K Shwetha; Roll No: 14ME10072

Assignment 3: Reduction with Matrices
HP3 CUDA Programming

Introduction

The goal of this assignment is to implement various reduction techniques for the problem of matrix addition.

Since matrix addition is an associative operation, the techniques for parallel reduction can be applied here. Your objective would be to implement a CUDA program which takes as input a set of n 2x2 matrices, perform matrix addition using reduction and return the final 2x2 matrix. The problem is very different from that of simple addition since you now have to accommodate a set of matrices in shared memory and perform element wise addition. The execution time of the program that I will be developing will therefore depend on how the set of matrices are stored in memory.

Device Properties

- Shared memory per block = 49152
- Warp size = 32
- Max threads per block = 1024
- Max threads per block x = 1024
- Max threads per block y = 1024
- Max threads per block z = 64
- Max grids x = 2147483647
- Max grids y = 65535
- Max grids z = 65535
- Total const memory = 6553
- Total multiprocessors = 15

Note:

I haven't used different approach for row-major and column-major implementation because I have stored the input data in such a way that all the (0,0) elements of all the matrices are stored together, and so on.

Naive Reduction

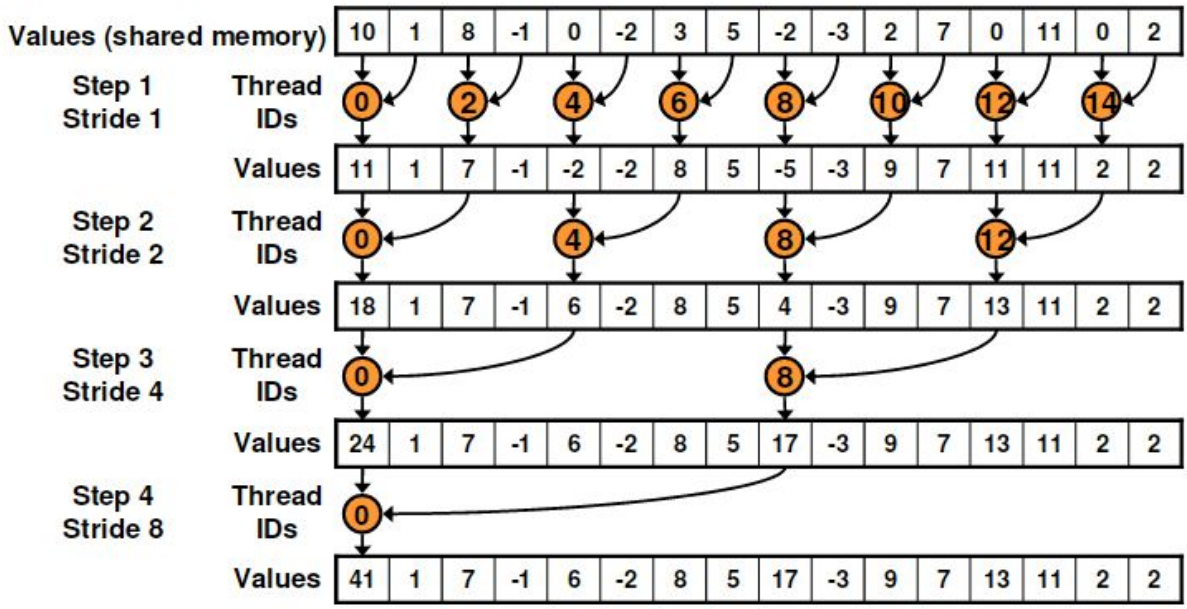
Kernel decomposition is done using tree-based approach as taught in class to handle large input sizes. In the kernel function, first the data is loaded, with each block containing all the elements to be added. The number of blocks per grid is constant and equal to 4, one block to calculate each element of the output matrix. I have used interlevead addressing of threads. In this, the number of active threads reduces by a factor of two in every loop iteration. This leads to a lot of inefficiency.

