Vector Processors : Consider a simple $Y = aX + Y$ operation

```
        L.D     F0,a        ;load scalar a
        DADDIU  R4,Rx,#512  ;last address to load
Loop:   L.D     F2,0(Rx)    ;load X[i]
        MUL.D   F2,F2,F0    ;a × X[i]
        L.D     F4,0(Ry)    ;load Y[i]
        ADD.D   F4,F4,F2    ;a × X[i] + Y[i]
        S.D     F4,9(Ry)    ;store into Y[i]
        DADDIU  Rx,Rx,#8    ;increment index to X
        DADDIU  Ry,Ry,#8    ;increment index to Y
        DSUBU   R20,R4,Rx   ;compute bound
        BNEZ    R20,Loop    ;check if done
```

(a) MIPS

```
L.D     F0,a        ;load scalar a
LV      V1,Rx       ;load vector X
MULVS.D V2,V1,F0    ;vector-scalar multiply
LV      V3,Ry       ;load vector Y
ADDVV.D V4,V2,V3    ;add
SV      V4,Ry       ;store the result
```

(b) VMIPS

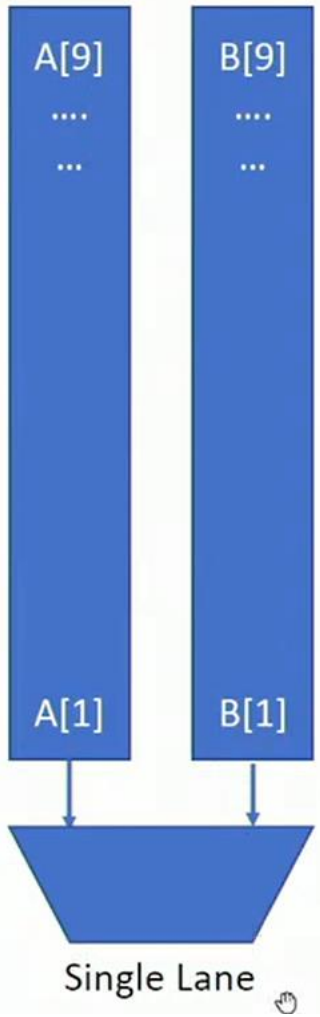
Figure: Assuming the data size $<$ vector storage (Ref: CoA: a quantitative approach (Hennessy & Patterson))

In non-vectorized code, every ADD.D must wait for a MUL.D, and every S.D must wait for the ADD.D

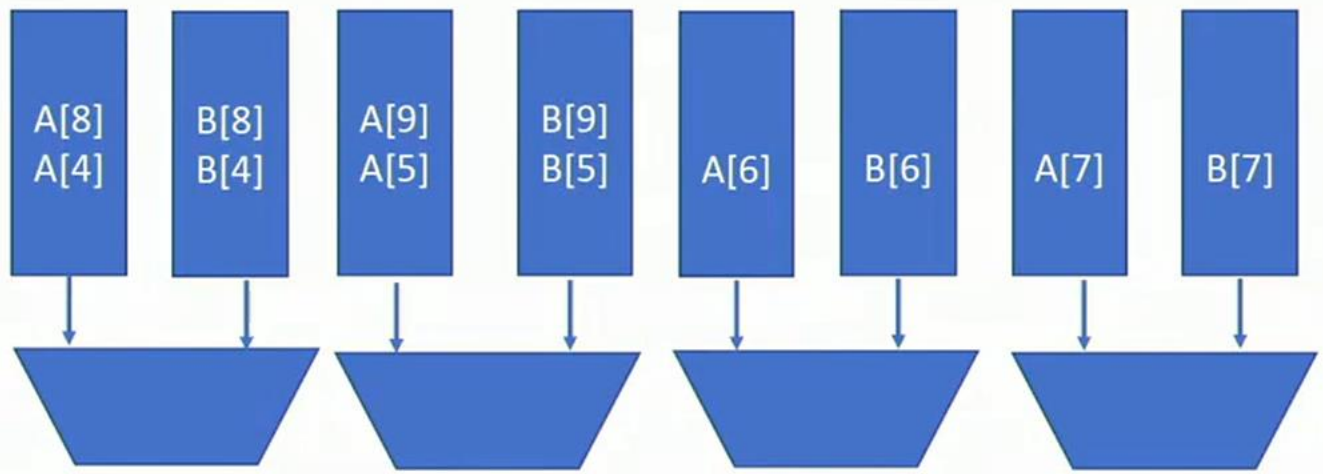
Vector Processors

- A vector instruction passes lot of parallel work to the hardware
- The FUs can be : fully parallel, or a combination of parallel and pipelined units
- If the clock rate of a vector processor is halved, doubling the number of lanes will retain the same potential performance.
- Work for compilers - loop vectorization, dependency handling

Vector Processors



Four add pipelines can complete four additions per cycle
Elements are interleaved



GPUs

Ideas from parallel instruction handling by vector architectures, ILP techniques etc were borrowed to accelerate graphics processing

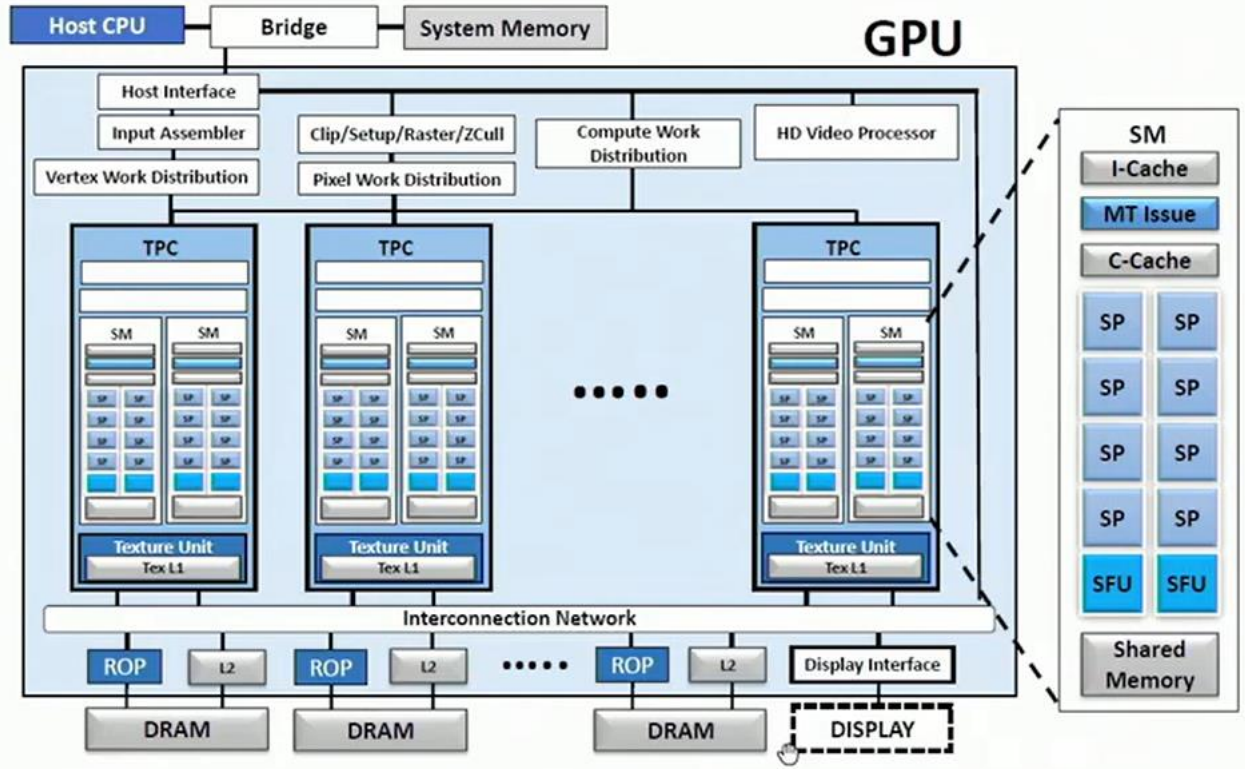


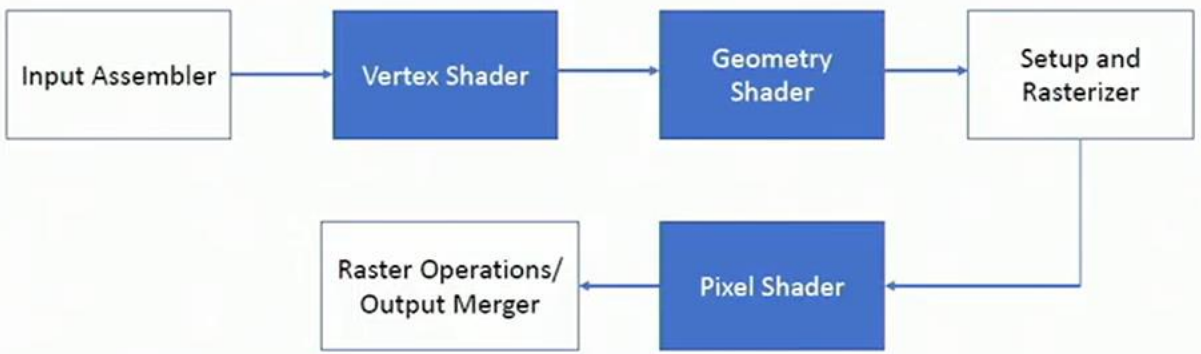
Figure: GPU systems (GeForce 8800) - Hennessy, Patterson (reproduced)

GPU Architecture (Tesla)

- Earlier figure depicts a GPU with an array of 128 streaming/scalar processor (SP) cores, organized as 16 multithreaded streaming multiprocessors (SM),
- Each SM has 8 SPs,
- 2 SMs together are arranged as independent processing units called texture/processor clusters (TPCs).

Early GPUs

Early GPUs accelerated the logical graphics pipeline



Graphics Logical Pipeline

Shader Programs

Graphics application sends the GPU a sequence of vertices grouped into geometric primitives—points, lines, triangles, and polygons.

- The input assembler collects vertices and primitives.
- Vertex shader programs map the position of vertices onto the screen, altering their position, color, or orientation.
- Geometry shader programs operate on geometric primitives (such as lines and triangles) defined by multiple vertices, changing them or generating additional primitives.

Shader Programs

Usually dataflow style, model how light interacts with different materials and to render complex lighting and shadows.

- The setup and rasterizer unit generates pixel fragments (which are potential contributions to pixels) that are covered by a geometric primitive.
- The pixel shader program fills the interior of primitives, including interpolating per-fragment parameters, texturing, and coloring.
- The raster operations processing (or output merger) stage : depth testing and stencil testing, color blending operation etc

Ref : "Computer Organization and Architecture" - Hennessy, Patterson (Appendix A on GPUs)

GPUs : massive multi-threading

Design goals

- Cover the latency of memory loads and texture fetches from DRAM
- Support fine-grained parallel graphics shader (and general parallel compute) programming models
- Virtualize the physical processors as threads and thread blocks to provide transparent scalability
- Simplify the parallel programming model to writing a serial program for one thread

First generation GPUs

- GeForce 256, introduced in 1999
- Contained fixed function vertex, pixel shaders programmed with OpenGL and the Microsoft DX7 API
- GeForce 3 - the first programmable vertex processor executing vertex shaders

- Ref for contents and here and subsequent places : "NVIDIA Tesla: A Unified Graphics and Computing Architecture" by Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym, (NVIDIA) IEEE Micro, Volume 28, Issue 2, March 2008

Trade-off

- Vertex processors were designed for low-latency, high-precision math operations
- pixel-fragment processors were optimized for high-latency, lower-precision texture filtering - typically more busy (considering large triangulation)
- if these are fixed function blocks - difficult to select a fixed processor ratio
- Primary design objective for Tesla architecture - execute vertex and pixel-fragment shader programs on the same unified processor.
- Unification helps in 1) dynamic load balancing of varying vertex- and pixel-processing workloads, 2) introducing other shaders

Tesla architecture

We come back to GeForce 8800 GPU with 128 SPs organized as 16 SMs

- external DRAM control and fixed-function raster operation processors (ROPs) perform color and depth frame buffer operations directly on memory
- The interconnection network carries computed pixel-fragment colors and depth values from SPs to the ROPs
- The network also routes texture memory read requests from the SP to DRAM and read data from DRAM through a level-2 cache back to the SPs

Graphics in Tesla

- The input assembler collects vertex work
- Vertex work distributor distributes vertex work packets to the various TPCs
- The TPCs execute vertex/geometry shader programs
- output data is written to on-chip buffers
- buffers then pass their results to the viewport/clip/setup/raster/zcull block

GPGPU

Each TPC has two SMs, each SM has

- eight streaming/scalar processor (SP) cores,
- two special function units (SFUs),
- a multi-threaded instruction fetch and issue unit (MT Issue),
- an instruction cache, a read-only constant cache,
- a 16-Kbyte read/write shared memory.

