CSE 5441

Programming Assignment 2

Vineeth Thumma (thumma.6)

1) Base version that directly accesses the elements of arrays A and B from global memory

Code:

```
_global__ void test_kernel(int N, double *A, double *B, double *C)
{

// Block Index along x & y
int bx = blockIdx.x; int by = blockIdx.y;
//Thread Index along x & y
int tx = threadIdx.x; int ty = threadIdx.y;

// Row & Column in resultant matrix C computed by the thread in the block
int Row = by * TILE_WIDTH + ty;
int Column = bx * TILE_WIDTH + tx;

double Pvalue = 0;

// Accumulate dot product
for (int k=0; k<N; ++k)
   Pvalue += A[Row*N+k]*B[Column*N+k];

// write final value to global memory
C[Row*N+Column] = Pvalue;

}
```

Performance obtained: 11.8 GFLOPS (TILE_WIDTH of 8) and 4.5 GFLOPS (TILE_WIDTH of 32)

In this base version, we are accessing elements of A & B from global memory all the time because of which the performance was low.

Also, for this case the performance was better for a TILE_WIDTH of 8 as compared to TILE_WIDTH of 32 even though the SM might be fully occupied for TILE_WIDTH of 32.
The reason might me because of more Cache Hits in SM in the former case compared to latter.

2) Using Shared Memory

Code:

```
__global__ void test_kernel(int N, double *A, double *B, double *C)

{
// using shared memory for storing TILES corresponding to this block in A & B
__shared__ double As[TILE_WIDTH][TILE_WIDTH];
__shared__ double Bs[TILE_WIDTH][TILE_WIDTH+1];

// Block Index along x & y
int bx = blockIdx.x; int by = blockIdx.y;

//Thread Index along x & y
int tx = threadIdx.x; int ty = threadIdx.y;

// Row & Column in resultant matrix C computed by the thread in the block
int Row = by * TILE_WIDTH + ty;
int Column = bx * TILE_WIDTH + tx;

double Pvalue = 0;

// breaking kernel into 'N/TILE_WIDTH' phases where 'm' is the current phase
for(int m=0; m<N/TILE_WIDTH; ++m) {

   // Bring elements from A & B to their corresponding shared memory
   As[ty][tx] = A[Row*N+(m*TILE_WIDTH+tx)];
   Bs[ty][tx] = B[((bx*TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];

   //wait for the entire tile in A & B to be in shared memory
   __syncthreads();

   // Accumulate subset of dot product
   for (int k=0; k<TILE_WIDTH; ++k) {
      Pvalue +=  As[ty][k]*Bs[tx][k];

   }
   // wait for the entire values corresponding to this phase is computed
      __syncthreads();
 }
   // write final value to global memory
   C[Row*N+Column] = Pvalue;
}
```

Performance obtained: 96.9 GFLOPS (TILE_WIDTH of 32)

In Shared Memory version, we are fetching TILES of matrices A & B from global memory (Coalesced access) and put them in Shared Memory (Coalesced). We then do access from shared memory (coalesced access for As – all threads in a warp access same element & coalesced access for Bs- due to Padding) to compute the dot product and reuse them instead of fetching from global memory every time. This explains the performance increase compared to base version.

3) 2-way loop unrolling along i, for the shared-memory version

Code:

```
__global__ void test_kernel(int N, double *A, double *B, double *C)

{
// using shared memory for storing TILES corresponding to this block in A & B. Each block is responsible
for computing TILE [2i][j] & TILE [2i+1][j] in C
__shared__ double As0[TILE_WIDTH][TILE_WIDTH];
__shared__ double As1[TILE_WIDTH][TILE_WIDTH];
__shared__ double Bs[TILE_WIDTH][TILE_WIDTH+1];

// Block Index along x & y
int bx = blockIdx.x; int by = blockIdx.y;

//Thread Index along x & y
int tx = threadIdx.x; int ty = threadIdx.y;

// Rows & Column in resultant matrix C computed by the threads in the block
int Row = by * 2 * TILE_WIDTH + ty;
int Column = bx * TILE_WIDTH + tx;

double Pvalue0 = 0;
double Pvalue1 = 0;

// breaking kernel into 'N/TILE_WIDTH' phases where 'm' is the current phase
for(int m=0; m<N/TILE_WIDTH; ++m) {

  // Bring elements from A & B to their corresponding shared memory
  As0[ty][tx] = A[Row*N+(m*TILE_WIDTH+tx)];
  As1[ty][tx] = A[(Row+TILE_WIDTH)*N+(m*TILE_WIDTH+tx)];

  Bs[ty][tx] = B[((bx*TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];

  //wait for the entire tiles in A & B to be in shared memory
  __syncthreads();

  // Accumulate subset of dot products
  for (int k=0; k<TILE_WIDTH; ++k) {
    Pvalue0 +=  As0[ty][k]*Bs[tx][k];
    Pvalue1 +=  As1[ty][k]*Bs[tx][k];
   }
  // wait for the entire values corresponding to this phase is computed
    __syncthreads();
  }
  // write final values to global memory
  C[Row*N+Column] = Pvalue0;
  C[(Row+TILE_WIDTH)*N+Column] = Pvalue1;

}
```