The data is stored in such a way that all the 0th elements of all the matrices are stored together in the global memory. This will reduce the number of warps required for fetching data from the global memory and storing it in the shared memory of the block.

```
// Initialize the host input vectors
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < 2; j++)
        {
            for(k = 0; k < 2; k++)
            {
                h_A[(j*2+k)*n + i] = (int)(RAND_MAX/rand());
             }
        }
    }
```

Fig. Code snippet for loading data

Naive Reduction Kernel Function



Naive Reduction Kernel Function

```
#include <stdio.h>

__global__ void add_kernel1(int *A, int *B, int n)
{
    __device__ __shared__ int sdata[1024];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

   sdata[tid]=A[i];
    __syncthreads();

    unsigned int s;

    for(s = 1; s<blockDim.x; s*=2)
    {
        if(tid % (2*s) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    if (tid==0)
        B[blockIdx.x] = sdata[0];
}
```

Results for Naive Reduction

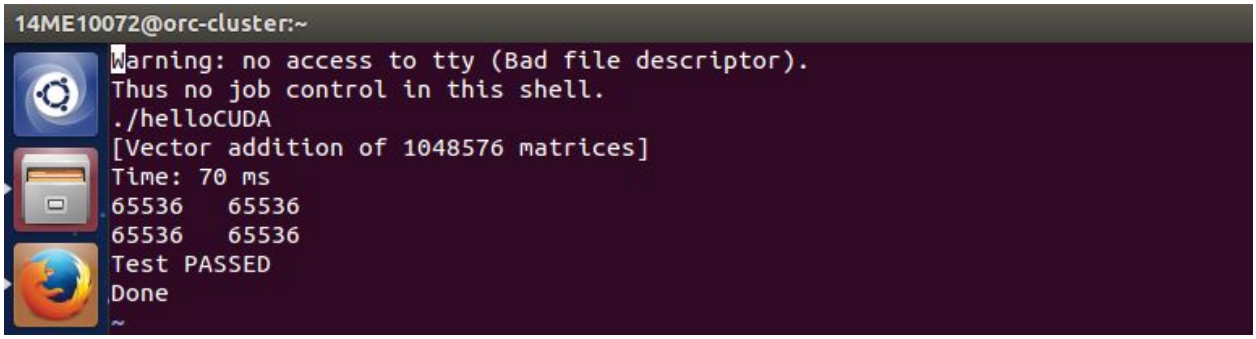


Fig. The naive reduction takes 70ms for 2^20 matrices.

