# HIGH PERFORMANCE PARALLEL PROGRAMMING (CS61064)

Soumyajit Dey
CSE, IIT Kharagpur

# Multi dimensional grid, block declaration

Consider the following host side code

```
dim3 X(2, 2, 1);
dim3 Y(4, 2, 2);
vecAddKernel<<<X, Y>>>(..);
```

The memory layout thus created in device when the kernel is launched is shown next

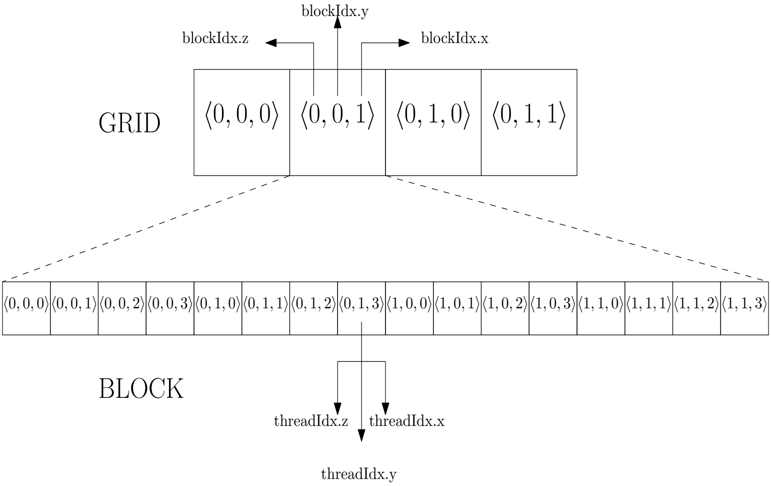
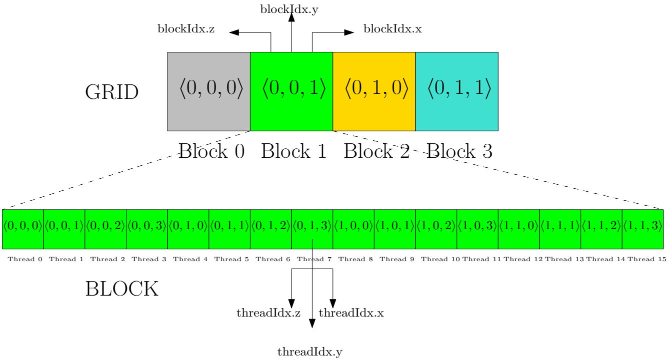



Figure: Grids and Blocks

| | Col 0 | Col 1 | Col 2 | Col 3 | Col 4 | Col 5 | Col 6 | Col 7 |
|---|---|---|---|---|---|---|---|---|
| Row 0 | | | | | | | | |
| Row 1 | | | | | | | | |
| Row 2 | | | | | | | | |
| Row 3 | | | | | | | | |
| Row 4 | | | | | | | | |
| Row 5 | | | | | | | | |
| Row 6 | | | | | | | | |
| Row 7 | | | | | | | | |

Figure: 2D Matrix




blockNum = blockIdx.z * (gridDim.x * gridDim.y) + blockIdx.y * gridDim.x + blockIdx.x

threadNum = threadIdx.z * (blockDim.x * blockDim.y) + threadIdx.y * (blockDIm.x) + threadIdx.x

globalThreadId = blockNum * (blockDim.x * blockDim.y * blockDim.z) + threadNum

Figure: Global Thread IDs

# Relations among variables

```
blockNum = blockIdx.z * (gridDim.x *
   gridDim.y) + blockIdx.y * gridDim.x
    + blockIdx.x;
threadNum = threadIdx.z * (blockDim.x
   * blockDim.y) + threadIdx.y * (
   blockDim.x) + threadIdx.x;
globalThreadId = blockNum * (blockDim.
   x * blockDim.y * blockDim.z) +
   threadNum;
```

| | Col 0 | Col 1 | Col 2 | Col 3 | Col 4 | Col 5 | Col 6 | Col 7 |
|---|---|---|---|---|---|---|---|---|
| Row 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Row 1 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Row 2 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| Row 3 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| Row 4 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| Row 5 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| Row 6 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| Row 7 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

i = globalThreadId / NumCols　　　j = globalThreadId % NumCols

NumRows * NumCols = gridDim.x * gridDim.y * gridDim.z * blockDim.x * blockDim.y * blockDim.z

