HIGH PERFORMANCE PARALLEL PROGRAMMING (CS61064)

Soumyajit Dey
CSE, IIT Kharagpur

Things that we will cover

- CUDA programming basics
- A bit of GPU architecture and organization (as and when required)
- Behavior of the CUDA Runtime system (as and when required)

- Fifteen years ago, Graphics on a PC were performed by a video graphics array (VGA) controller.
- VGAs evolved to more complex hardwares : accelerating graphics functions
- Early GPUs and their associated drivers implemented the OpenGL and DirectX models (APIs) of graphics processing.
- With time, HW functionality evolved as programmable SW

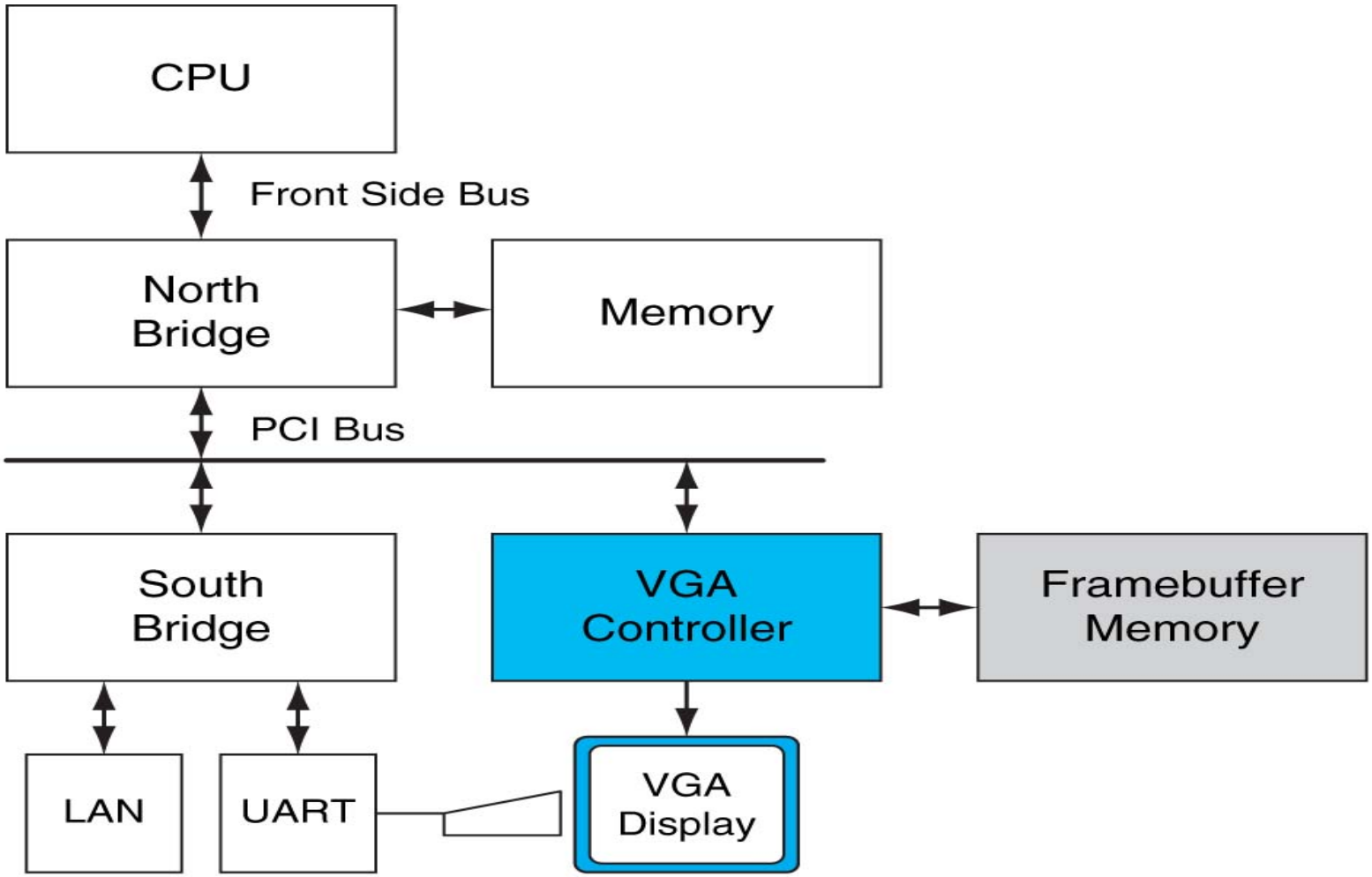


Figure: Historical PC. - Copyright 2009 Elsevier, Inc. All rights reserved - Hennessy and Patterson "Conputer Organization and Design"

- General Purpose computation on GPU (GPGPU) approach : programming the GPU using a graphics API and graphics pipeline to perform nongraphics tasks
- Is possible since internally there is a whole new ISA for the so called graphics pipeline.
- GPU computing : thinking of GPUs as just another kind of general architecture
- For now we will just think a GPU is a massively parallel processor (each core implements a generic ISA)