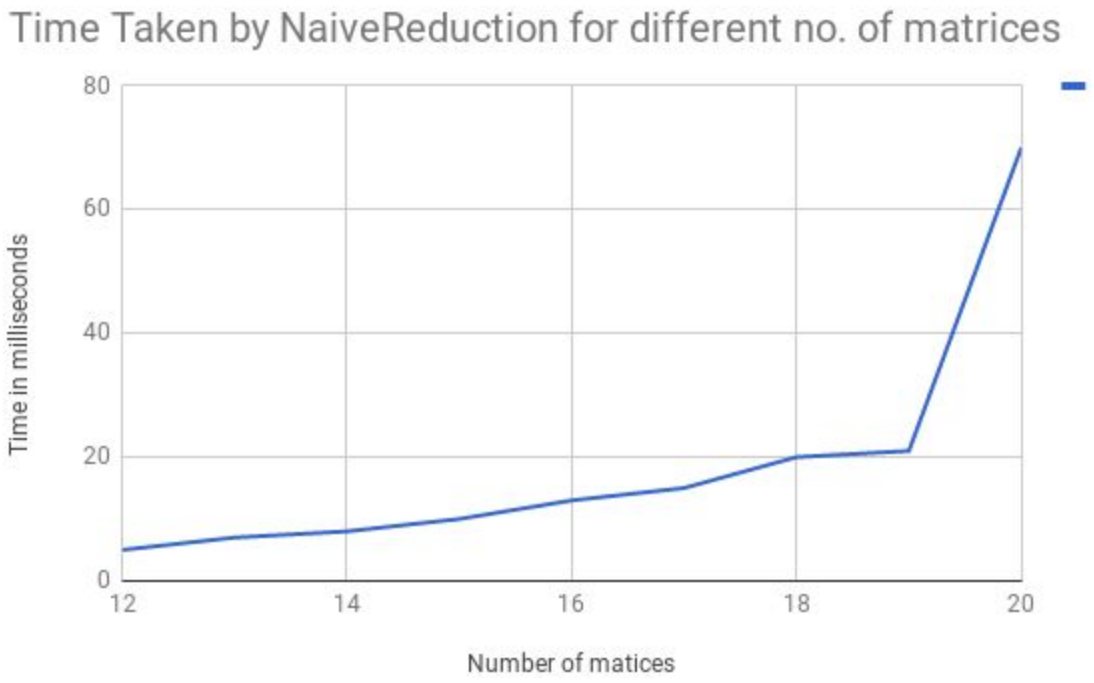


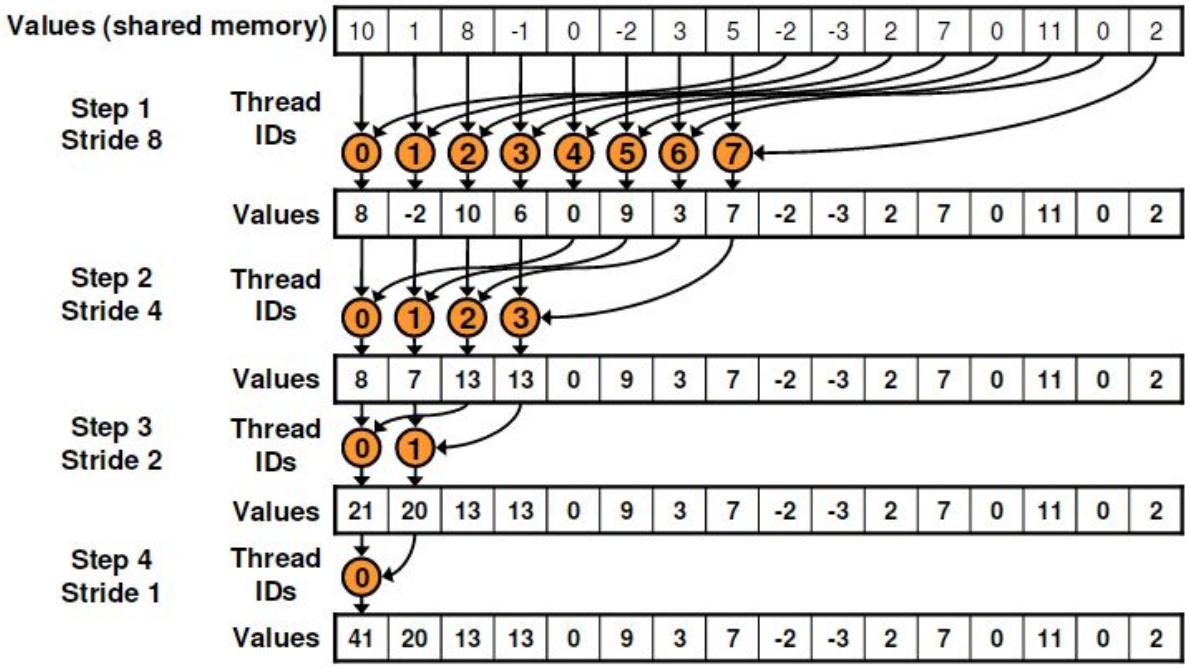
Fig. The time taken by Naive Reduction for different number of matrices.

Optimized Reduction

For the purpose of optimization, I tried different reduction techniques discussed in the class, and found that the sequential addressing along with addition at the first load works well with respect to speedup. Sequential addressing doesn't encounter any shared memory bank conflicts. Also, since the first addition is done during loading of the data into

shared memory, the problem of half of the threads being idle in the first iteration is also resolved.

