IMPLEMENTATION OF SORTING ALGORITHMS ON GPU

Karthik M 15CO221, Kapil Vashist 15CO123

NATIONAL INSTITUE OF TECHNOLOLGY KARNATAKA SURATHKAL

Table of Contents

PREFACE

This document is written by Karthik M and Kapil Vashist as part of their Heterogenous Parallel Computing course project on the topic *Analysis of Various GPU Sorting Algorithms.*

The contribution by each of the authors is given below:

Karthik M

Reduction
All four algorithms

Scan
All four algorithms

Sorting
Bitonic sort

CUDPP radix sort

Satish Radix sort

Satish Merge sort

Hybrid Vector Merge sort

Kapil Vashist

Sorting
Sample sort

Quick sort

Link for the codes
https://github.com/mkarthik2597/GPU-Sorting-Algorithms

PARALLEL REDUCTION

Source: *Optimising Parallel Reduction in CUDA, Mark Harris, NVIDIA Developer Technology*

1. Parallel Reduction with Divergent Branching

Algorithm

A thread is allocated to every element in the input array. During the first pass, only the even threads are active in calculating a partial sum of its own value and its adjacent value in the input array. In the next iteration, every fourth element will calculate a partial sum and after that every eighth thread. This goes on till the entire sum of the array elements is stored in the first array element.

Code

```
__global__ void reduction(int* input_d)
{
    /* Has the least performance because of control divergence */
    int tx=threadIdx.x,bx=blockIdx.x;
    int inx=bx*blockDim.x+tx;

    extern __shared__ int partialSum[];
    partialSum[tx]=input_d[inx];
    __syncthreads();

    for(int s=1;s<blockDim.x;s<<=1)
    {
        if(tx%(2*s)==0)
        partialSum[tx]+=partialSum[tx+s];

        __syncthreads();
    }

    if(tx==0)
    input_d[inx]=partialSum[0];
}
```

2. Parallel Reduction with Strided Index and Non-Divergent Branching

Algorithm

Each element is allocated a thread. In the first iteration, the threads of the first half of the block calculate the partial sum, in the next iteration the first quarter threads, the third iteration the first one-eighth and so on. This is achieved using reversed loop indexing

Code

```
__global__ void reduction(int* input_d)
{
    int tx=threadIdx.x,bx=blockIdx.x;
    int inx=bx*blockDim.x+tx;

    extern __shared__ int partialSum[];
    partialSum[tx]=input_d[inx];
    __syncthreads();

    /* Strided index and Non-divergent branching */
    for(int s=blockDim.x/2;s>0;s>>=1)
    {
        if(tx<s)
        partialSum[tx]+=partialSum[tx+s];

        __syncthreads();
    }

    if(tx==0)
    input_d[inx]=partialSum[0];
}
```

3. Parallel Reduction by Avoiding Idle Threads

Algorithm

Half of the threads were idle in the first loop of iteration. The number of threads are halved and each thread performs a single add before the start of the loop.

Code

```
__global__ void reduction(int* input_d)
{
    int tx=threadIdx.x,bx=blockIdx.x;
    int inx=2*bx*blockDim.x+tx;

    extern __shared__ int partialSum[];
    /* There are no idle threads in this reduction */
    partialSum[tx]=input_d[inx]+input_d[inx+blockDim.x];
    __syncthreads();

    for(int s=blockDim.x/2;s>0;s>>=1)
    {
        if(tx<s)
        partialSum[tx]+=partialSum[tx+s];

        __syncthreads();
    }

    if(tx==0)
    input_d[inx]=partialSum[0];
}
```

4. Parallel Reduction by Unrolling the Last Warp

Algorithm

In the above algorithm, control divergence occurs when the number of threads involved in calculating reduction becomes less than 32 (warp size). To improve performance, the last warp is unrolled.

