Improved performance through GPU shared memory

Recall: Matrix multiplication

$$\begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} & A_{4,1} \\ A_{1,2} & A_{2,2} & A_{3,2} & A_{4,2} \\ A_{1,3} & A_{2,3} & A_{3,3} & A_{4,3} \\ A_{1,4} & A_{2,4} & A_{3,4} & A_{4,4} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{2,1} & B_{3,1} & B_{4,1} \\ B_{1,2} & B_{2,2} & B_{3,2} & B_{4,2} \\ B_{1,3} & B_{2,3} & B_{3,3} & B_{4,3} \\ B_{1,4} & B_{2,4} & B_{3,4} & B_{4,4} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{2,1} & C_{3,1} & C_{4,1} \\ C_{1,2} & C_{2,2} & C_{3,2} & C_{4,2} \\ C_{1,3} & C_{2,3} & C_{3,3} & C_{4,3} \\ C_{1,4} & C_{2,4} & C_{3,4} & C_{4,4} \end{bmatrix}$$

Each element of C is the result of combining a row of A and a column of B:

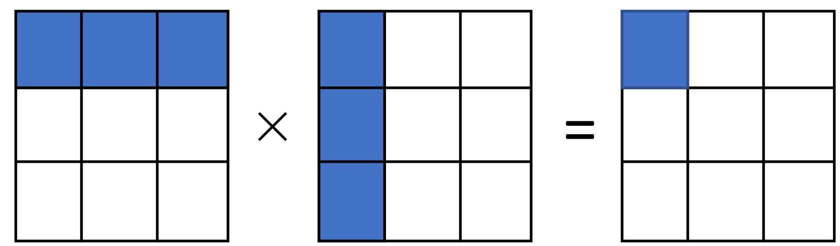
$$C_{i,j} = \sum_{k=1}^{n} A_{k,i} \times B_{j,k}$$

Basic CUDA implementation

```
__global__ void kernel(float* Md, float* Nd, float* Pd, int width) {
        int row = blockIdx.y*TILE_WIDTH + threadIdx.y;  //calculate indices of element
        int col = blockIdx.x*TILE_WIDTH + threadIdx.x;

        if(row >= width || col >= width)            //check that indices are in bounds
                return;

        float tmp = 0;  //local variable in which to accumulate the answer
        for(int k=0; k < width; ++k)
                tmp += Md[row*width + k] * Nd[k*width+col];
        Pd[row*width+col] =  tmp;
}
```

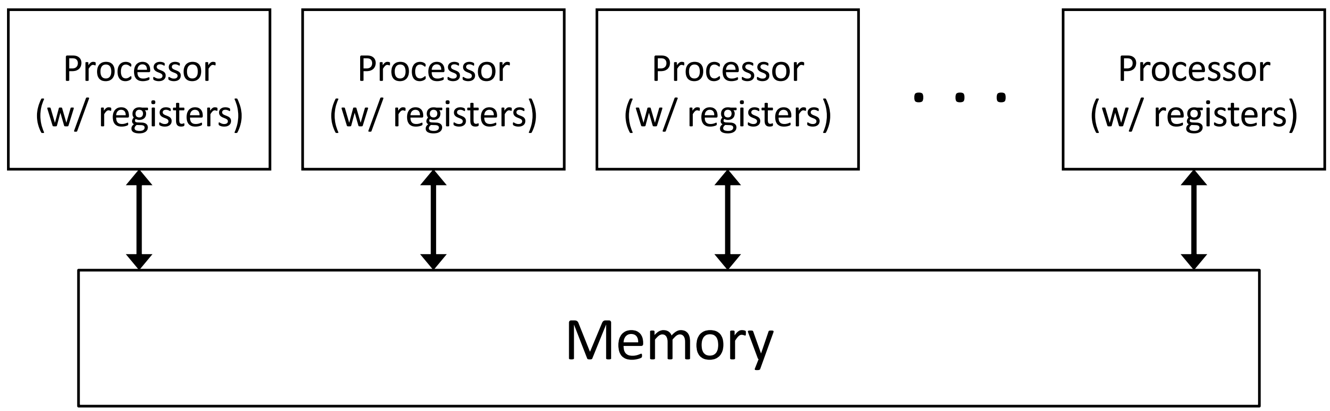
Tiling

- Divide matrix into “tiles” (submatrices)
- A tile of the output matrix depends on a row and a column of tiles respectively from the input matrices



- Reduce memory demand by computing the elements of an output tile together

Memory in CUDA so far



CUDA memory types

- Processors split between multiprocessors (each runs 1 block)
- Multiple types of memory
- Registers are per processor
- Shared memory is per multiprocessor
- Multiprocessors also have caches for constant and texture memory

