# Map Reduce

↳ Process large data sets in a distributed and parallel manner

↳ consists of 2 distinct tasks

↳ **Map** → divide into small problems → convert to (key, value) pairs } divide and conquer

↳ **Reduce** → combine all results parallely

↳ 2 essential Daemons → Process working in bg

↳ **Job Tracker**

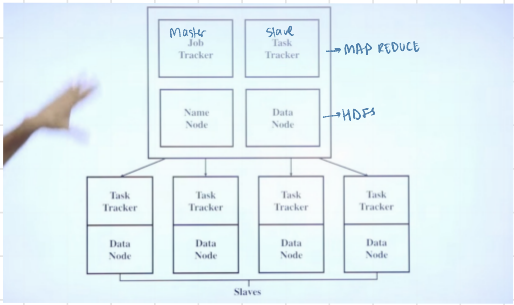
↳ schedule jobs

↳ provide resources

↳ monitor

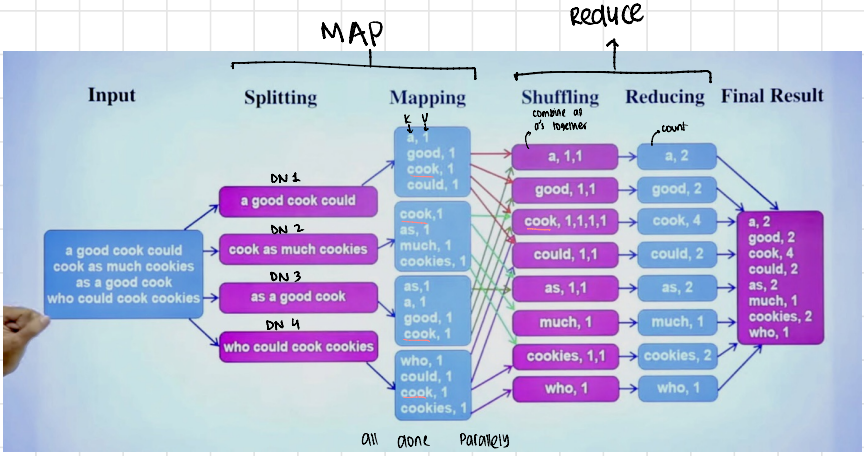
↳ **Task Tracker**

↳ does actual work on data




↳ PU goes to each data node parallely

Propogation Delay (PD): distance/velocity



## THE KEY CHALLENGES THAT MAPREDUCE AIMED TO ADDRESS WERE

### 1. Scalability

Traditional data processing systems were not scaling effectively with the rapidly increasing data sizes. MapReduce was designed to process large data sets across distributed computing resources efficiently.

### 2. Fault Tolerance

The likelihood of node failure in large-scale distributed computing is significant. MapReduce was built with mechanisms to handle failures gracefully, allowing data processing to continue despite node failures.

### 3. Simplicity of Data Processing

The framework simplified the programming model for large data sets. It abstracted the complexities of distributed computing, data partitioning, and fault tolerance, enabling developers to concentrate on the processing logic.

### 4. Data Locality

Transferring large data sets across a network is inefficient. MapReduce was optimized to process data on the nodes where it is stored, reducing network traffic and improving overall performance. This was a key part of the Google cluster environment: compute nodes and data storage nodes are not separate entities.

### 5. Flexibility

The framework was not overarching, which made it easy to understand and generic enough to be applicable to a broad range of applications, from web page indexing to machine learning tasks.