Figure: Mapping Threads to Matrix

# Mapping between kernels and data

The CUDA programming interface provides support for mapping kernels of any dimension (upto 3) to data of any dimension

- Mapping a 3D kernel to 2D kernel results in complex memory access expressions.
- Makes sense to map 2D kernel to 2D data and 3D kernel to 3D data

$$\text{NumCols} = \text{blockDim.x} * \text{gridDim.x}$$
$$\text{NumRows} = \text{blockDim.y} * \text{gridDim.y}$$

$gridDim = \langle 3, 2 \rangle$ $\quad$ $blockDim = \langle 5, 4 \rangle$

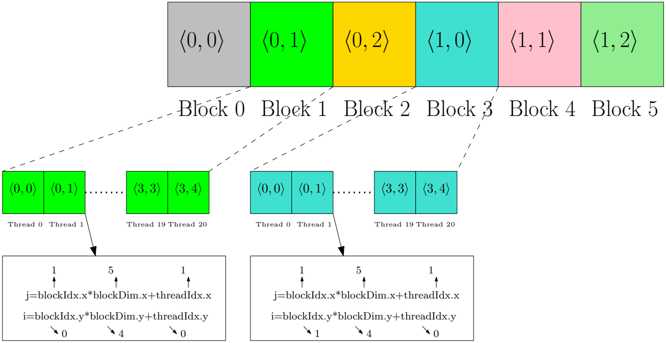



Figure: Two Dimensional Kernel

# 8 X 15 Matrix

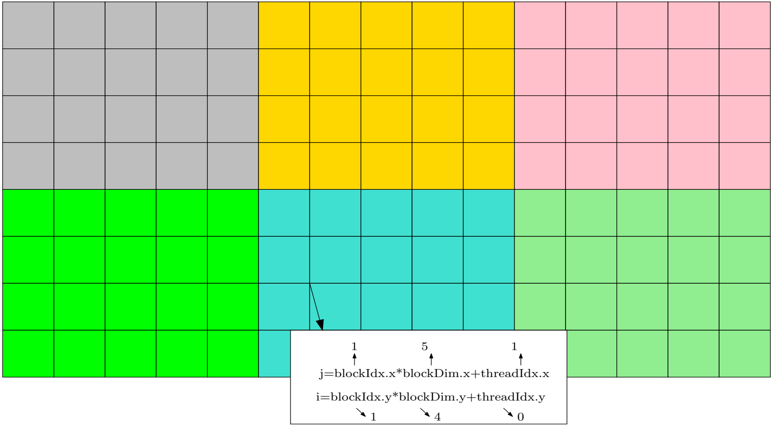



Figure: Two Dimensional Kernel-Data Mapping

$$nX = \text{blockDim.x} * \text{gridDim.x}$$
$$nY = \text{blockDim.y} * \text{gridDim.y}$$
$$nZ = \text{blockDim.z} * \text{gridDim.z}$$

$gridDim = \langle 2, 2, 2\rangle$ $blockDim = \langle 5, 4, 3\rangle$

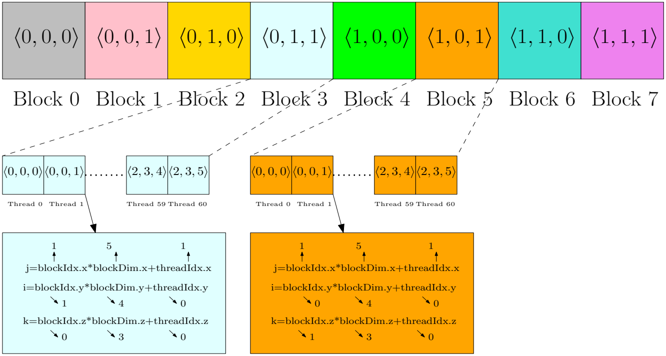
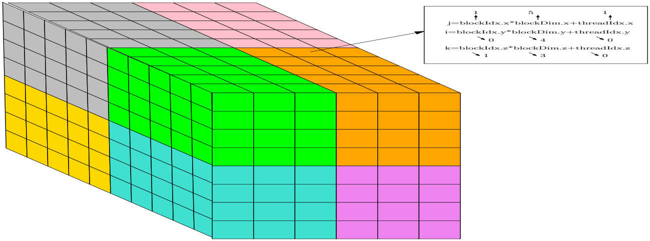



