Multi-Dimensional arrays in CUDA

Matrix Squaring (CPU)

```
void squarecpu(unsigned *matrix, unsigned *result,
                           unsigned matrixsize /* = 32*/) {
    for (unsigned ii = 0; ii < matrixsize; ++ii) {
    for (unsigned jj = 0; jj < matrixsize; ++jj) {

        for (unsigned kk = 0; kk < matrixsize; ++kk) {
            result[ii * matrixsize + jj] +=
                matrix[ii * matrixsize + kk] * matrix[kk * matrixsize + jj];
        }
    }
    }
}
```

Matrix Squaring (GPU-Version 2)

```
square<<<N, N>>>(matrix, result, N);    // N = 32
```

```
__global__ void square(unsigned *matrix,
                       unsigned *result,
                       unsigned matrixsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned ii = id / matrixsize;
    unsigned jj = id % matrixsize;
    for (unsigned kk = 0; kk < matrixsize; ++kk) {
        result[ii * matrixsize + jj] += matrix[ii * matrixsize + kk] *
                                        matrix[kk * matrixsize + jj];
}   }
```

Matrix Squaring (GPU-Version 3)

```
int main(){
    int i, j;
    int A[N][N];
    int B[N][N];
    dim3 BlockPerGrid(1, 1);
    dim3 ThreadPerBlock(N, N);

    // Initialize A to required values B to all zeros
   cudaMemcpyToSymbol(dA,A,size,0,cudaMemcpyDefault);
   cudaMemcpyToSymbol(dB,B,size,0,cudaMemcpyDefault);

  sqr <<< BlockPerGrid, ThreadPerBlock >>> (N);

  cudaMemcpyFromSymbol(A,dB,size,0,cudaMemcpyDefault);
 // Print the final results stored in A

return 0;
}
```

Matrix Squaring (GPU-Version 3)

```
sqr<<<BlockPerGrid,ThreadPerBlock>>>(N);   // N = 32
```

```
__device__ int dA[N][N];
__device__ int dB[N][N];

__global__ void sample(int matrixsize)
{
    int j = threadIdx.x + blockDim.x * blockIdx.x;
    int i = threadIdx.y + blockDim.y * blockIdx.y;

    //printf("%d %d\n",i, j);

    for(int k=0;k<matrixsize;k++)
        dB[i][j] += dA[i][k] * dA[k][j]  ;
}
```

2D Minimum Algorithm

2	3	4	5
3	5	6	2
2	8	4	5
1	4	8	9

2	3	2
2	4	2
1	4	4

- 2D operations like this are found in many fundamental algorithms like Interpolation, Convolution, Filtering
- Applications in seismic processing, weather simulation, image processing, etc

2D Minimum Algorithm

- Each output element is the minimum of input elements within the window

  O[i][j] = min(A[i-1][j], A[i-1][j+1], A[i][j], A[i][j+1])

- The rows are in the range 0..N-1 and columns in 0..M-1
- The computation boundaries remains intact.
- Initialize all the values in A using rand() function.

2D Minimum Algorithm (CPU)

```
void init(int grid[][N], int prevgrid[][N])
{
        for(int i=0;i<N;i++){
            for(int j=0;j<N;j++){
                int val = rand() % 10;
                grid[i][j] = val;
                prevgrid[i][j] = val;
            }
        }
}
void compute(int grid[][N], int prevgrid[][N])
{
        // preform the computation
        for(int i=1;i<N;i++){
                for(int j=0;j<N-1;j++){
                        grid[i][j] = min( min( prevgrid[i-1][j], prevgrid[i-1][j+1] ) ,
min( prevgrid[i][j], prevgrid[i][j+1] ) );
                }
        }

}
```

