CZ4011 Parallel Computing

Lab 3 Report

All Pair Shortest Path using CUDA

SESHADRI MADHAVAN
PRAJOGO TIO

Nanyang Technological University

Table of content

1. Introduction

In this report we discuss several CUDA implementations of parallelized Floyd Warshall algorithm. We study the effect of various parameterization of the CUDA execution on the algorithm's overall performance.

2. Code listing and implementation

In this section we present the code listings of our *.cu files that implements are the required kernels. Our inputs are generated using the function GenMatrix provided in MatUtil.h. We also perform correctness check of the results by comparing our results with ST_APSP instantiation, also provided in MatUtil.h.

2.1. Basic Floyd Warshall kernel

Below is the code listing for the basic kernel implementation for Floyd Warshall. It uses 2D grid and perform N round of kernel calls, where each kernel call computes in parallel d[i][j] that the current thread owns. No optimization is present.

basic_kernel.cu:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

extern "C" {
#include "MatUtil.h"
}

__device__
int Min(int a, int b) { return a < b ? a : b; }

__global__
void NaiveFloydWarshall(int* mat, int k, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i < N && j < N) {
        if (mat[i*N + k] != -1 && mat[k*N + j] != -1) {
            if (mat[i*N+j] == -1) {
                mat[i*N+j] = mat[i*N + k] + mat[k*N +j];
            } else {
                mat[i*N+j] = Min(mat[i*N + k] + mat[k*N + j], mat[i*N+j]);
            }
        }
    }
}

void NaiveFloydWarshallDriver(int* mat, int N, dim3 thread_per_block) {
    int* cuda_mat;
    int size = sizeof(int) * N * N;
    cudaMalloc((void**) &cuda_mat, size);
    cudaMemcpy(cuda_mat, mat, size, cudaMemcpyHostToDevice);
    dim3 num_block(ceil(1.0*N/thread_per_block.x),
                   ceil(1.0*N/thread_per_block.y));
    for (int k = 0; k < N; ++k) {
        NaiveFloydWarshall<<<num_block, thread_per_block>>>(cuda_mat, k, N);
    }
    cudaMemcpy(mat, cuda_mat, size, cudaMemcpyDeviceToHost);
    cudaFree(cuda_mat);
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char **argv) {
    if(argc != 5) {
        printf("Usage: test {N} {run_sequential_check: 'T' or 'F'} {thread_per_block.x}
{thread_per_block.y}\n");
        exit(-1);
    }
    char run_sequential_check = argv[2][0];
    dim3 thread_per_block(atoi(argv[3]), atoi(argv[4]));
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int*)malloc(sizeof(int)*N*N);
    GenMatrix(mat, N);

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    NaiveFloydWarshallDriver(result, N, thread_per_block);

    //compare your result with reference result
    if (run_sequential_check == 'T') {
        int *ref = (int*)malloc(sizeof(int)*N*N);
        memcpy(ref, mat, sizeof(int)*N*N);
        ST_APSP(ref, N);
        if(CmpArray(result, ref, N*N))
            printf("Your result is correct.\n");
        else
            printf("Your result is wrong.\n");
    }
}
```

2.2. Coalesced Floyd Warshall implementation

In this implementation, a 1D grid is used. We try to make each warp to have coalesced data access to the global memory by assigning threads following the matrix layout in memory (which is row-major format).

coalesced_kernel.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

extern "C" {
#include "MatUtil.h"
}

__device__
int Min(int a, int b) { return a < b ? a : b; }

__global__
void CoalescedFloydWarshall(int* mat, int k, int N) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N*N) {
        int i = idx/N;
        int j = idx - i*N;
        if (mat[i*N + k] != -1 && mat[k*N + j] != -1) {
            if (mat[idx] == -1) {
                mat[idx] = mat[i*N + k] + mat[k*N +j];
            } else {
                mat[idx] = Min(mat[i*N + k] + mat[k*N + j], mat[idx]);
            }
        }
    }
}

void CoalescedFloydWarshallDriver(int* mat, int N, int thread_per_block) {
    int* cuda_mat;
    int size = sizeof(int) * N * N;
    cudaMalloc((void**) &cuda_mat, size);
    cudaMemcpy(cuda_mat, mat, size, cudaMemcpyHostToDevice);
    int num_block = ceil(1.0*N*N/(thread_per_block));
    for (int k = 0; k < N; ++k) {
        CoalescedFloydWarshall<<<num_block, (thread_per_block)>>>(cuda_mat, k, N);
    }
    cudaMemcpy(mat, cuda_mat, size, cudaMemcpyDeviceToHost);
    cudaFree(cuda_mat);
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
```

```
{
    if(argc != 4) {
        printf("Usage: test {N} {run_sequential_check: 'T' or 'F'} {thread_per_block}\n");
        exit(-1);
    }
    char run_sequential_check = argv[2][0];
    int thread_per_block = atoi(argv[3]);
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int*)malloc(sizeof(int)*N*N);
    GenMatrix(mat, N);

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    CoalescedFloydWarshallDriver(result, N, thread_per_block);

    //compare your result with reference result
    if (run_sequential_check == 'T') {
        int *ref = (int*)malloc(sizeof(int)*N*N);
        memcpy(ref, mat, sizeof(int)*N*N);
        ST_APSP(ref, N);
        if(CmpArray(result, ref, N*N))
            printf("Your result is correct.\n");
        else
            printf("Your result is wrong.\n");
    }
}
```

2.3 Coalesced Floyd Warshall with segmentation

In this implementation, we try to improve the coalescing by allowing a thread to fetch a few more additional data to its right, hence owning a "segment" of data (instead of just one value). In other words, each thread will be assigned a segment [j, j+1, ..., j+segment_size-1] and will compute d[i][J] for all J in that segment. In our experiment, we set the segment_size to 2 because it achieves better performance.

coalesced_kernel_with_segment.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

extern "C" {
#include "MatUtil.h"
}

__device__
int Min(int a, int b) { return a < b ? a : b; }

__global__
void CoalescedFloydWarshall(int* mat, int k, int N, int segment_size) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (segment_size*idx < N*N) {
        for (int offset = 0; offset < segment_size && offset + segment_size*idx < N*N;
++offset) {
            int i = (segment_size*idx + offset)/N;
            int j = segment_size*idx + offset - i*N;
            if (mat[i*N + k] != -1 && mat[k*N + j] != -1) {
                if (mat[i*N+j] == -1) {
                    mat[i*N+j] = mat[i*N + k] + mat[k*N +j];
                } else {
                    mat[i*N+j] = Min(mat[i*N + k] + mat[k*N + j], mat[i*N+j]);
                }
            }
        }
    }
}

// Each thread will access 'segment_size' values to improve coalescing.
// Each block now handles thread_per_block * segment_size values.
// Hence the number of blocks needed is N*N/(segment_size*thread_per_block).
void CoalescedFloydWarshallDriver(int* mat, int N, int thread_per_block, int segment_size) {
    int* cuda_mat;
    int size = sizeof(int) * N * N;
    cudaMalloc((void**) &cuda_mat, size);
    cudaMemcpy(cuda_mat, mat, size, cudaMemcpyHostToDevice);
    int num_block = ceil(1.0*N*N/(thread_per_block*segment_size));
    for (int k = 0; k < N; ++k) {
        CoalescedFloydWarshall<<<num_block, thread_per_block>>>(cuda_mat, k, N,
segment_size);
```

```
    }
    cudaMemcpy(mat, cuda_mat, size, cudaMemcpyDeviceToHost);
    cudaFree(cuda_mat);
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
{
    if(argc != 5) {
        printf("Usage: test {N} {run_sequential_check: 'T' or 'F'} {thread_per_block}
{segment_size}\n");
        exit(-1);
    }
    char run_sequential_check = argv[2][0];
    int thread_per_block = atoi(argv[3]);
    int segment_size = atoi(argv[4]);
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int*)malloc(sizeof(int)*N*N);
    GenMatrix(mat, N);

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    CoalescedFloydWarshallDriver(result, N, thread_per_block, segment_size);

