Introduction to GPU

L03: GPU ARCHITECTURE

Dina Tantawy
Computer Engineering Department
Cairo University

Agenda

- Review
- Architecture of GPU
  - SMs
  - Thread Scheduling
    - Warps
    - Barriers
  - Shared memory.

Vector Addition, Explicit Memory Management

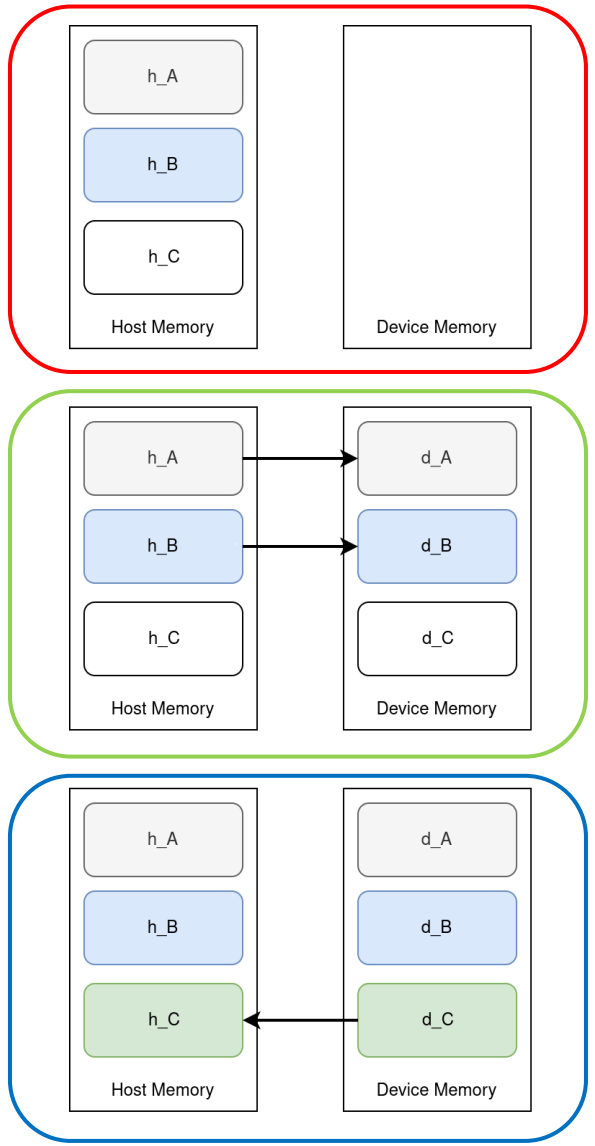
```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation code – to be shown later
    vecAddKernel<<<1,256>>>(d_A, d_B, d_C, n);

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



Example: Vector Addition Kernel

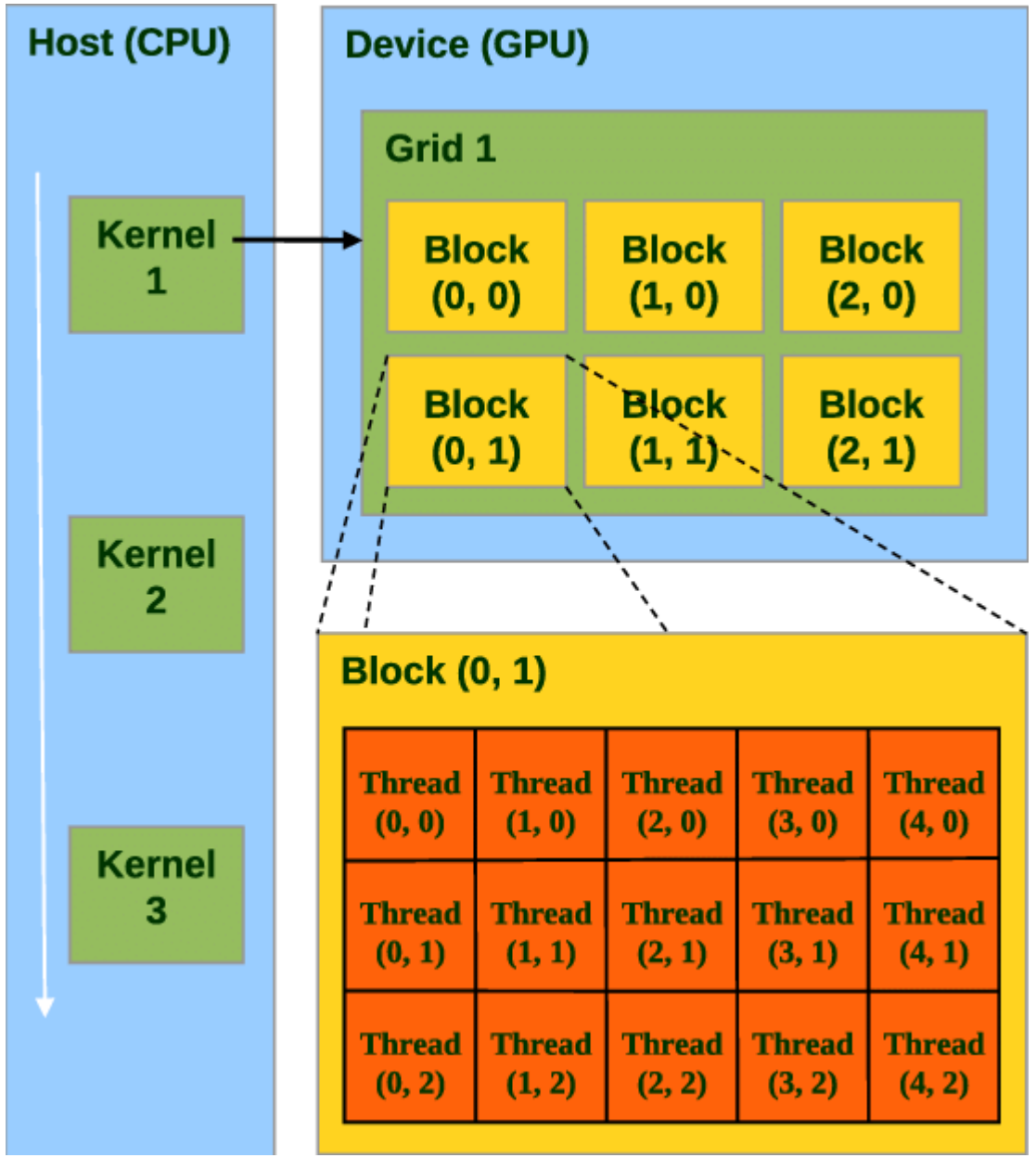
Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__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];
}
```

