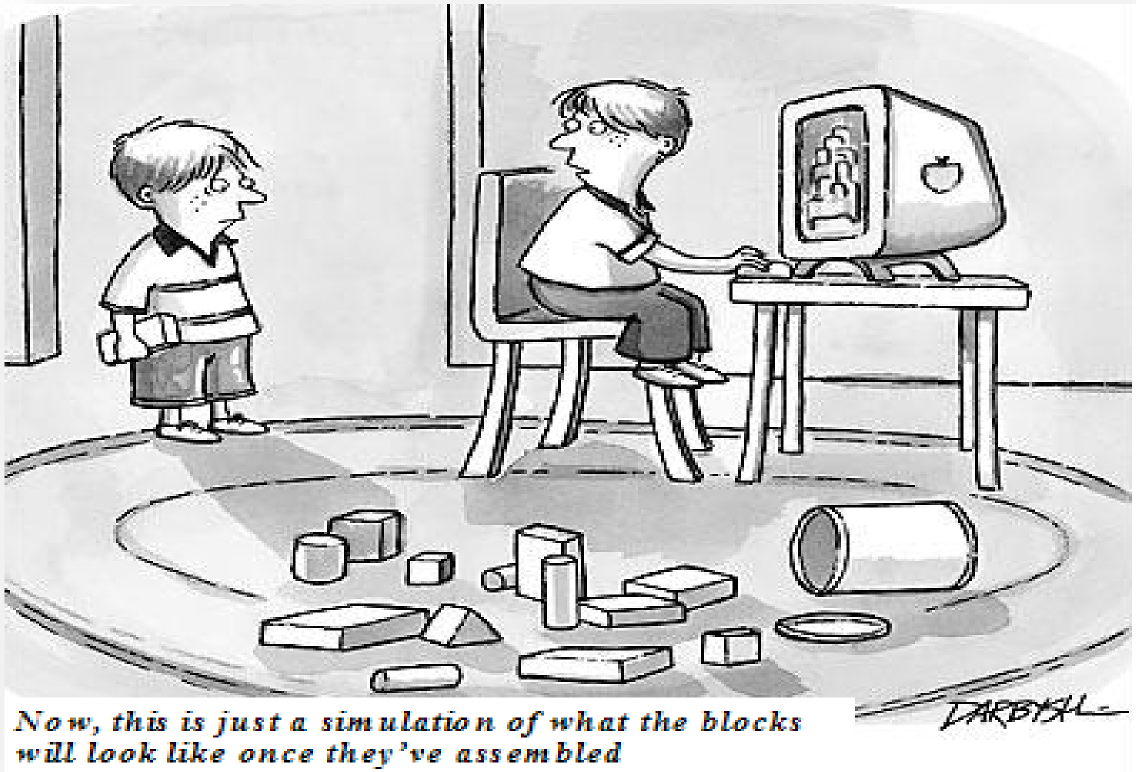
ECE569
Module 21



Now, this is just a simulation of what the blocks will look like once they've assembled

- Matrix Multiplication

Matrix Multiplication

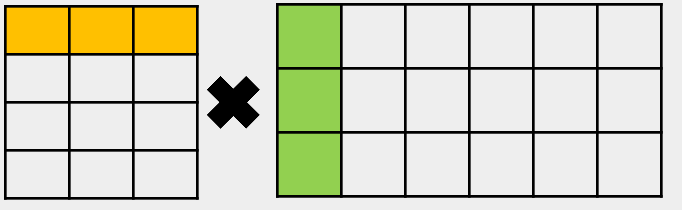
Goal: *Learn ways to reduce the limiting effect of memory bandwidth on parallel kernel performance*

- **This module:**
  - Global memory usage
  - Naïve implementation
  - Expose performance bottleneck
- **Later**
  - understand the design of a tiled parallel algorithm for matrix multiplication
  - Shared memory
    - Loading a tile
    - Phased execution
  - Barrier Synchronization

Matrix Multiplication

- **Vector multiplication**
  - One Row in M with One Column in N
  - Between M(r1,c1) and N(r2,c2), generates P(r1,c2) matrix