    //compare your result with reference result
    if (run_sequential_check == 'T') {
        int *ref = (int*)malloc(sizeof(int)*N*N);
        memcpy(ref, mat, sizeof(int)*N*N);
        ST_APSP(ref, N);
        if(CmpArray(result, ref, N*N))
            printf("Your result is correct.\n");
        else
            printf("Your result is wrong.\n");
    }
}
```

2.4 Floyd Warshall with Shared Memory

In this implementation, we make use of tiling strategy to allow a block of [i..i+TILE_HEIGHT-1] x [j...j+TILE_WIDTH-1] to first prefetch collaboratively the values d[I][k] and d[k][J] for all I and J owned by the block. After the values are prefetched from global memory to shared memory, computation resumes as per normal. The choice of TILE_WIDTH and TILE_HEIGHT for this experiment are 8x8, 16x16, and 32x32.

shared_memory_kernel.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

extern "C" {
#include "MatUtil.h"
}

#define TILE_WIDTH 32
#define TILE_HEIGHT 32


__device__
int Min(int a, int b) { return a < b ? a : b; }

__global__
void SharedMemoryFloydWarshall(int* mat, int k, int N) {
    __shared__ int dist_i_k[TILE_HEIGHT];
    __shared__ int dist_k_j[TILE_WIDTH];
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i < N && j < N) {
        int dist_i_j = mat[i*N + j];
        if (i % TILE_HEIGHT == 0) {
            dist_k_j[j % TILE_WIDTH] = mat[k*N + j];
        }
        if (j % TILE_WIDTH == 0) {
            dist_i_k[i % TILE_HEIGHT] = mat[i*N + k];
        }
        __syncthreads();
        if (dist_i_k[i % TILE_HEIGHT] != -1 && dist_k_j[j % TILE_WIDTH] != -1) {
            int new_dist = dist_i_k[i % TILE_HEIGHT] + dist_k_j[j % TILE_WIDTH];
            if (dist_i_j != -1) {
                new_dist = Min(new_dist, dist_i_j);
            }
            mat[i*N + j] = new_dist;
        }
    }
}

void SharedMemoryFloydWarshallDriver(int* mat, int N, dim3 thread_per_block) {
    int* cuda_mat;
    int size = sizeof(int) * N * N;
    cudaMalloc((void**) &cuda_mat, size);
    cudaMemcpy(cuda_mat, mat, size, cudaMemcpyHostToDevice);
    dim3 num_block(ceil(1.0*N/thread_per_block.x),
                   ceil(1.0*N/thread_per_block.y));
    for (int k = 0; k < N; ++k) {
        SharedMemoryFloydWarshall<<<num_block, thread_per_block>>>(cuda_mat, k, N);
    }
    cudaMemcpy(mat, cuda_mat, size, cudaMemcpyDeviceToHost);
    cudaFree(cuda_mat);
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
{
    if(argc != 3) {
        printf("Usage: test {N} {run_sequential_check: 'T' or 'F'}\n");
        exit(-1);
    }
    char run_sequential_check = argv[2][0];
    dim3 thread_per_block(TILE_HEIGHT, TILE_WIDTH);
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int*)malloc(sizeof(int)*N*N);
    GenMatrix(mat, N);

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    SharedMemoryFloydWarshallDriver(result, N, thread_per_block);

    //compare your result with reference result
    if (run_sequential_check == 'T') {
        int *ref = (int*)malloc(sizeof(int)*N*N);
        memcpy(ref, mat, sizeof(int)*N*N);
        ST_APSP(ref, N);
        if(CmpArray(result, ref, N*N))
            printf("Your result is correct.\n");
        else
            printf("Your result is wrong.\n");
    }
}
```

2.5 Floyd Warshall with Full Optimization

For the fully optimized kernel, we combine the use of shared memory and data coalescing to improve the performance of the kernel. The key observation is that for each row i, each thread that computes d[i][j] needs the same d[i][k], hence d[i][k] can be fetched to shared memory once and be used by the rest of the threads in that row. Each thread in the row will still fetch d[k][j] and d[i][j] by themselves to their registers. However, since in a warp both d[k][j] and d[i][j] are accessed in uniform fashion, their accesses are coalesced.

full_optimization_kernel.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes CUDA
#include <cuda_runtime.h>

extern "C" {
#include "MatUtil.h"
}

__device__
int Min(int a, int b) { return a < b ? a : b; }

__global__
void FullyOptimizedFloydWarshall(int* mat, int k, int N) {
    __shared__ int dist_i_k;
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i < N && j < N) {
        int dist_i_j = mat[i*N + j];
        int dist_k_j = mat[k*N + j];
        if (threadIdx.y == 0) {
            dist_i_k = mat[i*N + k];
        }
        __syncthreads();
        if (dist_i_k != -1 && dist_k_j != -1) {
            int new_dist = dist_i_k + dist_k_j;
            if (dist_i_j != -1) {
                new_dist = Min(new_dist, dist_i_j);
            }
            mat[i*N + j] = new_dist;
        }
    }
}

void FullyOptimizedFloydWarshallDriver(int* mat, int N, dim3 thread_per_block) {
    int* cuda_mat;
    int size = sizeof(int) * N * N;
    cudaMalloc((void**) &cuda_mat, size);
    cudaMemcpy(cuda_mat, mat, size, cudaMemcpyHostToDevice);
    dim3 num_block(ceil(1.0*N/thread_per_block.x),
                   ceil(1.0*N/thread_per_block.y));
    for (int k = 0; k < N; ++k) {
        FullyOptimizedFloydWarshall<<<num_block, thread_per_block>>>(cuda_mat, k, N);
    }
    cudaMemcpy(mat, cuda_mat, size, cudaMemcpyDeviceToHost);
    cudaFree(cuda_mat);
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////

int main(int argc, char **argv)
{
    if(argc != 4) {
        printf("Usage: test {N} {run_sequential_check: 'T' or 'F'} {segment_size}\n");
        exit(-1);
    }
    char run_sequential_check = argv[2][0];
    int segment_size = atoi(argv[3]);
    dim3 thread_per_block(1, segment_size);
    //generate a random matrix.
    size_t N = atoi(argv[1]);
    int *mat = (int*)malloc(sizeof(int)*N*N);
    GenMatrix(mat, N);

    //compute your results
    int *result = (int*)malloc(sizeof(int)*N*N);
    memcpy(result, mat, sizeof(int)*N*N);
    //replace by parallel algorithm
    FullyOptimizedFloydWarshallDriver(result, N, thread_per_block);

    //compare your result with reference result
    if (run_sequential_check == 'T') {
        int *ref = (int*)malloc(sizeof(int)*N*N);
        memcpy(ref, mat, sizeof(int)*N*N);
        ST_APSP(ref, N);
        if(CmpArray(result, ref, N*N))
            printf("Your result is correct.\n");
        else
            printf("Your result is wrong.\n");
    }
}
```

2.6 GPU capabilities

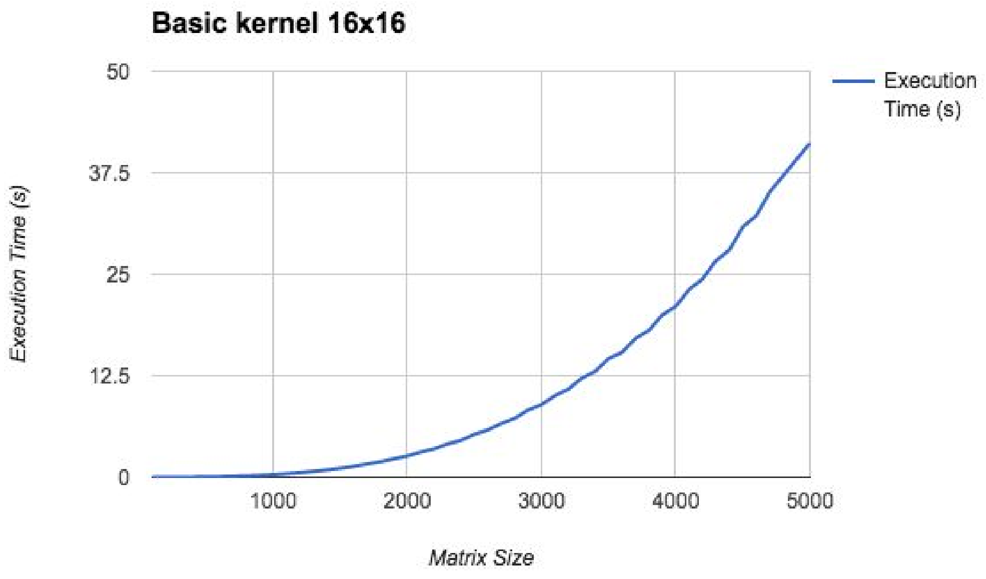
Below we present the GPU specifications used for our experiment.

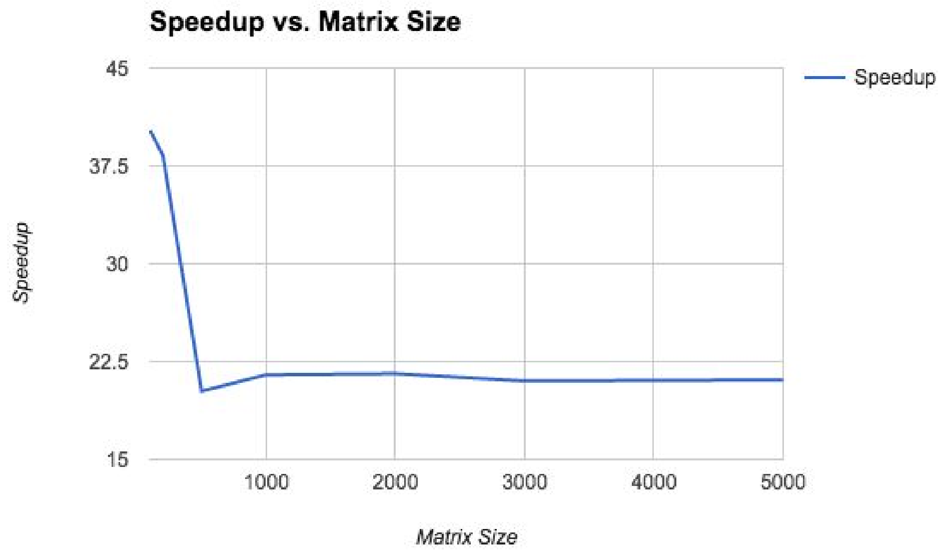
```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GTX 680
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432     6113.4
 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432     6538.7