Code

```
for(int s=blockDim.x/2;s>=32;s>>=1)
{
    if(tx<s)
    partialSum[tx]+=partialSum[tx+s];

    __syncthreads();
}

if(tx<32)
{
    partialSum[tx]+=partialSum[tx+16];
    __syncthreads();
    partialSum[tx]+=partialSum[tx+8];
    __syncthreads();
    partialSum[tx]+=partialSum[tx+4];
    __syncthreads();
    partialSum[tx]+=partialSum[tx+2];
    __syncthreads();
    partialSum[tx]+=partialSum[tx+1];
    __syncthreads();
}
```

Performance Comparison

PARALLEL SCAN

Source: *Parallel Prefix Sum (Scan) with CUDA, Mark Harris, NVIDIA Developer Technology*

1. **Naïve Parallel Scan**

Algorithm

This scan algorithm works only for block sizes which are equal to warp size. The scan is performed in place on the array. The time complexity is *O(nlgn)* and hence is not work-efficient

Code

```
__global__ void scan(int* input_d)
{
    int tx=threadIdx.x;

    extern __shared__ int partialSum[];
    partialSum[tx]=input_d[tx];
    __syncthreads();

    for(int s=1;s<blockDim.x;s<<=1)
    {
        if(tx>=s)
        partialSum[tx]+=partialSum[tx-s];
    }

    input_d[tx]=partialSum[tx];
}
```

2. Naïve Parallel Scan with Double Buffering

Algorithm

Since the algorithm above was in place and hence limited to input size equal to warp size, double buffering eliminates this problem. However, this algorithm will run only on a single block of threads. The time complexity is *O(nlgn)* and hence is not work-efficient.

Code

```
    for(int s=1;s<blockDim.x;s<<=1)
    {
        if(tx>=s)

partialSum[outputRow*BLOCK_SIZE+tx]=partialSum[inputRow*BLOCK_SIZE+tx]+partia
lSum[inputRow*BLOCK_SIZE+tx-s];
        else

partialSum[outputRow*BLOCK_SIZE+tx]=partialSum[inputRow*BLOCK_SIZE+tx];

        __syncthreads();

        outputRow=1-outputRow;
        inputRow=1-inputRow;
    }
```

3. Work Efficient Intra Block Scan

Algorithm

This algorithm performs the scan operation in *O(n)* time. Balanced trees are used on the input data. The algorithm consists of two phases: an upsweep phase of reduction and a down sweep phase building the scan

Code

```
/* Upsweep phase (Reduction)*/
for(int s=1;s<blockDim.x;s<<=1)
{
    if((tx+1)%(2*s)==0)
    partialSum[tx]+=partialSum[tx-s];

    __syncthreads();
}

if(tx==BLOCK_SIZE-1)
partialSum[tx]=0;

/* Downsweep phase*/
int temp;
for(int s=blockDim.x/2;s>0;s>>=1)
{
    if((tx+1)%(2*s)==0)
    {
        temp=partialSum[tx];
        partialSum[tx]+=partialSum[tx-s];
        partialSum[tx-s]=temp;
    }

    __syncthreads();
}
```

4. Performing Scan on Arbitrary Array Sizes

Algorithm

The input array is divided into blocks that each can be scanned by a single thread block, scan the blocks and write the total sum of each block to another array of block sums. The block sums are again scanned generating an array of block increments that are added to all elements in their respective blocks.

Code

```
__global__ void blocksumScan(int* input_d, int* auxilliary_d)
{
    int tx=threadIdx.x;

    extern __shared__ int partialSum[];
    partialSum[tx]=auxilliary_d[tx];
    __syncthreads();

    /* Upsweep phase*/
    for(int s=1;s<blockDim.x;s<<=1)
    {
        if((tx+1)%(2*s)==0)
        partialSum[tx]+=partialSum[tx-s];

        __syncthreads();
    }

    if(tx==blockDim.x-1)
    partialSum[tx]=0;

    /* Downsweep phase*/
    int temp;
    for(int s=blockDim.x/2;s>0;s>>=1)
    {
        if((tx+1)%(2*s)==0)
        {
            temp=partialSum[tx];
            partialSum[tx]+=partialSum[tx-s];
            partialSum[tx-s]=temp;
        }

        __syncthreads();
    }

    if(tx!=0)
    {
        for(int i=0;i<BLOCK_SIZE;i++)
        input_d[(tx)*BLOCK_SIZE+i]+=partialSum[tx];
    }
}
```