<u>Performance obtained:</u> 127.5 GFLOPS (TILE_WIDTH of 32)

In this version, along with using shared memory to bring 2 TILES of A and 1 TILE of B, we are 2-way unrolling along ‘i’ . In this case the no of blocks along ‘x’ would be the same and blocks along ‘y’ is halved. Each block is responsible for computing two TILES in C. In this case, we are reducing global memory access for B by half, as one block is computing values of two TILES in C and we re-use the values in shared memory.
This explains performance jump as compared to **2)**

4) 2-way loop unrolling along j, for the shared-memory version

```
_global__ void test_kernel(int N, double *A, double *B, double *C)
{
// using shared memory for storing TILES corresponding to this block in A & B. Each block is responsible
for computing TILE [i][2j] & TILE [i][2j+1] in C
__shared__ double As[TILE_WIDTH][TILE_WIDTH];
__shared__ double Bs0[TILE_WIDTH][TILE_WIDTH+1];
__shared__ double Bs1[TILE_WIDTH][TILE_WIDTH+1];

// Block Index along x & y
int bx = blockIdx.x; int by = blockIdx.y;

//Thread Index along x & y
int tx = threadIdx.x; int ty = threadIdx.y;

// Row & Columns in resultant matrix C computed by the threads in the block
int Row = by * TILE_WIDTH + ty;
int Column = bx * 2 * TILE_WIDTH + tx;

double Pvalue0 = 0;
double Pvalue1 = 0;

// breaking kernel into 'N/TILE_WIDTH' phases where 'm' is the current phase
for(int m=0; m<N/TILE_WIDTH; ++m) {
   // Bring elements from A & B to their corresponding shared memory
   As[ty][tx] = A[Row*N+(m*TILE_WIDTH+tx)];

   Bs0[ty][tx] = B[((bx*2*TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];
   Bs1[ty][tx] = B[((bx*2*TILE_WIDTH+TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];

   //wait for the entire tiles in A & B to be in shared memory
   __syncthreads();

   // Accumulate subset of dot products
   for (int k=0; k<TILE_WIDTH; ++k) {
      Pvalue0 +=  As[ty][k]*Bs0[tx][k];
      Pvalue1 +=  As[ty][k]*Bs1[tx][k];
    }
    // wait for the entire values corresponding to this phase is computed
      __syncthreads();
   }
   // write final values to global memory
   C[Row*N+Column] = Pvalue0;
   C[Row*N+Column+TILE_WIDTH] = Pvalue1;
}
```

Performance obtained: 128.4 GFLOPS (TILE_WIDTH of 32)

In this version, along with using shared memory to bring 1 TILE of A and 2 TILES of B, we are 2-way unrolling along ‘j’ . In this case the no of blocks along ‘y’ would be the same and blocks along ‘x’ is halved compared to **2)**. Each block is responsible for computing two TILES in C. In this case, we are reducing global memory access for A by half, as one block is computing values of two TILES in C and we re-use the values in shared memory.
This explains performance jump as compared to **2)**