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432     151984.0

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU
Boost is enabled.


Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 680"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2047 MBytes (2146762752 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1058 MHz (1.06 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096,
4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 3 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
>

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5,
NumDevs = 1, Device0 = GeForce GTX 680
Result = PASS
```

3. Results

3.1 Analysis on Matrix Size

This section shows the effect of matrix size on the execution time of a basic Floyd Warshall kernel. Each block is fixed at size of 16x16.





Analysis on Execution Time

The execution time graph shows an exponential shape function, which is expected because GPU based parallel computing also allows only linear speedup in general especially when N is large.

Analysis on Speedup

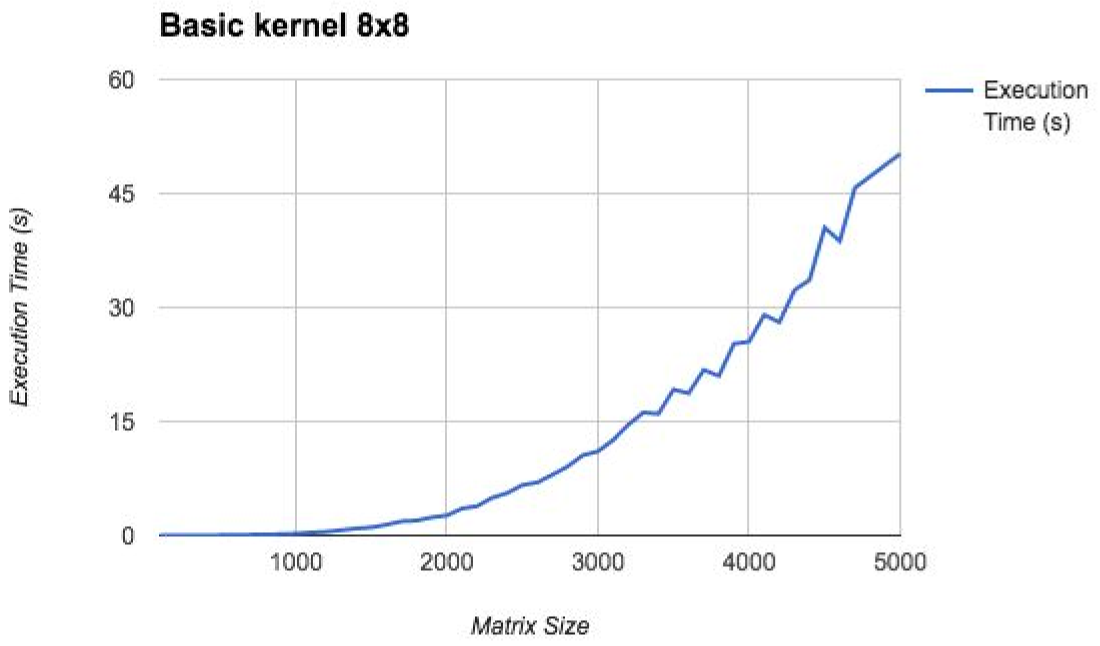
The basic CUDA based implementation shows a magnificent speedup of around 20x for N > 1000, which is significantly larger than what OpenMP and MPI based implementation can achieve. However the speedup degrades from around 38x to around 20x quickly for N < 500, and stays constant afterwards. This is different from the speedup profile we have seen in OpenMP and MPI based implementation which shows linear speedup for small N and sublinear speedup to larger N.

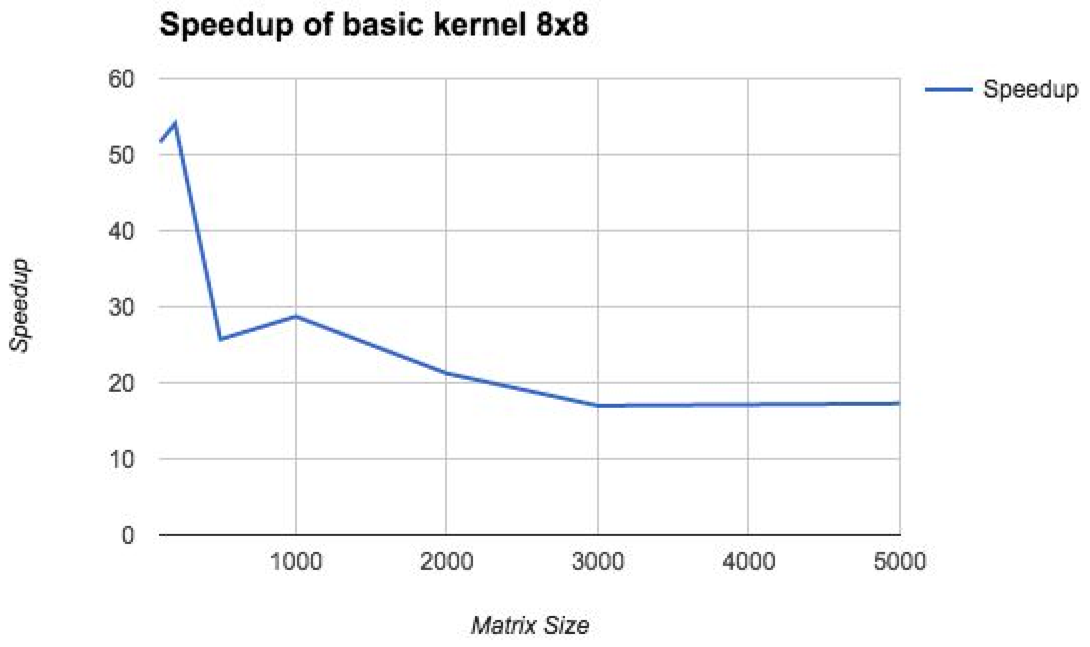