Compute Unified Device Architecture

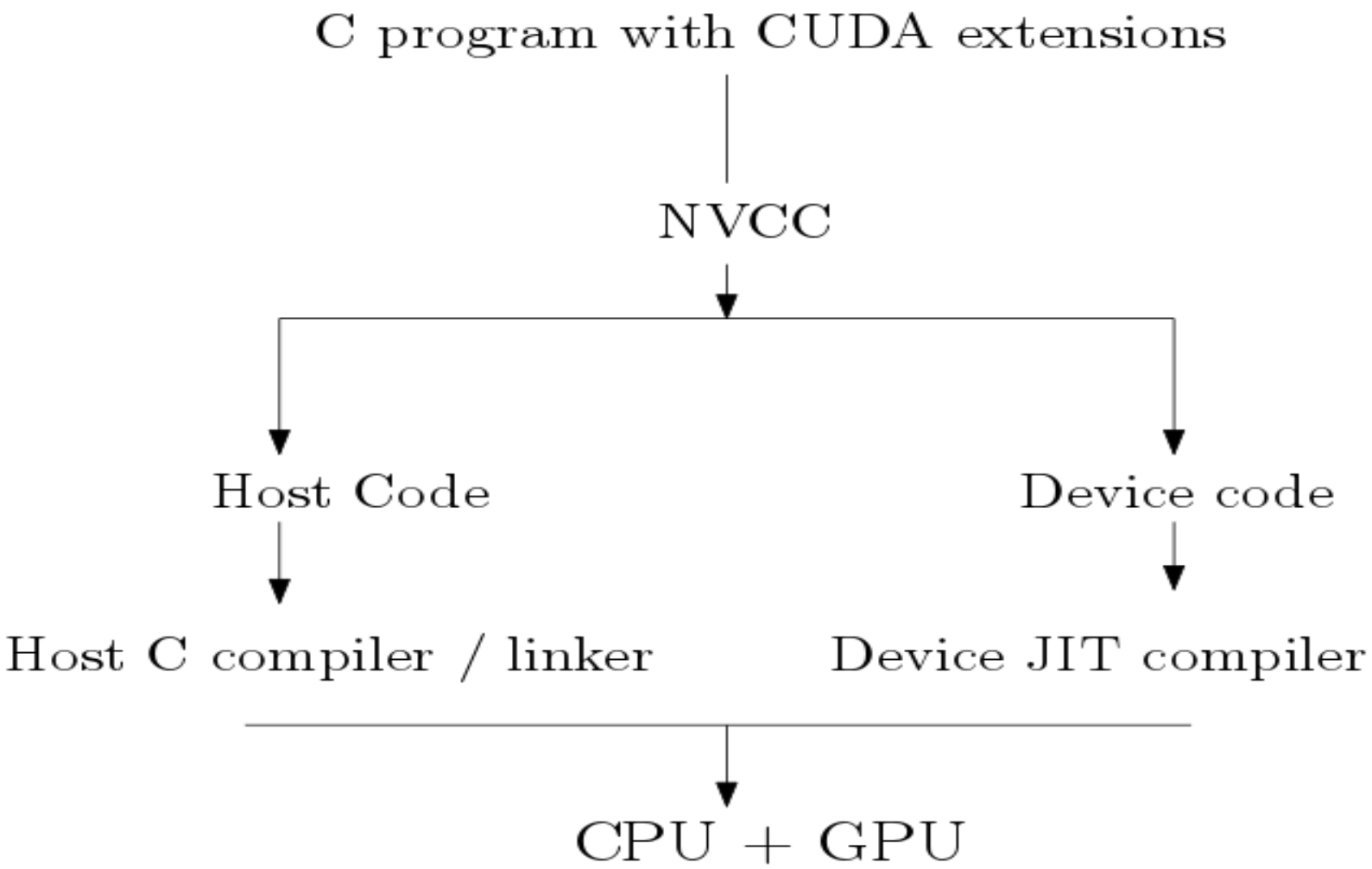
- CUDA C is an extension of C programming language with special constructs for supporting parallel computing
- CUDA programmar perspective - CPU is a *host* : dispatches parallel jobs to GPU *devices*

CUDA program structure

- *host code* for a host device (CPU)
- device code for GPU(s)
- Any C program is a valid CUDA host code
- In general CUDA programs (host + device) code cannot be compiled by standard C compilers

NVIDIA C compiler (NVCC)

The compilation flow



The execution flow

CPU serial code

$\Downarrow$

GPU parallel kernel

$\Downarrow$

CPU serial code

$\Downarrow$

GPU parallel kernel

Examples : Vector addition CPU only

```
void vecAdd(float* h_A, float* h_B,
float* h_C, int n)
{
 for (i = 0; i < n; i++)
   h_C[i] = h_A[i] + h_B[i];
}
int main()
{
  float *h_A,*h_B,*h_C;
  int n;
  h_A=(float*)malloc(n*sizeof(float))
  h_B=(float*)malloc(n*sizeof(float))
  h_C=(float*)malloc(n*sizeof(float))
  vecAdd(h_A, h_B, h_C, N);
}
```