Computational Grid

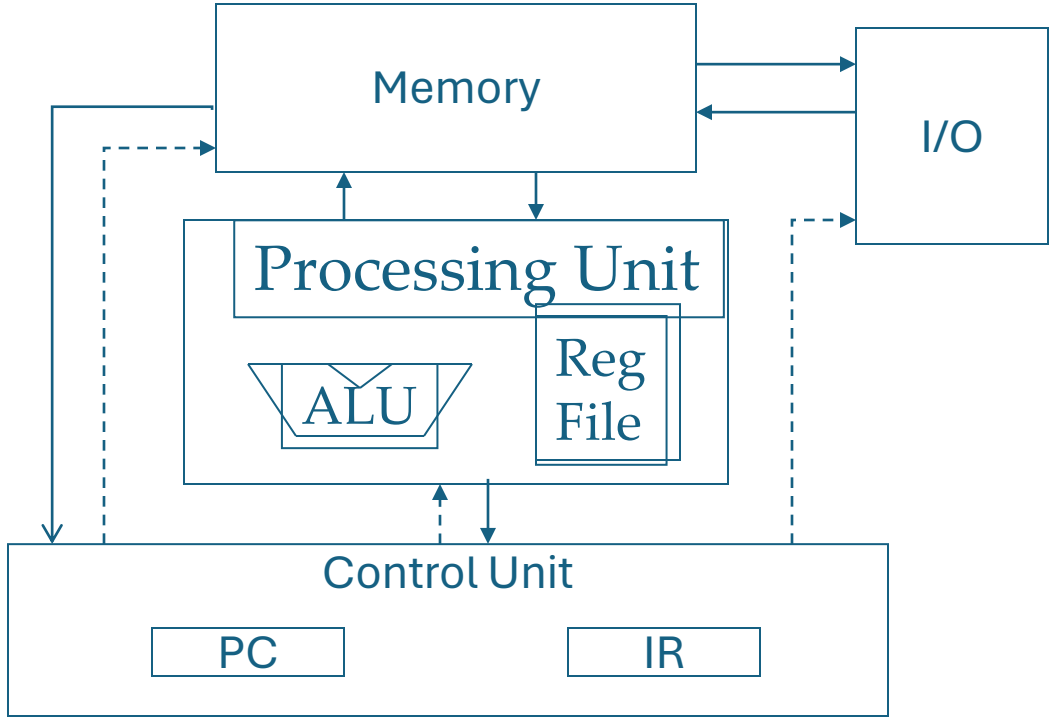
- Computational Grid consists of a number of blocks
- Each block consists of a number of threads.



A Thread as a Von-Neumann Processor

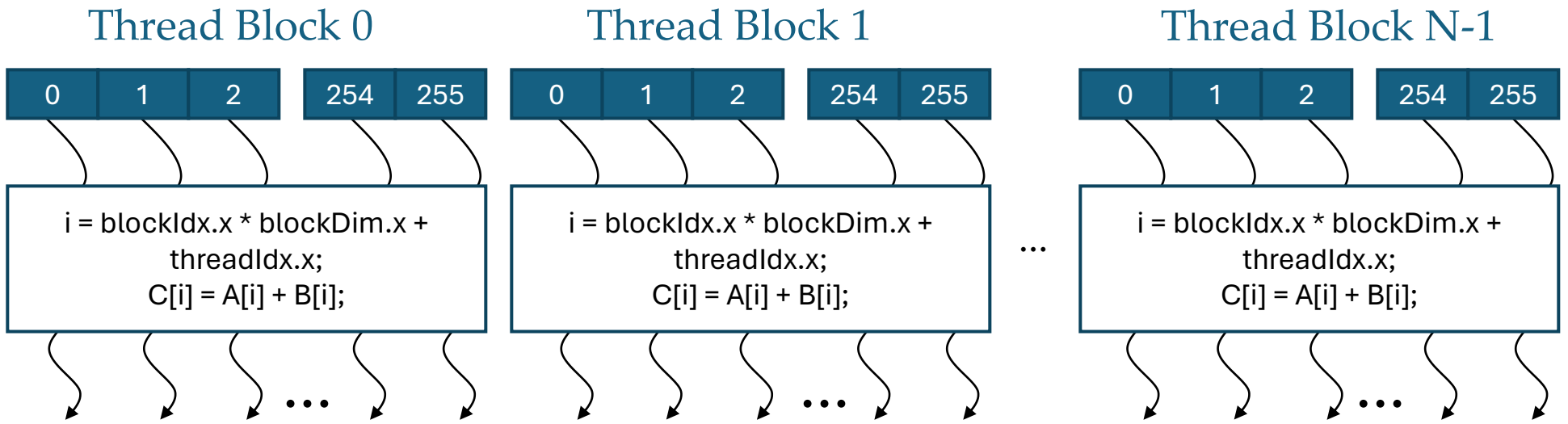
A thread is a “virtualized” or “abstracted” Von-Neumann Processor

kol thread shayfa el denya bnsblha el shkl da.