The constant speedup after N > 1000 represents the bottleneck on data transfer bandwidth and number of FLOPs that the GPU can perform. The speedup that we can get is highly dependent on the parameters used (grid dimension, number of thread per block, data access pattern, warp divergence avoidance). In the next section we present the effect of varying various parameters on execution time.

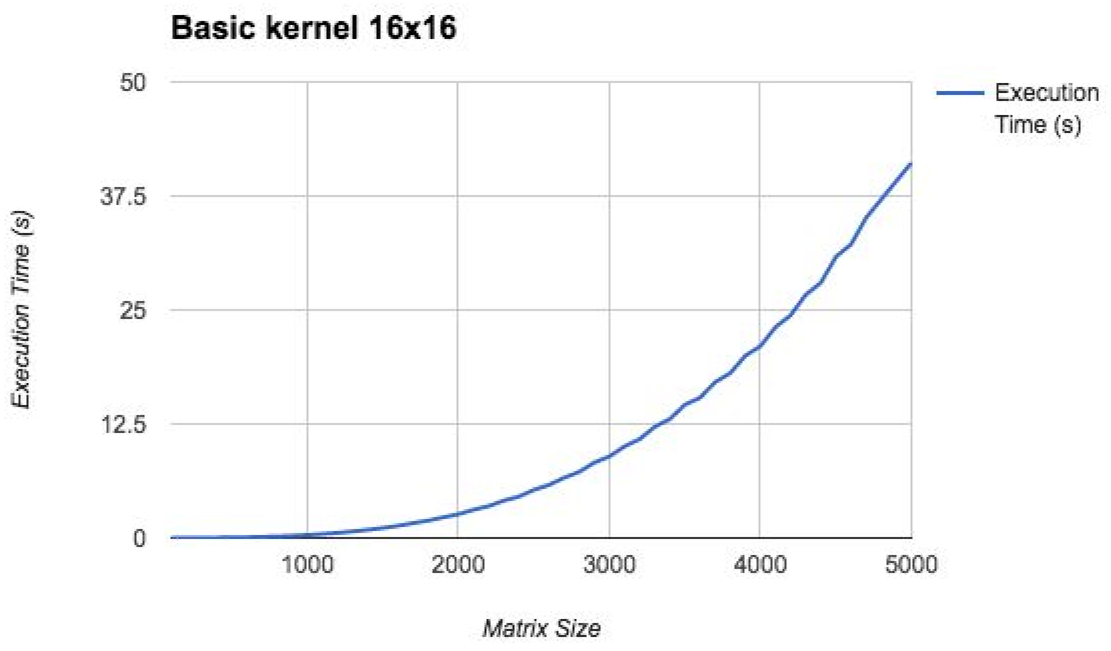
Although for larger N the time needed to send data from host to device and vice versa will take longer, in our current implementation it is only done once at the beginning and end of the algorithm. Also, the constant speedup of around 20x at larger N values indicate that the data transfer overhead to initialize and retrieve the computed data is actually small in comparison to the total work done by the GPU.

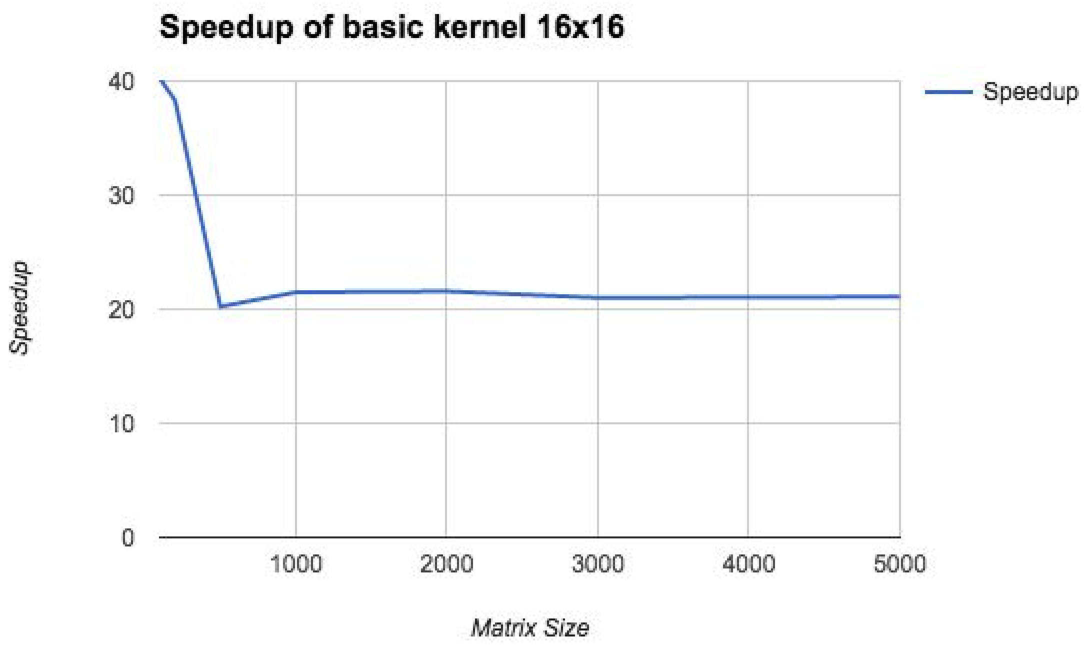
It is worth noting that even the most naive implementation of GPU based algorithm can outperform a carefully designed MPI/OpenMP code (which can at most achieve 4-6x speedup using common commodity computer).

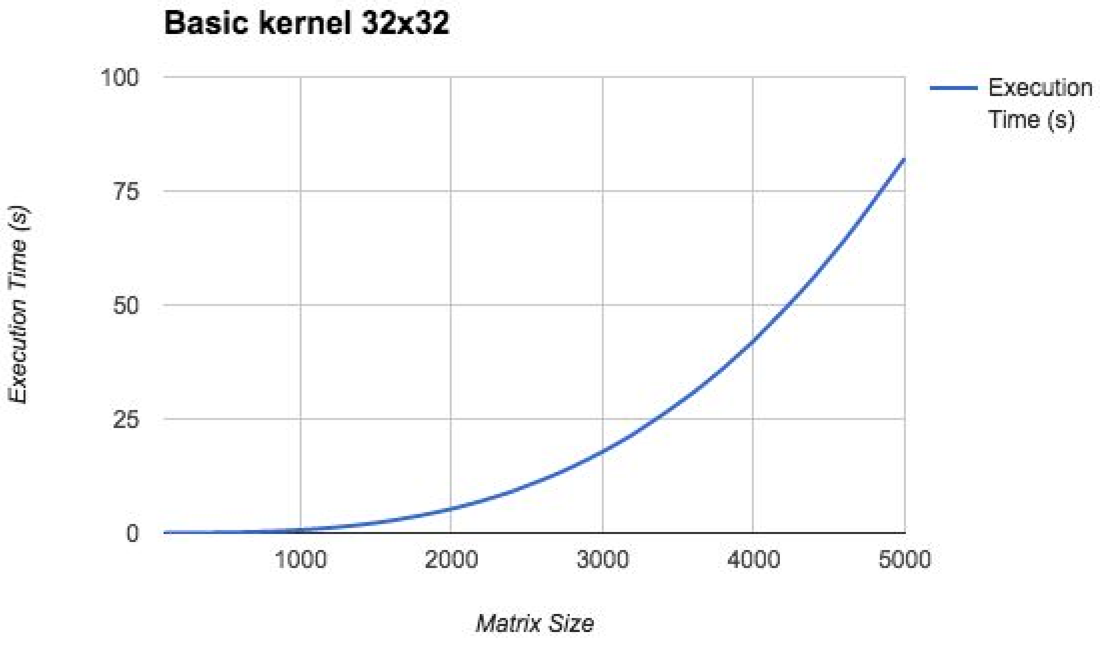
3.2 Effect of block size on the basic kernel

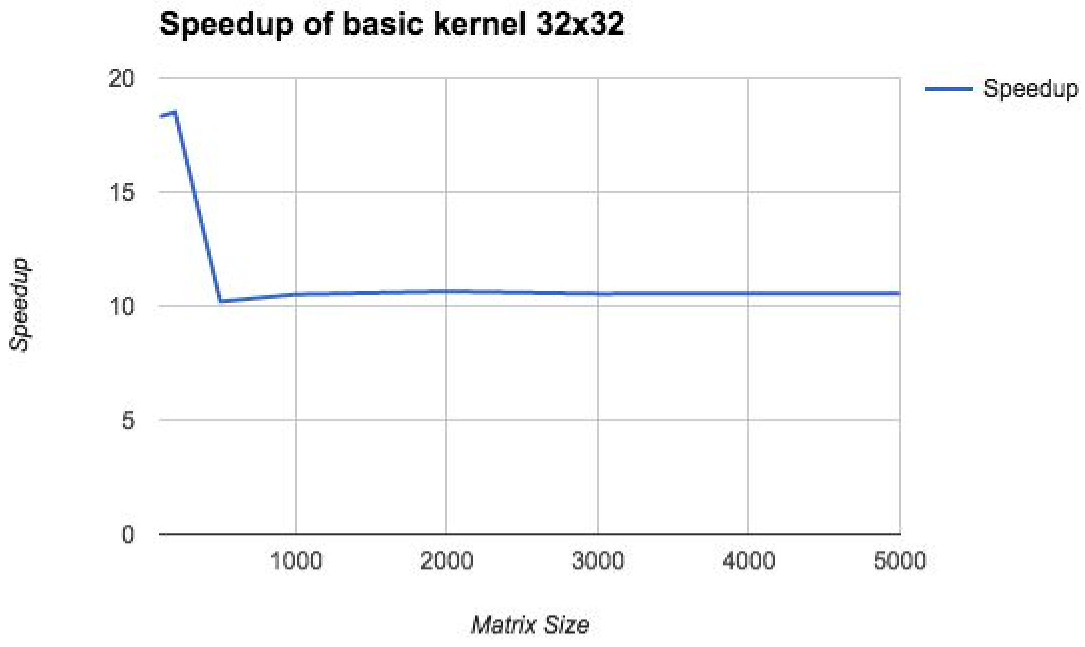





Basic kernel 16x16
Execution Time (s)
50
37.5
25
12.5
0
1000
2000
3000
4000
5000
Matrix Size
Execution Time (s)


Speedup of basic kernel 16x16
Speedup
40
30
20
10
0
1000
2000
3000
4000
5000
Matrix Size
Speedup





<u>Analysis</u>
Based on the graphs, we see that using block size of 16x16 results in the best speedup profile for N > 1000, achieving speed up of 21.1 for N = 5000 (where sequential run take 867.7s while the kernel takes 41.1s). Block size of 8x8 achieves poorer result of 17.3 speedup for N = 5000,