Figure: Three Dimensional Kernel




8 X 15 Matrix

Figure: Three Dimensional Kernel-Data Mapping

# Matrix Multiplication (CPU only)

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

   for(int i=0;i<N;i++)
     for(int j=0;j<N;j++) {
       float Pvalue=0.0;
       for (int k = 0; k < N; ++k) {
         Pvalue += M[i][k]*N[k][j];
       }
      P[i][j] = Pvalue;
    }
}
```

# Matrix Multiplication Host Program

```
int main()
{
  int size = 16*16;
  cudaMemcpy(d_M, M, size*sizeof(float),
  cudaMemcpyHostToDevice);
  cudaMemcpy(d_N, N, size*sizeof(float),
  cudaMemcpyHostToDevice);
  dim3 grid(2,2,1);
  dim3 block(8,8,1);
  int N=16; //N is the number of rows and
      columns
  MatrixMulKernel<<<grid,block>>>(d_M,d_N
     ,d_P,N)
  cudaMemcpy(P, d_P, size*sizeof(float),
  cudaMemcpyDeviceToHost);

}
```

# 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;
 }
}
```

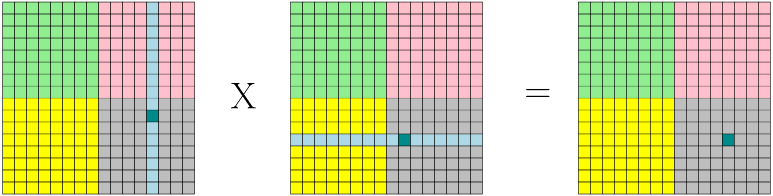



Figure: Matrix Multiplication

$$d\_P[i * N + j] = \sum_{k=0}^{N} d\_M[i * N + k] * d\_N[k * N + j]$$

# Synchronization

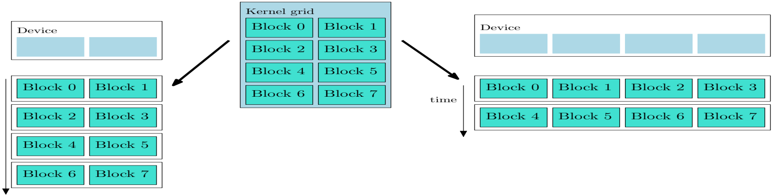



Figure: Mapping Blocks to Hardware

- Each block can execute in any order relative to other blocks.
- Lack of synchronization constraints between blocks enables scalability.

# Synchronization

- Synchronization constraints can be enforced to threads inside a thread block.
- Threads may co-operate with each other and share data with the help of local memory (more on this later)
- CUDA construct __synchthreads() is used for enforcing synchronization.

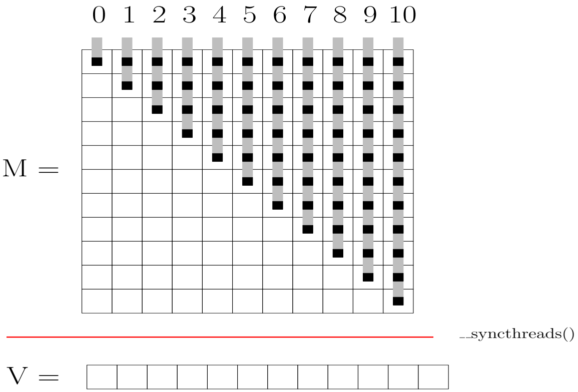



Figure: Input: A 11 X 11 matrix, Output: A vector of size 12 where each element represents the column sums and the last element represents the sum of the column sums.