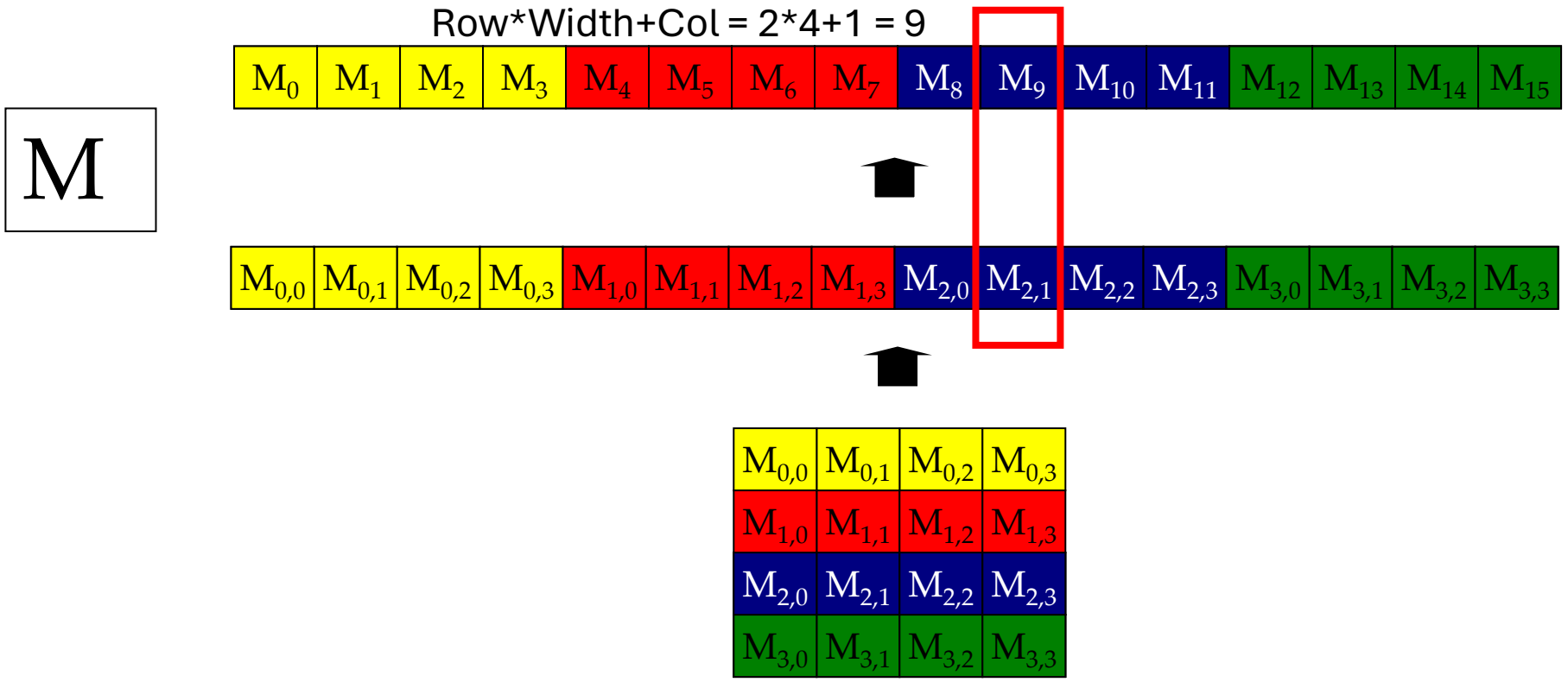


Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  - Threads in different blocks do not interact



Row-Major Layout in C/C++ (2D)



Into the GPU

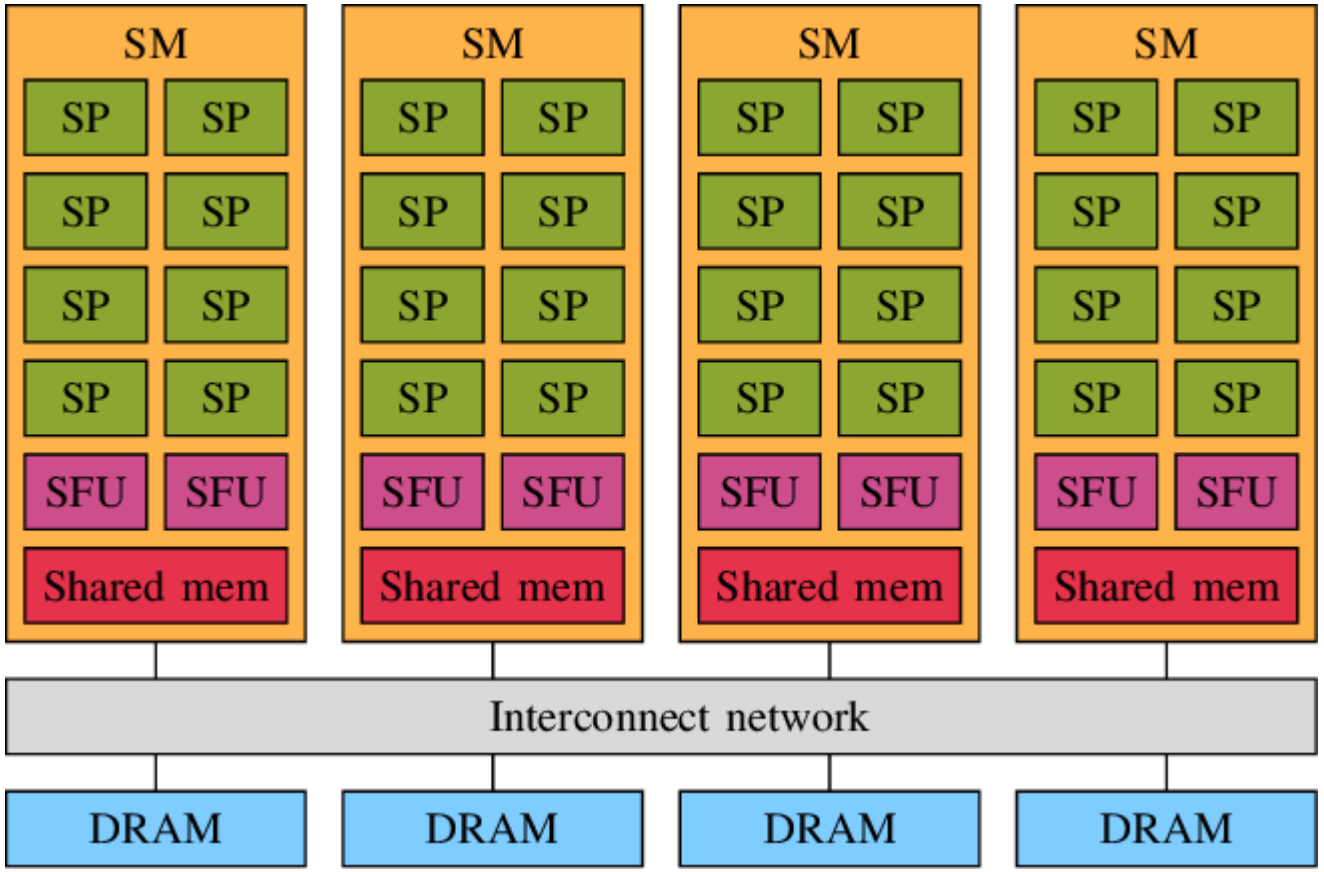
3dd el threads elly mwguda theoritically, hwa 3dd el cores elly 3ndna.