while block 32x32 achieves even poorer result of 10.6 speedup for the same matrix size. This suggests that using 16x16 gives us the best occupancy of the SMs.

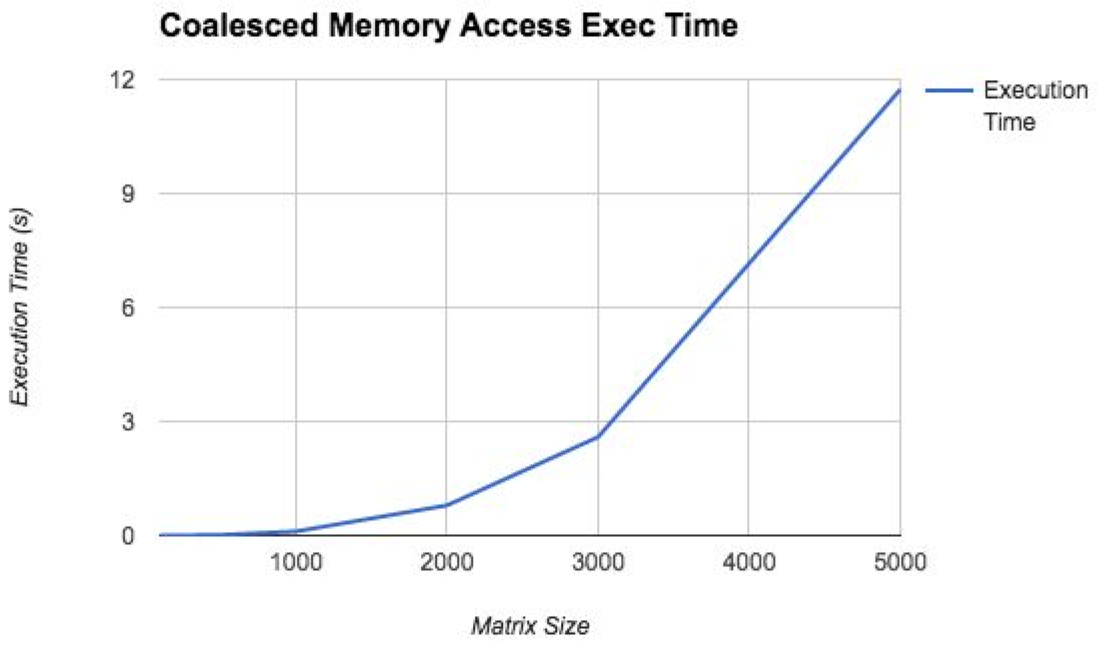
For 8x8 block size, we will need more blocks per SM to fully occupy the SM because each block only has 64 threads. However, SM imposes a certain limit to the maximum number of blocks it can contain, usually 8. Hence in this case choosing 8x8 will result in underutilisation of SM, resulting in lower speedup observed.

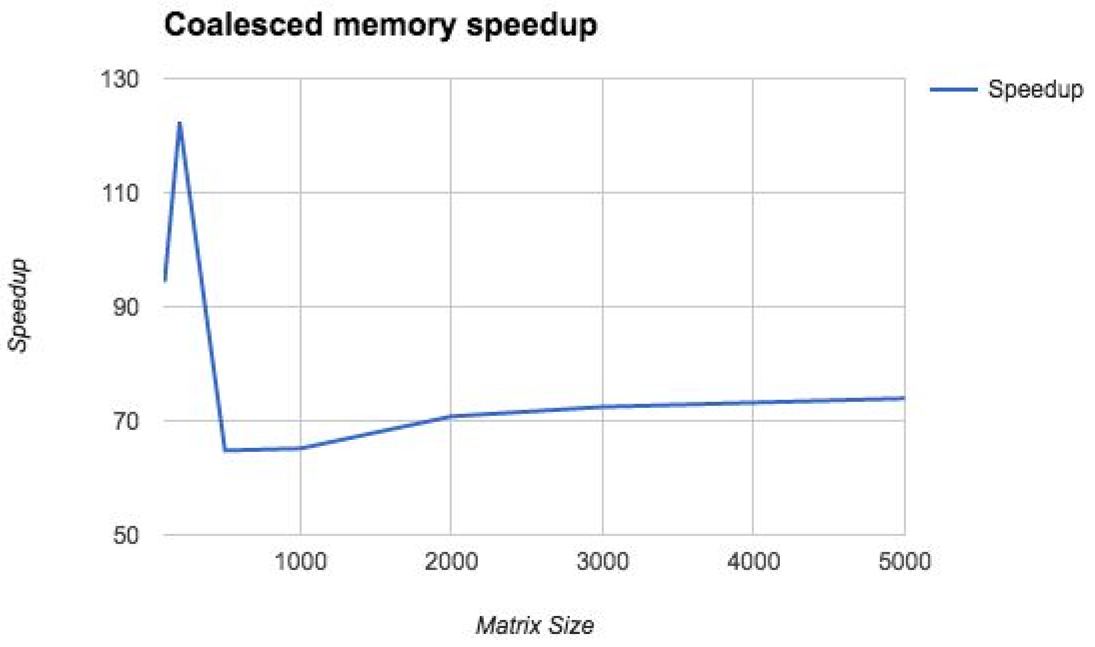
For 16x16 block size, each block contains 256 threads. Using 8 blocks, we can fully occupy SM which has maximum number of threads of 2048.

For 32x32 block size, each block will contain 1024 threads. In the GPU specification, we see that the maximum number of threads per SM is 2048, hence theoretically 2 blocks will fully occupy the SM. However we see that the performance is significantly lower than when we have block size of 16x16. Hence other factors such as highly uncoalesced accesses to global memory and high memory contention could have influenced the results.

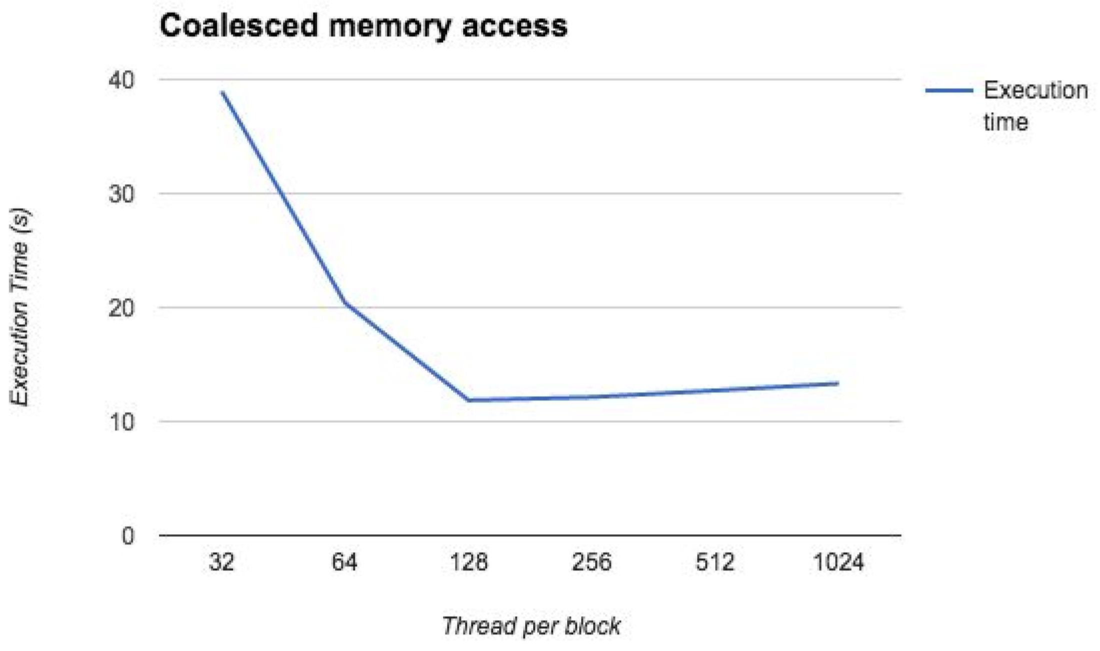
3.3. Effect of Coalescing memory access

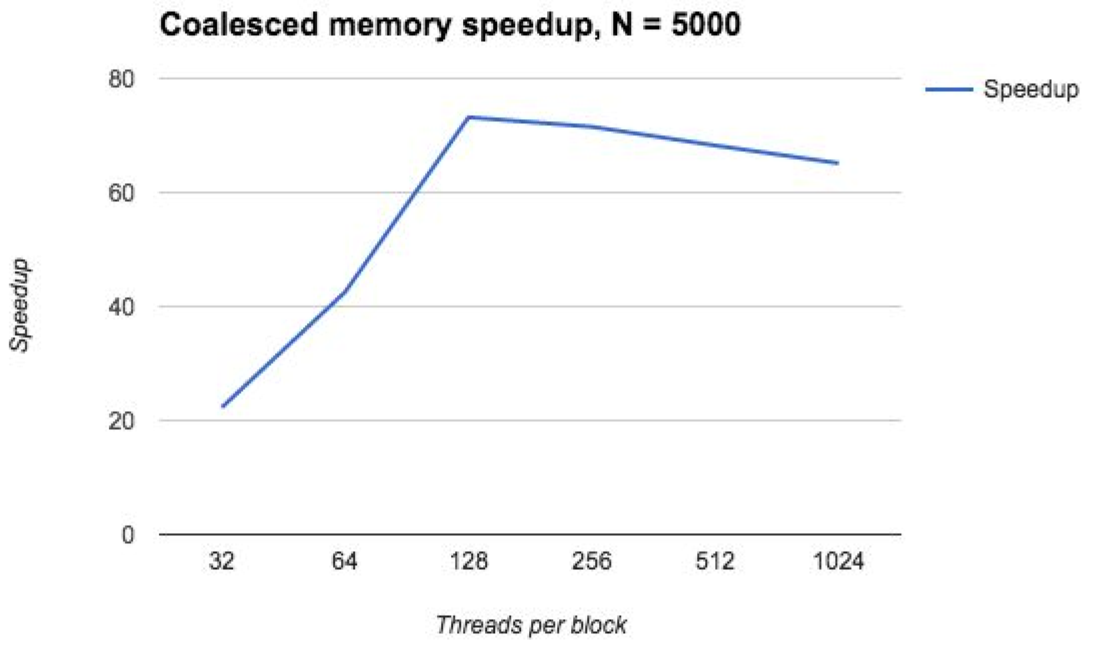
Below is the benchmark profile for coalesced memory access over a range of N values:





Below is the execution time and speedup profile for N = 5000.





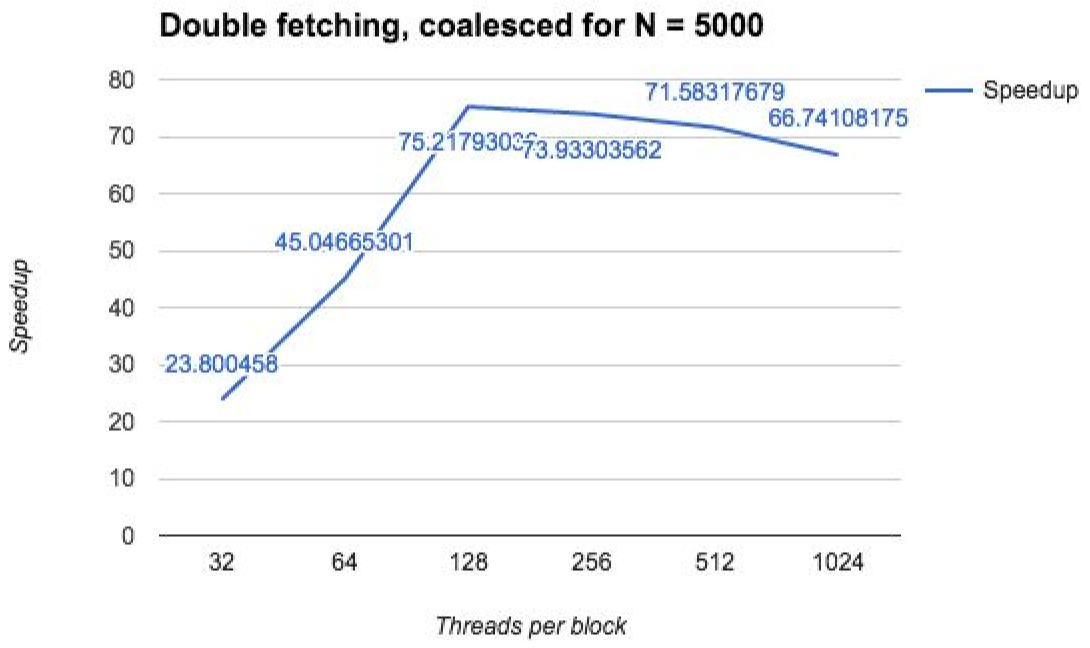
<u>Analysis</u>