Examples : Vector addition CPU-GPU

```
#include <cuda.h>
__global__
void vecAddKernel(float* A, float* B,
float* C, int n){
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
      C[i] = A[i] + B[i];
}
void vecAdd(float* A, float*B,
float* C, int n){
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void)&d_A, size);
    cudaMalloc((void)&d_B, size);
    cudaMalloc((void)&d_C, size);
```

Examples : Vector addition CPU-GPU

```
    cudaMemcpy(d_A, A, size,
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size,
    cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256),256>>>
    (d_A, d_B, d_C, n)

    cudaMemcpy(C, d_C, size,
    cudaMemcpyDeviceToHost);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Observations

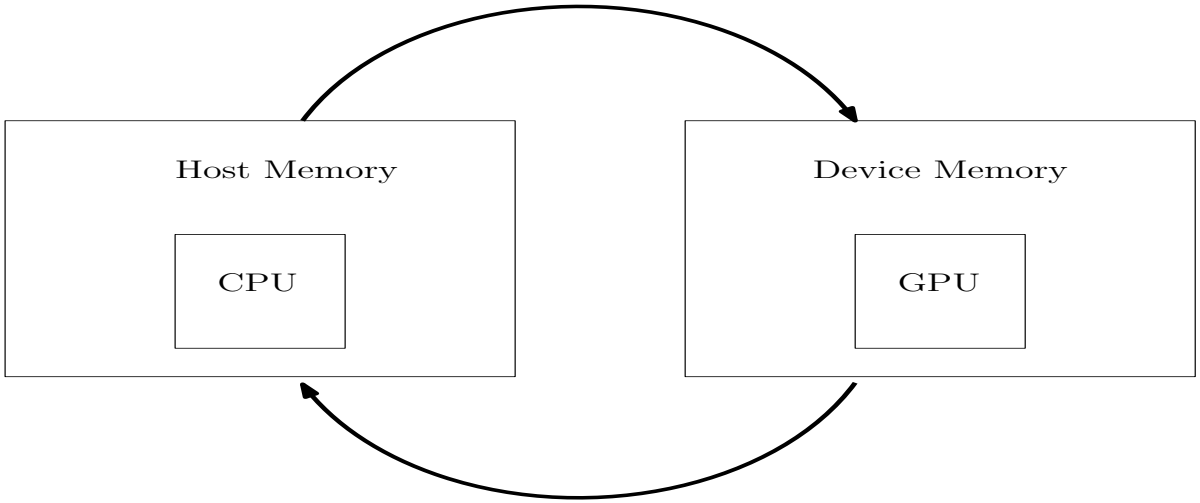


Figure: CPU/GPU Mem Layout

- cuda.h → includes during compilation CUDA API functions and CUDA system variables
- A, B, C → arrays mapped to main memory locations

Observations

```
cudaMalloc((void)&d_A, size);
//allocate memory segment from GPU
    global memory
//expects a generic pointer (void **)
//the low level function is common for
     all object types
cudaMemcpy(d_A, A, size,
    cudaMemcpyHostToDevice);
//transfer data from CPU to GPU memory
//d_A cannot be dereferenced in host
    code
```

Observations

```
//d_A cannot be dereferenced in host
    code
cudaMemcpy(C, d_C, size,
    cudaMemcpyDeviceToHost);
//transfer data from GPU to CPU memory
//can also transfer among different
    device mem locations
//can also transfer data host to host
    - we do not need that
//cannot transfer data among different
    GPU devices
cudaFree(d_A);
//free GPU global memory
```

Function declaration Keywords