GPGPU

- Each SP core contains a scalar multiply-add (MAD) unit, giving the SM eight MAD units
- The SM uses its two SFU units for transcendental functions
- Each SFU also contains four floating-point multipliers
- In total an SM has eight MAD and floating-point multipliers

SIMT

GPU execution model

- SIMT architecture is similar to SIMD design, which applies one instruction to multiple data lanes.
- The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes.
- A SIMD instruction controls a vector of multiple data lanes together, a SIMT instruction controls the execution and branching behavior of one thread.

SIMT

- In contrast to SIMD vector architectures, SIMT enables programmers to write thread level parallel code for independent threads as well as data-parallel code for coordinated threads
- SIMT - essentially a single thread of SIMD instructions
- Each SM's multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*
- Each SM manages a pool of 24 warps, with a total of 768 threads
- Each SM maps warp threads to the SP cores

Warp execution

- In each operation cycle, the SM warp scheduler selects one of the 24 warps
- An issued warp executes over four processor cycles
- The SP cores and SFU units execute instructions independently

ISA

- Support for floating-point, integer, bit, conversion, transcendental, flow control, memory load/store
- Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers
- Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root.
- Bitwise operators include shift left, shift right, logic operators, and move

Register File

Each SIMD processor (SM)

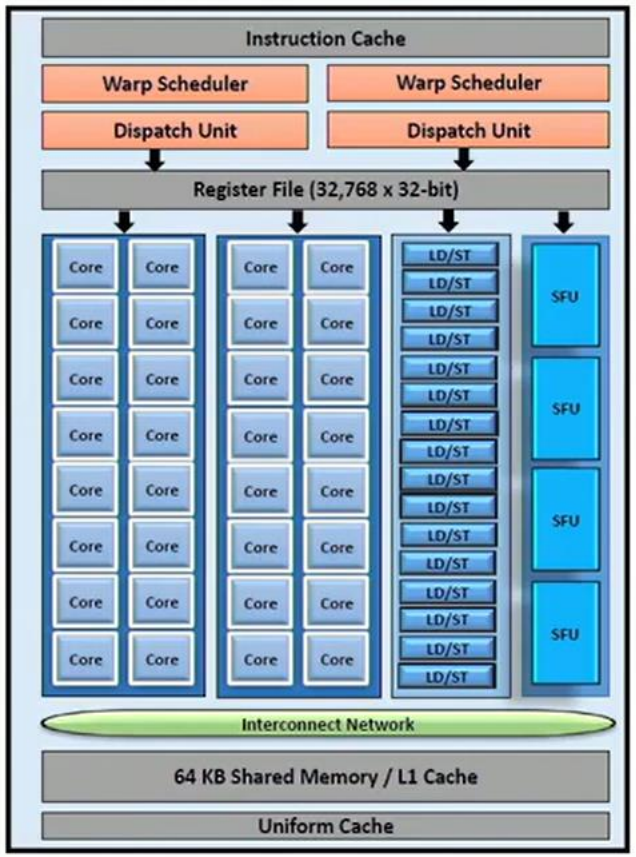
- has a large vector register file
- like a vector processor, these registers are divided logically across the SIMD Lanes, i.e. the SPs
- These numbers vary across across architecture families.

Fermi GTX 480 GPU

Has

- 16 SMs, total 512 CUDA cores
- Each SM has 32 SPs, 32,768 32-bit registers divided logically across executing threads
- Each SIMD Thread is limited to no more than 64 registers
- A warp has access to 64×32 registers which are 32 bit,

Fermi Streaming Multiprocessor (SM)



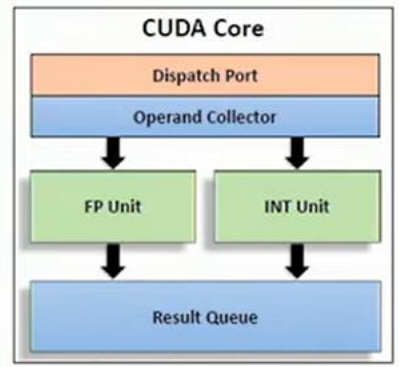


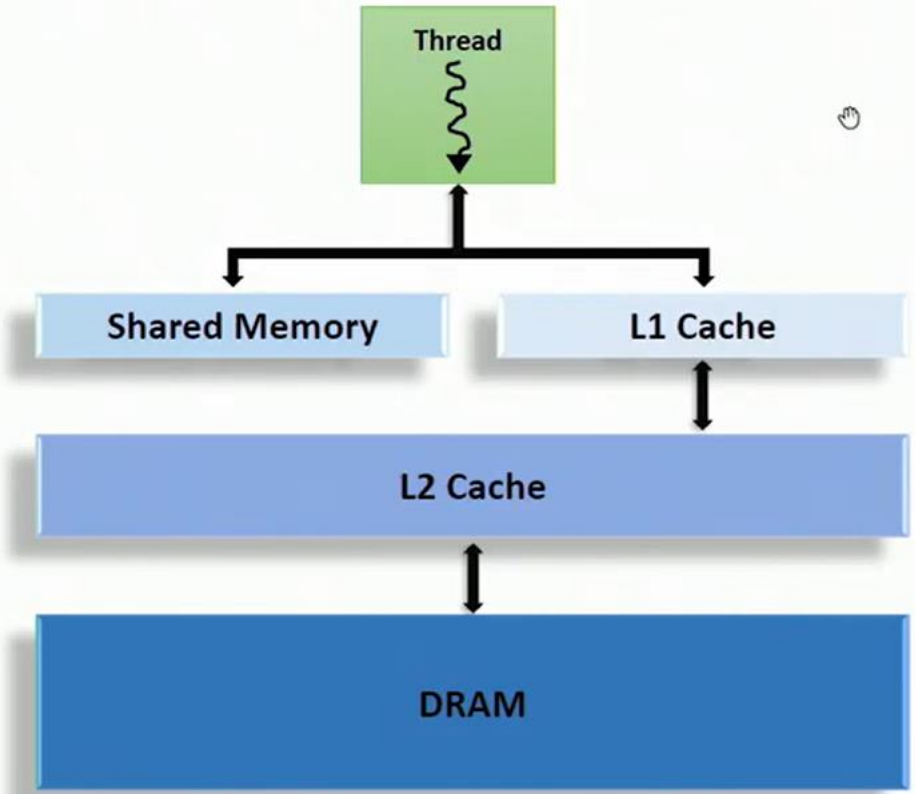
Figure: Single SP

- Each SM has 16 Load/store units (load/store data at each address to cache or DRAM.) - 16 SIMD lanes
- Each lane has 2048 registers
- Each SM has 4 SFUs, Each SP has one FP, one Integer ALU.
- ALUs also support Boolean, shift, move, compare, convert, bit-field

Memory Hierarchy

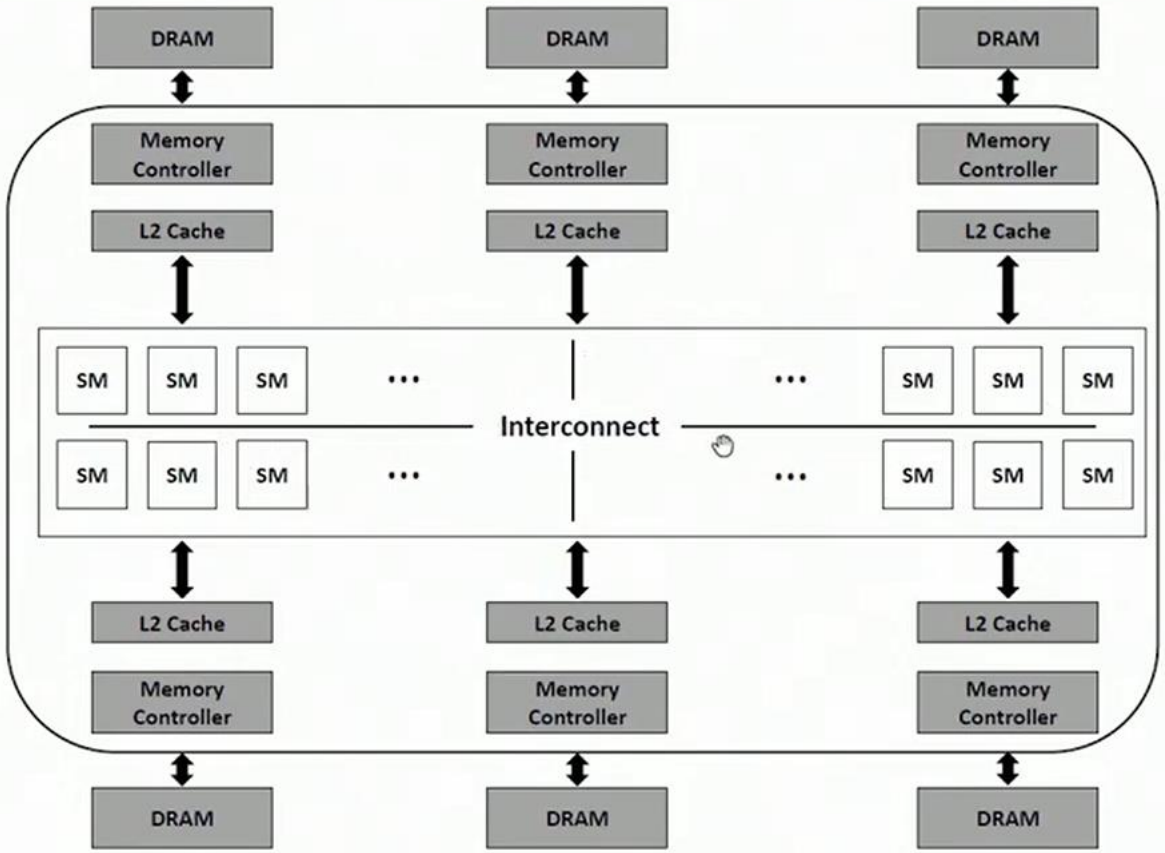
- Local memory for per-thread, private, temporary data (implemented in external DRAM)
- Shared memory for low-latency access to data shared by threads in the same SM
- Global memory for data shared by all threads of a computing application (implemented in external DRAM)

Fermi Memory Hierarchy



- Shared memory enables threads to cooperate, facilitates reuse of on-chip data, and reduces off-chip traffic.
- Each SM has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache.
- Source : NVIDIA Whitepaper

Fermi Memory Hierarchy



- L1 (Data) cache + Shared memory is private to SMs along with read-only texture and constant caches
- L2 is unified for all SMs, 6 high-bandwidth DRAM channels
- Compared to CPU, GPUs have larger register file, smaller L1/L2 cache with higher bandwidth
- Ref : "The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing" - Manish Arora

GPU ISA

- The instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set
- PTX (Parallel Thread Execution) provides a instruction set for compilers that remains same for different generations of GPUs
- PTX code gets translated to target hardware instructions while being loaded to GPU

PTX instructions

- format : opcode.type d, a, b, c;
- a,b,c, are source; d is destination operand
- Source operands are 32-bit or 64-bit registers or a constant value
- All instructions can be predicated by 1-bit predicate registers, which can be set by a set predicate instruction (setp)

GPUs becoming ubiquitous

GPUs have started finding wide usage in several domains where workloads have become intensive

- Mobile GPUs : ARM Mali, Adreno GPUs (Qualcom) - accelerate graphics as well as compute tasks
- NVIDIA in embedded space : Jetson TX/ Nano / AGX Xavier $\Rightarrow$ Multi core ARM CPU + 128-512 core GPU targeting AI / Deep Learning tasks
- NVIDIA Drive : for implementing autonomous car and ADAS functionality powered by deep learning (Tesla cars !!)