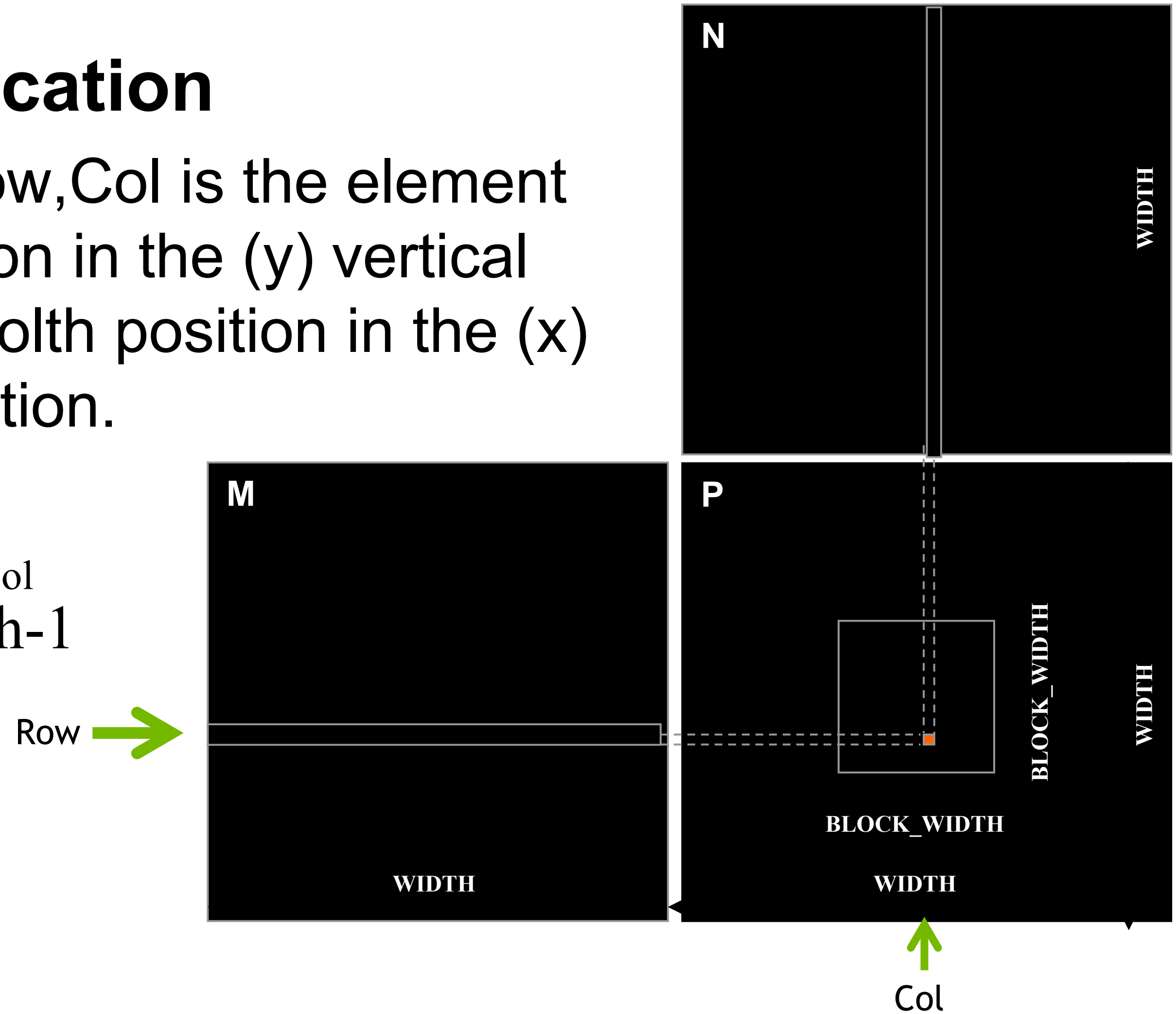
```
// Multiplying matrices M and N and
// storing result in P matrix
for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
     for(k=0; k<c1; ++k)
     {
      P[i][j]+=M[i][k]*N[k][j];
     }
```



Matrix Multiplication (Square Matrix!)

- **Vector multiplication**
  - Convention: Row,Col is the element at Rowth position in the (y) vertical direction and Colth position in the (x) horizontal direction.