Sorting on the GPU

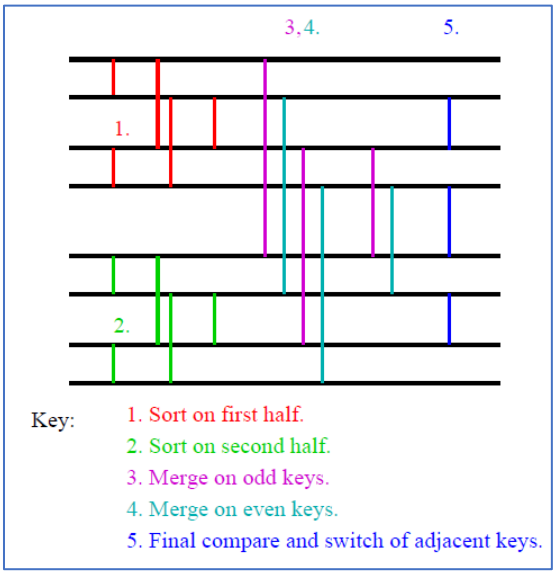
1. Bitonic Sort

Source: *Fast in-place sorting in CUDA based on bitonic sort, Hagan Peters et al.*

Algorithm

This is a classical parallel algorithm for sorting. It sorts in time $O(n\lg^2 n)$. A sequence is called Bitonic if it is first increasing, then decreasing. We start by forming 4-element bitonic sequences from consecutive 2-element sequence. We then concatenate the two pairs to form a 4 element bitonic sequence. Next, we take two 4 element bitonic sequences, sorting one in ascending order, the other in descending order (using the Bitonic Sort), and so on, until we obtain the bitonic sequence.

A sorting network is a fixed collection of comparison-switches, so that all comparisons and switches are between keys at locations that have been specified from the beginning. The Bitonic Sort is a sorting network and can be schematically represented as shown below:



Code

```
__global__ void BitonicSort(int* arr_d,int j, int k)
{
    int tx=threadIdx.x, bx=blockIdx.x, i=bx*blockDim.x+tx;
    int ixj=i^j;

    if(i<ixj)
    {   // Sort ascending
        if((i&k)==0)
        {
            if(arr_d[i]>arr_d[ixj])
            {
                int temp=arr_d[i];
                arr_d[i]=arr_d[ixj];
                arr_d[ixj]=temp;
            }
        }
        // Sort descending
        else
        {
            if(arr_d[i]<arr_d[ixj])
            {
                int temp=arr_d[i];
                arr_d[i]=arr_d[ixj];
                arr_d[ixj]=temp;
            }
        }
    }

}
```

2. CUDPP Radix Sort

Source: *Designing Efficient Sorting Algorithms for Manycore GPUs, Satish et al.*

Algorithm

Let the input be an array of b-bit keys. The radix sort version implemented in the CUDA Data Parallel Primitive library (CUDPP) processes the keys 1-bit at a time from the least significant bit. For each bit, a histogram is prepared in the GPU. The histograms are then processed to determine the rank of each key in the output array. After b such passes, the output array becomes sorted.

Code

```
__global__ void RadixSort(int* array_d, int exp, int* device_histo)
{
    /* Histogram Calculation*/
    int tx=threadIdx.x,bx=blockIdx.x;
    int inx=bx*blockDim.x+tx;
    if(inx==0)
    {
        device_histo[0]=0;
        device_histo[1]=0;
    }

    __shared__ int shared_histo[2];
    if(tx==0)
    {
        shared_histo[0]=0;
        shared_histo[1]=0;
    }
    __syncthreads();

    atomicAdd(&shared_histo[digit(array_d[inx],exp)],1);
    __syncthreads();

    if(tx==0)
    {
        atomicAdd(&device_histo[0],shared_histo[0]);
        atomicAdd(&device_histo[1],shared_histo[1]);
    }
}
```

3. Satish Radix Sort

Source: *Designing Efficient Sorting Algorithms for Manycore GPUs, Satish et al.*

Algorithm

The CUDPP radix sort is not efficient when the arrays are in DRAM. For 32-bit keys, it will perform scatter operations that reorder the entire sequence. To reduce the number of global memory scatters, the digit size is increased to more than 1. Data blocking is done to maximize the coherence of scatters. Each block of data is transferred on to the shared memory and local sorting is done. This converts scattered writes to global memory into scattered writes to on-chip memory. The local sorting is done using local histograms. The local histograms are combined to form a global histogram used for global sorting.