SMs

The NVIDIA GPU consists of several Streaming Multiprocessors (SMs), four are shown in this figure. Each SM holds several Scalar Processors (SPs), usually eight.

Each SM also has two Special Function Units (SFUs) and a small shared memory (Shared mem).

All SMs access the off-chip DRAM via the interconnect network

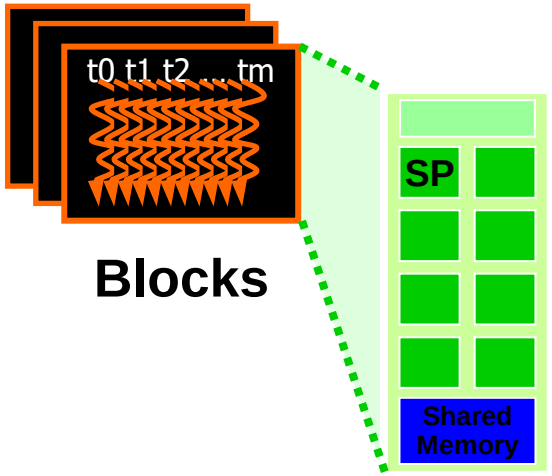


bn2sm el data le agza2 ktera, w da 34an nsr3 3mlyt el retrieving, 34an mntzn2sh fe el width bta3 el busses bta3 el ram el wahda.

bma enu 3ndy 8, yeb2a my2drsh yshghl aktur mn 8 threads per cycle, tb omal ezay 3ndi 2048 threads?
da 34an bbasata, lw fe thread menhom hya5ud w2t aktur, nsebo w nru7 ngeb wahed tany 3lama da y5ls, 34an yeb2a shghal very highly pipelined.

Executing Thread Blocks

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to 32 blocks to each SM as resources allow.
  - Volta SM can take up to 2048 threads
    - Could be 256 (threads/block) * 8 blocks
    - Or 512 (threads/block) * 4 blocks, etc.
    - A max of 1024 thread per block
- SM maintains thread/block idx
- SM manages/schedules thread execution
- The basic unit of execution is “warp”



All numbers in those slides are examples, those numbers are architecture-specific according to compute capabilities

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 4X4, we have 16 threads per Block.
  - Each SM can take up to 2048 threads, which translates to 128 Blocks.
  - However, each SM can only take up to 32 Blocks, so only 512 threads will go into each SM!

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 8X8, we have 64 threads per Block.
  - Since each SM can take up to 2048 threads, it can take up to 32 Blocks and achieve full capacity unless other resource considerations overrule.

  - For 30X30, we would have 900 threads per Block. Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

el n5rug beh mn el 7eta de, en t2semt el blocks, wl threads, btfr2 m3aya gedan fl performance bta3y, ana dayman 3ndy max number of threads, w max number of blocks.
w mynf34 eny a2sm block, 7ata lw kan feh msa7a kbera fadya, lazm ast5dm el block ka unit, mynnf34 yb2a 3ndy 3.5 blocks y3ny, f34an keda bn7awl dayman nst8l msa7t el block kolaha, w mndy34 ay msa7a fl block. 34an ntl3 b a7sn efficiency.

Block Granularity Considerations

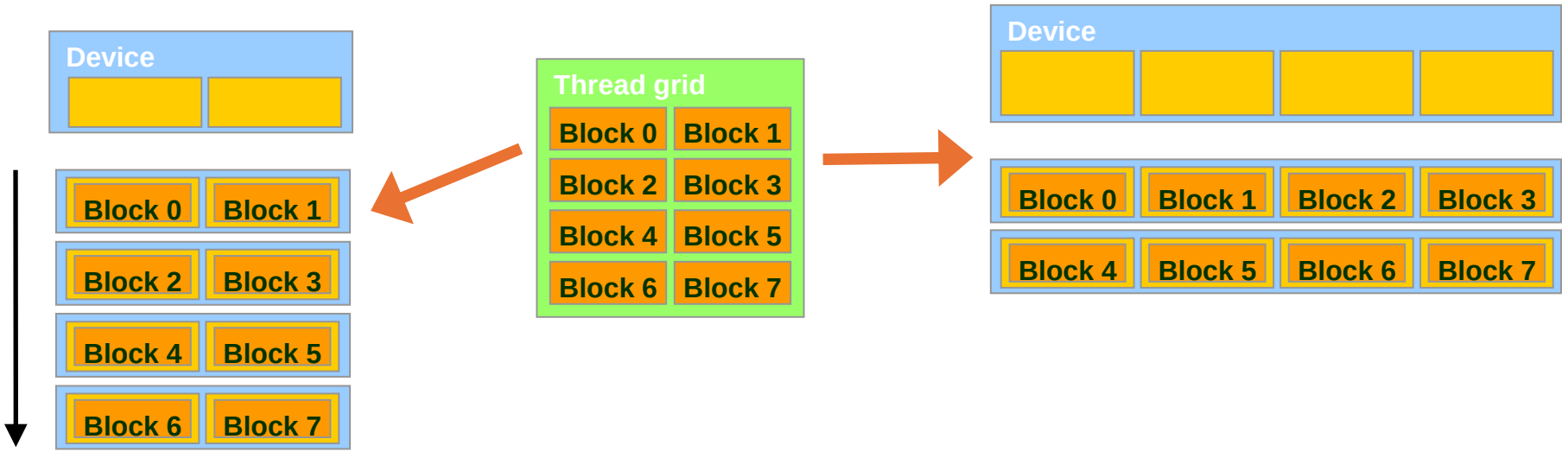
- For Matrix Multiplication using multiple blocks, should each block have 4X4, 8X8 or 30X30 threads for Volta?