```
__global__
void vecAddKernel(float* A, float* B,
    float* C, int n)
```

Table: CUDA Keywords for functions and their scope

Keywords and Functions	Executed on the	Only callable from the
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

CUDA functions

- Every function is a default __host__ function (if not having any CUDA keywords)
- A function can be declared as both __host__ and __device__ function
  - "__host__ __device__ fn()"
  - Runtime system generates two object files, one can be called from host fn()s, another from device fn()s
- __global__ functions can also be called from the device using CUDA kernel semantics (<<< ... >>>) if you are using *dynamic parallelism* - that requires CUDA 5.0 and compute capability 3.5 or higher.

CUDA functions : more observations

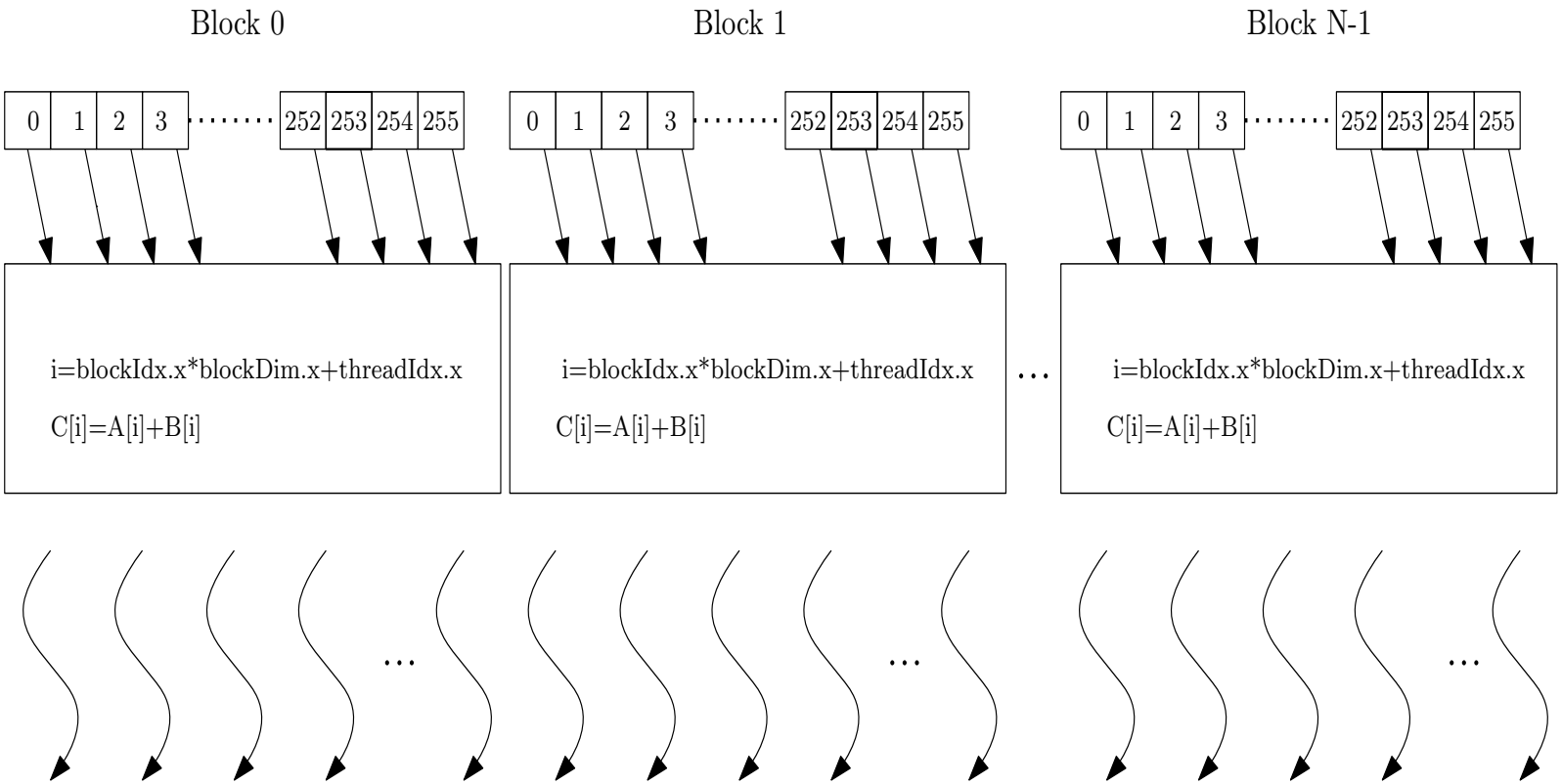
- __device__ functions can have a return type other than void but __global__ functions must always return void
- __global__functions can be called from within other kernels running on the GPU to launch additional GPU threads (as part of CUDA dynamic parallelism model) while __device__ functions run on the same thread as the calling kernel.

CUDA kernel

A CUDA kernel when invoked launches multiple threads arranged in a 2 level hierarchy, check the device fn call.