Code

```
__global__ void RadixSort(int* array_d, int exp, int* device_histo,int*
BlockWiseHistograms_d)
{
    int tx=threadIdx.x,bx=blockIdx.x;
    int inx=bx*blockDim.x+tx;

    extern __shared__ int shared_histo[];

    if(inx==0)
    {
        for(int i=0;i<HISTO_SIZE;i++)
        device_histo[i]=0;
    }

    if(tx==0)
    {
        for(int i=0;i<HISTO_SIZE;i++)
        shared_histo[i]=0;
    }
    __syncthreads();

    atomicAdd(&shared_histo[digit(array_d[inx],exp)],1);
    __syncthreads();

    if(tx==0)
    {
        for(int i=0;i<HISTO_SIZE;i++)
        {
            atomicAdd(&device_histo[i],shared_histo[i]);
            BlockWiseHistograms_d[bx*HISTO_SIZE+i]=shared_histo[i];
        }
    }
}
```

4. Satish Merge Sort

Source: *Designing Efficient Sorting Algorithms for Manycore GPUs, Satish et al.*
Advanced Parallel Algorithms, Lecture notes by Prof. Nodari Sitchinava

Algorithm

The input array is divided into blocks of elements, each of which will be loaded on the shared memory. The shared memory elements are sorted internally using bitonic sort. To simplify the parallel merge procedure, it is assumed that the arrays do not contain any duplicate elements. The sorted blocks are merged pairwise parallelly. If two sorted arrays A and B are to be merged, each thread is allotted one element and a binary

search is done to find the rank of its element in the other array. The value of this rank combined with the rank in its own array will give the index to which the element in the output array belongs to.

Code

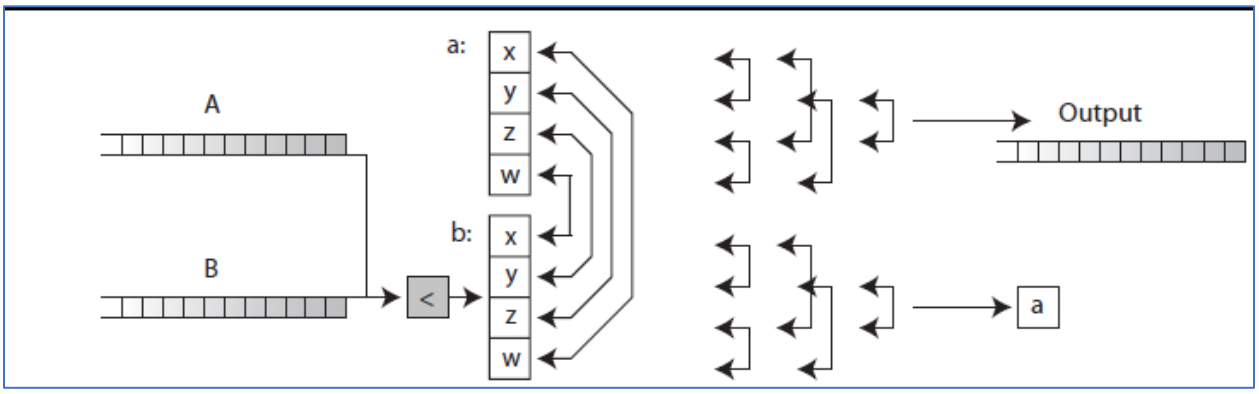
```
__global__ void MergeSort(int* arr_d, int* result_d)
{
    int tx=threadIdx.x,bx=blockIdx.x,key=VALUE(arr_d,bx,tx);

    /* partner is the block where the parallel binary search should take
place */
    int partner=bx^1;
    int rank=0;

    /* Each thread performs a binary search in its partner block*/
    int low=0,high=blockDim.x-1;

    while(low<=high)
    {
        int mid=(low+high)/2;

        if(VALUE(arr_d,partner,mid)<key)
        {
            if(mid+1<blockDim.x && VALUE(arr_d,partner,mid+1)<key)
            low=mid+1;

            else
            {
                rank=mid+1;
                break;
            }
        }

        else
        {
            if(mid-1>=0 && VALUE(arr_d,partner,mid-1)>key)
            high=mid-1;

            else
            {
                rank=mid;
                break;
            }
        }
    }

    rank+=tx;
    if(bx<partner)
    VALUE(result_d,bx,rank)=key;
    else
    VALUE(result_d,partner,rank)=key;
}
```