2D Minimum Algorithm (GPU)

```
void compute(int grid[][N], int prevgrid[][N])        // N = 32
{
        dim3 BlockPerGrid(1, 1);
        dim3 ThreadPerBlock(N, N);
        size_t size = N * N * sizeof(int);


        cudaMemcpyToSymbol(dgrid, grid, size, 0, cudaMemcpyDefault);
        cudaMemcpyToSymbol(dprevgrid, prevgrid, size, 0,
cudaMemcpyDefault);

        sample <<<BlockPerGrid, ThreadPerBlock>>> (N);

        cudaMemcpyFromSymbol(grid, dgrid, size, 0, cudaMemcpyDefault);

}
```

2D Minimum Algorithm (GPU)

```
sample <<<BlockPerGrid, ThreadPerBlock>>> (N);    // N = 32
```

```
__device__ int dgrid[N][N];
__device__ int dprevgrid[N][N];

__global__ void sample(int matrixsize){
    int j = threadIdx.x + blockDim.x * blockIdx.x;
    int i = threadIdx.y + blockDim.y * blockIdx.y;

    //printf("%d %d\n",i, j);

    if( i > 0 && j < N-1 )
        dgrid[i][j] = min( min( dprevgrid[i-1][j], dprevgrid[i-1]
[j+1] ) , min( dprevgrid[i][j], dprevgrid[i][j+1] ) );

}
```