## KEY/VALUE PAIRS: HOW AND WHERE

Keys allow MapReduce to distribute and parallelize load

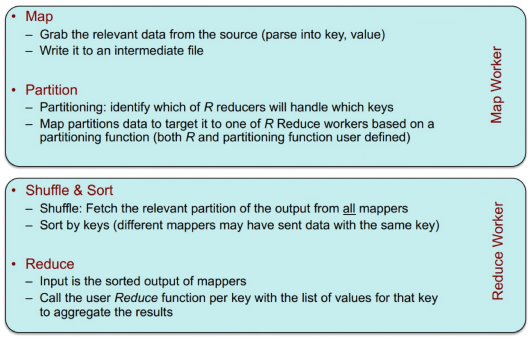
```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

Core abstraction: data can be partitioned by key, there is no locality between keys.

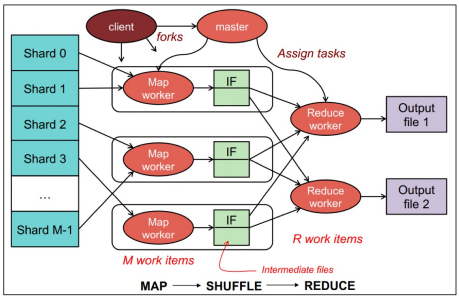
In the original paper

- Each mapper writes a local file for each key in k2, and reports its files to a master node
- The master node tells the reducer for k2 where the all of the k2 files are
- The reducer reads all of the k2 files from the nodes that ran the mappers and writes its own output locally, reporting this to the master node

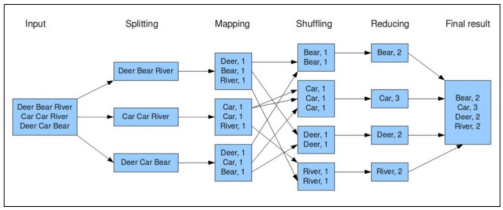
## MAPREDUCE: WHAT HAPPENS IN BETWEEN?



**MAPREDUCE: THE COMPLETE PICTURE**




**WORD COUNT EXAMPLE IN DETAIL**




## Execution: Phases

**Splitting**

- Input key-value pairs (documents) are parsed and prepared

**Mapping**

- **Map function** is executed for each input document
- Intermediate key-value pairs are emitted

**Shuffling**

- Intermediate key-value pairs are **grouped and sorted** according to the keys

**Reducing**

- **Reduce function** is executed for each intermediate key
- Final output is generated

## Execution: Components

**Input reader**

- Reads data from a stable storage (e.g. a distributed file system)
- Splits the data into appropriate size blocks (splits)
- Parses these blocks and **prepares input key-value pairs**

**Map function**

**Partition function**

- Determines Reduce task for an intermediate key-value pair
  - E.g. hash of the key modulo the overall number of reducers

**Compare function**

- Compares two intermediate keys, used during the shuffling

**Reduce function**

**Output writer**

- Writes the output of the Reduce function to stable storage

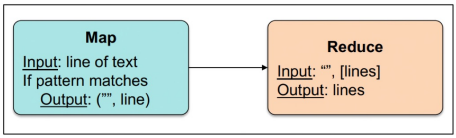
**WORD COUNT EXAMPLE**

Consider the problem of counting the numb
of documents. The user would write code sir

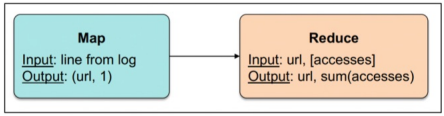
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

**MORE EXAMPLES**

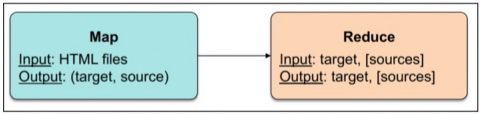
1. **Distributed Grep:**
   - The map function emits a line if it matches a supplied pattern.
   - The reduce function is an identity function that just copies the supplied intermediate data to the output.




2. **Count of URL Access Frequency:**
   - Find the count of each URL in web logs.
   - The map function processes log of web page requests and outputs <URL, 1>.
   - The reduce function adds together all values for the same URL and emits a <URL, total count> pair.

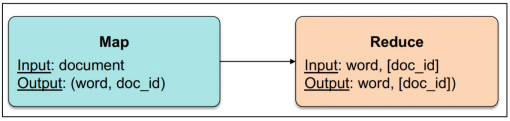



3. **Reverse Web-Link Graph:**
   - Find where page links come from
   - The map function outputs <target, source> pairs for each link to a target URL found in a page named source.
   - The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>




4. **Term-Vector per Host:**
   - A term vector summarizes the most important words that occur in a document or a set of documents as a list of <word, frequency> pairs.