Speedup gain from coalesced memory access for thread per block 128 for large N peaks at 73.9x speedup, which is significantly larger than that of basic kernel at 20x speedup. This shows

that by performing coalesced data access to global memory, memory bandwidth are utilized more effectively, which in turn increases throughput.

Also, at thread per block of 128, we see that the performance of this particular kernel achieves its best performance, which indicates that this block size has best SM occupancy. Setting the value lower than 128 causes significant drop in performance and speedup as theoretically more blocks are needed to fully occupy an SM, while there is a limit of around 8 blocks per SM, hence SM will be under utilized. Increasing the block size larger than 128 leads to modest drop in performance as higher possibility of bank conflict and warp divergence.

Double fetching

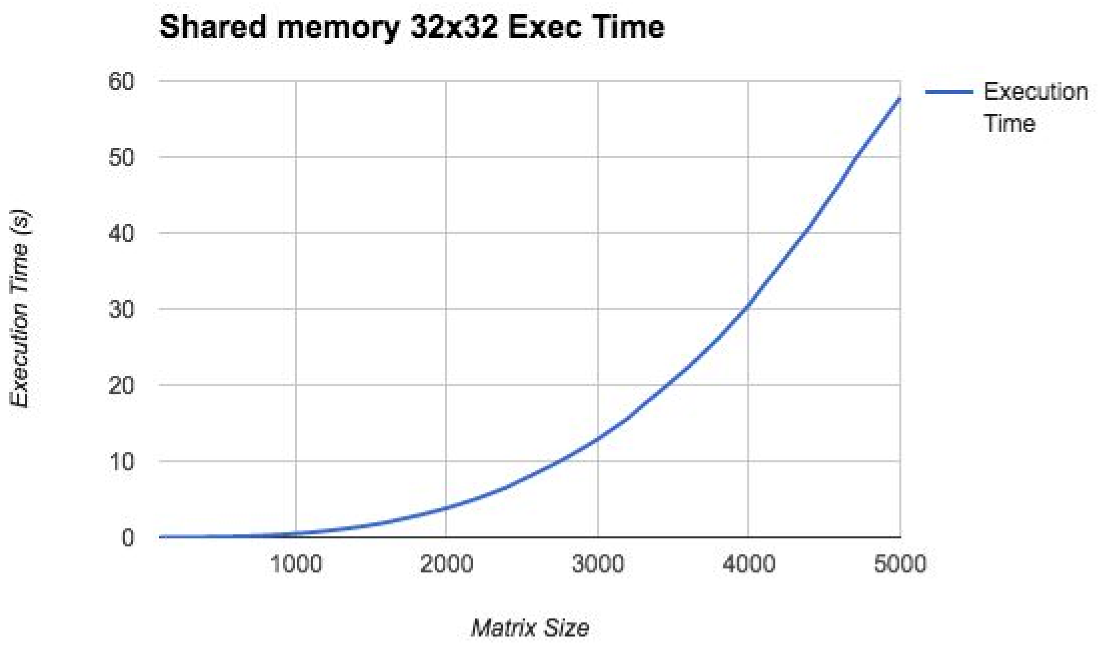
By allowing each thread to fetch two values d[i][j] and d[i][j+1], we observe an improvement in performance to the coalesced memory approach as seen in the graph below:

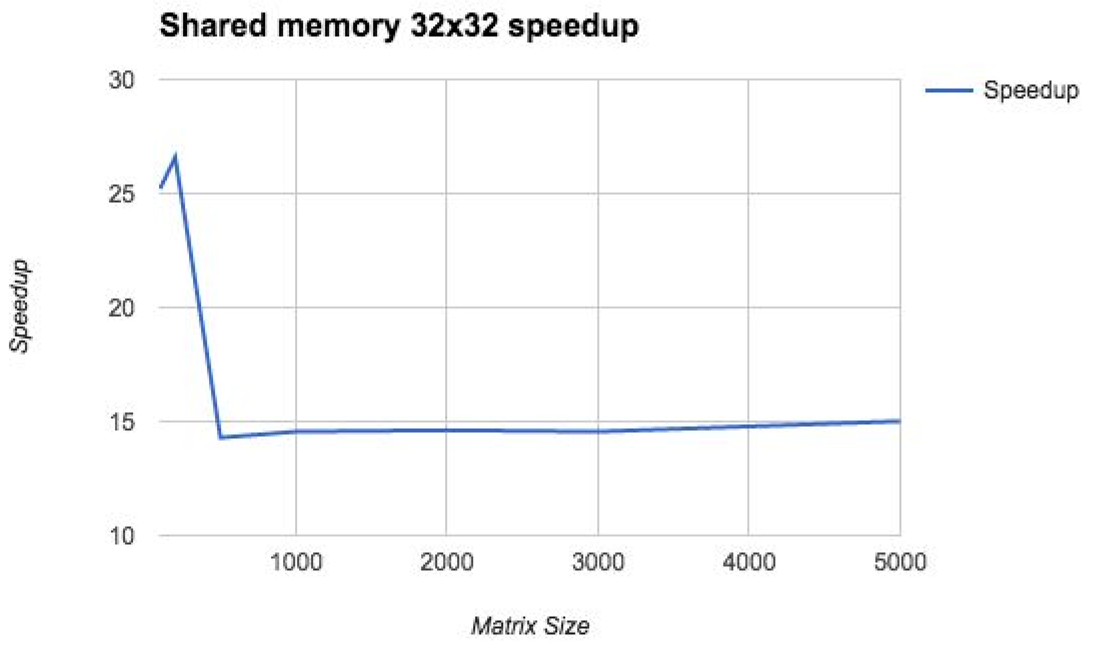


The maximum speedup is 75.21x, higher than the previous peak at 73.9x. This suggests that allowing threads to exploit temporal locality can further improve performance.

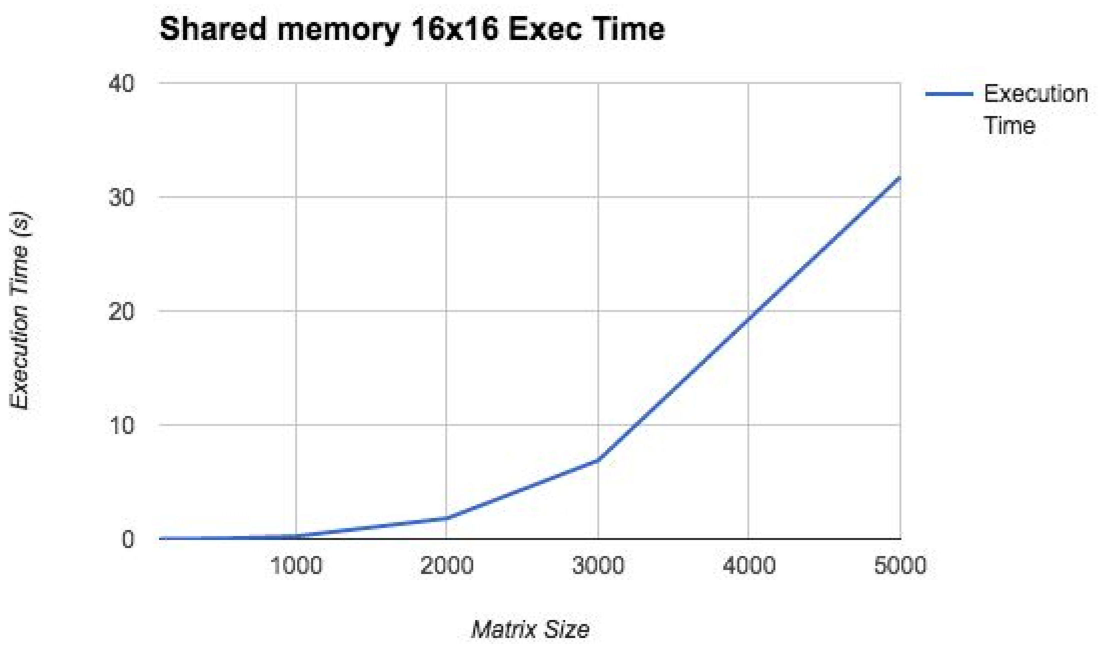
3.4. Effect of Shared Memory

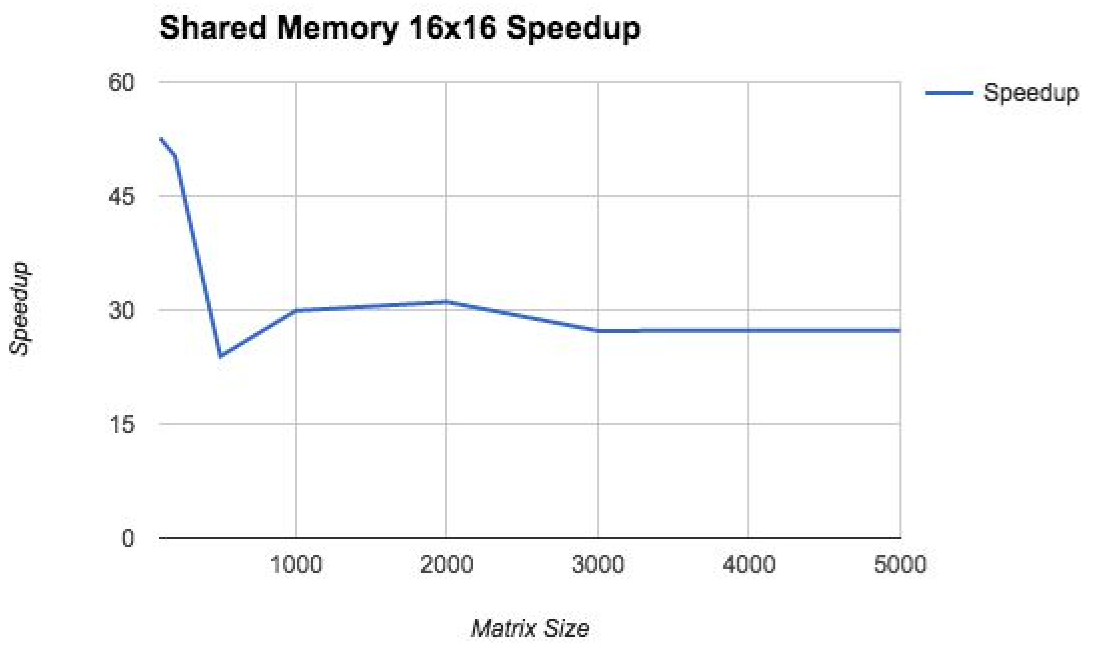
Below is the execution time and speedup profile for shared memory approach using tiling of size 32x32 to maximize shared memory usage.





Next, we present the benchmark profile for shared memory approach using tiling size 16x16.