```
vecAddKernel<<<ceil(n/256),256>>>
(d_A, d_B, d_C, n)
```

- The call specifies a **grid** of threads to be launched
- the grid is arranged in a hierarchical manner
- (no. of blocks, no. of thread per block)
- all blocks contain same no. of threads (max 1024)
- blocks can be numbered as (_,_,_) triplets : more on this later

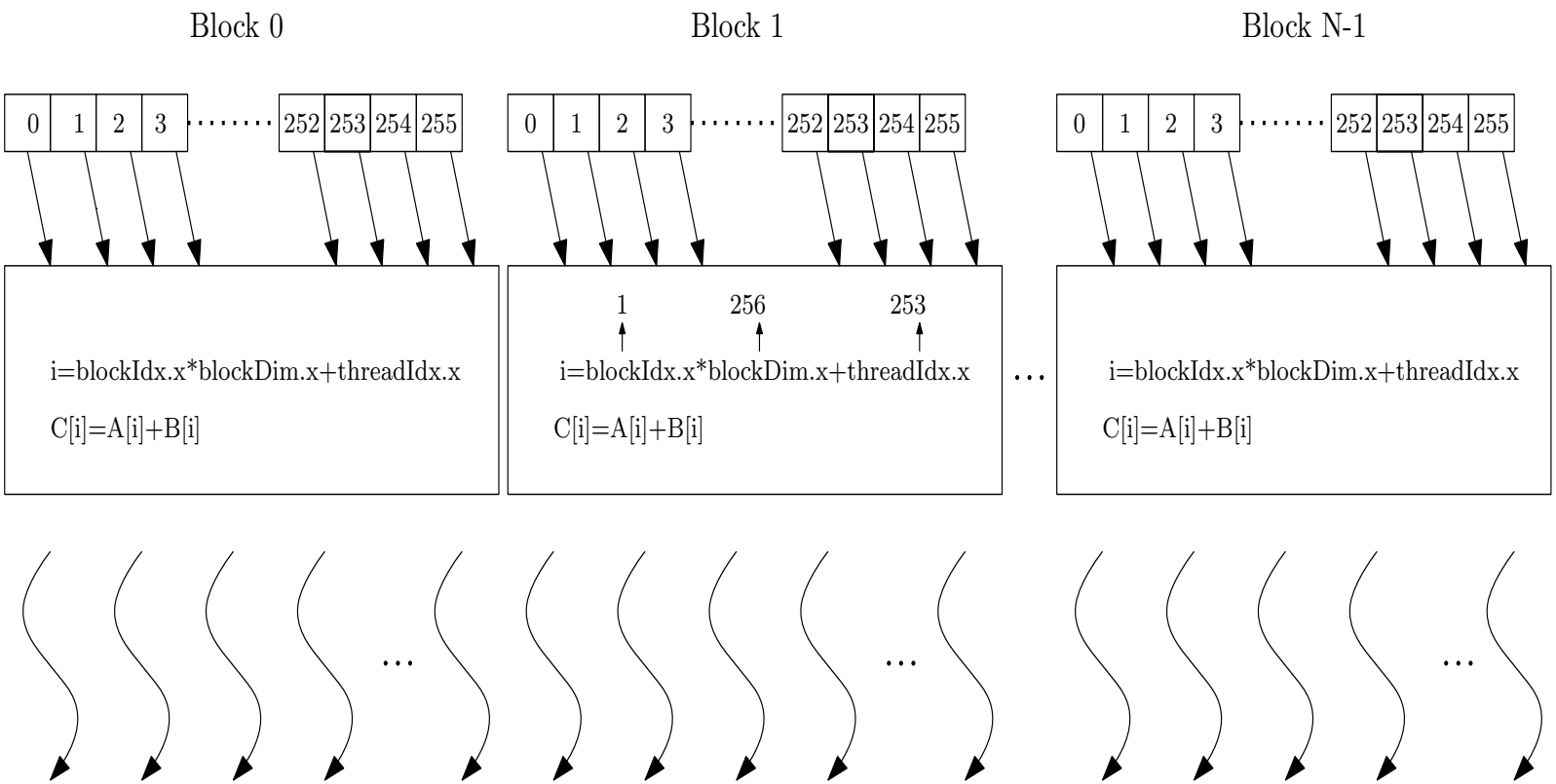

Block 0
Block 1
Block N-1
0 1 2 3 ..... 252 253 254 255
i=blockIdx.x*blockDim.x+threadIdx.x
C[i]=A[i]+B[i]

Kernel specific system vars

- **gridDim** - no. of blocks in the grid
- **gridDim.x** - no. of blocks in dimension x of multi-dim grid !!
- **blockDim** - no. of threads/block
- **blockDim.x** - no. of threads/block in dimension x of multi-dim block !!
- For single dimension defn of block composition in grid, **blockDim** = **blockDim.x**
- **blockidx.x** = block number for a thread
- **threadidx.x** = thread no. inside a block

```
__global__
void vecAddKernel(float* A, float* B,
float* C, int n){
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
      C[i] = A[i] + B[i];
}
```

- The code is executed by all the threads in the grid
- Every thread has a unique combination of (blockIdx.x, threadIdx.x) which maps to a unique value of **i**
- **i** is private to each thread


Block 0
Block 1
Block N-1
0 1 2 3 ..... 252 253 254 255
i=blockIdx.x*blockDim.x+threadIdx.x
C[i]=A[i]+B[i]
1
256
253

Multi dimensional block

In general

- a grid is a 3-D array of blocks
- a block is a 3-D array of threads
- specified by C `struct` type `dim3`
- unused dimensions are set to 1

Multi dimensional grid, block