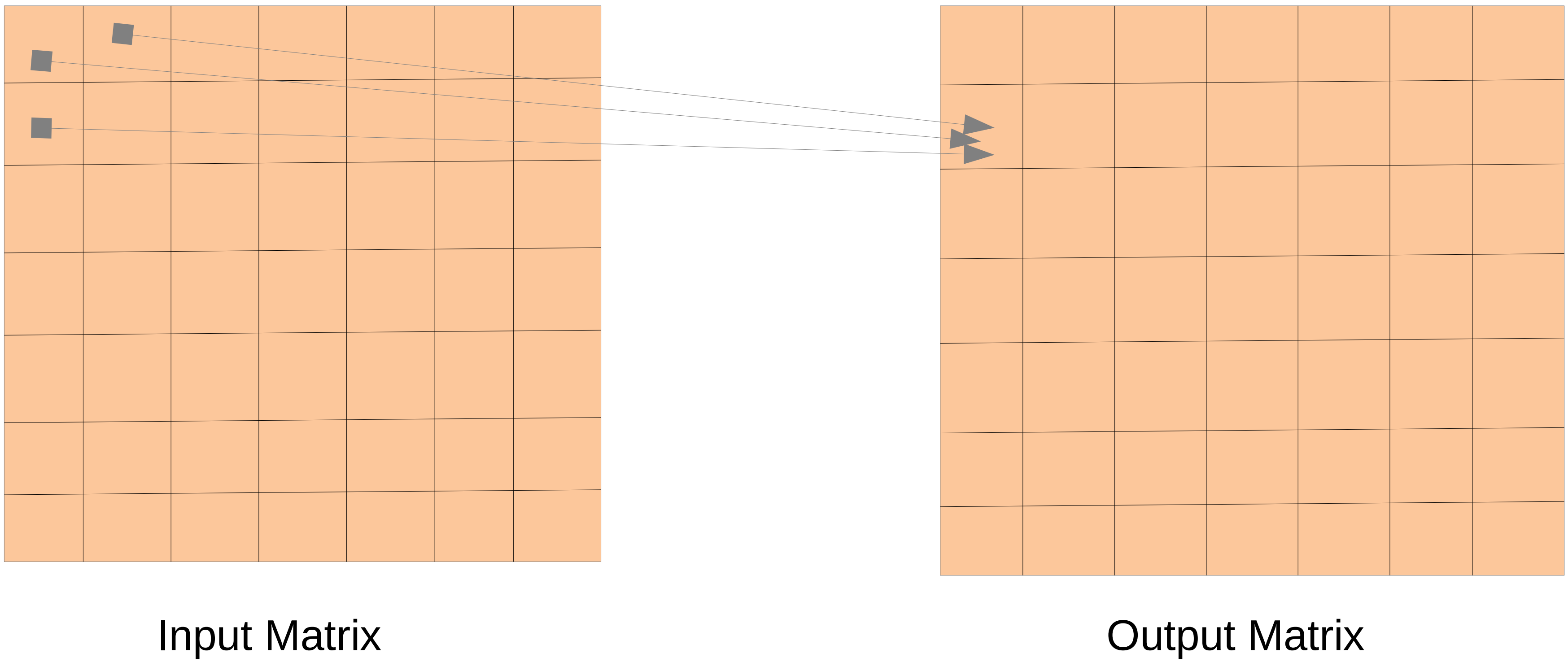
Stencil Computation

- Design and implement a stencil computation wherein in each iteration, cells compute their values using the neighboring ones, according to the following formula:

  A[i][j] += A[i-1][j] + A[i-1][j+1]

- The rows are in the range 0..N-1 and columns in 0..M-1
- The computation boundaries remains intact.
- Initialize all the values in A to all 1s.
- Your program should run for ten iterations.

Stencil Computation (in-place)



Stencil Computation (CPU)

```
// Perform the computation
 for(int itr=0;itr<10;itr++){

        // preform the computation for the current iteration
        for(int i=1;i<N;i++){
                for(int j=0;j<N-1;j++){
                        stencil[i][j] += prevstencil[i-1][j] + prevstencil[i-1][j+1];
                }
        }

        // copy the result to the current iteration to the prevstencil so that can
be used for next iteration
        for(int i=0;i<N;i++){
                for(int j=0;j<N;j++){
                        prevstencil[i][j] = stencil[i][j];
                }
        }

 }
```

Stencil Computation (GPU)

```
dim3 BlockPerGrid(1, 1);
dim3 ThreadPerBlock(N, N);
size_t size = N * N * sizeof(int);

int itr;
// Perform the computation
for(itr=0;itr<10;itr++)
{
        cudaMemcpyToSymbol(dstencil, stencil, size, 0,
cudaMemcpyDefault);
        cudaMemcpyToSymbol(dprevstencil, prevstencil, size, 0,
cudaMemcpyDefault);
        sample <<<BlockPerGrid, ThreadPerBlock>>> (N);
        cudaMemcpyFromSymbol(prevstencil, dstencil, size, 0,
cudaMemcpyDefault);
        cudaMemcpyFromSymbol(stencil, dstencil, size, 0,
cudaMemcpyDefault);
}
```

Stencil Computation (GPU)

```
sample <<<BlockPerGrid, ThreadPerBlock>>> (N);    // N = 32
```

```
__device__ int dstencil[N][N];
__device__ int dprevstencil[N][N];

__global__ void sample(int matrixsize)
{
    int j = threadIdx.x + blockDim.x * blockIdx.x;
    int i = threadIdx.y + blockDim.y * blockIdx.y;

    //printf("%d %d\n",i, j);

    if( i > 0 && j < N-1 )
        dstencil[i][j] += dprevstencil[i-1][j] + dprevstencil[i-1][j+1];
}
```

Stencil Computation(Original)

- Design and implement a stencil computation wherein in each iteration, cells compute their values using the neighboring ones, according to the following formula:

  A[i][j] += A[i-1][j] + A[i-1][j+1]

- The rows are in the range 0..N-1 and columns in 0..M-1
- The computation boundaries remains intact
- Initialize all the values in A to all 1s
- Your program should run for ten iterations

Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix.