  - For 30X30, we would have 900 threads per Block.
  - Only two blocks could fit into an SM for Volta, so only 1800/2048 of the SM thread capacity would be utilized.

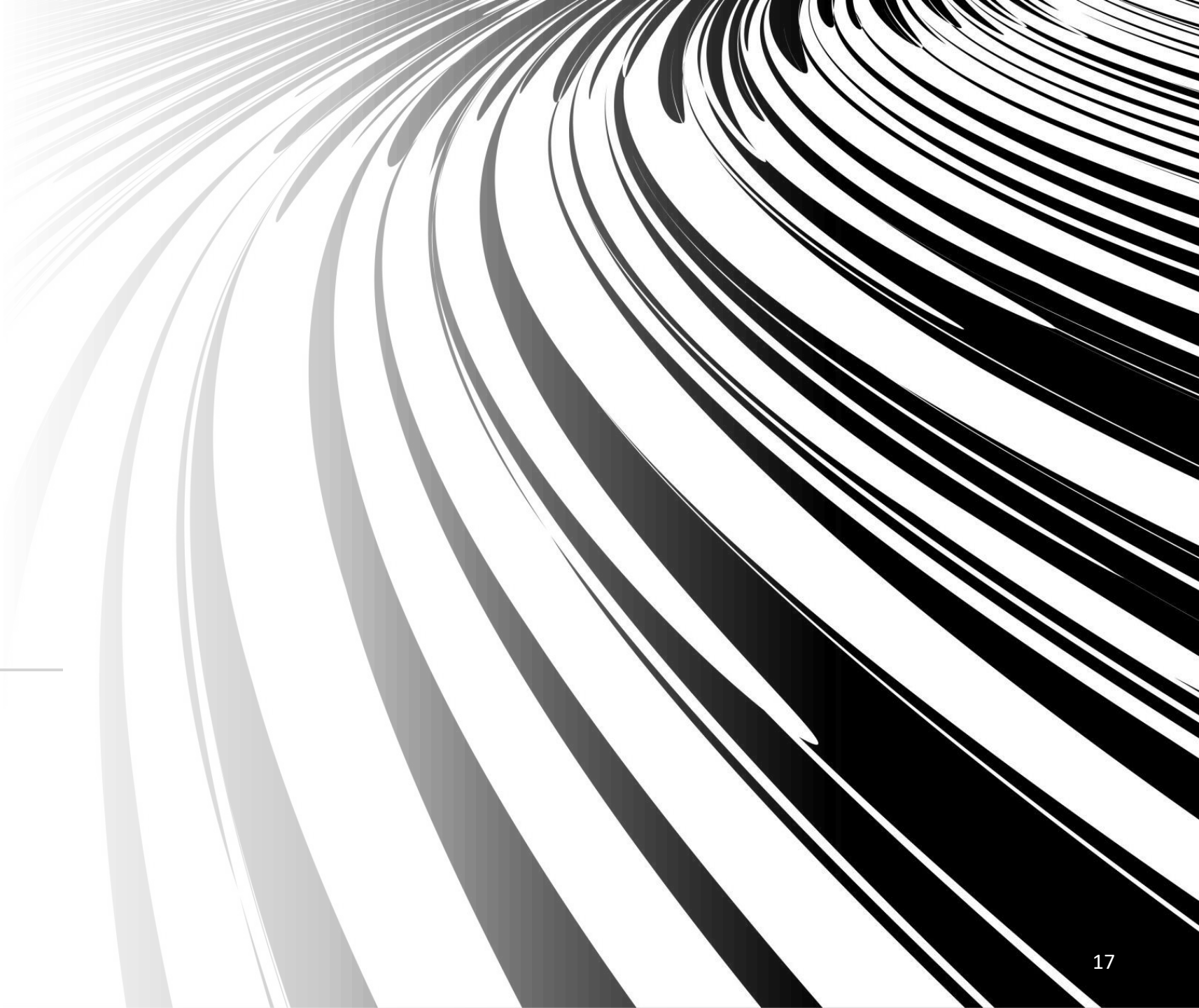
el GPU mo3tmd -> edeny data ktera, w ana hatsrf.

Transparent Scalability

- It is the ability of the system and applications to expand in scale without change to the system structure or the application algorithms.
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors



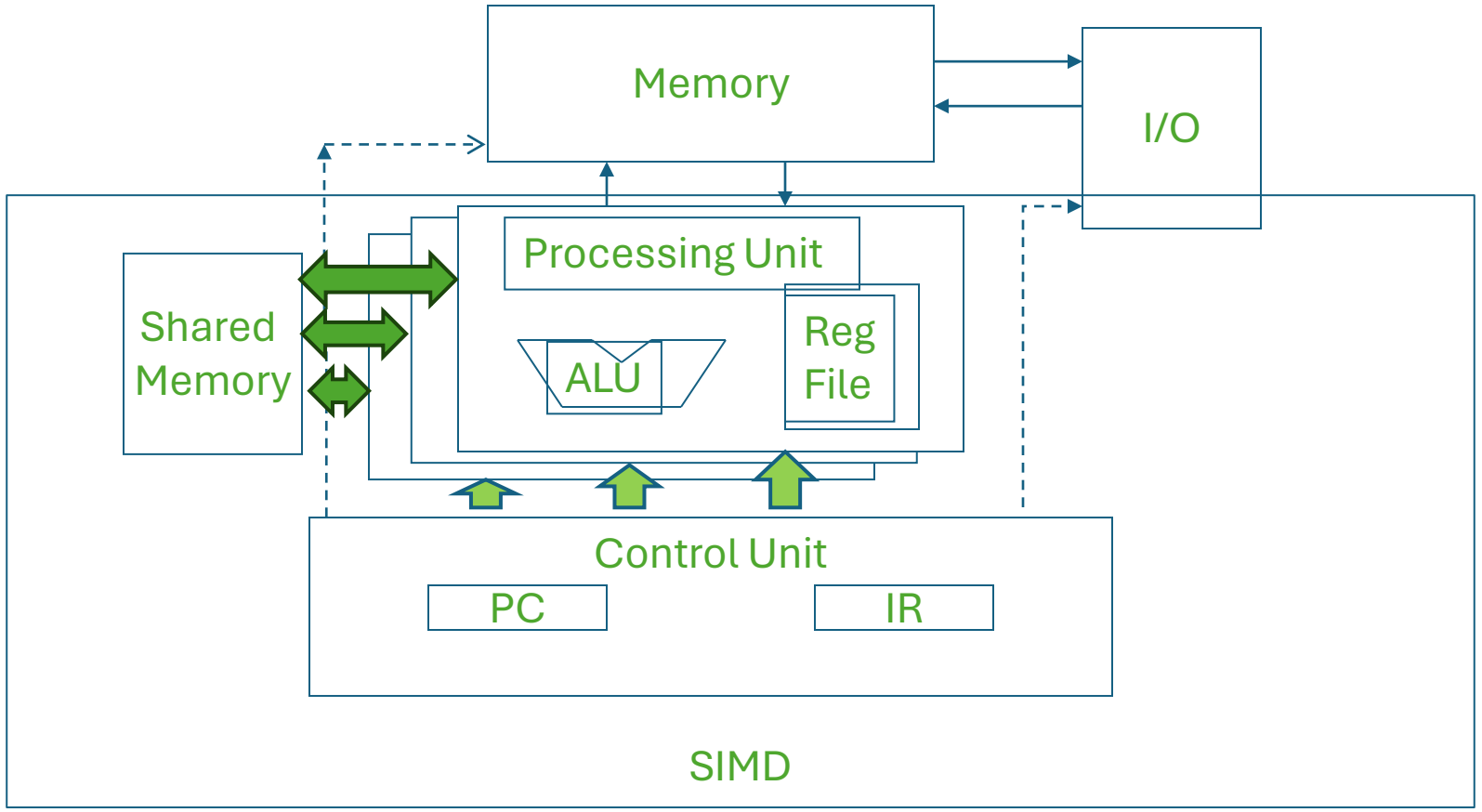
Warps



Warps as Scheduling Units

- Each Block is executed as 32-thread Warps (architecture-specific)
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

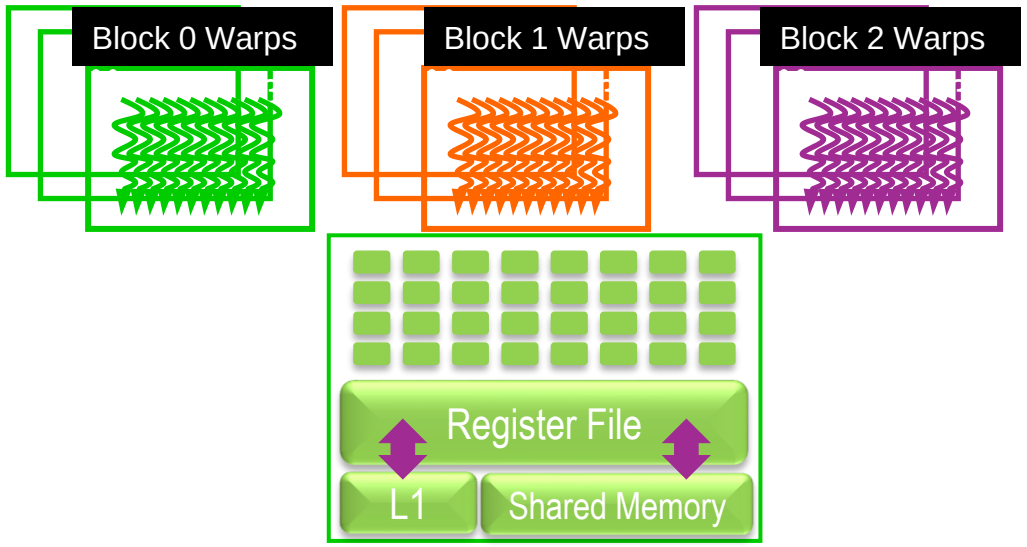
The Von-Neumann Model with SIMD units (Warps)



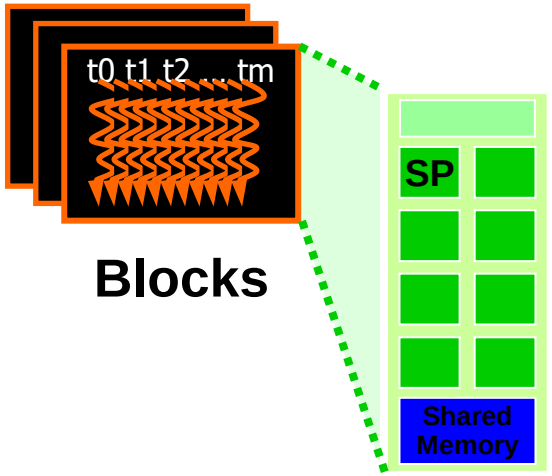
Warp Example

– If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?

- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps
- Total number of threads = 768



Scheduling in GPU



- SM schedule units of warps
- Each warp executes on one SM (obviously ..)
- Each clock cycle, the core can fetch one thread*
- But what if the SM has less core than a number of threads per warp??
  - Example, SM has 8 cores and Warp has 32 thread!
  - How can the 32 thread be implemented?
    - Each instruction will take 4 cycles, the first cycle will execute the first 8 threads, the second cycle next 8, the third cycle next 8 and the final cycle will be the last 8.

Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

- When a warp needs a lengthy operation like fetching from global memory, the other ready warps are executed.

- Thus, it hides latency because there are many warps to be executed and there must be someone ready to execute.

Threads Collaboration

Vector-Matrix Multiplication

A vector 1xN multiply by a matrix NxM = 1 x M

```
for (i = 0; i<N; i++)
    {
        for (j = 0; j<M; j++)
        {
            result_vector[j] += vector[i] * matrix[i][j];
        }
    }
```

e7na hnb2a 3auzen kul thread tshtghl 3la one column.

el loop de ms2ola 3n el columns, fa momken nshelhha, w n3mlhum parallel.

tb hgeb el j idx ezay? => el equation el 7elwa bta3tna.

Let's make M threads, where each thread calculates one output

Vector-Matrix Multiplication (1xN * NxM)

```
int j = threadIdx.x + blockIdx.x*blockDim.x;


if (j < M )
    for (i = 0; i<N; i++)
    {
        result_vector[j] += vector[i] * matrix[i*M+j];
    }
```

Can we optimize it more ?

Vector-Matrix Multiplication (1xN * NxM)

```
 int j = threadIdx.x +
blockIdx.x*blockDim.x;


 if (j < M )
    for (i = 0; i<N; i++)
    {
        result_vector[j] +=
vector[i] * matrix[i*M+j];
    }
```

Can we optimize it more?

1. We access global memory each time we use result_vector?
2. All threads read the same elements of the vector from global memory, can't they collaborate to bring the vector closer?
3. All threads perform another loop of calculation can we use parallel threads to make this second loop

bdl ma kol mara, aru7 ageb el vector kol mara mn el global memory., w kolohom ygebu nfs el data, fa mmkn akhly kol thread ygeb goz2 mn el vector, w y7utoh fl shared memory, fa keda el mwdu3 baa asr3, lw e7na ka talba el doc tlbt enna ngeb 10 e2lam mn mdam samia, el approach el 3ady en kul wa7d fena yru7 ygeb el 10 a2lam mn mdam samia, el bntklm feh baa, enna ntf2 m3 b3d kol wa7d fena yru7 ygeb alam wahed bs, wlma ngeb el 10, khlas keda m7dsh ygeb haga tany.

dayman awl 7aga bn3mlha, enna n2ml parallelizing lel memory., 34an hya dayman btb2a el bottelneck.

Vector-Matrix Multiplication (1xN * NxM)

```
 int j = threadIdx.x +
blockIdx.x*blockDim.x;
 float temp = 0;
 if (j < M )  {
    for (i = 0; i<N; i++)
    {
        temp += vector[i] *
matrix[i*M+j];
    }

result_vector[j] = temp; }
```

1. We access global memory each time we use result_vector?
2. All threads read the same elements of the vector from global memory, can't they collaborate to bring the vector closer?
3. All threads perform another loop of calculation can we use parallel threads to make this second loop

Vector-Matrix Multiplication (1xN * NxM)

2. Bring common data closer

```
 int j = threadIdx.x + blockIdx.x*blockDim.x;
 float temp = 0;
 if (j < M )  {
    for (i = 0; i<N; i++)
    {
        temp += vector[i] * matrix[i*M+j];
    }

result_vector[j] = temp; }
```

Vector-Matrix Multiplication (1xN * NxM)

2. Bring common data closer

```
 int j = threadIdx.x + blockIdx.x*blockDim.x;
 __shared__ float v[10000];    //allocated shared memory
float temp = 0;
 if (j < M )  {
    for (i = 0; i<N; i++)
    {
        temp += vector[i] * matrix[i*M+j];
    }

result_vector[j] = temp; }
```

Vector-Matrix Multiplication (1xN * NxM)

2. Bring common data closer

```
int j = threadIdx.x + blockIdx.x*blockDim.x;

__shared__ float v[10000];    //allocated shared memory
//let each thread fill part of the vector
int amount_per_thread = Ceil(N /blockDim.x);
    int start_index = j*amount_per_thread;
    int end_index = min(start_index + amount_per_thread,N);
    for (int k = start_index ; k <end_index ; k++ ) {
        v[k] = vector[k];
    }
//if a thread is finished can it start before others? What if it is in
a different warp? Will it still wait others ???
```

Vector-Matrix Multiplication (1xN * NxM)

el sM el wa7da feha shared memory wahda.

that is how we create a shared memory.

2. Bring common data closer

```
 __shared__ float v[10000];
    int j = threadIdx.x + blockIdx.x*blockDim.x;
    int amount_per_thread = Ceil(N /blockDim.x);
    int start_index = j*amount_per_thread;
    int end_index = min(start_index +
amount_per_thread,N);
    for (int k = start_index ; k <end_index ;
k++ )    {
        v[k] = vector[k];
    }
    __syncthreads();
```

HW: ezay n5ly el size dynamic -> momken ndeha el size mn bara. -> dwr 3leha w eb3tha lel DR.

bnshof 3ndy kam thread, w a2sm 7gm el shared memory 3lehom, b7es kol thread tgeb 7aba w hya gaya, w b2smhom b est5dam el index.

bstna en el nas kulaha tgeb nfs el 7aga, fa bn3ml barrier 34an kol el nas tu2af 3ndu, w b3den nkml.