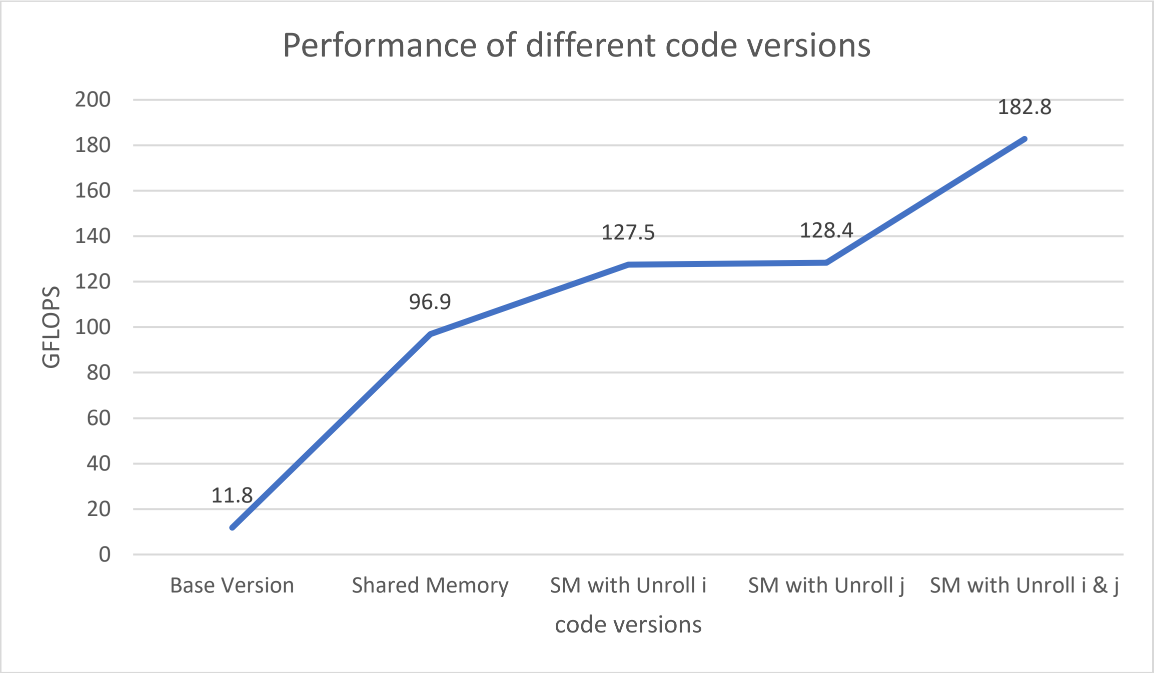
5) 2-way loop unrolling along i and j, for the shared-memory version

```
_global__ void test_kernel(int N, double *A, double *B, double *C)
{
// using shared memory for storing TILES corresponding to this block in A & B. Each block is responsible
for computing 4 TILES in C
__shared__ double As0[TILE_WIDTH][TILE_WIDTH];
__shared__ double As1[TILE_WIDTH][TILE_WIDTH];
__shared__ double Bs0[TILE_WIDTH][TILE_WIDTH+1];
__shared__ double Bs1[TILE_WIDTH][TILE_WIDTH+1];

// Block Index along x & y
int bx = blockIdx.x; int by = blockIdx.y;

//Thread Index along x & y
int tx = threadIdx.x; int ty = threadIdx.y;
// Rows & Columns in resultant matrix C computed by the threads in the block
int Row = by * 2 *TILE_WIDTH + ty;
int Column = bx * 2 * TILE_WIDTH + tx;

double Pvalue0 = 0;
double Pvalue1 = 0;
double Pvalue2 = 0;
double Pvalue3 = 0;

// breaking kernel into 'N/TILE_WIDTH' phases where 'm' is the current phase
for(int m=0; m<N/TILE_WIDTH; ++m) {

   // Bring elements from A & B to their corresponding shared memory
   As0[ty][tx] = A[Row*N+(m*TILE_WIDTH+tx)];
   As1[ty][tx] = A[(Row+TILE_WIDTH)*N+(m*TILE_WIDTH+tx)];

   Bs0[ty][tx] = B[((bx*2*TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];
   Bs1[ty][tx] = B[((bx*2*TILE_WIDTH+TILE_WIDTH+ty)*N)+(m*TILE_WIDTH+tx)];

   //wait for the entire tiles in A & B to be in shared memory
   __syncthreads();

   // Accumulate subset of dot products
   for (int k=0; k<TILE_WIDTH; ++k) {

      Pvalue0 +=  As0[ty][k]*Bs0[tx][k];
      Pvalue1 +=  As0[ty][k]*Bs1[tx][k];
      Pvalue2 +=  As1[ty][k]*Bs0[tx][k];
      Pvalue3 +=  As1[ty][k]*Bs1[tx][k];

   }
   // wait for the entire values corresponding to this phase to be computed
      __syncthreads();

   }
    // write final values in C computed by this block to global memory
   C[Row*N+Column] = Pvalue0;
   C[Row*N+Column+TILE_WIDTH] = Pvalue1;
   C[(Row+TILE_WIDTH)*N+Column] = Pvalue2;
   C[(Row+TILE_WIDTH)*N+Column+TILE_WIDTH] = Pvalue3;
}
```

Performance obtained: 182.8 GFLOPS (TILE_WIDTH of 32)

In this version, along with using shared memory to bring 2 TILES of A and 2 TILES of B, we are 2-way unrolling each along ‘i’ and ‘j’. In this case the no of blocks along ‘y’ is halved and blocks along ‘x’ is halved compared to **2)**. Each block is responsible for computing four TILES in C. In

this case, we are reducing global memory access for A by half and B by half, as one block is computing values of four TILES in C and we re-use the values in shared memory.
This explains performance jump as compared to **3)** and **4)**


Performance of different code versions
GFLOPS
Base Version
Shared Memory
SM with Unroll i
SM with Unroll j
SM with Unroll i & j
code versions