Sequential Addressing



Kernel Function for Sequential Addressing

```
26  __global__ void add_kernel4(int *A, int *B, int n)
27  {
28      __device__ __shared__ int sdata[1024];
29      unsigned int tid = threadIdx.x;
30      unsigned int i = blockIdx.x*(blockDim.x) + threadIdx.x;
31      sdata[tid] = A[i] + A[i+blockDim.x];
32      __syncthreads();
33
34      unsigned int s;
35
36      for (s=blockDim.x/2; s>0; s>>=1)
37      {
38          if(tid < s)
39          {
40              sdata[tid] += sdata[tid + s];
41          }
42      __syncthreads();
43      }
44
45      if (tid==0)
46          B[blockIdx.x] = sdata[0];
47  }
```

Results for Optimized Reduction

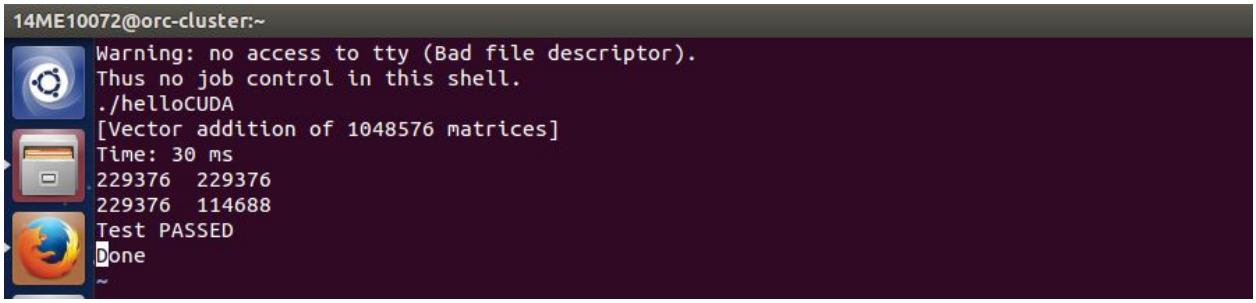
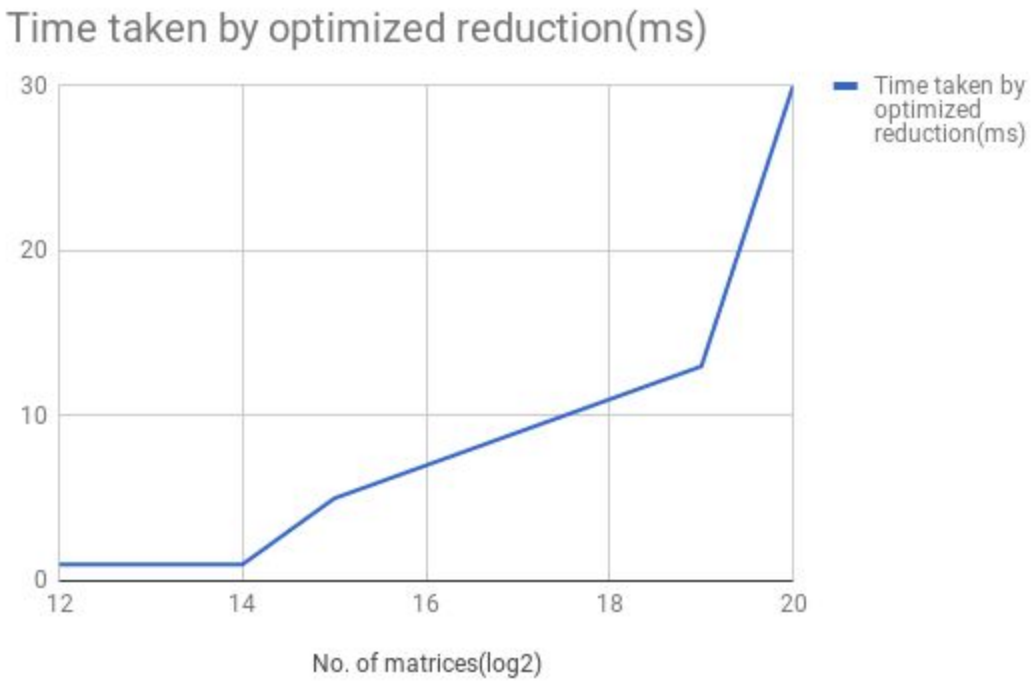
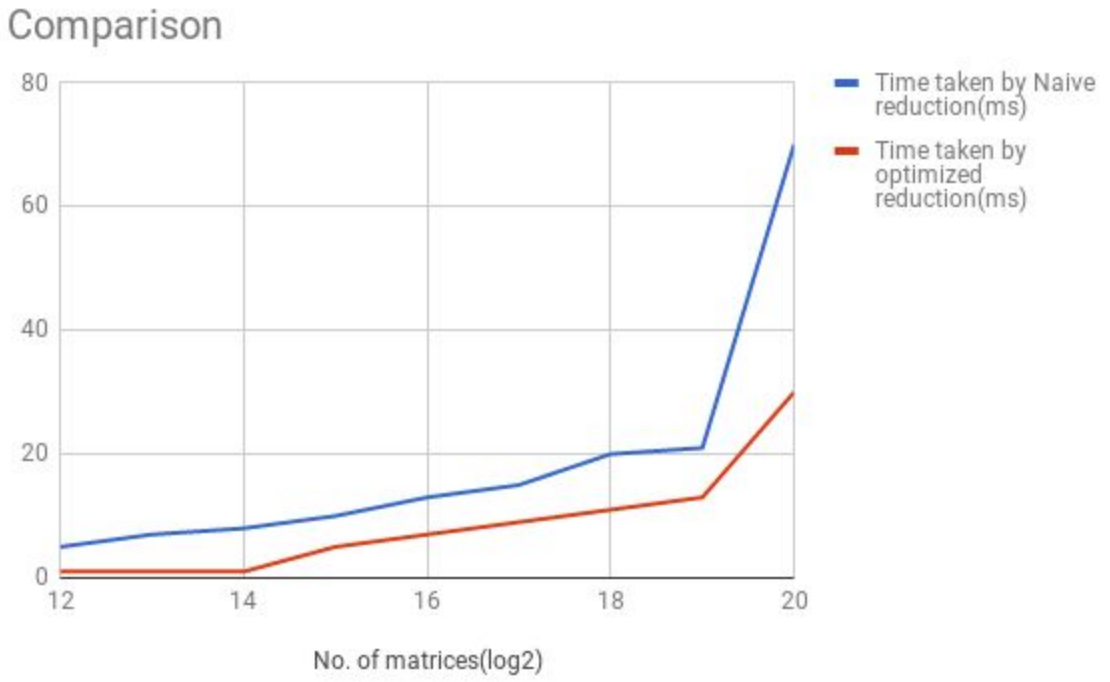
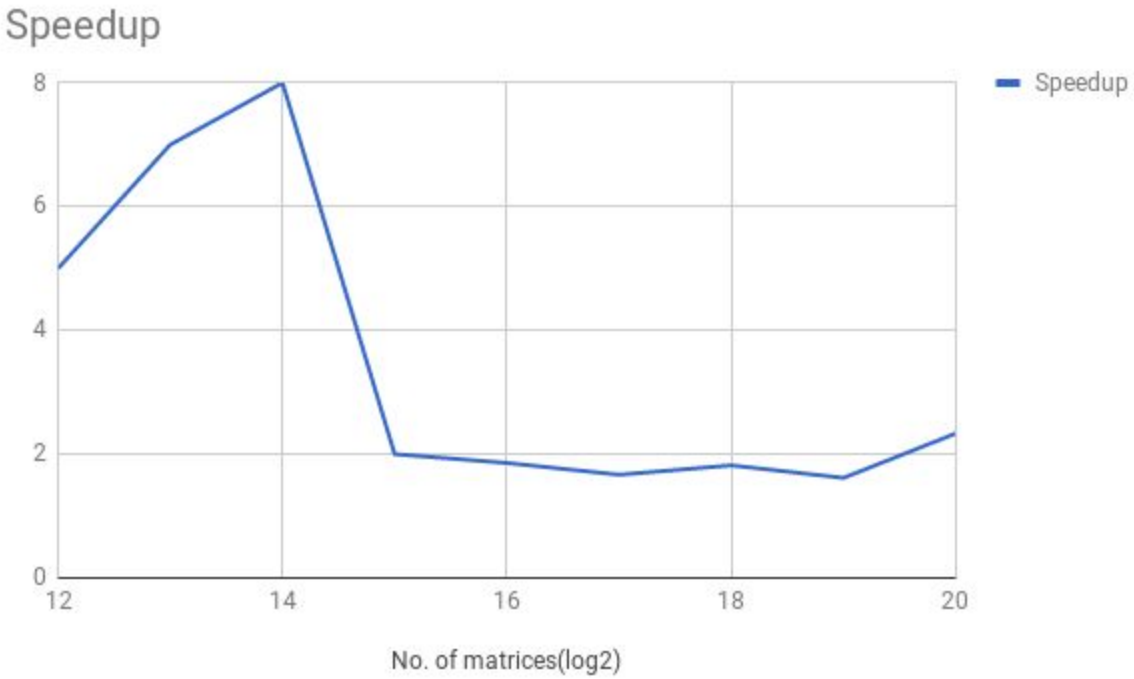