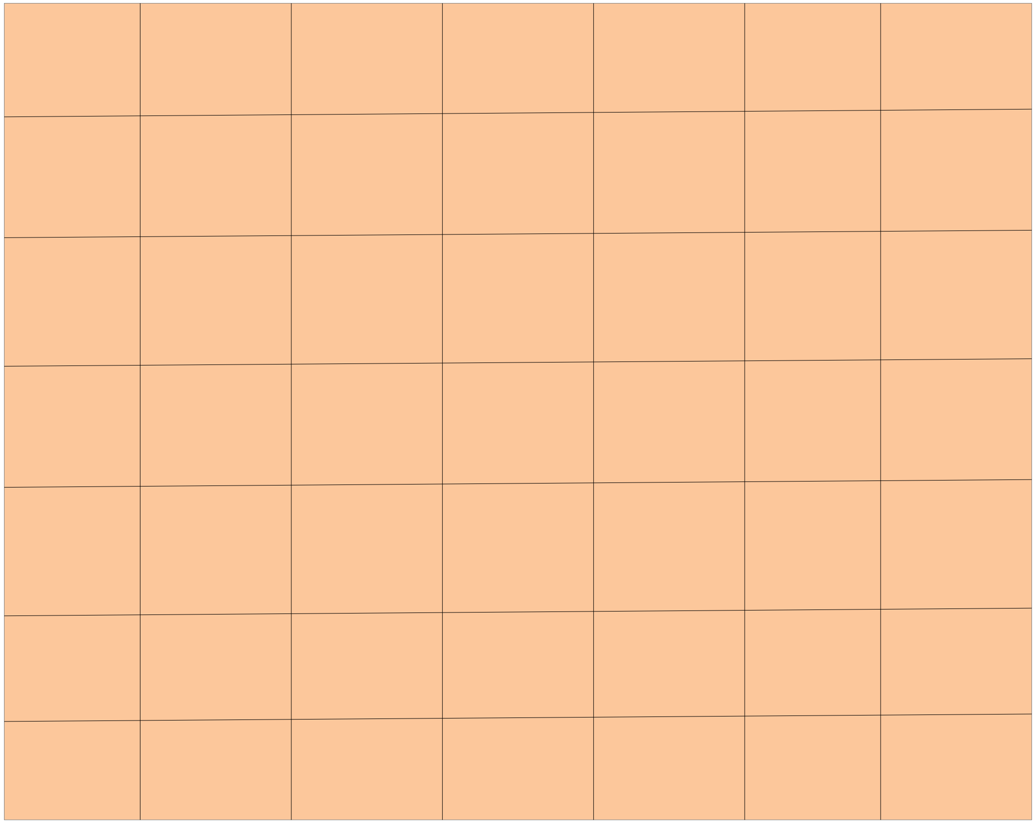
  On CPU:

```
for(int itr=0;itr<10;itr++){
        for(int i=N-1;i>=1;i--){
            for(int j=0;j<N-1;j++){
                stencil[i][j] += stencil[i-1][j] + stencil[i-1][j+1];
            }
        }
}
```

Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix

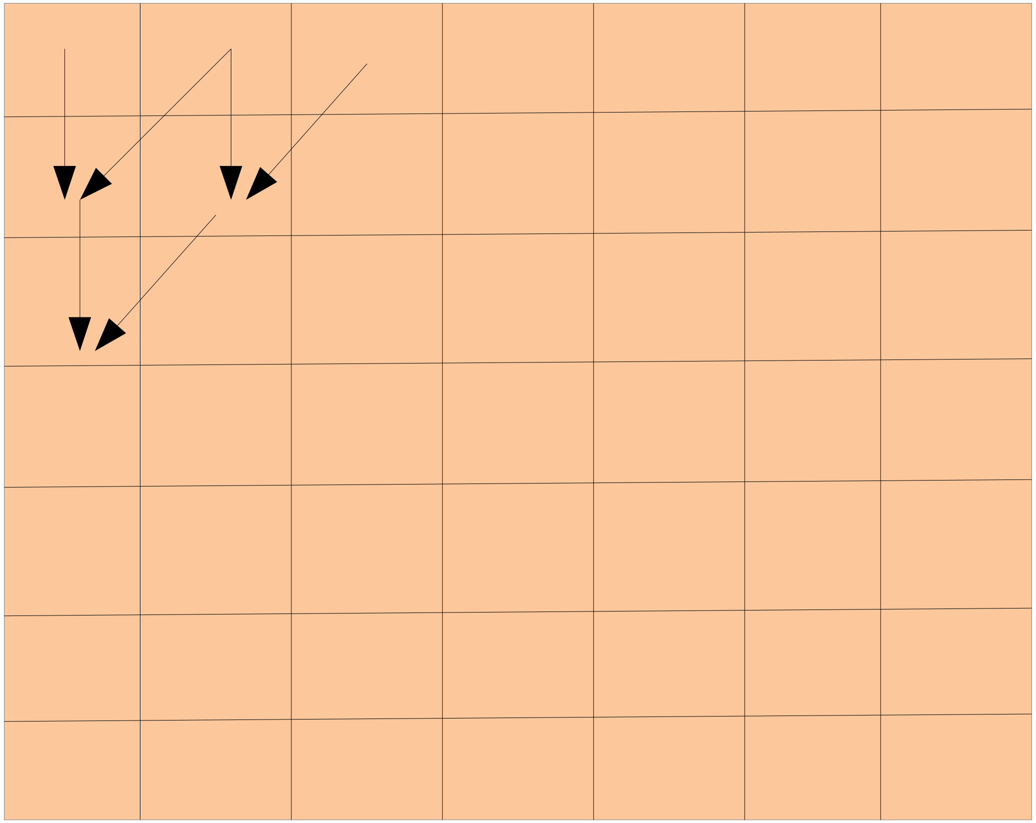
On GPU: stencil[7][7]



Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix

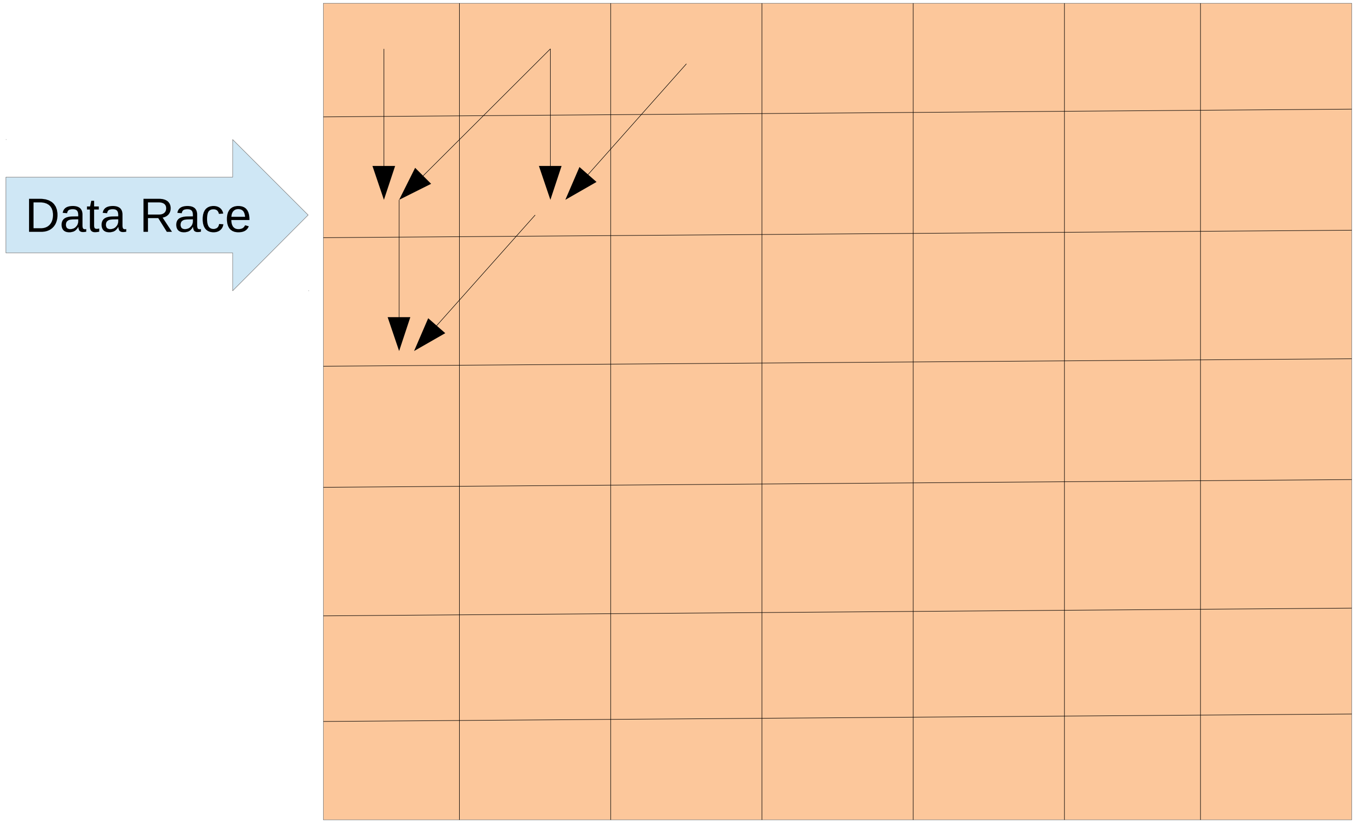
On GPU: stencil[7][7]



Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix

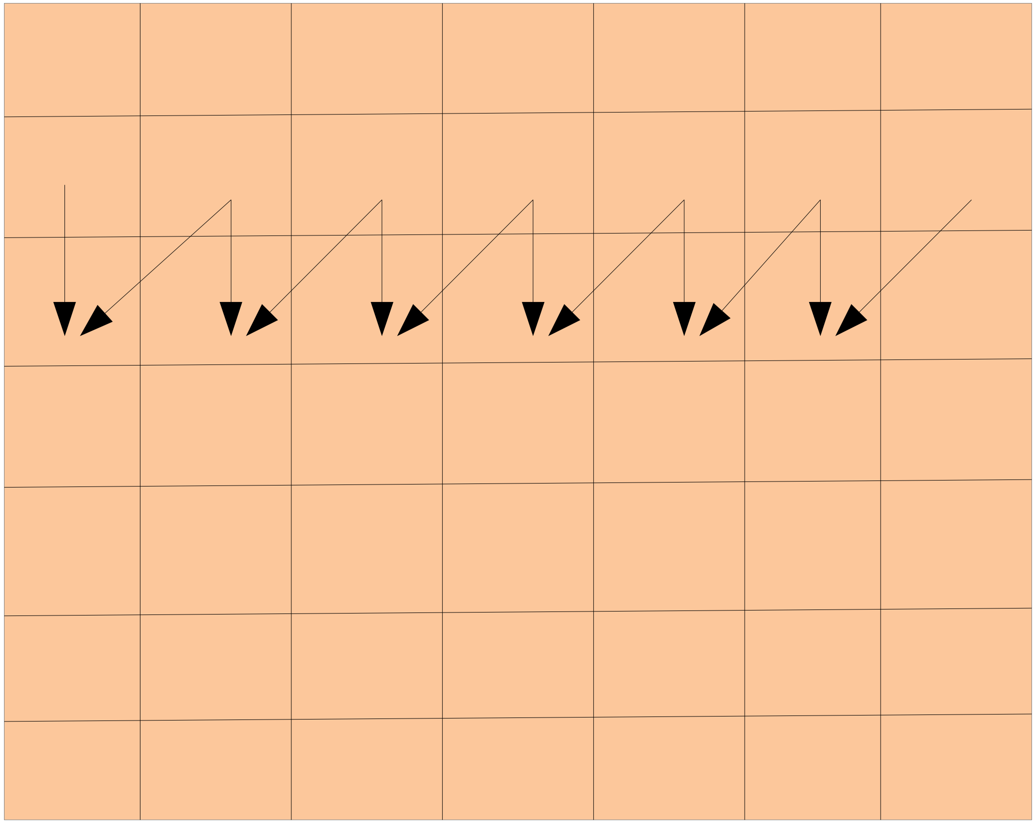
On GPU: stencil[7][7]



Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix

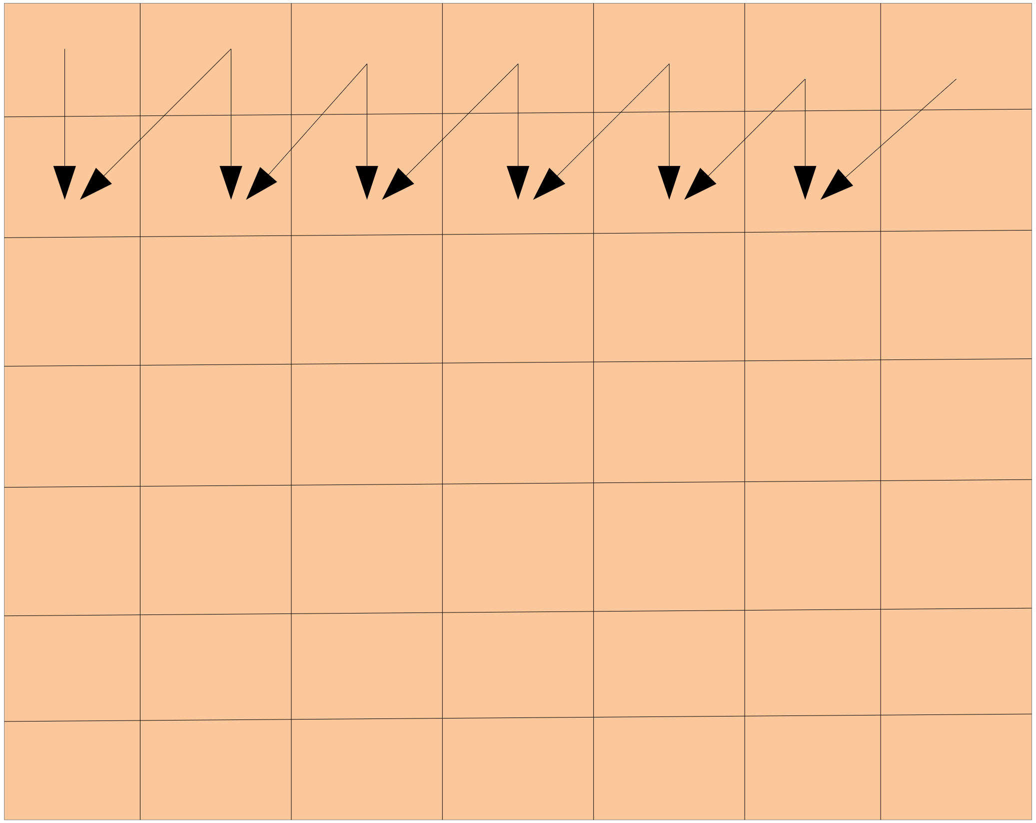
On GPU: stencil[7][7]



Stencil Computation (in-place)

- The updates should happen in-place inside the input matrix

On GPU: stencil[7][7]



Stencil Computation (in-place)

```
    size_t size = N * N * sizeof(int);
    int itr;

    // Perform the computation
     for(itr=0;itr<10;itr++)
     {
            cudaMemcpyToSymbol(dstencil, stencil, size, 0, cudaMemcpyDefault);

            for(int i=N-1;i>=1;i--)
            {
               cudaMemcpyToSymbol(dstencil[i], stencil[i], N, 0,
cudaMemcpyDefault);
               sample <<<1, N>>> (N,i);
               cudaMemcpyFromSymbol(stencil[i], dstencil[i], N, 0,
cudaMemcpyDefault);
            }

            cudaMemcpyFromSymbol(stencil, dstencil, size, 0,
cudaMemcpyDefault);
     }
```

Stencil Computation (in-place)

```
sample <<<1, N>>> (N,i);            // N = 32, i = [1,N-1]
```

```
__device__ int dstencil[N][N];

__global__ void sample(int matrixsize, int i)
{
    int j = threadIdx.x + blockDim.x * blockIdx.x;

    //printf("%d %d\n",i, j);

    if( i > 0 && j < N-1 )
        dstencil[i][j] += dstencil[i-1][j] + dstencil[i-1][j+1];

}
```