$P_{Row,Col}=\Sigma M_{Row,k} * N_{k,Col}$
Where k = 0,1,…Width-1



$P_{Row,Col}$ is the inner product of the Row^{th} row of M and the Col^{th} column of N

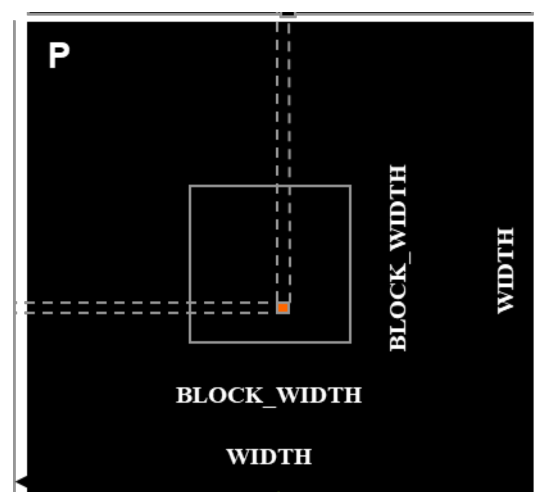
P1,5 = M1, 0*N0,5 + M1,1*N1,5 + M1,2*N2,5 +…. + M1,Width-1*NWidth-1,5

Basic Matrix Multiplication (Square Matrix!)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {

  // Calculate the row index of the P element and M

  int Row =

  // Calculate the column index of P and N

  int Col =








}
```

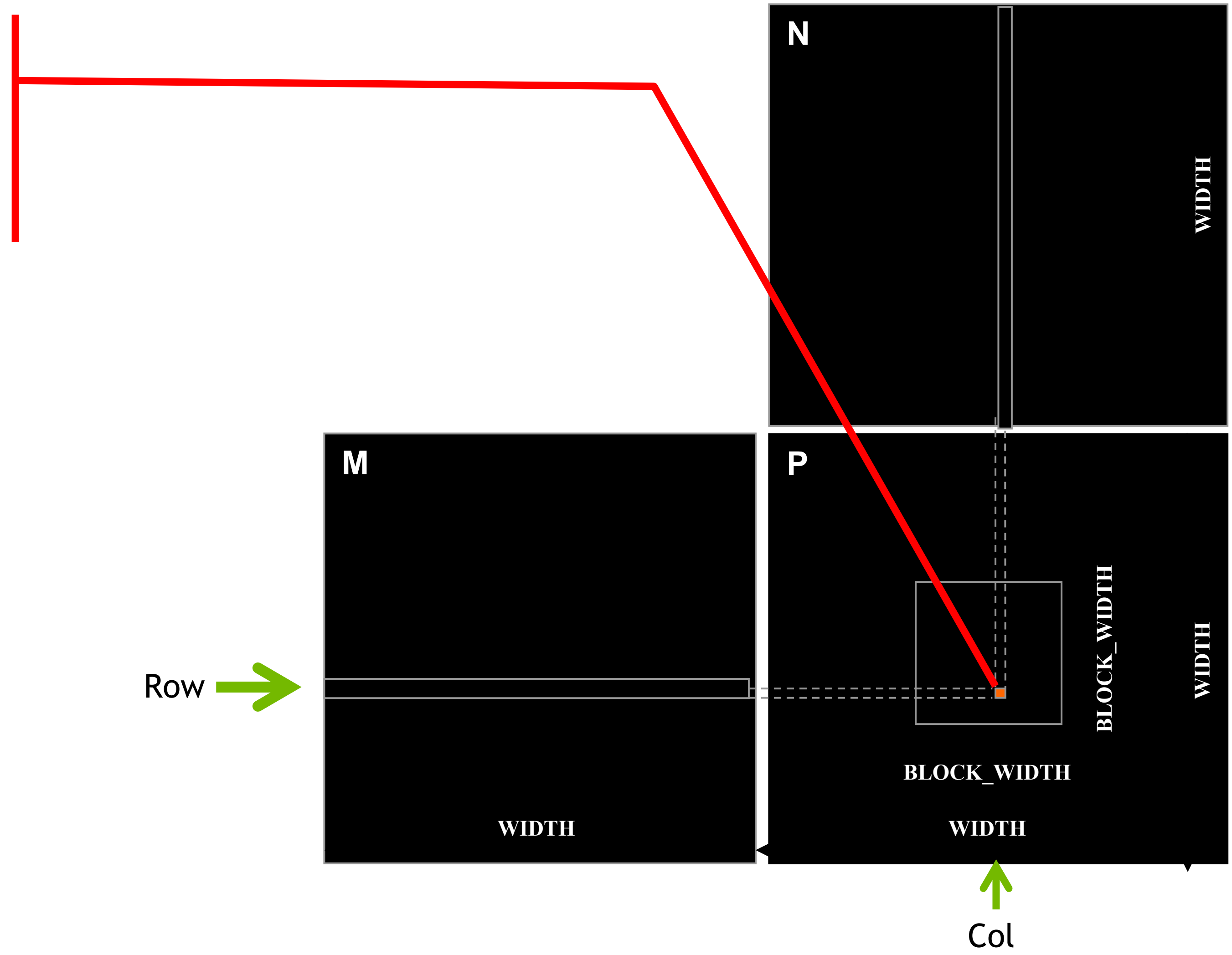
Basic Matrix Multiplication (Square Matrix!)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {

  // Calculate the row index of the P element and M

  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N

  int Col = blockIdx.x*blockDim.x+threadIdx.x;

}
```