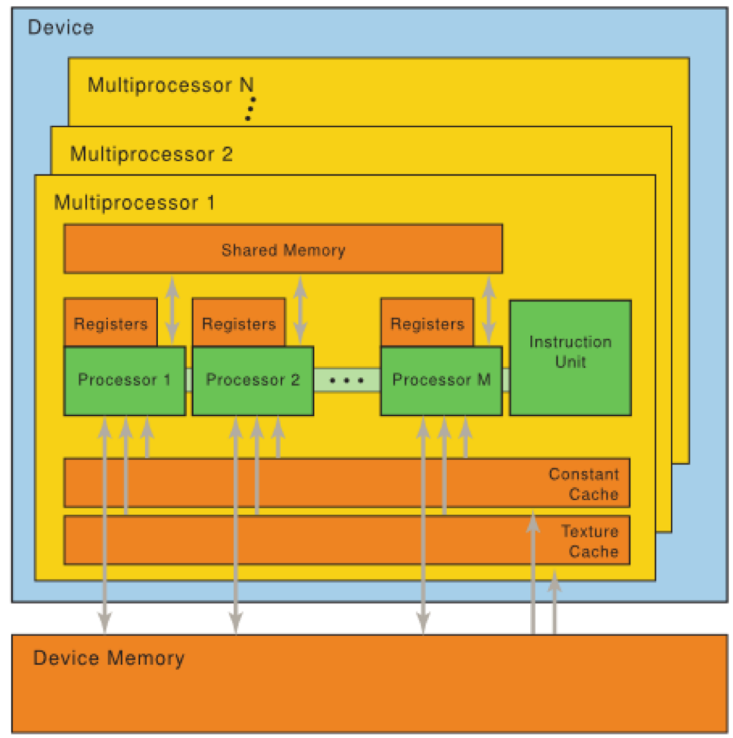


Image: https://docs.nvidia.com/cuda/parallel-thread-execution

Idea of shared memory implementation

- Use shared memory as programmer managed cache
  - Each block computes a tile of the output matrix
  - Each thread of the block is responsible for one element of the tile
  - Load one tile of each input matrix (1 element per thread), each thread computes their contribution to the output, and then move on to next tiles

Idea of shared memory implementation

- Use shared memory as programmer managed cache
  - Each block computes a tile of the output matrix
  - Each thread of the block is responsible for one element of the tile
  - Load one tile of each input matrix (1 element per thread), each thread computes their contribution to the output, and then move on to next tiles

- Between steps, need

  `__syncthreads();  //blocks until all threads in the block reach the call`

Writing the tiled kernel

```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
        …
}
```

Step 1: Allocate the cache to store a tile of each matrix

Writing the tiled kernel

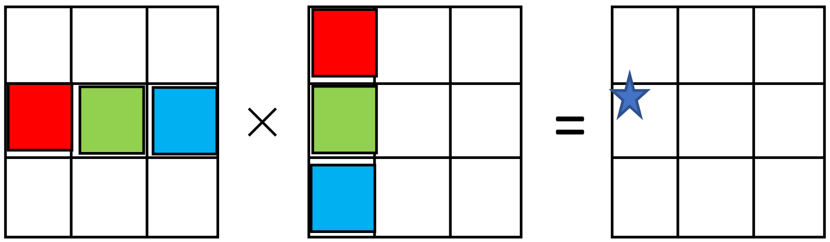
```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        …
        int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
        int col = blockIdx.x * TILE_WIDTH + threadIdx. x;

        int num_tiles = (width+TILE_WIDTH-1)/TILE_WIDTH;
        …
}
```

Step 2: Figure out the index for which this thread is responsible

(Block and tile have the same width)

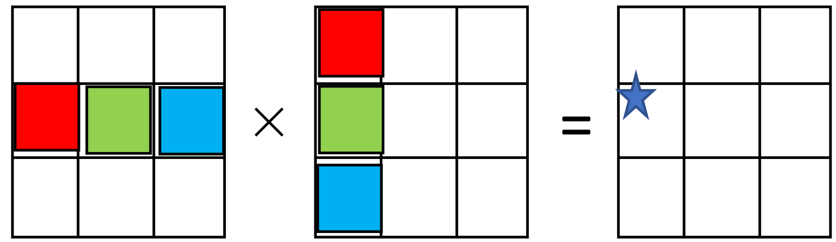
Figure out the width of the matrix in tiles

Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {

}
```

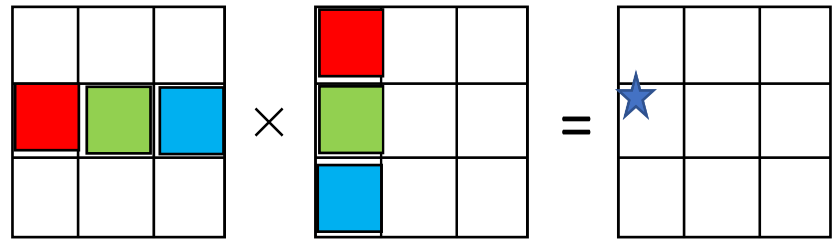
Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {
        if(/* in bounds */)
                Mds[threadIdx.y][threadIdx.x] = …;
        if(/* in bounds */)
                Nds[threadIdx.y][threadIdx.x] = …;
         __syncthreads();




}
```