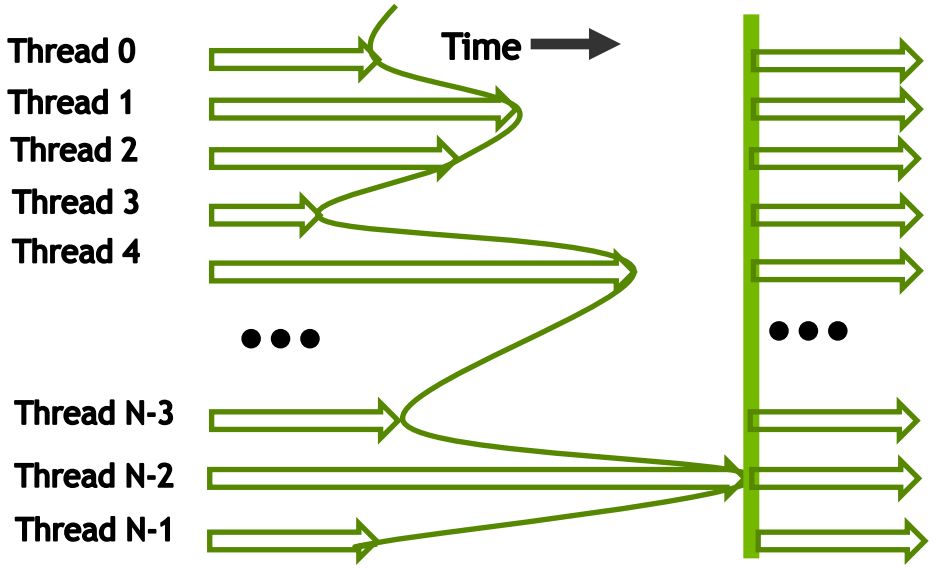
```
float temp = 0;
if (j < M )  {
for (i = 0; i<N; i++)
{
    temp += v[i] * matrix[i*M+j];
}
result_vector[j] = temp;
}
```

Barrier Synchronization

- All threads stop at the same point waiting for the others.

- Imagine going out with your family to the shopping
- You will all gather in the car then when all are there you will move.

- Take care that you all should gather at the **same car !!**



Barrier Synchronization

```
if (threadIdx < 10 ) {
        //dosomething
        __syncthreads();
    } else {
        //do another thing
        __syncthreads();   //waiting at the different car !
    }
```

keda kol wahed hyu2f 3nd no2ta mo5talefa.
lazm nfs el satr kol el threads yu2afu 3leh.

Why Getting Data Closer

- All threads access global memory for their input matrix & vector elements
  - two memory accesses (8 bytes) per floating-point addition
  - 8B/s of memory bandwidth/FLOPS
- Assume a GPU with
  - Peak floating-point rate 1,600 GFLOPS with 600 GB/s DRAM bandwidth
  - 8*1,600 = 12,800 GB/s required to achieve peak FLOPS rating
  - The 600 GB/s memory bandwidth limits the execution at 75 GFLOPS

- This limits the execution rate to 4.7% (75/1600) of the peak floating-point execution rate of the device!

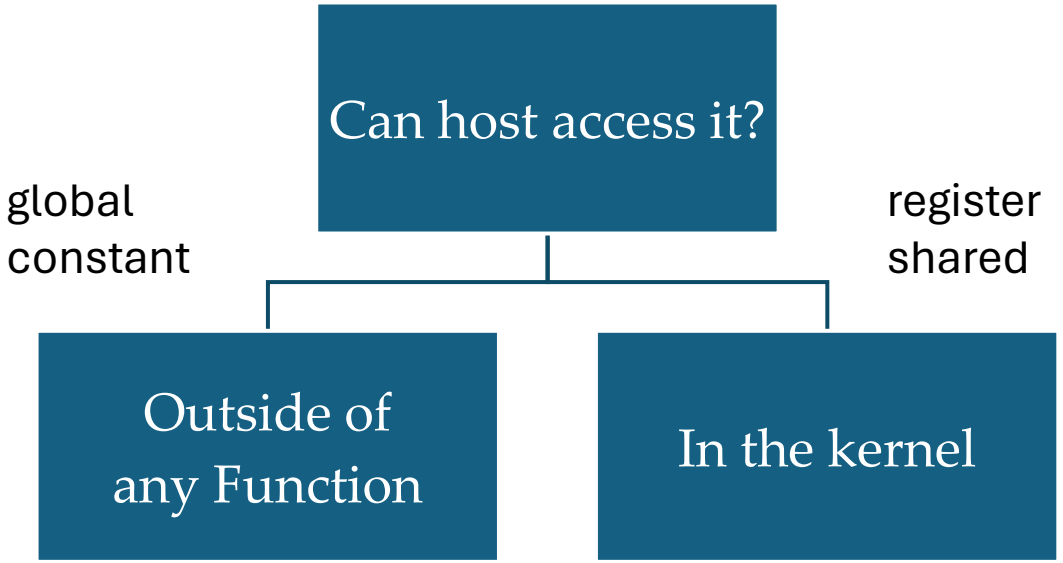
- Need to drastically cut down memory accesses to get close to the1,600 GFLOPS

Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **`__device__`** is optional when used with **`__shared__`**, or **`__constant__`**
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

Where to Declare Variables?



Summary

- Architecture of GPU
  - SMs
  - Thread Scheduling
    - Warps
    - Barriers
  - Shared memory.

Summary of components

- What can't be changed → **Constant per GPU**
  - #SMs is constant
  - #Cores per SM is constant
  - Shared Memory is constant
  - #Threads per warp is constant
  - Others to be added

What can be dynamically allocated:

- #Blocks per SM is variable with an upper bound
- #Threads per Block is variable with upper bound.
- #Threads per SM have upper bound
- …others to be added

Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture

Vector-Matrix Multiplication (1xN * NxM)

```
 int j = threadIdx.x +
blockIdx.x*blockDim.x;
 float temp = 0;
 if (j < M )  {
    for (i = 0; i<N; i++)
    {
        temp += vector[i] *
matrix[i*M+j];
    }

result_vector[j] = temp; }
```

1. We access global memory each time we use result_vector?
2. All threads read the same elements of the vector from global memory, can't they collaborate to bring the vector closer?
3. All threads perform another loop of calculation can we use parallel threads to make this second loop (Think about it ..)

Thank you