GPUs as mobile workload accelerators

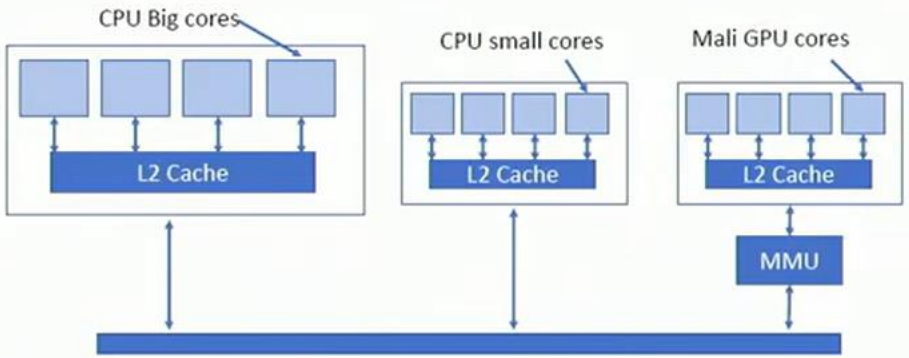


Figure: Typical architecture of an ARM based Mobile SoC

- Objective : Maximize performance and reduce power consumption
- Developers need to map the workload across the whole CPU + GPU system
- RenderScript for Android SDK, OpenCL - language support for data parallel computation on Mobile devices

Integrated GPUs in Desktop Systems

With the release of AMD's Fusion and Intel's Ivy Bridge architecture (i3, i5, i7) in 2011, the trend of fused CPU-GPU architectures started

- CPU and GPU access the same physical memory such that zero-copy transfers can be employed
- Zero- copy transfers ensure coherency; translate pointers to memory buffers for the common CPU and GPU address space, but do not actually transfer data.
- Bad effect - CPU and GPU compete for memory bandwidth of the shared physical memory

Integrated GPUs in Desktop Systems

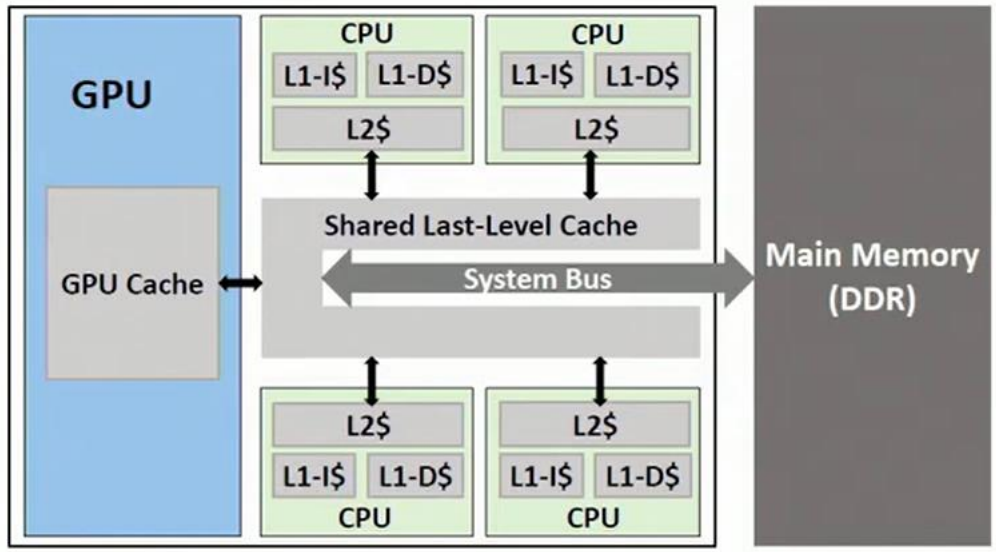


Figure: Fused CPU-GPU with shared LLC

- In more recent architectures, Intel Broadwell and beyond, CPU and GPU were further integrated
- They access the shared last level cache (LLC)
- This helps in CPU and GPU executing computational kernels on the same data in parallel collaboratively (LLC enables cache coherence between CPU and GPU)
- "Co-Scheduling on Fused CPU-GPU Architectures with Shared Last Level Caches" - Henkel et. al.

Jetson Series from NVIDIA

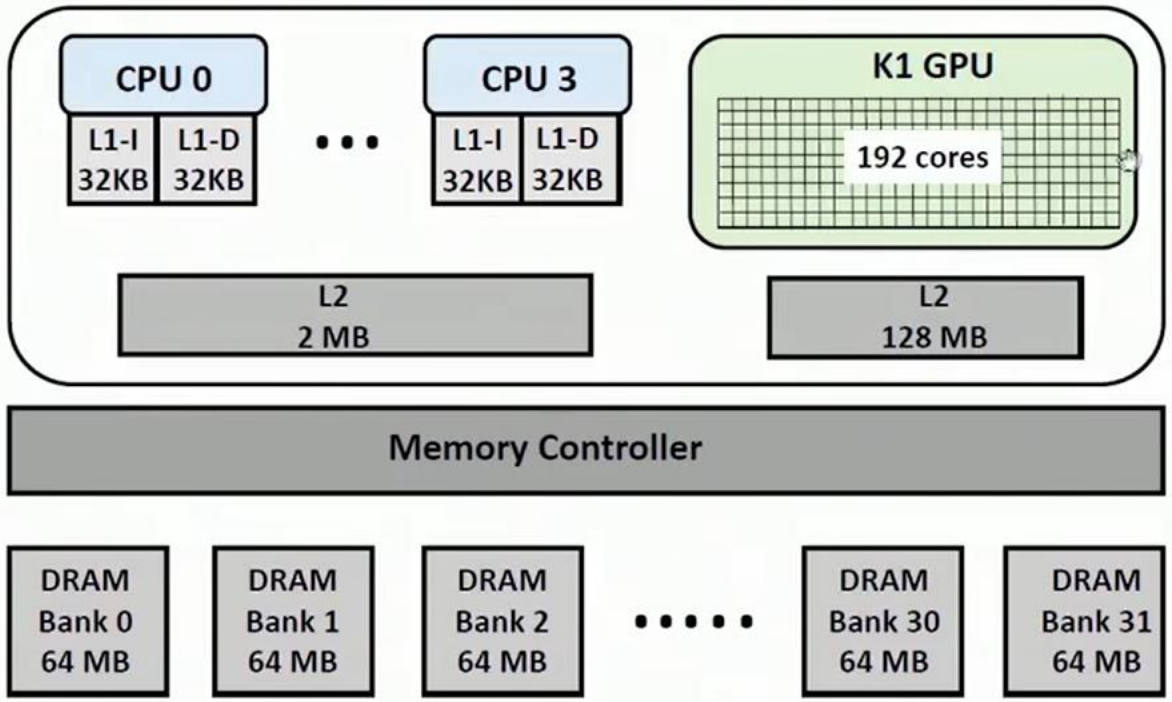


Figure: Jetson TK1

- TK1 SOC incorporates a quad-core 2.32 GHz 32-bit ARM machine and an integrated Kepler GK20a GPU
- The CPUs share a 2-MB L2 cache
- The GPU has 192 cores and a 128-KB L2 cache
- The CPU also has 'little' ARM cores (not shown) - low power, low performance

NVIDIA Drive series of systems

Figure: Source- Wiki, NVIDIA Drive PX Platform

- The Nvidia Drive PX 2 is based on 1/2 Tegra SoCs where each SoC contains 2 Denver cores, 4 ARM A57 cores and a GPU from the Pascal generation
- Useful for implementing high throughput real time neural net processing - self driving / drive assist systems

Compute Unified Device Architecture [CUDA]

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

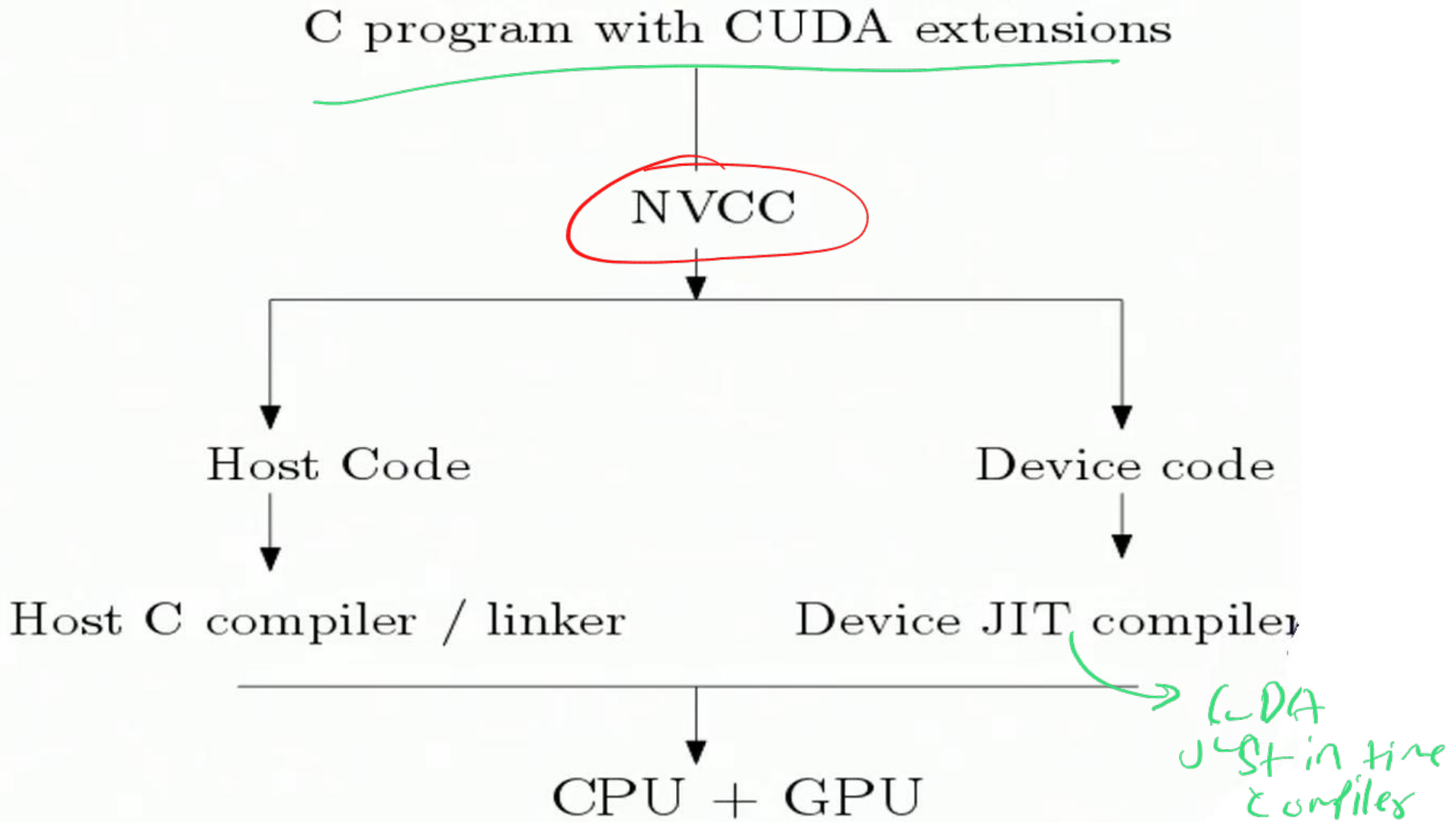
CUDA program structure

- Host code →
  i) executes on CPU
  ii) controls flow and initiate calls to GPU code
- Device code →
  i) executes on GPU
  ii) perform N computations