# Synchronization Host Program

```
int main()
{
  int N=11;
  int size_M=N*N;
  int size_V=N+1;

  cudaMemcpy(d_M,M,size_M*sizeof(float),
  cudaMemcpyHostToDevice);
  cudaMemcpy(d_V, V, size_V*sizeof(float)
      ,
  cudaMemcpyHostToDevice);
  dim3 grid(1,1,1);
  dim3 block(11,1,1);
  sumTriangle<<<grid,block>>>(d_M,d_V,N);
  cudaMemcpy(V,d_V,size_V*sizeof(float),
  cudaMemcpyDeviceToHost);

}
```

# Kernel

```
__global__
void sumTriangle(float* M, float* V, int
    N){

int j=threadIdx.x;
float sum=0.0;
for (int i=0;i<j;i++)
    sum+=M[i*N+j];

V[j]=sum;
__syncthreads();
```

# Kernel



```
if(j == N-1)
{    sum = 0.0;
     for(i=0;i<N;i++)
          sum =sum + V[i];
     V[N] = sum;
}
```

Once each thread finishes computing sum across columns, the total sum is computed by the last thread.

# Synchronization Program Variant I

Modification: Only elements at odd indices are summed.

```
__global__
void sumTriangle(float* M, float* V,
    int N){

int j=blockIdx.x;
float sum=0.0;
for (int i=0;i<j;i++)
    if(i%2) // Check for odd indices
        sum+=M[i*N+j];

V[j]=sum;
__syncthreads();
```

# Synchronization Program Variant I

Addition still carried out by the last thread.

```
if(j == N)
{
    sum = 0.0;
    for(i=0;i<N;i++)
        sum =sum + V[i];
    V[N+1] = sum;
}
```

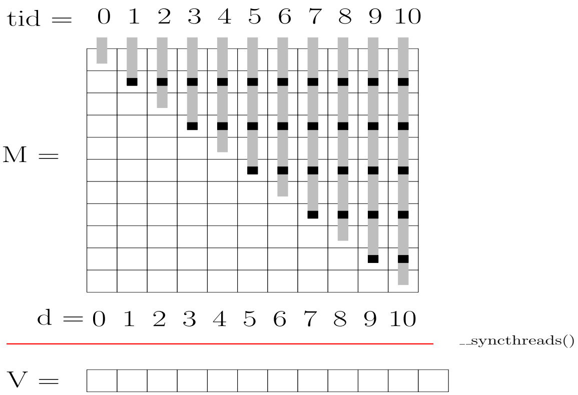



Figure: A variant of SumTriangle where only the elements at odd indices of a column are added

# Synchronization Program Variant II



Modification: Consider summing all indices again. But use all threads for final reduction.

```
__global__
void sumTriangle(float* M, float* V,
    int N){

int j=threadIdx.x;
float sum=0.0;
for (int i=0;i<j;i++)
sum+=M[i*N+j];

V[j]=sum;
__syncthreads();
```

# Synchronization Program Variant II



Reduction possible since addition is an associative operation.

```
    for(unsigned int s=1;s<N;s*= 2)
    {
        if (j %(2*s)==0 && j+s < N)
            V[j]+=V[j+s];
        __syncthreads();
    }
}
```

Once each thread finishes computing sum across columns, the total sum is computed by all the threads.

# Reduction

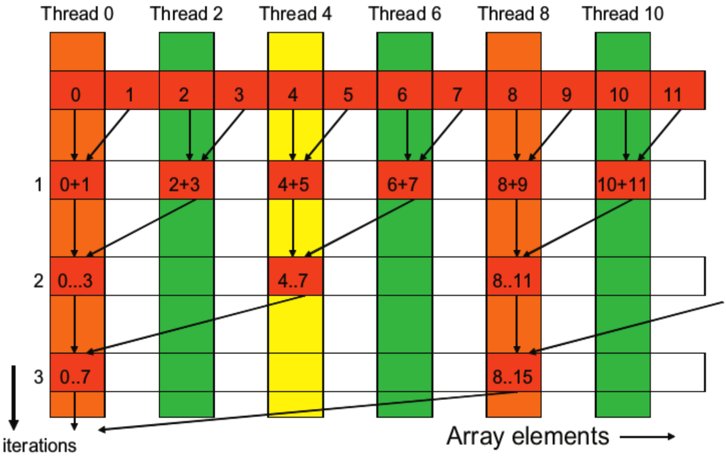



Figure: Reducing an array of 12 elements