HIGH PERFORMANCE PARALLEL PROGRAMMING (CS61064)

Soumyajit Dey
CSE, IIT Kharagpur

Recap: Memory Spaces

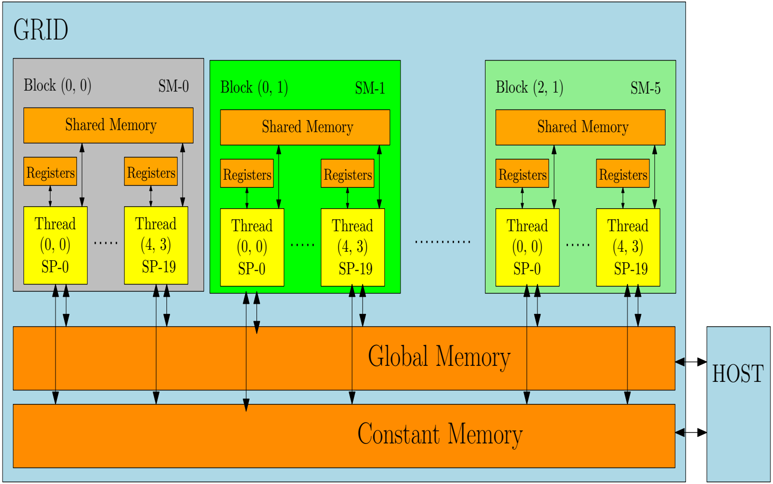


Figure: Global Memory Accesses

Recap: Memory Spaces

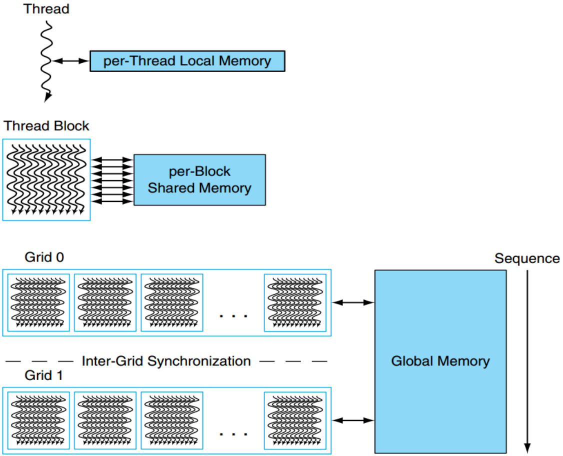


Figure: Memory Access Scopes

Recap: Memory Spaces

Latency of accesses differ for different memory spaces

- Global Memory (accessible by all threads) is the slowest
- Shared Memory (accessible by threads in a block) is very fast.
- Registers (accessible by one thread) is the fastest.

Warp Requests to Memory

- A warp typically requests 32 aligned 4 byte words in one memory transaction.
- Reducing number of global memory transactions by warps is one of the keys for optimizing execution time
- Efficient memory access expressions must be designed by the user for the same

Access Expressions

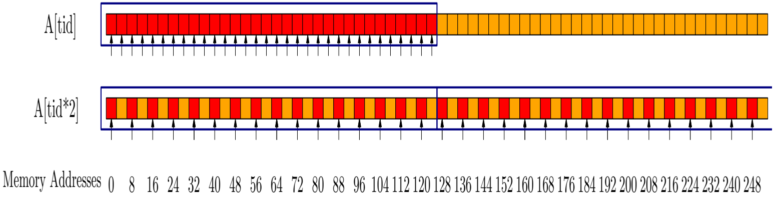


Figure: Global Memory Accesses

Using Shared Memory

- Applications typically require different threads to access the same data over and over again (data reuse)
- Redundant global memory accesses can be avoided by loading data into shared memory.