Fig. The optimized reduction takes 30ms for 2^20 matrices.



Speedup and Comparison





Limitations of Hardware

Limitations of hardware were bypassed by using kernel decomposition. This approach involves recursively calling the kernel function for smaller blocks. The code snippet is shown below.

Since the maximum number of threads in a block is 1024, the number of threads per block reduce by a factor of 1024 in each iteration.

Also, since the data is stored in such a way that all the same position elements of all the matrices are stored in consequent memory locations, the number of slower global memory transactions is also reduced significantly.

```
88      while(n>1)
89      {
90          for(i = 0; i<n; i+=1024)
91          {
92              if(n-i>=1024)
93              {
94                  size = 1024*numElements*sizeof(int);
95                  threadsPerBlock = 1024;
96              }
97              else
98              {
99                  size = (n-i)*numElements*sizeof(int);
100                 threadsPerBlock = n-i;
101             }
102
103             // Copy the host input vectors A and B in host memory to the device input vectors in device memory
104             err = cudaMemcpy(d_A, h_A + i, size, cudaMemcpyHostToDevice);
105
106             if (err != cudaSuccess)
107             {
108                 fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n", cudaGetErrorString(err));
109                 exit(EXIT_FAILURE);
110             }
111
112             // Launch the Vector Add CUDA Kernel
113             add_kernel1<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, n);
114             err = cudaGetLastError();
115
116             if (err != cudaSuccess)
117             {
118                 fprintf(stderr, "Failed to launch swap kernel (error code %s)!\n", cudaGetErrorString(err));
119                 exit(EXIT_FAILURE);
120             }
121
122             // Copy the device result vector in device memory to the host result vector in host memory.
123             err = cudaMemcpy(h_B, d_B, 4*sizeof(int), cudaMemcpyDeviceToHost);
124
125             if (err != cudaSuccess)
126             {
127                 fprintf(stderr, "Failed to copy vector B from device to host (error code %s)!\n", cudaGetErrorString(err));
128                 exit(EXIT_FAILURE);
129             }
130
131             temp[4*((i+1)/64)] = h_B[0];
132             temp[4*((i+1)/64)+1] = h_B[1];
133             temp[4*((i+1)/64)+2] = h_B[2];
134             temp[4*((i+1)/64)+3] = h_B[3];
135         }
136         n = n/1024;
137         h_A = temp;
138     }
139
```

Sensitivity to Hardware Properties

- My optimization is dependent on the maximum number of threads in a block and the warp size. If the maximum number of threads is increased, the kernel decomposition will occur at a faster rate, increasing the parallelism of the program, and hence, decreasing the execution time significantly.
- As the warp size increases, the number global memory accesses required for storing the data in shared memory decreases, hence decreasing the execution time.