```
dim3 X(ceil(n/256.0), 1, 1);
dim3 Y(256, 1, 1);
vecAddKernel<<<X, Y>>>(..);
vecAddKernel<<<ceil(n/256), 256>>>(..);
//CUDA compiler is smart enough to
    realise both as equivalent
```

Multi dimensional grid, block

- **gridDim.x/y/z** $\in [1, 2^{16}]$
- (**blockIdx.x, blockIdx.y, blockIdx.z**) is one block
- All threads in the block sees the same value of system vars **blockIdx.x, blockIdx.y, blockIdx.z**
- **blockIdx.x/y/z** $\in [0,$ **gridDim.x/y/z** -1]

Multi dimensional grid, block

block dimension is limited by total number of threads possible in a block – 1024.

- (512, 1, 1) - $\surd$
- (8, 16, 4) - $\surd$
- (32, 16, 2) - $\surd$
- (32, 32, 32) - $\times$

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

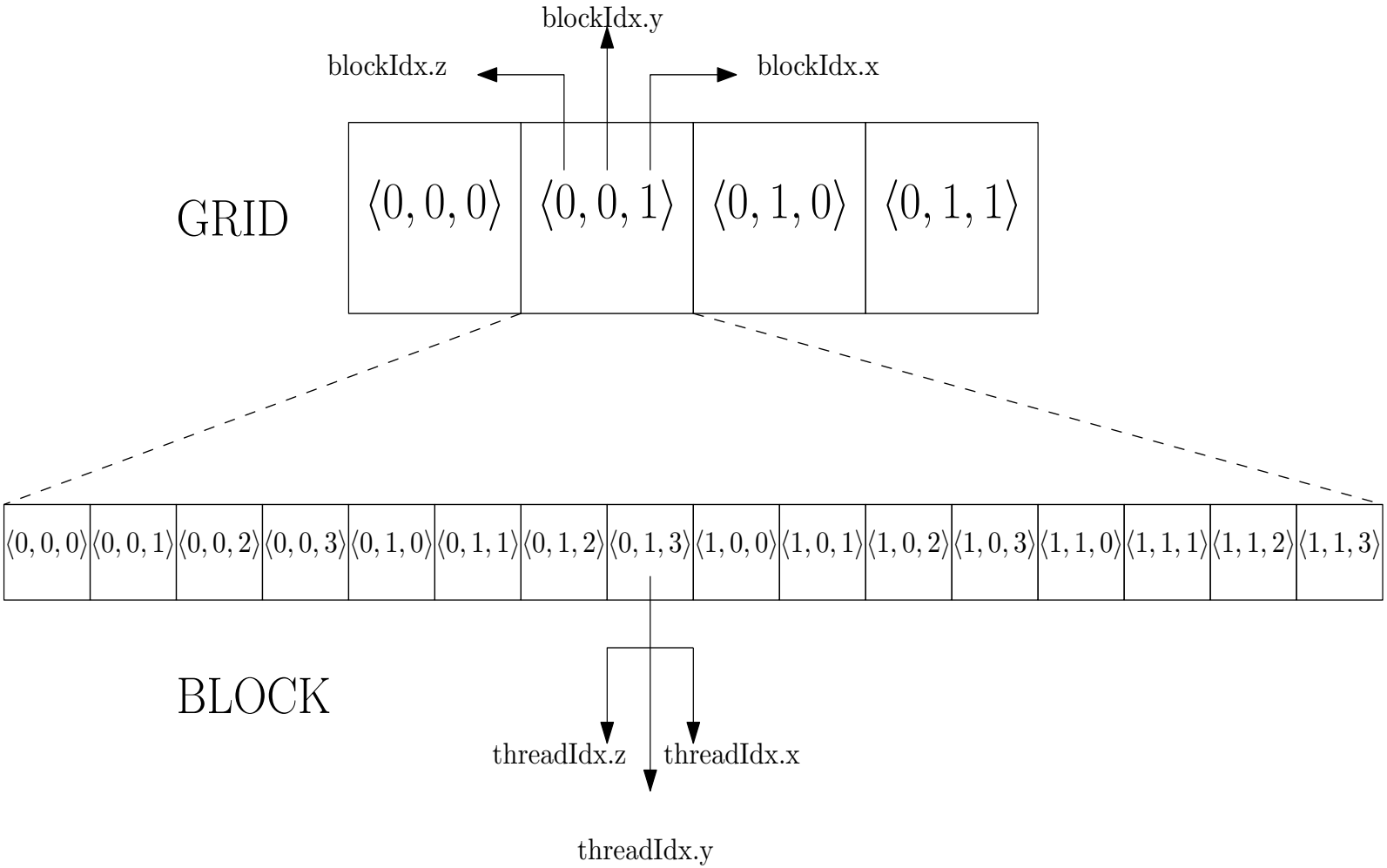


Figure: Grids and Blocks