5. Hybrid Vector Merge Sort

Source: *Fast Parallel GPU Sorting Using a Hybrid Algorithm, Erik Sintorn, Ulf Assarsson*

Algorithm

The input listed is split into buckets of equal length using pivot points based on bucket sort. For simplicity in choosing the perfect pivot points, the input elements are ensured to be distinct. Each bucket is then divided into vectors of 4 elements, each of which is sorted using bitonic sort. The vector arrays are then pairwise merged using a custom vector merge sort. Sorting on an array of 4 element vectors improves the speed of memory reads 4 times compared to merge sorting on single floats.



Code

```
__global__ void MergeSort(int* arr_d, int* result_d, int listSize)
{
    int index=blockIdx.x*BLOCKSIZE+2*listSize*threadIdx.x;
    int vec[2][4];

    int a=0,b=1;

    int a_idx=index,b_idx=index+listSize;

    Copy(vec[a],0,arr_d,a_idx);
    Copy(vec[b],0,arr_d,b_idx);
    a_idx+=4;
    b_idx+=4;

    int i=0;
    while(1)
    {
        VectorMerge(vec[a],vec[b]);
        InternalSort(vec[a]);
        InternalSort(vec[b]);

        Copy(result_d,index+i,vec[a],0);
        i+=4;

        Copy(vec[a],0,vec[b],0);

        if(a_idx!=index+listSize && b_idx!=index+2*listSize)
        {
            if(arr_d[a_idx]<arr_d[b_idx])
            {
                Copy(vec[b],0,arr_d,a_idx);
                a_idx+=4;
            }
            else
            {
                Copy(vec[b],0,arr_d,b_idx);
                b_idx+=4;
            }
        }
        else if(a_idx==index+listSize && b_idx!=index+2*listSize)
        {
            Copy(vec[b],0,arr_d,b_idx);
            b_idx+=4;
        }
        else if(b_idx==index+2*listSize && a_idx!=index+listSize)
        {
            Copy(vec[b],0,arr_d,a_idx);
            a_idx+=4;
        }
        else
        {
            break;
        }
    }

    Copy(result_d,index+i,vec[a],0);
}
```

6. Sample Sort

Source:

http://www.ipdps.org/ipdps2010/ipdps2010-slides/session-05/gpusample.pdf
https://arxiv.org/pdf/0909.5649.pdf

Algorithm

Code

```
tid = threadIdx.x;
//Sorting the bucket using Parallel Bubble Sort
for(phase = 0; phase < bucketLength; phase ++) {
    if(phase % 2 == 0) {
        while((tid < bucketLength) && (tid % 2 == 0)) {
            if(localBucket[tid] > localBucket[tid +1]) {
                temp = localBucket[tid];
                localBucket[tid] = localBucket[tid + 1];
                localBucket[tid + 1] = temp;
            }
            tid += offset;
        }
    }
    else {
        while((tid < bucketLength - 1) && (tid %2 != 0)) {
            if(localBucket[tid] > localBucket[tid + 1]) {
                temp = localBucket[tid];
                localBucket[tid] = localBucket[tid + 1];
                localBucket[tid + 1] = temp;
            }
            tid += offset;
        }
    }
}
tid = threadIdx.x;
while(tid < bucketLength) {
    outData[(blockIdx.x * bucketLength) + tid] = localBucket[tid];
    tid += offset;
```

7. Quick Sort

Source:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.156.9622&rep=rep1&type=pdf

Algorithm

GPU-Quicksort is designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and interthread synchronization needed. It achieves this by:

(i) using a two-pass design to keep the inter-thread synchronization low,
(ii) coalescing read operations and constraining threads so that memory accesses