<u>Analysis</u>

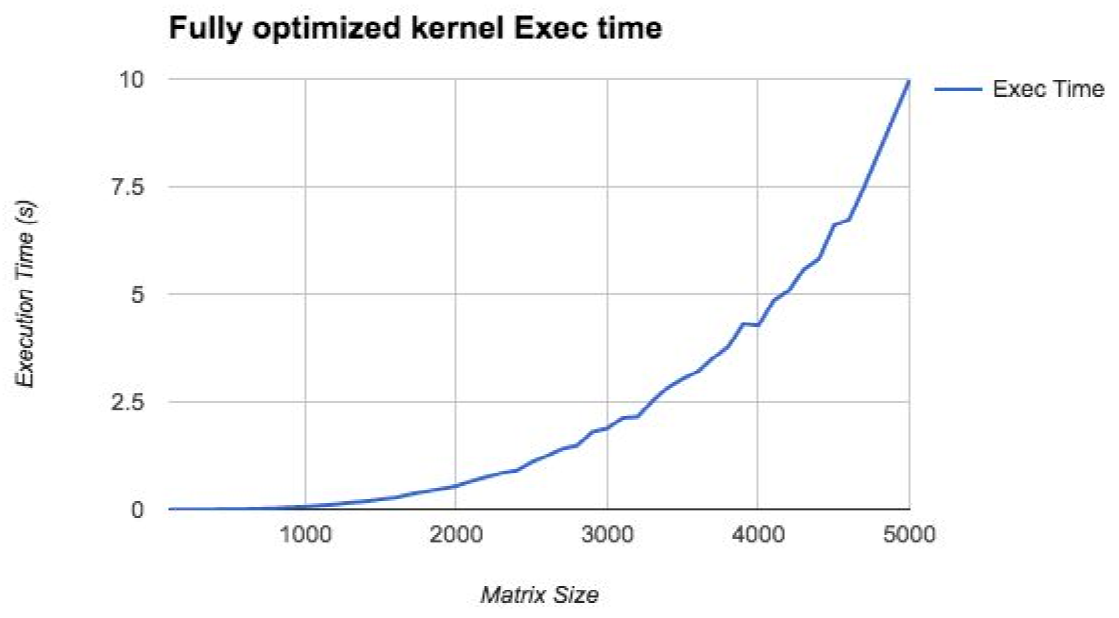
Shared memory approach helps to reduce the effect of uncoalesced memory accesses by first transferring the entries in global memory that will be accessed multiple times during the computation to shared memory, hence reducing the global memory access by a certain constant factor. In case of tiling of size 16x16, we observe a better performance in general in comparison to the basic kernel, where the shared memory approach achieves speedup of around 27.3x for large N values, while basic kernel only barely passes 20x speedup.

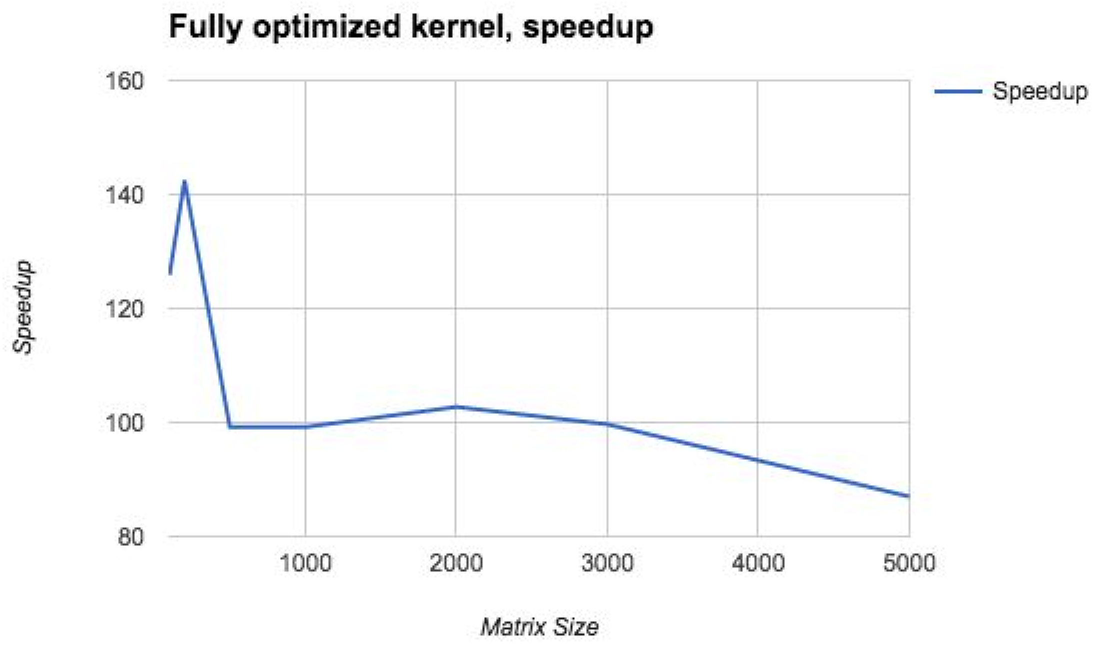
However, setting the tiling size to 32x32 degrades the performance significantly, down to 15x speedup. Hence there is a need to balance between improving the memory bandwidth usage through shared memory and occupancy. Also, larger tiles may require more expensive synchronization overhead, which may lead to poorer performance as observed.

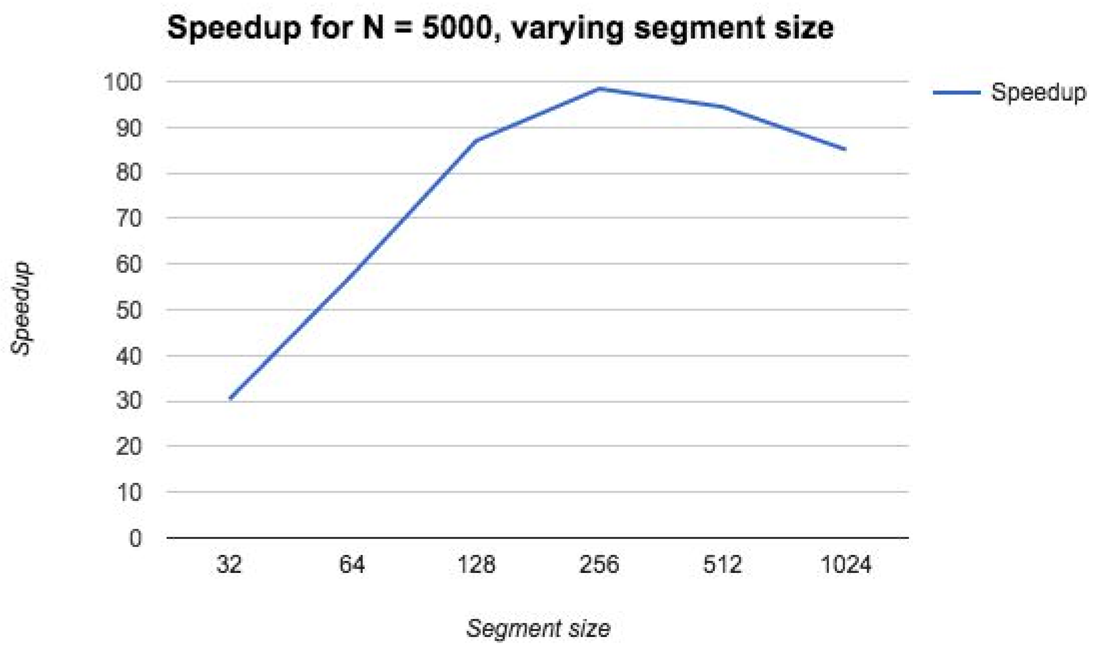
In comparison with coalescing memory accesses, we find that the shared memory approach offers less improvement in performance. Tiling approach does not perform as well because only d[i][k] (data along the k column) is being reused by the threads in the tile. Hence a better approach would be to only use one shared memory per segment of threads which are on the same row. This approach has the benefit of forcing the memory accesses to be coalesced as well. In the next section, we will show the results based on the improved approach combining the use from shared memory and enforcing memory coalescing.

3.5 Fully optimized kernel

Below is the benchmark profile of our fully optimized kernel code, exploiting shared memory effectively and making sure that memory accesses are coalesced. The following results are for segment size of 128 (where 128 consecutive threads share one prefetched d[i][k]).







Analysis
By combining the shared memory approach effectively, and ensuring that data are being accessed by each warp of threads in coalesced manner, we are able to achieve speedup of 102x for N = 2000 and 87.0x for N = 5000, significantly higher than the basic kernel.

We also analyze the most effective segment size to use, which controls how many consecutive threads share the same d[i][k]. By fixing the size of the matrix at N = 5000, we try different values of segment size and discover that the kernel has the most optimal performance at segment size of 256 where we achieve a speedup of almost 100x.

4. Conclusion

Kernel performance is very sensitive to the execution parameters used, as underutilization of bandwidth and suboptimal threads occupancy can lead to significant degradation of GPU performance. Choosing the right values for the thread per block, block sizes, and segment sizes requires careful profiling and benchmarking. Furthermore, some parameters work best for certain implementation, while other values would lead to better performance for others. However, the general principles of coalescing memory accesses and reducing global memory accesses through shared memory have direct implication on the practical speedup that we can obtain, as the experiment results have shown.