Step 3a: Load cell of
cached submatrices
(then wait)

Step 3: Main loop



```
float tmp = 0;
for (int m=0; m < num_tiles; m++) {
        if(/* in bounds */)
                Mds[threadIdx.y][threadIdx.x] = …;
        if(/* in bounds */)
                Nds[threadIdx.y][threadIdx.x] = …;
         __syncthreads();

        for(k=0; k < TILE_WIDTH; k++)
                tmp += …
         __syncthreads();
}
```

Step 3a: Load cell of cached submatrices (then wait)

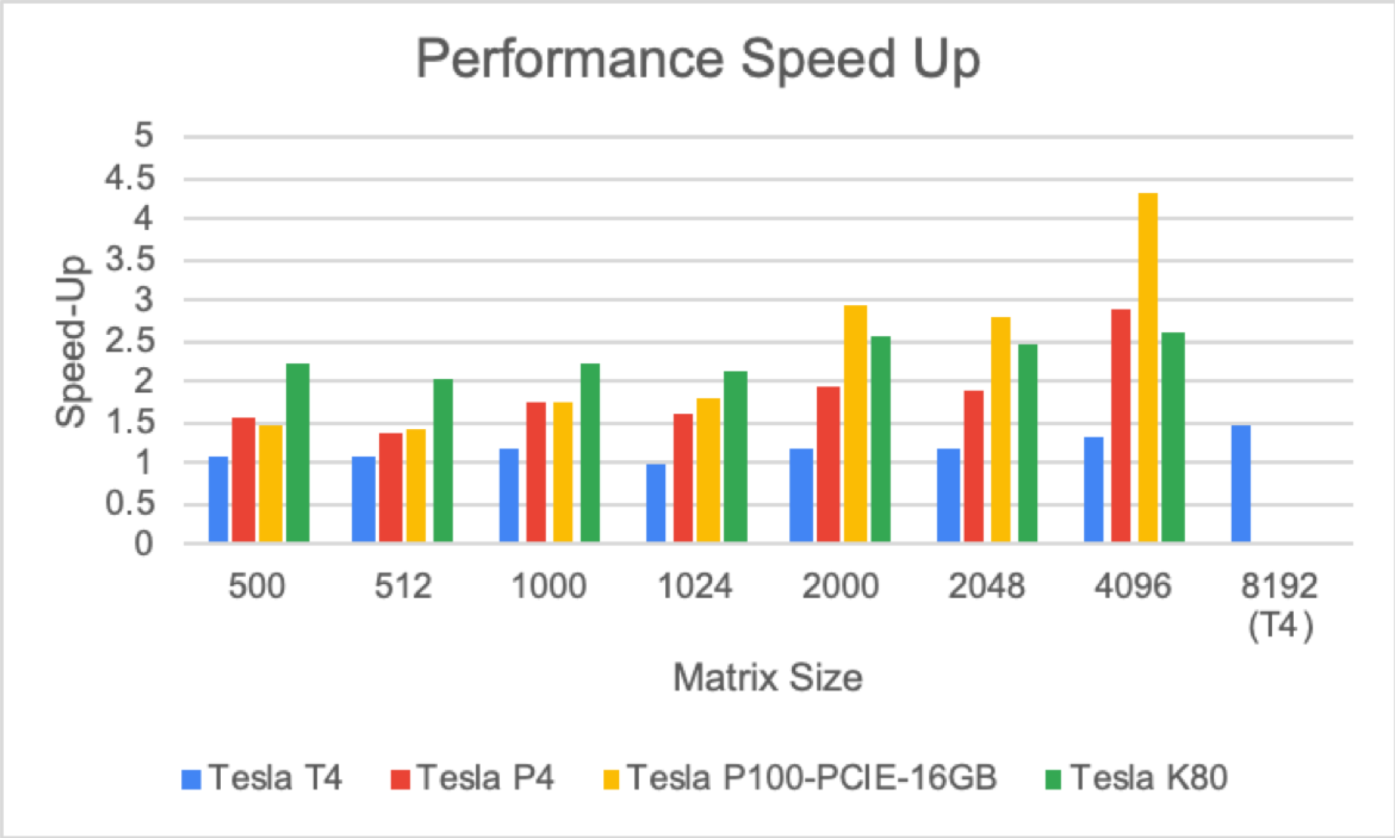
Step 3b: Calculate submatrix contribution (then wait)

Writing the tiled kernel

```
__global__ void tiledkernel(float* Md, float* Nd, float* Pd, int width) {
        …
        for (int m=0; m < num_tiles; m++) {
                …
        }

        if (row < width && col < width)
                Pd[row*width+col] = tmp;
}
```

Step 4: Write the value into the appropriate cell

Performance results



Speed-up = Time for untiled version / time for tiled version