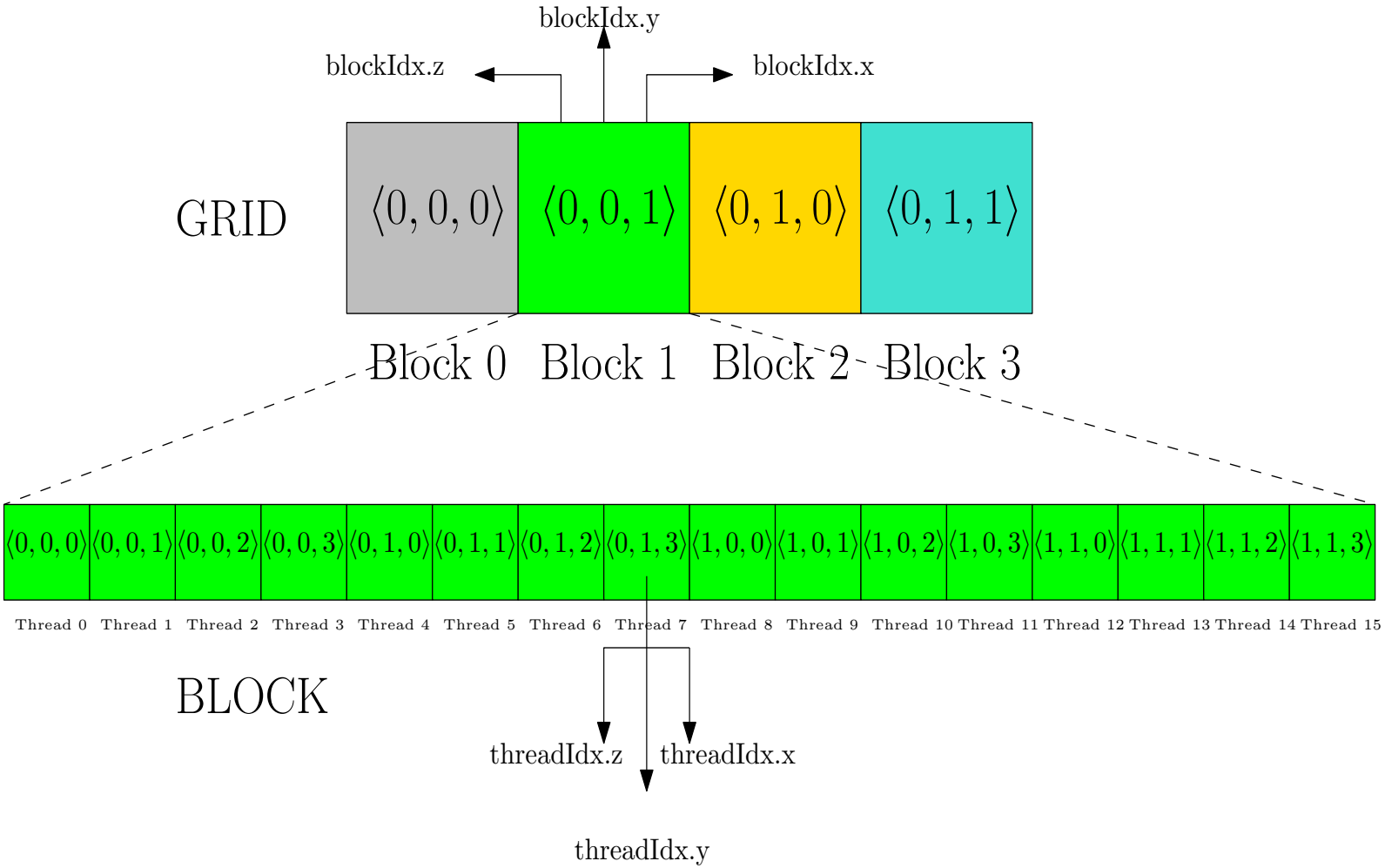


Figure: Global Thread IDs

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

NumCols = blockDim.x * gridDim.x

NumRows = blockDim.y * gridDim.y

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

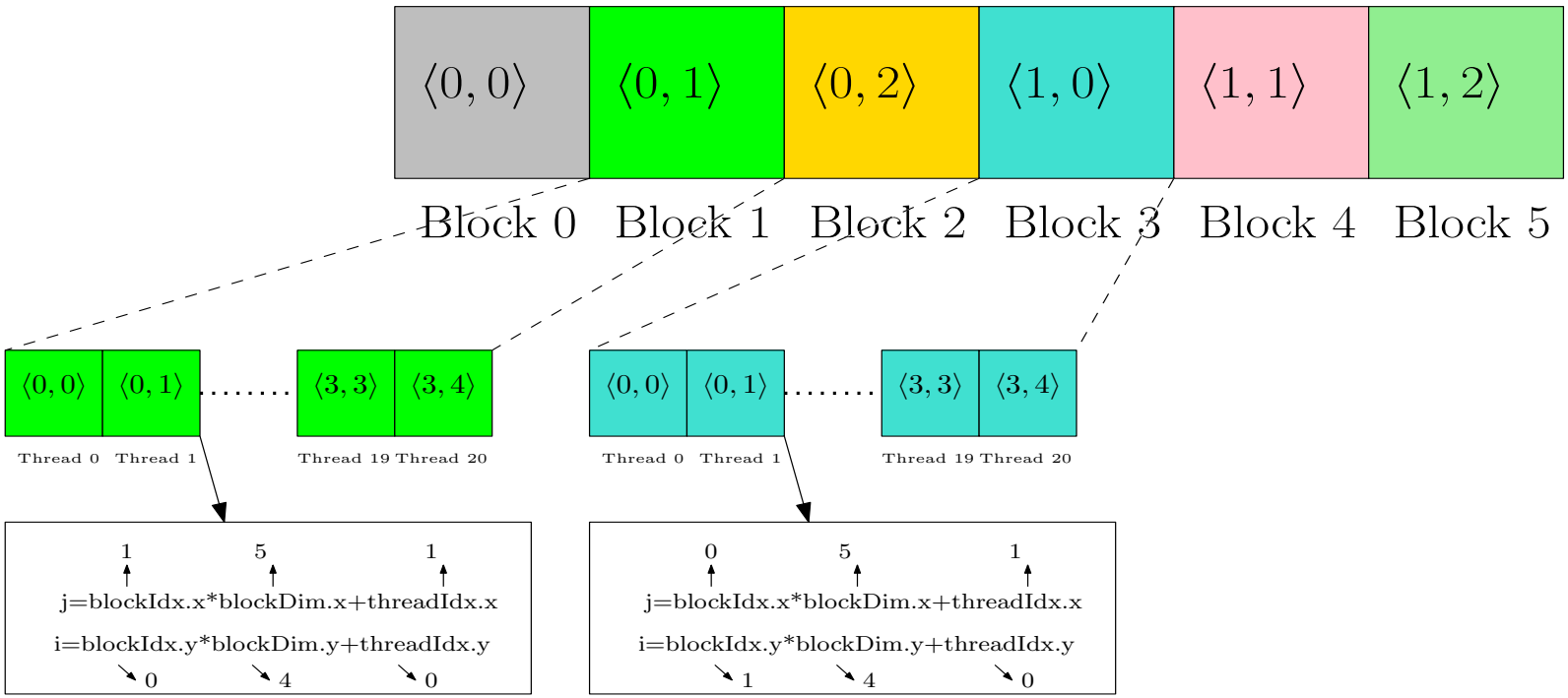


Figure: Two Dimensional Kernel

8 X 15 Matrix

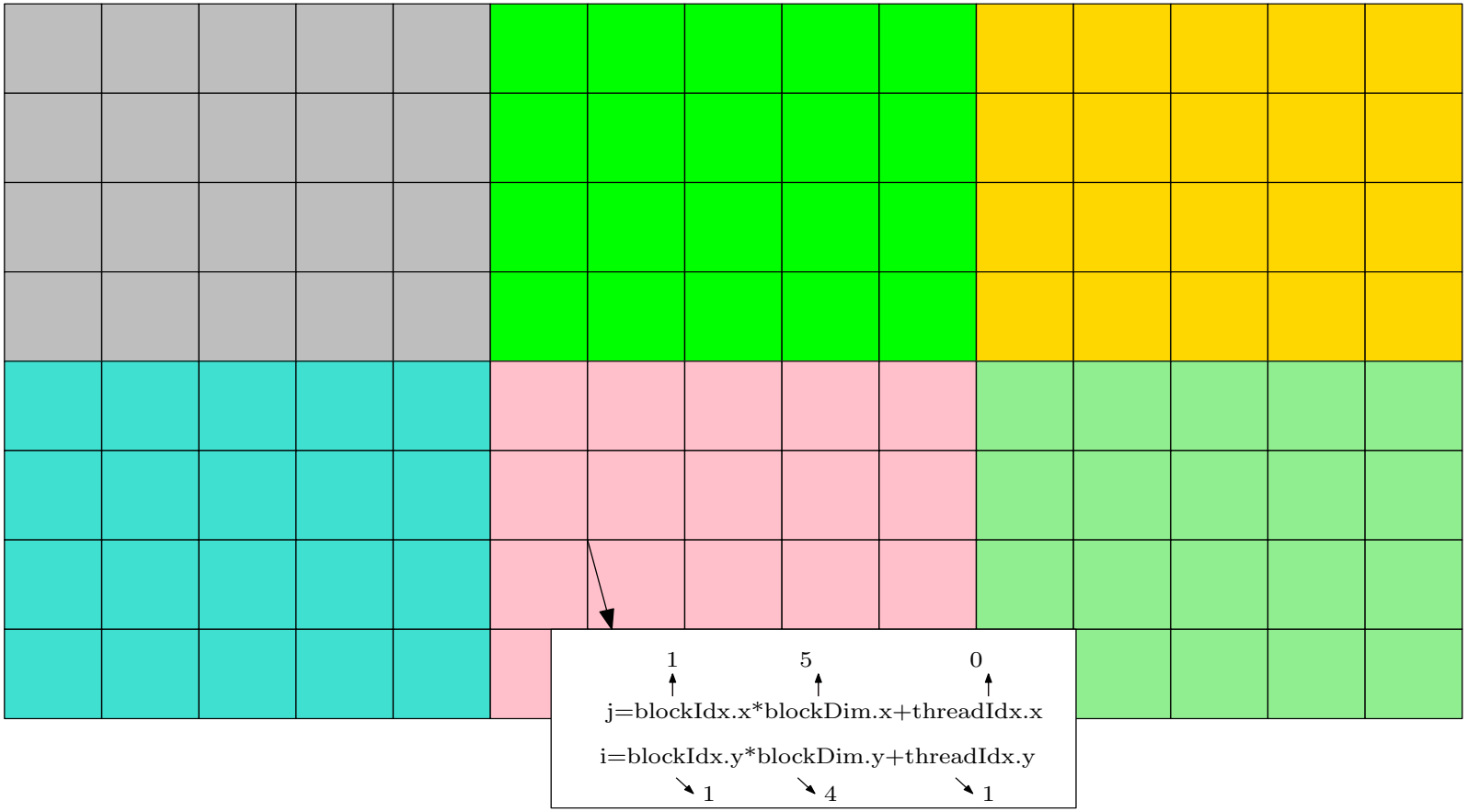


Figure: Two Dimensional Kernel-Data Mapping