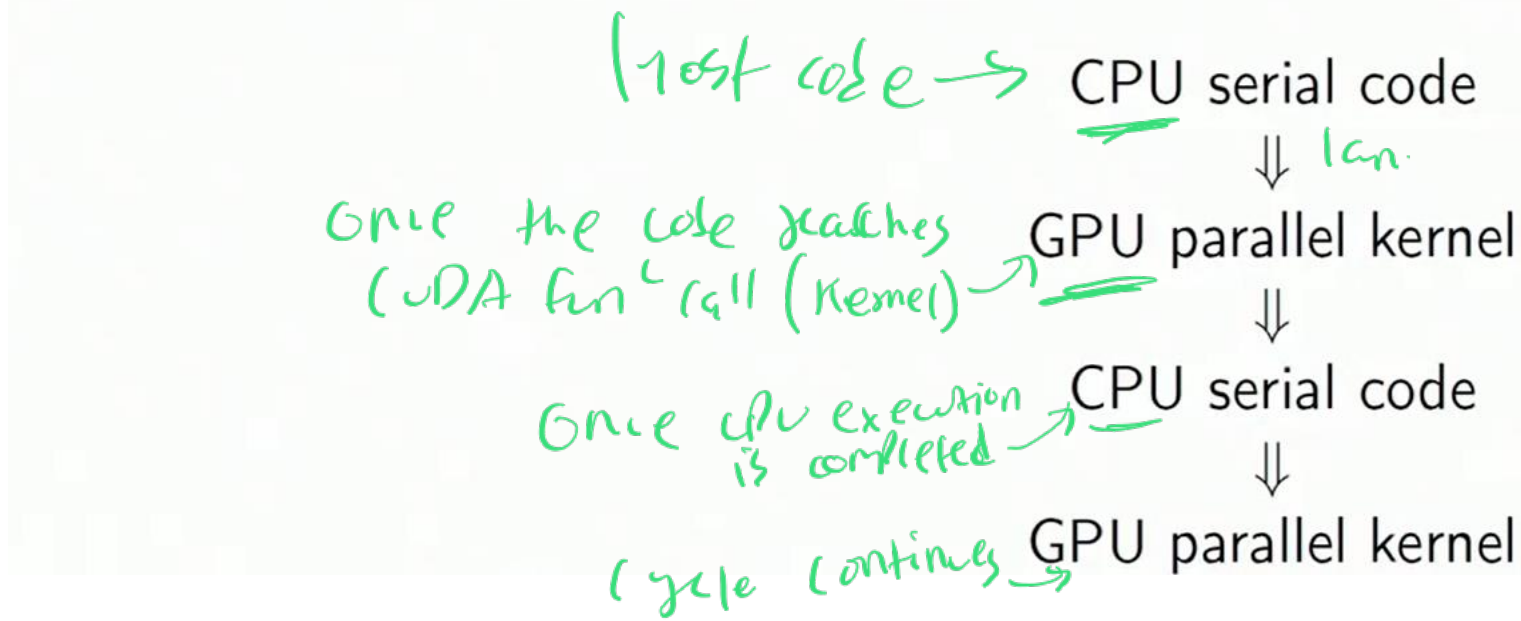
- *host code* for a host device (CPU)
- device code for GPU(s)
- Any C program is a valid CUDA host code
- In general CUDA programs (host + device) code cannot be compiled by standard C compilers

NVIDIA C compiler (NVCC)

The compilation flow



The execution flow



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

Headers & Kernel declaration

kernel definition

CPU side initiation

```
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(float*, float*, float*, int);
/*------------------------------------------------------------*/
__global__
void vectorAdd(float* A, float* B,
float* C, int n){ //CUDA kernel definition
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
         C[i] = A[i] + B[i];
}
/*------------------------------------------------------------*/
void vecAdd(float* h_A, float*h_B,
float* h_C, int n)
{//host program
   int size = n* sizeof(float);
   float *d_A=NULL, *d_B=NULL, *d_C=NULL;

   // Error code to check return values for CUDA calls
   cudaError_t err = cudaSuccess;
```

9i1

Device Memory Allocation (GPU - side)

```
err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_B, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector C (error code %s)!
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
```

Host to Device Data Transfer

```
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector A from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector B from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Kernel Launch

```
int threadsPerBlock = 256;
int blocksPerGrid =(n+threadsPerBlock-1)/threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n",threadsPerBlock,
    blocksPerGrid);
vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A, d_B, d_C, n);
err = cudaGetLastError();
// device function (CUDA kernel) called from host does not have return type
//CUDA runtime functions (execute in host side) can have return type

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n".
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Device to Host Memory Transfer

```
printf("Copy output data from the output device to the host memory\n");
 err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 if (err != cudaSuccess)
 {
   fprintf(stderr, "Failed to copy vector C from device to host (error code %s
       )!\n", cudaGetErrorString(err));
   exit(EXIT_FAILURE);
 }
 cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
 // Verify that the result vector is correct
 for (int i = 0; i < n; ++i)
 {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        {
          fprintf(stderr, "Result verification failed at element %d!\n", i);
          exit(EXIT_FAILURE);
        }
 }
 printf("Test PASSED");
} // End of Function
```

Compile and Run

```
nvcc kernel.cu host.cu -o output

./output
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
```

Observations

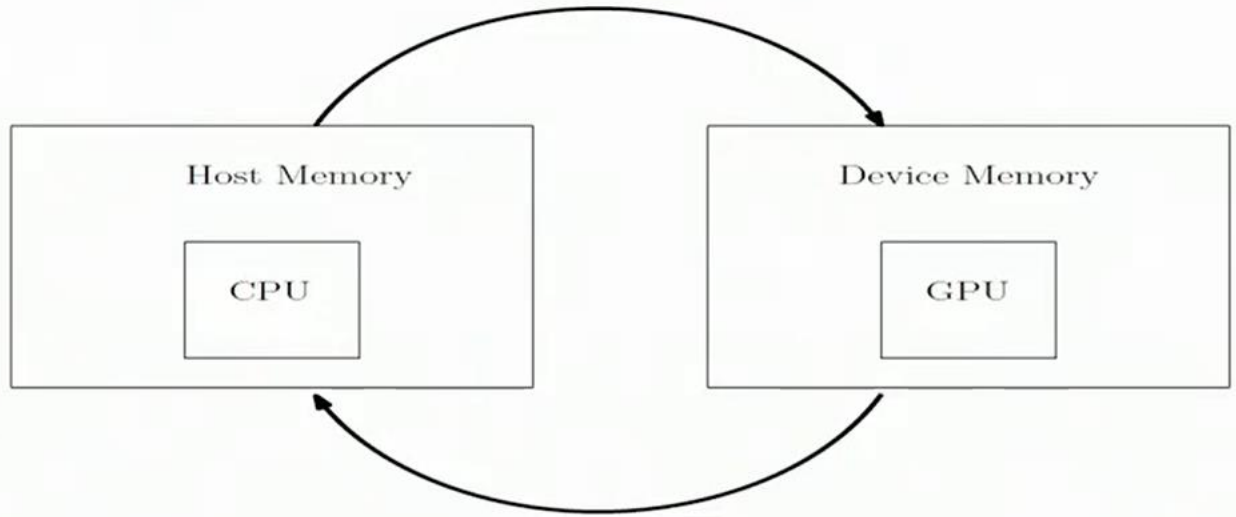


Figure: CPU/GPU Mem Layout

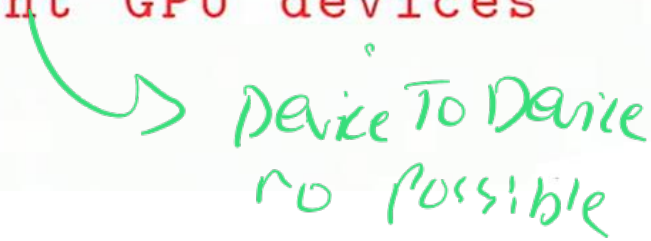
- cuda.h → includes during compilation CUDA API functions and CUDA system variables
- h_A, h_B, h_C → arrays mapped to main memory locations