   - The map function emits a <hostname, term vector> pair for each input document (where the hostname is extracted from the URL of the document).
   - The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final <hostname, term vector> pair.

5. **Inverted Index:**
   - Find what documents contain a specific word
   - The map function parses each document, and emits a sequence of <word, document Id> pairs.
   - The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list (document ID)> pair. The set of all output pairs forms a simple inverted index.
   - It is easy to augment this computation to keep track of word positions.




6. **Distributed Sort:**
   - The map function extracts the key from each record, and emits a <key, record> pair.
   - The reduce function emits all pairs unchanged.

# CHP 3

**Parallelism**: act of managing multiple computations simultaneously

## Data Parallelism

↳ same operation on diff data

↳ scalable

↳ distribute data

## Task Parallelism

↳ diff operations on same/diff data

↳ not scalable

↳ distribute threads

↳ two tasks that can be done independently

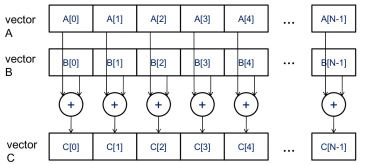



**FIGURE 3.1**
Data parallelism in vector addition.

→ CONS
↳ overhead of
↳ allocating device memory
↳ transfer host to device
↳ transfer device to host
↳ de-allocating device memory
can make code slower

## CUDA Program Structure

↳ each source file has host + device code

↳ compiled by NVCC (NVIDA C Compiler)

↳ CUDA keywords seperate host and device code

C Program: Cuda Program with only host code

### HOST code

↳ ANSI code
+
↳ C/C++ code

↳ run as traditional CPU process

### DEVICE code

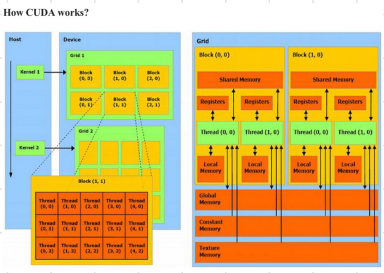
↳ marked with CUDA keywords

↳ to label kernels → data parallel functions

↳ to label their association data structures

↳ compiled by runtime component of NVCC

↳ executed on GPU device




## CUDA Program Execution

→ sequential thread execution

In CUDA, the execution of each thread is sequential as well. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying runtime mechanisms to create many threads that process different parts of the data in parallel.

→ allows to exploit data parallelism

1. first CPU execute host code
2. when kernel function called
   ↳ it's executed by large no. of threads (→ grids) on a device
   ↳ after all threads execution completed
   ↳ corresponding grid terminates
   ↳ execution continues on host
3. Repeat step 2 if kernel fuction called again

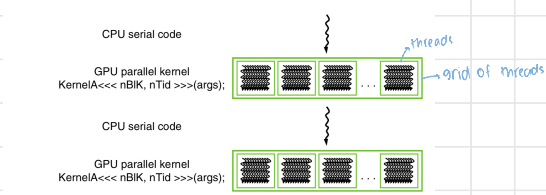



**FIGURE 3.3**
Execution of a CUDA program. 2 grids of threads

# Vector Addition

cuda code

c code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

P1  1. allocate memory
    2. copy host to device
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

P2  perform actual vector addition
    // Kernel invocation code - to be shown later
    ... parallel execution

P3  copy C to host from device
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

**FIGURE 3.9**

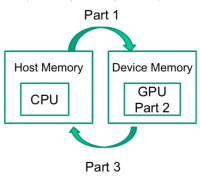
A more complete version of `vecAdd()`.