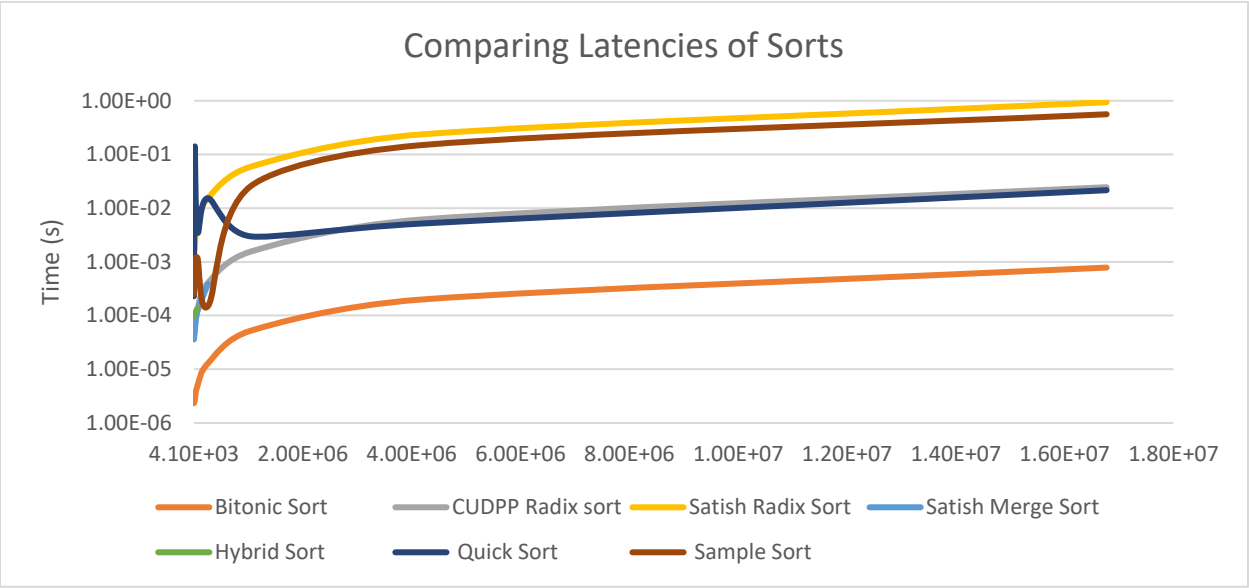
are kept to a minimum. It can also take advantage of the atomic synchronization primitives found on newer hardware to, when available, further improve its performance.

Code

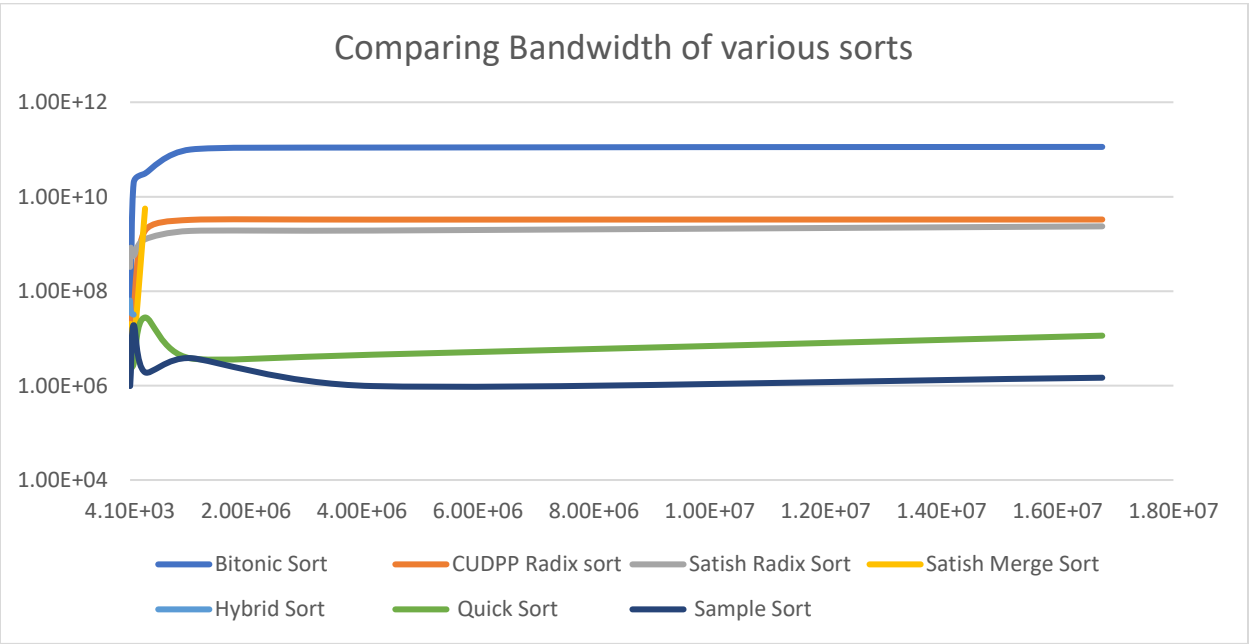
```
void quickSortIterative (int arr[], int l, int h)
{
    int lstack[ h - l + 1 ], hstack[ h - l + 1];

    int top = -1, *d_d, *d_l, *d_h;

    lstack[ ++top ] = l;
    hstack[ top ] = h;

    cudaMalloc(&d_d, (h-l+1)*sizeof(int));
    cudaMemcpy(d_d, arr,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);

    cudaMalloc(&d_l, (h-l+1)*sizeof(int));
    cudaMemcpy(d_l, lstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);

    cudaMalloc(&d_h, (h-l+1)*sizeof(int));
    cudaMemcpy(d_h, hstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);
    int n_t = 1;
    int n_b = 1;
    int n_i = 1;
    while ( n_i > 0 )
    {
        partition<<<n_b,n_t>>>( d_d, d_l, d_h, n_i);
        int answer;
        cudaMemcpyFromSymbol(&answer, d_size, sizeof(int), 0,
cudaMemcpyDeviceToHost);
        if (answer < 1024)
          {
            n_t = answer;
          }
        else
          {
            n_t = 1024;
            n_b = answer/n_t + (answer%n_t==0?0:1);
          }
        n_i = answer;
        cudaMemcpy(arr, d_d,(h-l+1)*sizeof(int),cudaMemcpyDeviceToHost);
    }
}
```