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, h_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(h_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
```



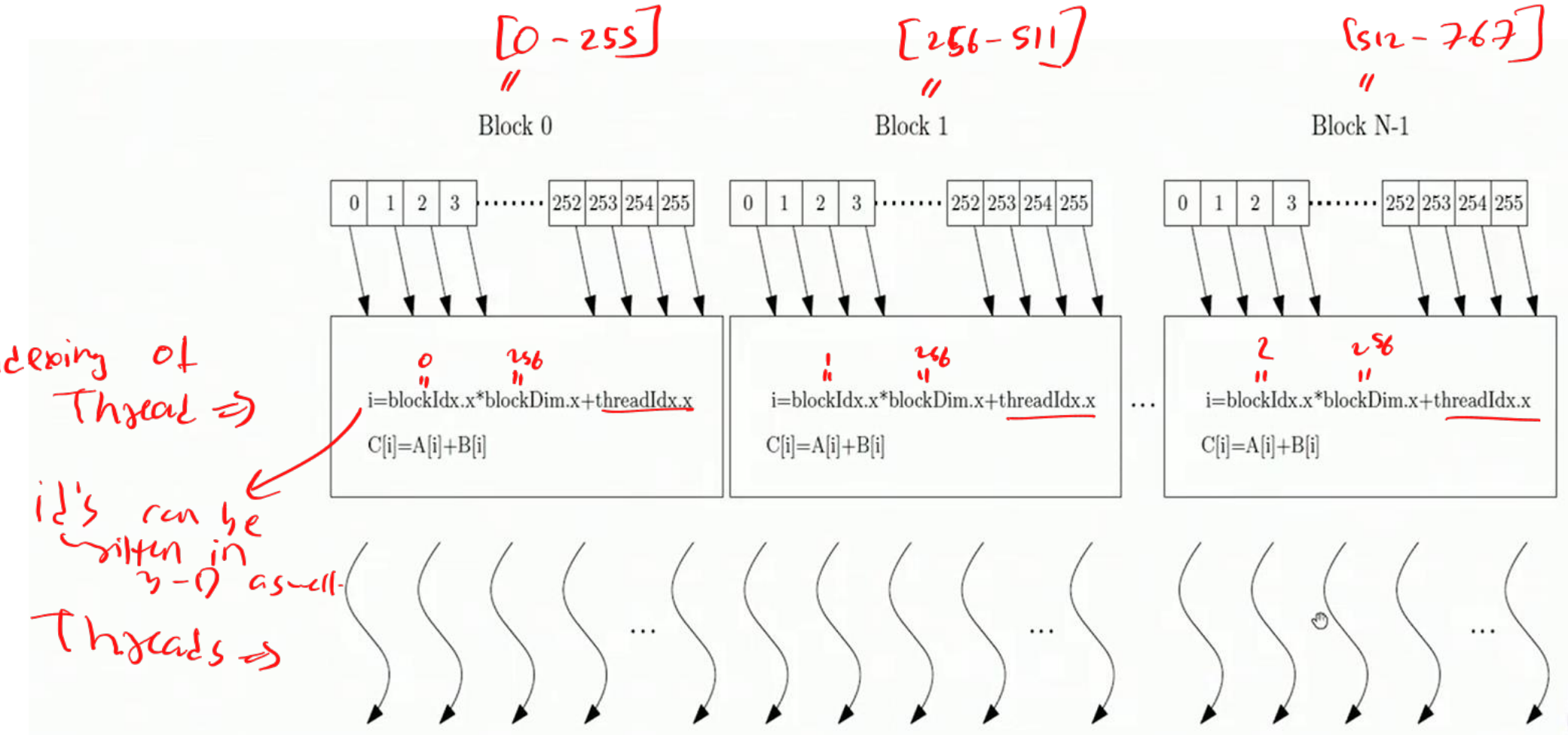
CUDA kernel

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

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

High level = n/256 blocks
low level = 256 threads

- 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


[0 - 255]
[256 - 511]
[512 - 767]
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]
0
256
1
256
2
258
Indexing of Thread ⇒
id's can be written in 3-D as well-
Threads ⇒

Kernel specific system vars

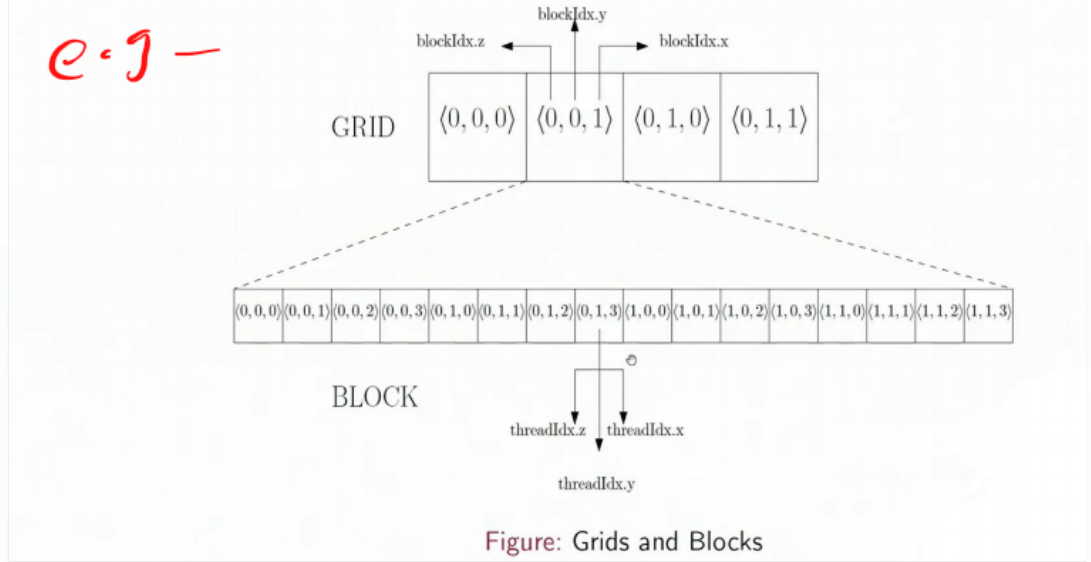
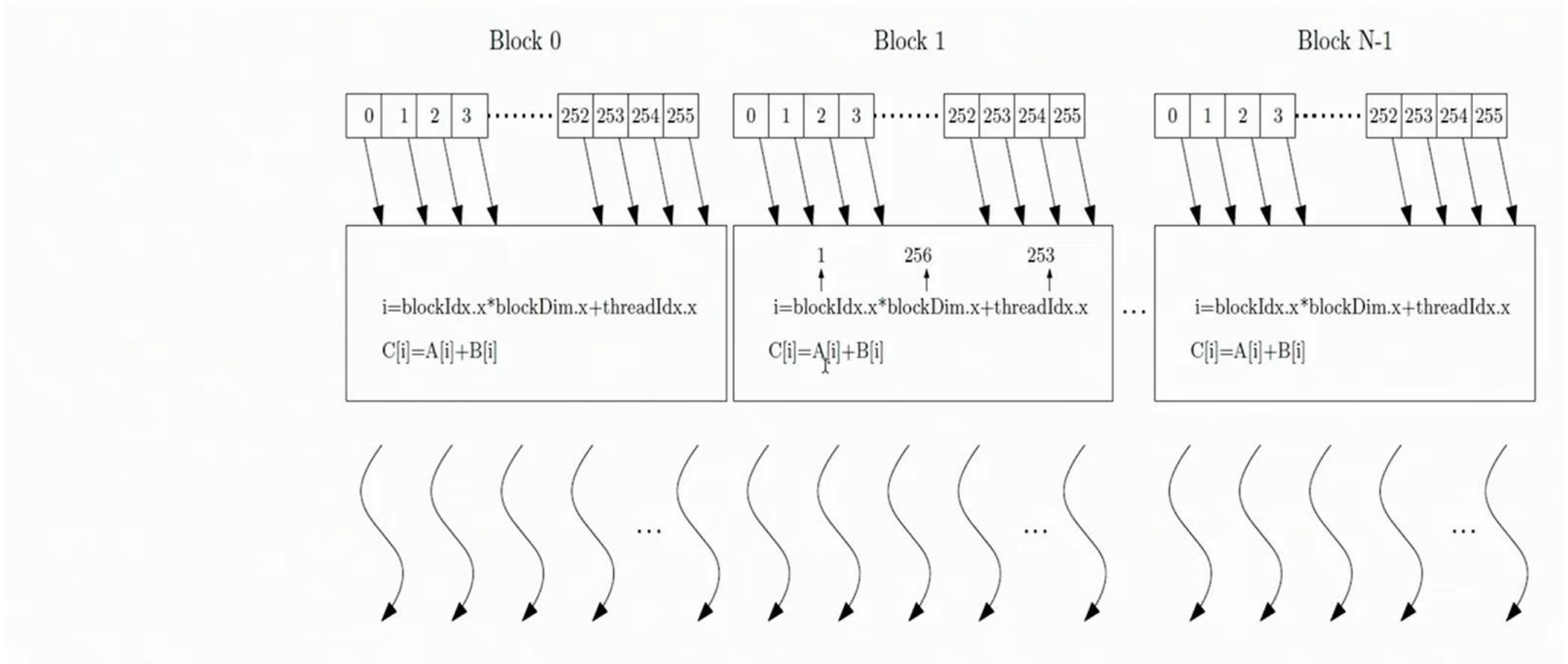


Figure: Grids and Blocks

- **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 vectorAdd(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
1 256 253
i=blockIdx.x*blockDim.x+threadIdx.x
C[i]=A[i]+B[i]

Function declaration Keywords

```
__global__
void vectorAdd(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.