```
// Compute vector sum h_C = h_A+h_B
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()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

array A, array B, output array

**FIGURE 3.4**

A simple traditional vector addition C code example.



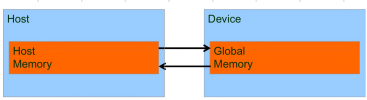




**FIGURE 3.6**

Host memory and device global memory.

# CUDA API Functions

cast generic pointer; address of pointer; size of object

```
cudaMalloc((void**) &d_a, size);
```

↳ allocates object in device memory

pointer to freed object

```
cudaFree(d_a);
```

↳ frees object in device memory

pointer to destination; pointer to source; size of data; transfer direction: H→H, H→D, D→H, D→D

```
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

↳ data transfer b/w host and device

```
cudaError_t err = cudaMalloc();
```

↳ tests for error conditions

```
    if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString( err),
      __FILE__, __LINE__);
    exit(EXIT_FAILURE);
    }
```

This way, if the system is out of device memory, the user will be informed about the situation.

One would usually define a C macro to make the checking code more concise in the source.

# Kernel functions

- specifies code to be executed by all threads
- SPMD (single program, multiple data) — NOT SIMD; doesn't need to execute same instruction
  Parallel programming style
- when called by host, CUDA runtime system generates a grid of threads

# Grid

loop in C is replaced by grids in CUDA

- kernel function generated collective threads
- organised into array of thread blocks → blockIdx
- all blocks of same size → max possible size 1024 threads → blockDim
- each thread in block has unique thread id → threadIdx

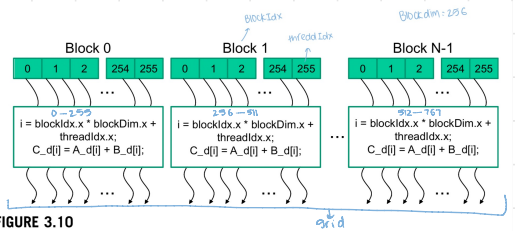



**FIGURE 3.10**

All threads in a grid execute the same kernel 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];
}
```

automatic (local) variable visible to each thread — each thread's id — total no of threads out — extra threads won't perform addition

**FIGURE 3.11**

A vector addition kernel function and its launch statement.

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

1. allocate memory
2. copy host to device
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

Perform vector addition
    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

copy C to host from device
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        // Free device memory for A, B, C
free vectors in device
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

grid Dim — block Dim

<< >> → used for calling kernel function

**FIGURE 3.14**

A complete version of `vecAdd()`.

### Function Declarations

CUDA extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Figure 3.12. Using one of `__global__`, `__device__`, or `__host__`, a CUDA programmer can instruct the compiler to generate a kernel function, a device function, or a host function. All function declarations without any of these keywords are defaulted to host functions. If both `__host__` and `__device__` are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA extension keyword, the function defaults into a host function.

| | Executed on the: | Only callable from the: |
|---|---|---|
| `__device__ float DeviceFunc()` | device | device |
| `__global__ void KernelFunc()` | device | host |
| `__host__ float HostFunc()` | host | host |

**FIGURE 3.12**

CUDA C keywords for function declaration.

Note that there is an `if (i<n)` statement in `addVecKernel()` in Figure 3.11. This is because not all vector lengths can be expressed as multiples of the block size. For example, if the vector length is 100, the smallest efficient thread block dimension is 32. Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their `i` values against `n`, which is 100. With the `if (i<n)` statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

## 1D Grids

data type — no of blocks — all 1Ds have 1 here

```
dim3 dimGrid (32,1,1);
dim3 dimBlock (128,1,1);
vecAddkernel<< dimGrid, dimBLOCK >>....;
```
(no of threads)

OR

```
dim3 dimGrid ( ceil(n/256),1,1);
dim3 dimBlock (256,1,1)
vecAddkernel<< dimGrid, dimBLOCK >>....;
```

1D grid
128 blocks
32 threads each

total no of threads
= 128 x 32
= 4,096 threads

if n=1000, 4 blocks
if n=4000, 16 blocks

OR

```
vecAddkernel<< ceil(n/256), 256 >>....;
```

↳ max range of gridDim.x, gridDim.y, gridDim.z = 1 ⟶ 65,536
↳ max range of blockDim.x, blockDim.y, blockDim.z = 0 ⟶ gridDim.x -1
(respectively)

## Blocks

↳ are organised into 3D array of threads

* grid can have higher dim than its blocks and vice versa

**1D block**: blockDim.y = 1, blockDim.z = 1
**2D block**: blockDim.z = 1
**3D block**:

↳ max size of block = 1024 threads

1. (512,1,1) ✓
2. (8,16,4) ✓
3. (32,16,2) ✓
4. (32,32,2) ✗ as threads exceed 1024

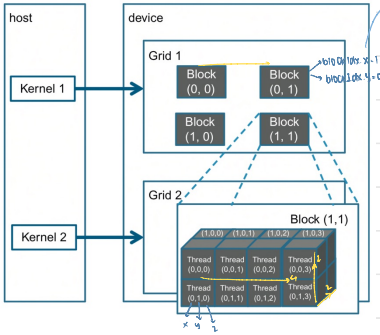



**FIGURE 4.1**
A multidimensional example of CUDA grid organization.

innermost dimension comes first ?