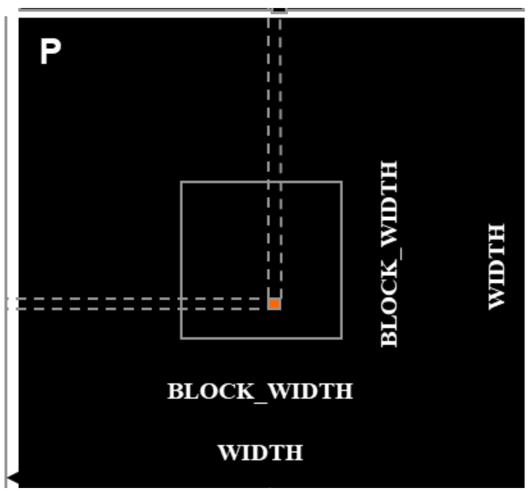
Mapping thread to a P(Row,Col)

- P[???]



Basic Matrix Multiplication (Square Matrix!)

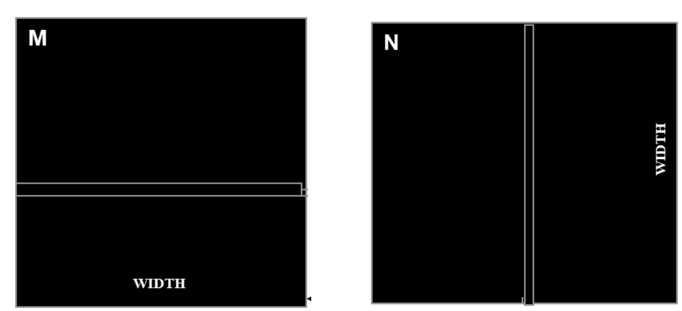
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  // Boundary condition
  if (                                              ) {

    // each thread computes one element of the block sub-matrix
    // in P[Row*Width+Col]
```




```
}
```

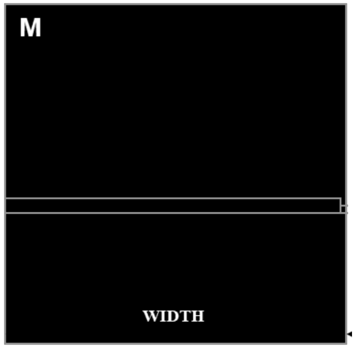
Basic Matrix Multiplication (Square Matrix!)

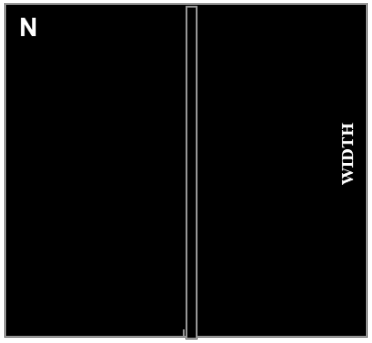
```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    // each thread computes one element of the block sub-matrix
    // in P[Row*Width+Col]
    // Insert multiplication expression in a loop



}
```



Basic Matrix Multiplication (Square Matrix!)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```





Basic Matrix Multiplication (Square Matrix!)

```
if ((Row < Width) && (Col < Width)) {
  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; ++k) {
    Pvalue += M[Row*Width+k]*N[k*Width+Col];
  }
  P[Row*Width+Col] = Pvalue;
}
```

```
P100: 9300GFLOPS, 720GB/sec
How far are we from peak FLOPS for the matrix multiplication
with global memory?
```

Basic Matrix Multiplication (Square Matrix!)

```
if ((Row < Width) && (Col < Width)) {
  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; ++k) {
    Pvalue += M[Row*Width+k]*N[k*Width+Col];
  }
  P[Row*Width+Col] = Pvalue;
}
```

2 Global memory accesses
1 floating point multiply operation
1 floating point addition

1 memory access per FP operation
P100: 9300GFLOPS, 720GB/sec

(720GB/sec)/(4B/FPop) = 180GFLOPS max out of 9300
Massive under utilization!

Next

- **Tiling method**