Results of Sorting Algorithms

Latency comparison

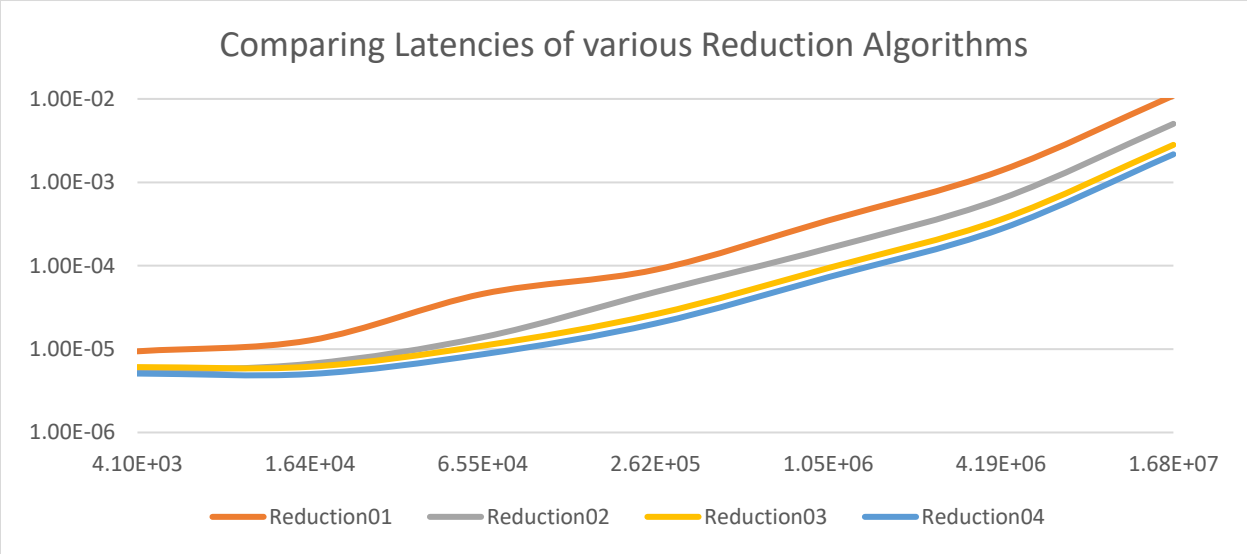


Bandwidth Comparison



Results of Reduction Algorithms

Latency Comparison



Bandwidth Comparison