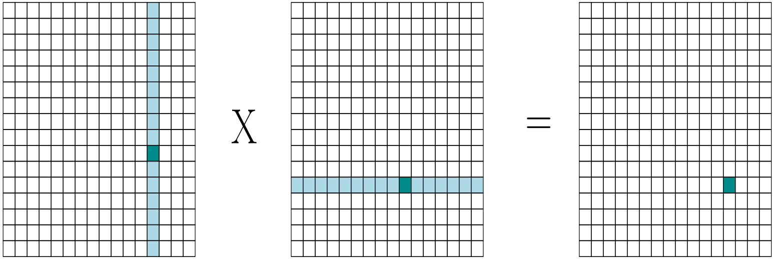
Recap: Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float*
   d_N, float* d_P, int N){
int i=blockIdx.y*blockDim.y+threadIdx.y;
int j=blockIdx.x*blockDim.x+threadIdx.x;
if ((i<N) && (j<N)) {
  float Pvalue = 0.0;
  for (int k = 0; k < N; ++k) {
     Pvalue += d_M[i*N+k]*d_N[k*N+j];
  }
  d_P[i*N+j] = Pvalue;
 }
}
```

Recap Matrix Multiplication Kernel

- Number of threads launched is equal to the number of elements in the matrix
- The same row and column is accessed multiple times by different threads.
- Redundant global memory accesses are a bottleneck to performance

Recap: Matrix Multiplication Kernel



Total Mem. accesses required

$= N^2\,(\mathrm{N} + \mathrm{N}/32)$

$\approx N^3$

Figure: Number of memory accesses

Matrix Multiplication Kernel using Tiling

An alternative strategy is to use shared memory for reducing global memory traffic

- Partition the data into subsets called tiles so that each tile fits into shared memory
- Threads in a block collaboratively load tiles into shared memory before they use the elements for the dot-product calculation

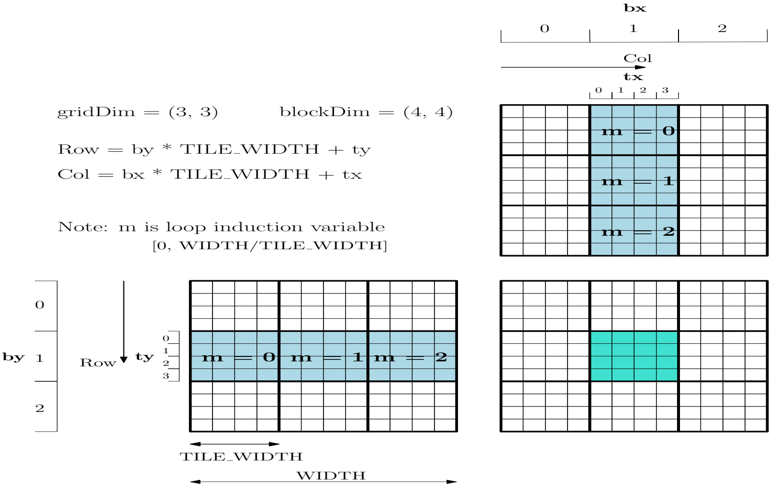


Figure: Access Expressions

Matrix Multiplication Kernel using Tiling


```
__global__
void MatrixMulKernel(float* d_M, float*
    d_N, float* d_P,int Width) {.

     __shared__ float Mds[TILE_WIDTH][
        TILE_WIDTH];
     __shared__ float Nds[TILE_WIDTH][
        TILE_WIDTH];

      int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;
```

```
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;
  for (int m = 0; m < Width/TILE_WIDTH;
      ++m) {
   Mds[ty][tx] = d_M[Row*Width + m*
      TILE_WIDTH + tx];
   Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty
      )*Width + Col];
   __syncthreads();
   for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += Mds[ty][k] * Nds[k][tx];
   __syncthreads();
 }
 d_P[Row*Width + Col] = Pvalue;
 }
```

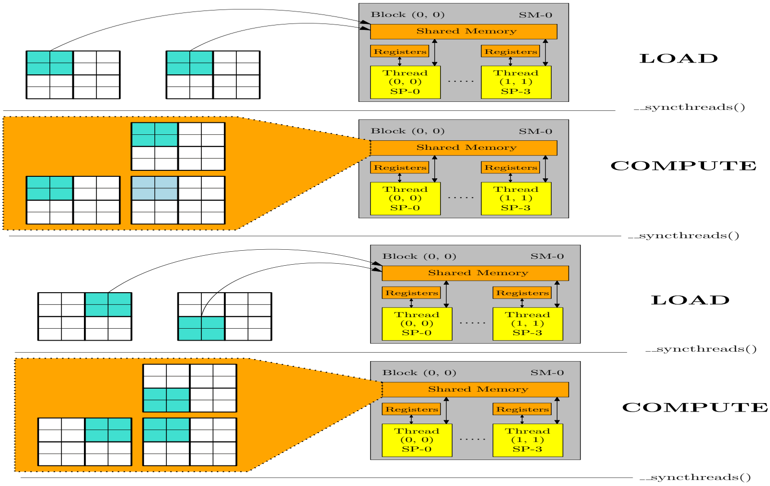


Figure: Load and compute tiles in shared memory

Mem. accesses for computing a tile in C = (# Mem. accesses to load a tile) **x** (# Tiles to load from A & B)= (W/32 x W) x (2N/W)

Total Mem. Accesses = (# Mem. accesses for computing a tile in C) x (# Tiles)
= (W/32 x W) x (2N/W) x (N^2/W^2)
= ($N^3/16W$)

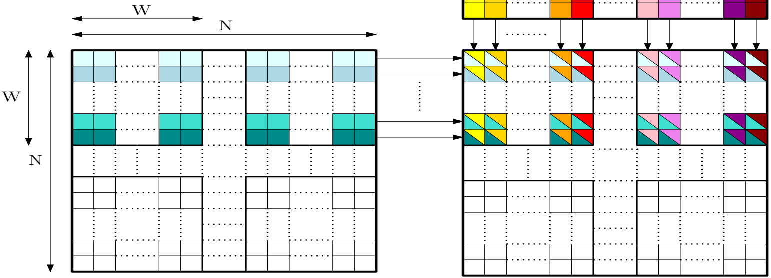


Figure: Number of memory accesses

1-D Convolution

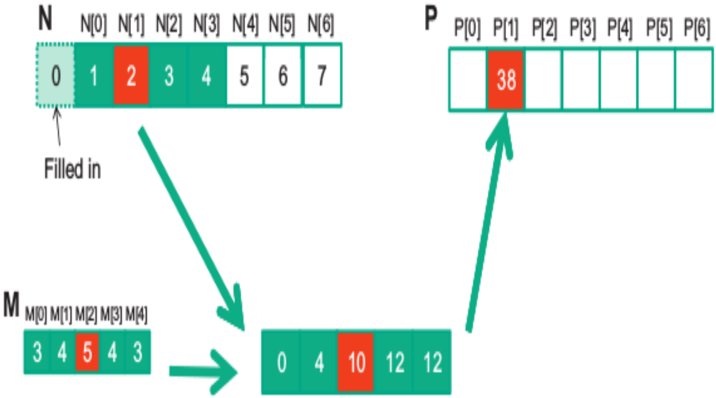


Figure: 1D Convolution

1-D Convolution

