandesh
2.Scripting :	
1. Development of a generic Perl/Python Script	Sandesh
2. Testing with various Benchmarks	Sandesh
3.Result Gathering and Analytics	Arnab, Sandesh
4.Documentation	Arnab

Project Complexity

The complex elements of the project are the following:

- Development of a pseudo code.
- Implementation of the pseudo code.
- Rigorous testing of the code for corner cases involving concurrency and synchronization in addition to correct functionality.
- Developing a generic script to embed the input function description and interval range in some parts of the C implementation.
- Testing of the script for several benchmarks(against gelpia with serial solver + aided with Evolutionary algorithm)
- Gathering data on run times of functions and comparing results against the CPU based implementation of the same code.

References:

[1]Finding and Proving the optimum:Cooperative Stochastic and Deterministic Search – Jean-Marc Alliot, Nicolas Durand, David Gianazza, Jean-Baptiste Gotteland
[2]Introduction to interval analysis – Ole Caprani, Kaj Madsen, Hans Bruun Nielson
[3]Efficient Search for inputs causing high floating point errors – Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Alexey Solovyev
[4]Interval Analysis – R.E.Moore
[5]A Scalabale Heterogeneous Parallelization Framework for Iterative Local Searches – Martin Burtscher, Hassan Rabeti