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;
  }
}
```

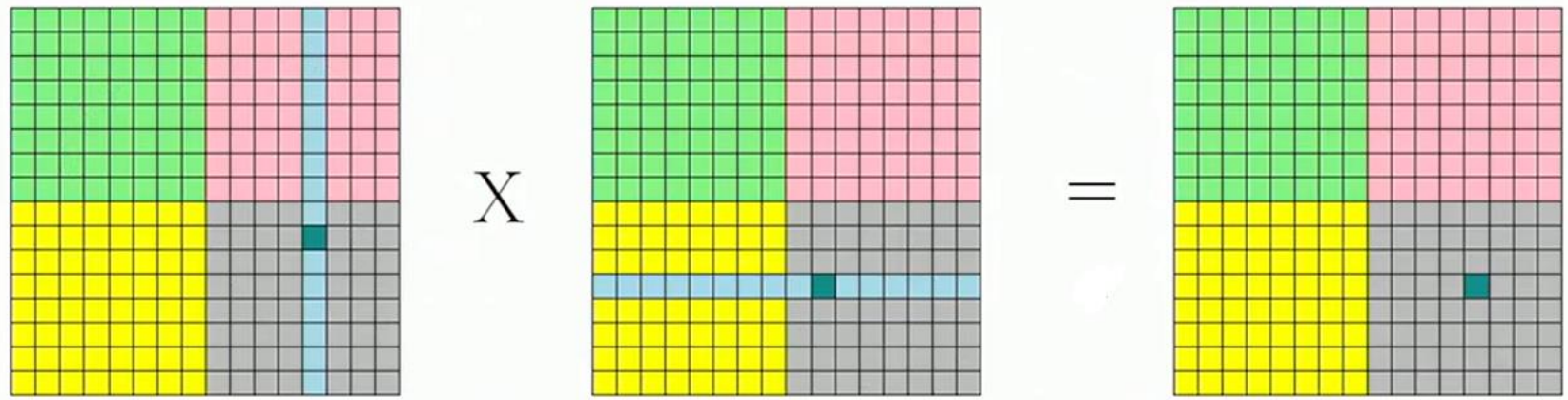


Figure: Matrix Multiplication

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

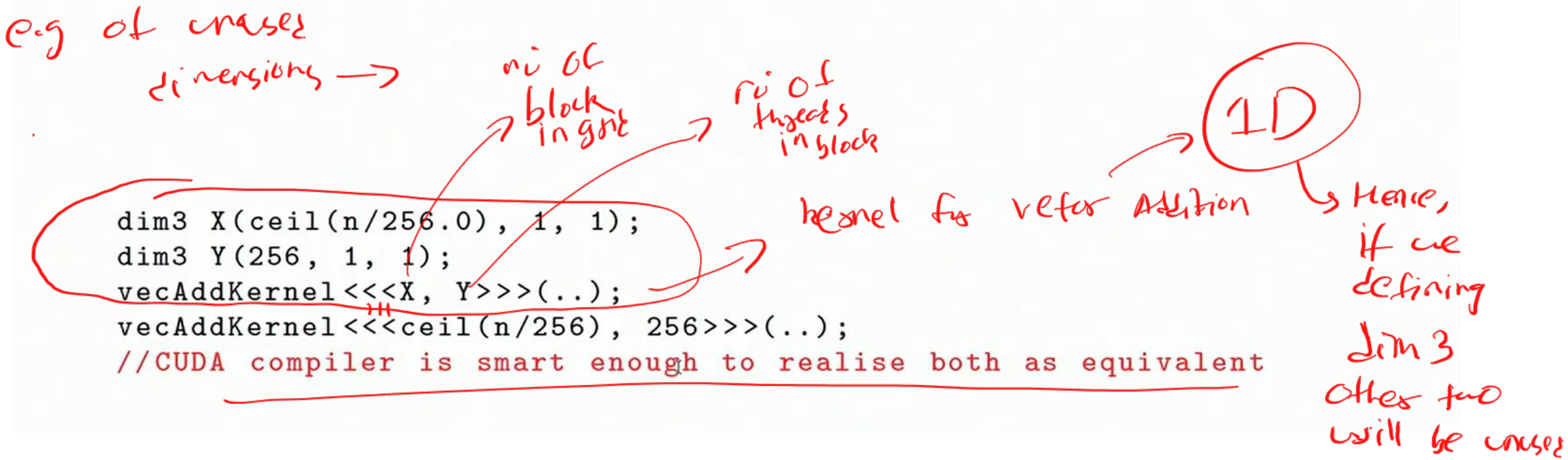
Multi dimensional block

when a Kernel is launched ⇒ multiple threads are launched
↓
These threads are arranged in form of blocks
↓
And these blocks are further arranged in form of grid

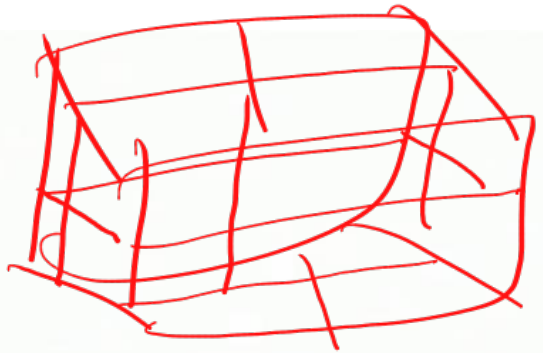
In general

- 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



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) - √ 512 × 1 × 1 threads
- (8, 16, 4) - √ 8 × 16 × 4 —
- (32, 16, 2) - √ 32 × 16 × 2 —
- (32, 32, 32) - × 32 × 32 × 32 —

Multi dimensional grid, block declaration

e.g – Kernel launch of 3-D block

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

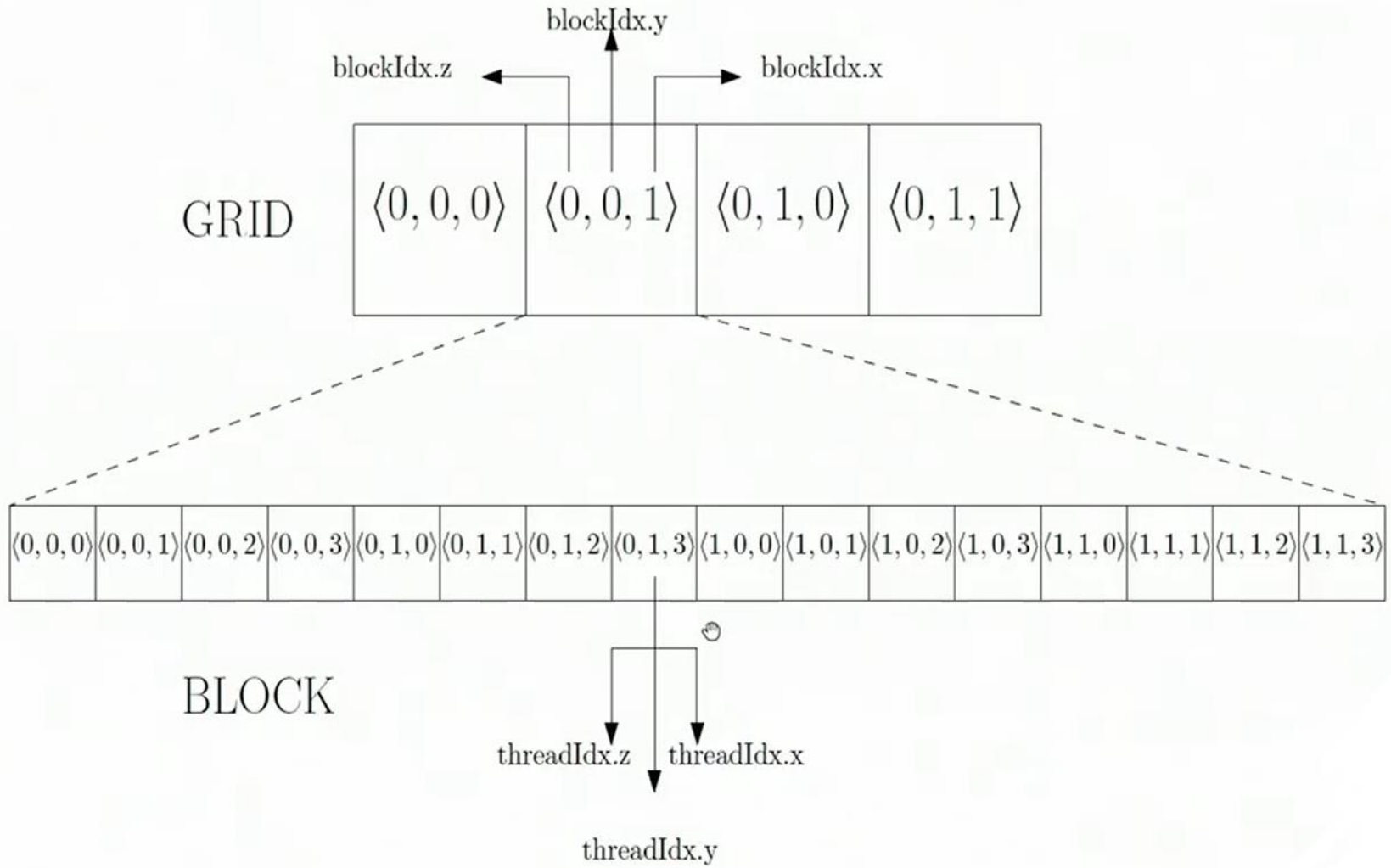
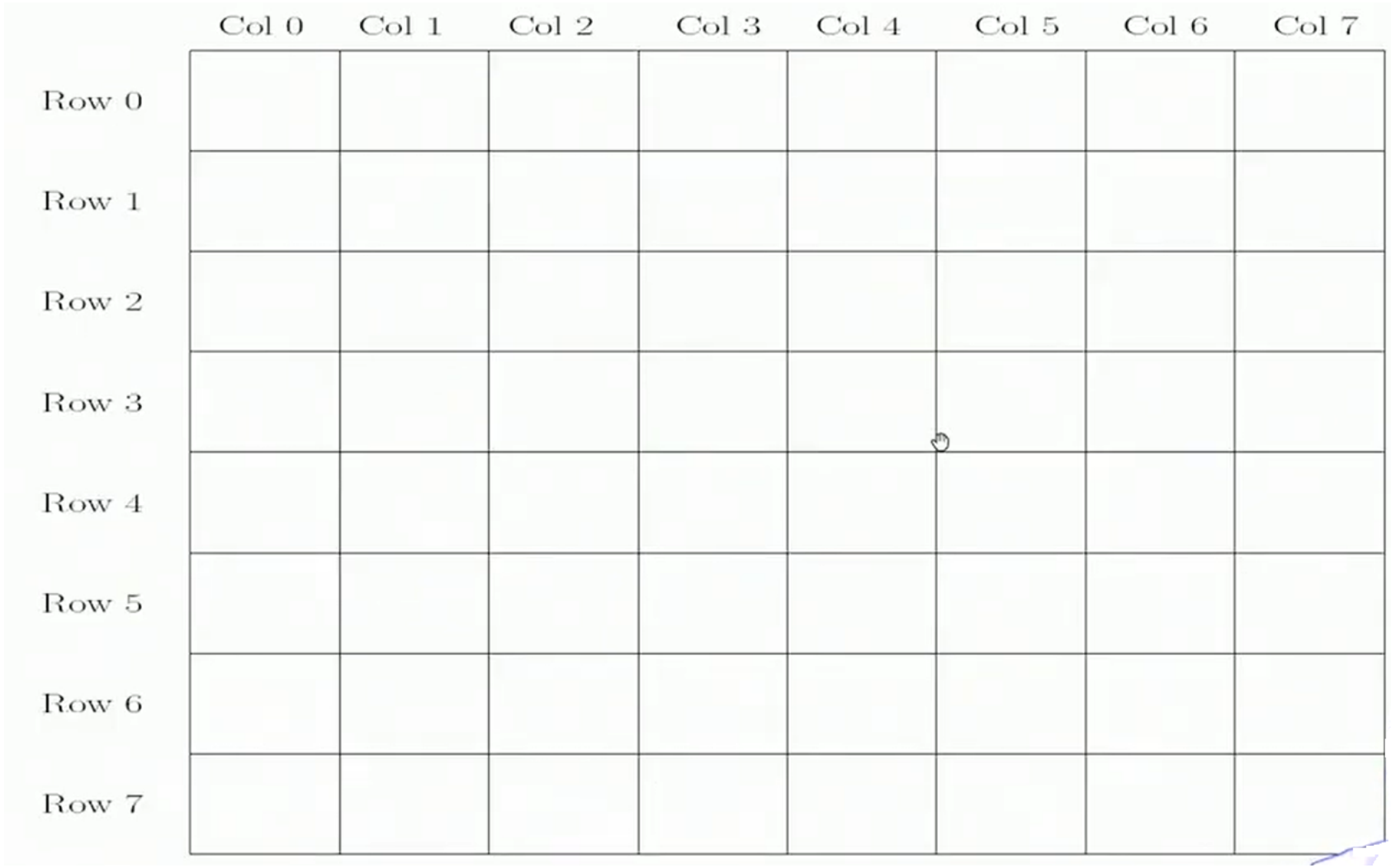


Figure: Grids and Blocks

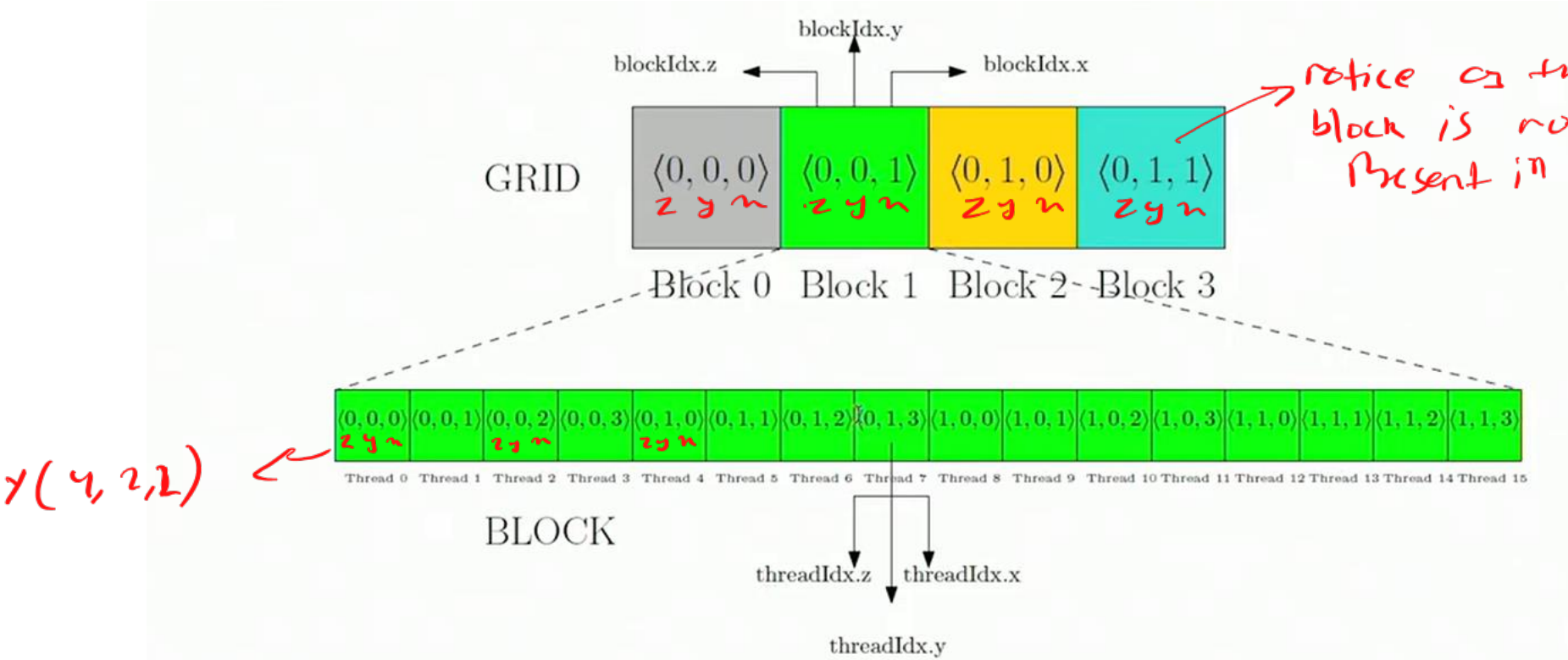

Col 0
Col 1
Col 2
Col 3
Col 4
Col 5
Col 6
Col 7
Row 0
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7

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



$$\text{blockNum} = \text{blockIdx.z} * (\text{gridDim.x} * \text{gridDim.y}) + \text{blockIdx.y} * \text{gridDim.x} + \text{blockIdx.x}$$

$$\text{threadNum} = \text{threadIdx.z} * (\text{blockDim.x} * \text{blockDim.y}) + \text{threadIdx.y} * (\text{blockDIm.x}) + \text{threadIdx.x}$$