```
__global__
void convolution_1D_basic_kernel(float
    *N, float *M, float *P,
int Mask_Width,
int Width) {
int i=blockIdx.x*blockDim.x+threadIdx.
   x;
float Pvalue = 0;
int N_start_point=i-(Mask_Width/2);
for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 &&
        N_start_point + j < Width)
 Pvalue+=N[N_start_point + j]*M[j];
 }
P[i] = Pvalue;
}
```

1-D Convolution with tiling

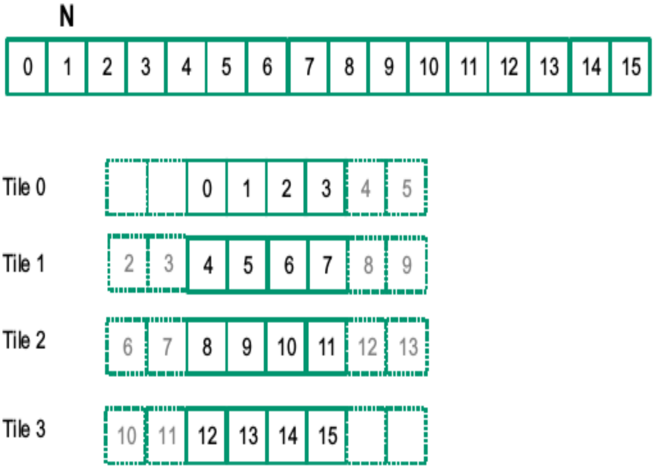


Figure: 1D Convolution

1-D Convolution with tiling

```
__global__
void convolution_1D_Tiling(float *N,
    float *P, int Mask_Width,
int Width){
  int i = blockIdx.x*blockDim.x +
      threadIdx.x;
  __shared__ float N_ds[TILE_SIZE +
      MAX_MASK_WIDTH -1];
  int n = Mask_Width/2;
  int halo_index_left = (blockIdx.x - 1)*
      blockDim.x + threadIdx.x;
  if(threadIdx.x>=blockDim.x-n)
    N_ds[threadIdx.x-(blockDim.x - n)]=
    (halo_index_left < 0)?0:N[
        halo_index_left];

  N_ds[n + threadIdx.x]=N[blockIdx.x*
      blockDim.x+threadIdx.x];
```

```
int halo_index_right=(blockIdx.x+1)*
   blockDim.x+threadIdx.x;
if (threadIdx.x<n)
N_ds[n+blockDim.x+threadIdx.x] =
(halo_index_right>=Width)?0:N[
   halo_index_right];

__syncthreads();
float Pvalue = 0;

for(int j=0;j<Mask_Width;j++) {
    Pvalue+=N_ds[threadIdx.x+j]*M[j];
}
P[i] = Pvalue;
}
```