$$\text{globalThreadId} = \text{blockNum} * (\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z}) + \text{threadNum}$$

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

Block 0.

Block 1

Block 2

Block 3

i = globalThreadId / NumCols j = globalThreadId % NumCols

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

↳ To identify Position in the terple of global thread Id

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

e.g— Mapping 2D-kernel

for Kernel-Data mapping

NumCols = blockDim.x * gridDim.x

NumRows = blockDim.y * gridDim.y

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

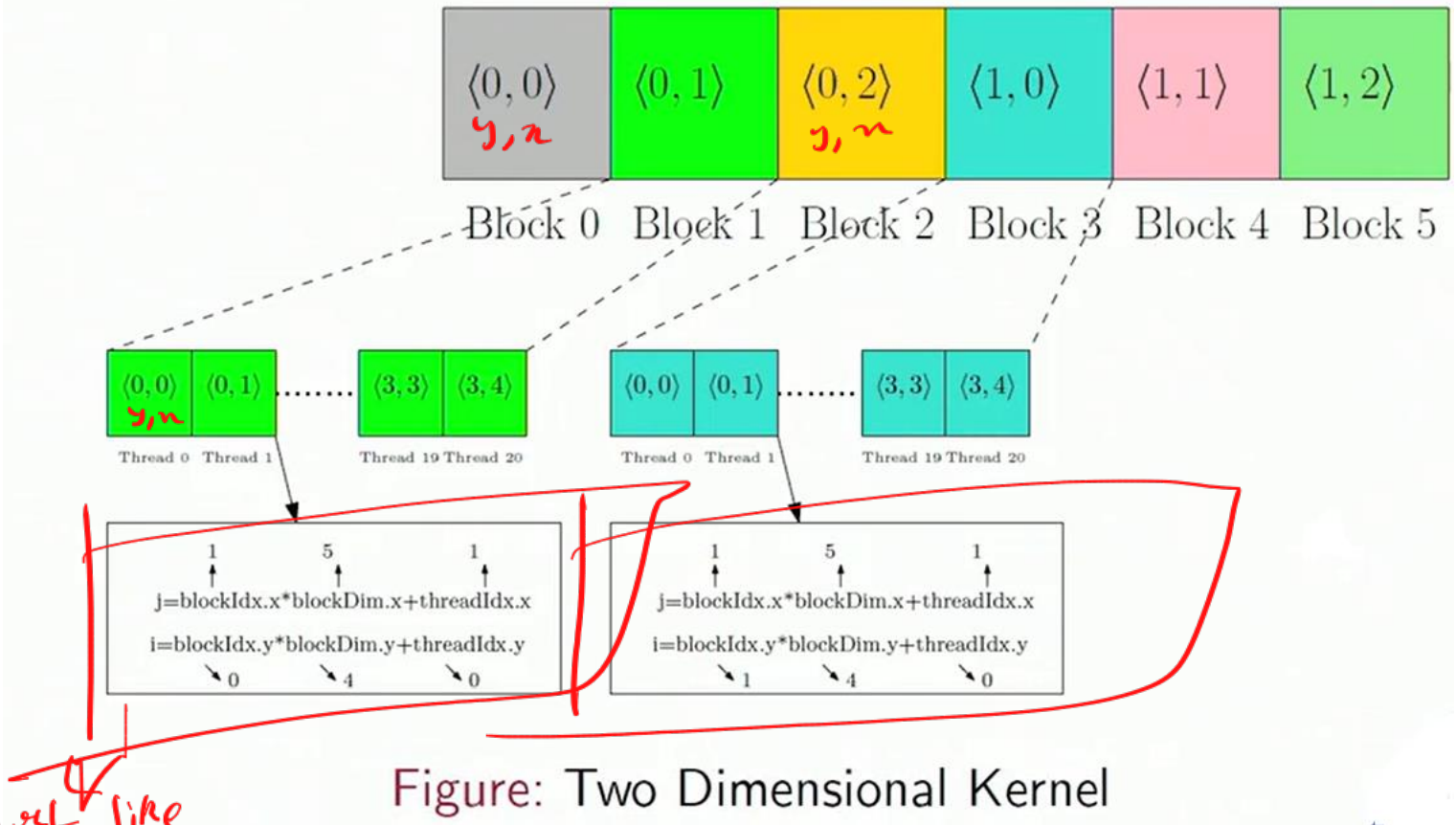


Figure: Two Dimensional Kernel

Just like matrix mul kernel

e.g- Mapping 3-D Kernel

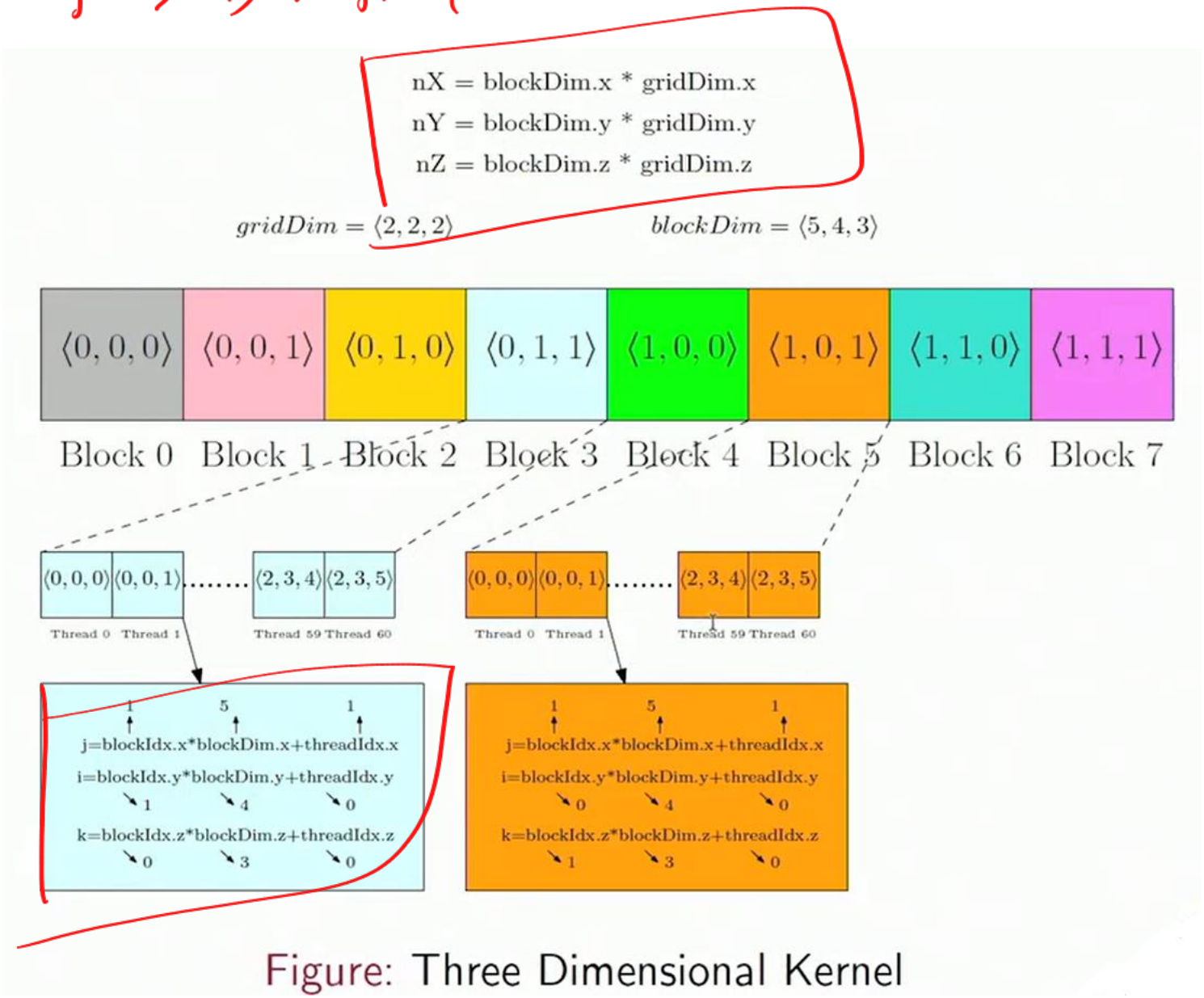


Figure: Three Dimensional Kernel

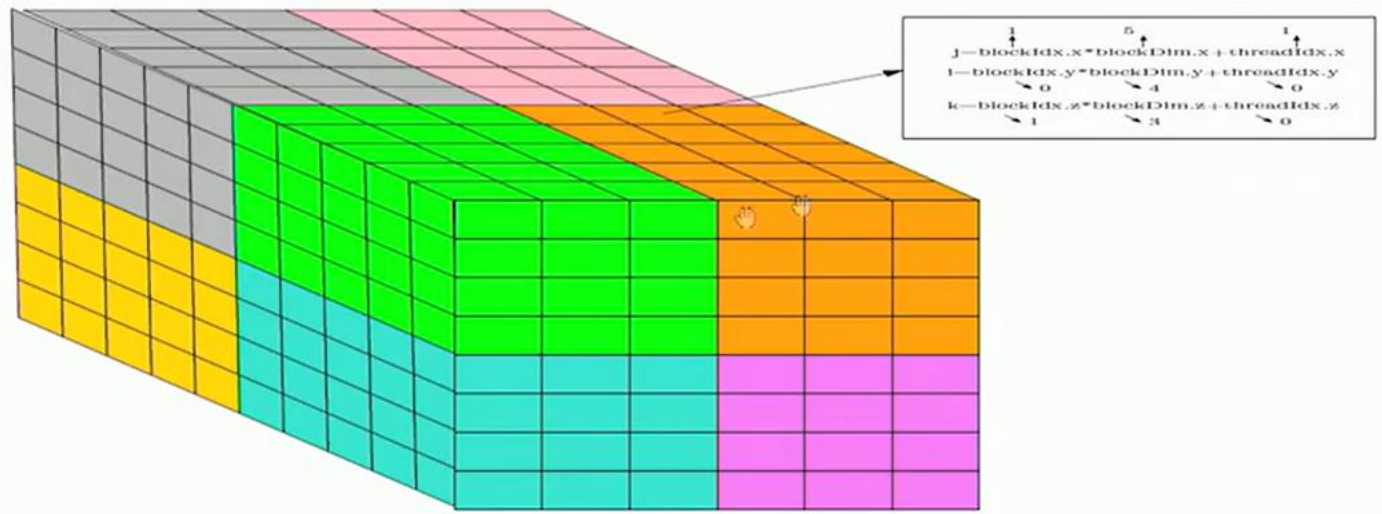


Figure: Three Dimensional Kernel-Data Mapping

Synchronization

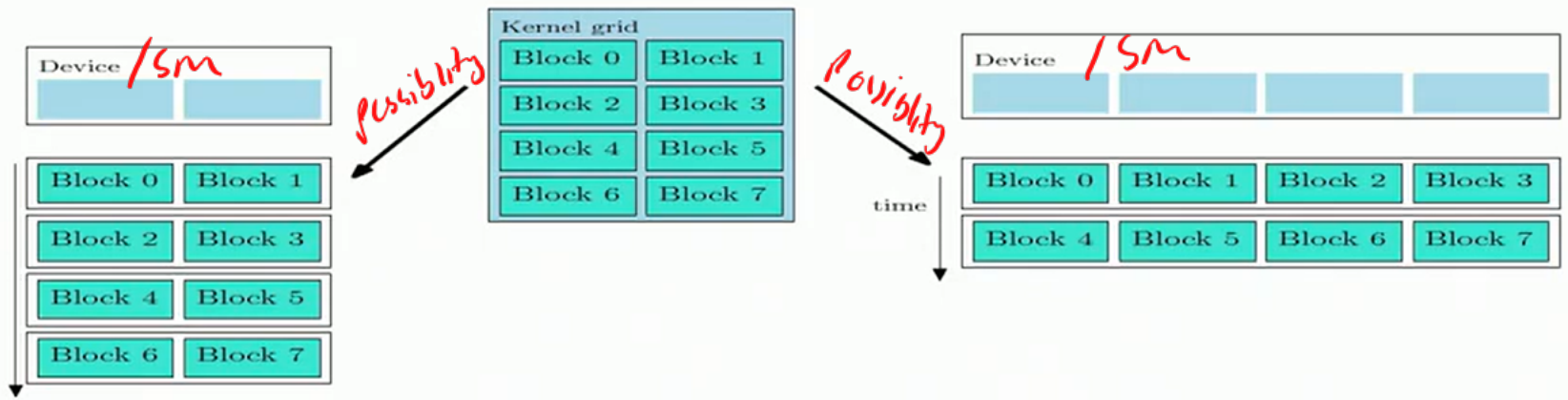


Figure: Mapping Blocks to Hardware

- 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.

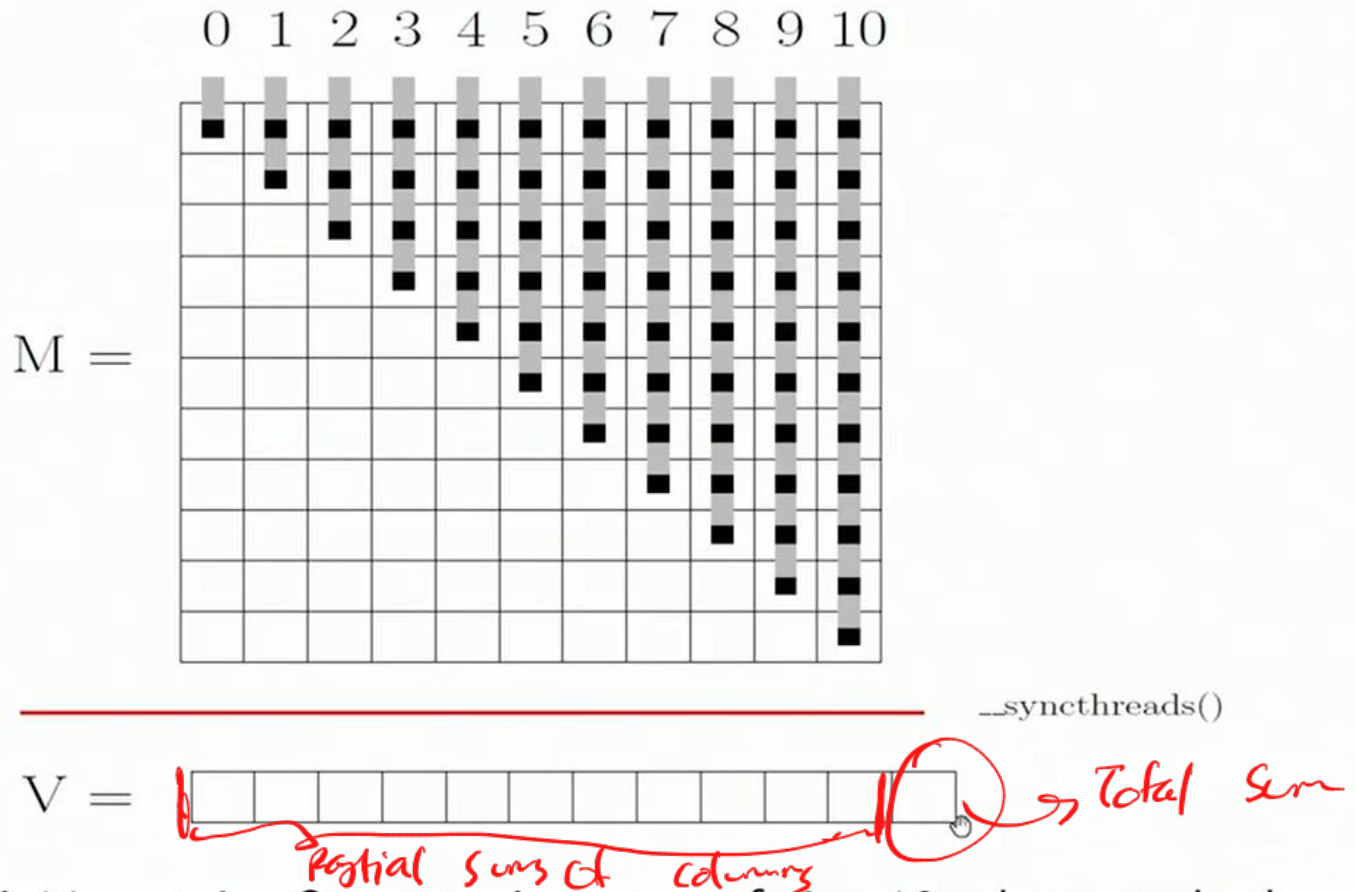


Figure: Input: A 11 X 11 matrix, Output: A vector of size 12 where each element represents the column sums and the last element represents the sum of the column sums.

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){
        Thread Idx.
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();
```

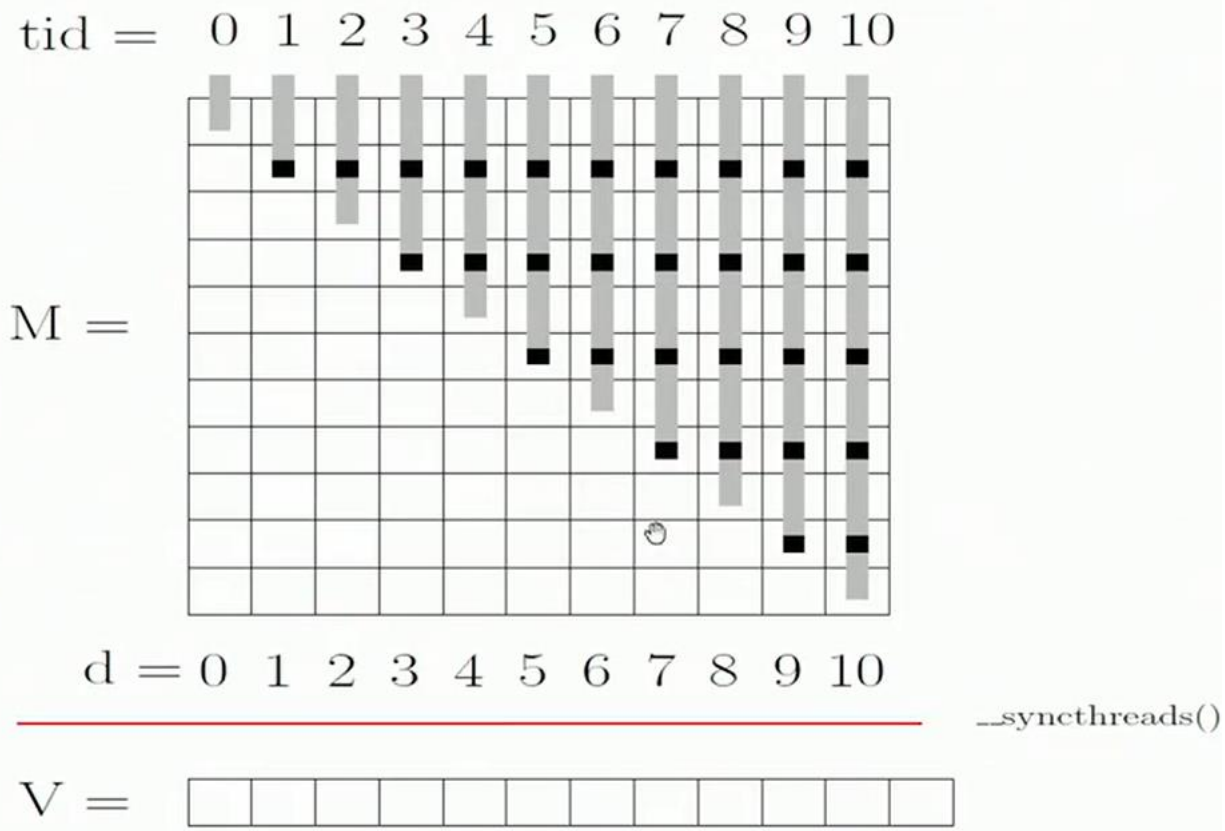


Figure: A variant of SumTriangle where only the elements at odd indices

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;
}
```

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();
```

Reduction

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

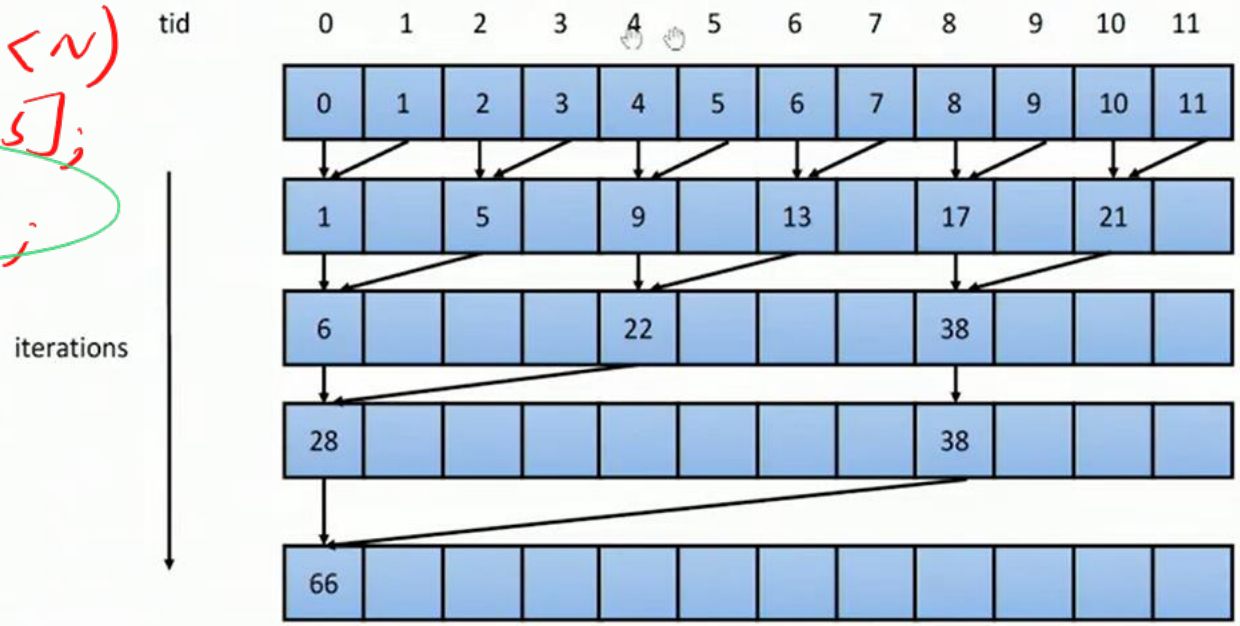


Figure: Reducing an array of 12 elements