⟶ its code of 2D grid

```
dim3 dimGrid(4,2,2); → 3D block
dim3 dimBlock(2,2,1); → 2D grid
vecAddkernel<< dimGrid, dimBLOCK >>....;
```
(no of blocks; no of threads)

blocks = 4
mreads = 4x2x2 = 16
total no of mreads = 64

4.2

# Mapping threads to multidimensional data

## Pictures are a 2D array of Pixels

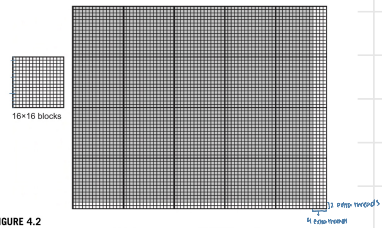



FIGURE 4.2
Using a 2D grid to process a picture.

76 x 62 Picture Pixel
we used
Threads = 16 x 16 = 256
Blocks = 5 x 4 = 20
total no of threads = 256 x 20 = 5120 threads
to process 76 x 62 = 4712 pixels

↳ its code for 2D kernel (no of blocks in x, no of blocks in y)

```
dim3 dimGrid(16,16,1); →2D
dim3 dimBlock(ceil(n/16), ceil(m/16), 1); →2D
Picture kernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

(n: no of pixels in x direction; m: no of pixels in y direction)

d_Pin[i][i] → a 2D array, where no of columns be known at compile time

but this is a dynamically allocated array so not possible in CUDA C

SOLUTION → Linearize a 2D allocated array (aka flatten)

↳ convert dynamically allocated arrays to 1D array
↳ translate multidimensional index to 1D offset

TWO WAYS

1. ROW MAJOR LAYOUT (used by CUDA C) → transpose of each other ← 2. COLUMN MAJOR LAYOUT (used by FORTRAN compiler)

↳ Place all elements in same row consecutively
↳ rows are placed one after the other in the memory space

↳ Place all elements in same columns consecutively
↳ columns are placed one after the other in the memory space

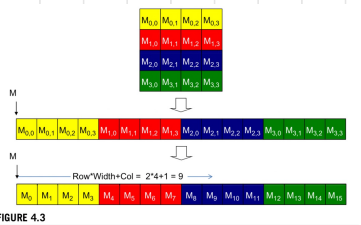



FIGURE 4.3
Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression Row*Width + Col for an element that is in the Row$^{th}$ row and Col$^{th}$ column of an array of Width elements in each row.

4x4 matrix linearized into 16 element 1D array

index = j x 4 + i (j: chooses row; i: chooses column)

$M_{2,1}$ index = 2 x 4 + 1
= 9

# Scale Picture Pixel by factor of 2 and map

```
__global__ void PictureKernel(float *d_Pin, float *d_Pout, int n, int m)
{
  int row = blockIdx.y * blockDim.y + threadIdx.y
  int col = blockIdx.x * blockDim.x + threadIdx.x

  if ((row<m) && (col < n))
  { d_Pout[row*n + col] = 2 * d_Pin[row*n + col]; }
}
```

→ Calculate no of rows ≥ no of vertical pixels

→ Calculate no of columns ≥ no of horizontal pixels

→ makes sure within range

create 1D index for pixel

scale every pixel by factor of 2

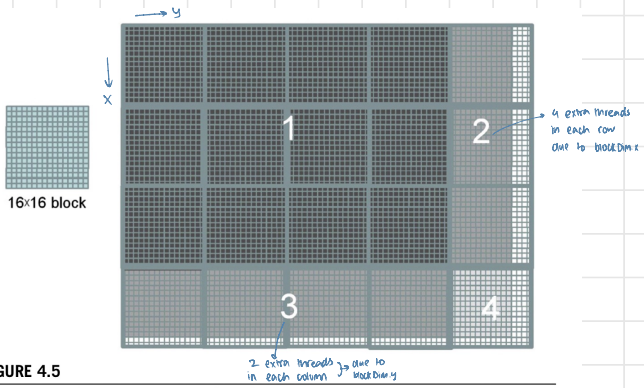



**FIGURE 4.5**
Covering a 76 × 62 picture with 16 × 16 blocks.

## 2D arrays to 3D arrays

```
int Plane = blockIdx.z * blockDim.z + threadIdx.z
P[Plane * m * n + Row * n + Col]
```

```
ROW = blockIdx.x * blockDim.x + threadIdx.x
Column = blockIdx.y * blockDim.y + threadIdx.y
Plane = blockIdx.z * blockDim.z + threadIdx.z
```

[row * n + col] → convert to 1D

u3/ Matrix Multiplication

↳ using square matrices so use width instead of i,j

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }
}
```

sum ↳ linearized in 2D array

1-D equivalent index expression

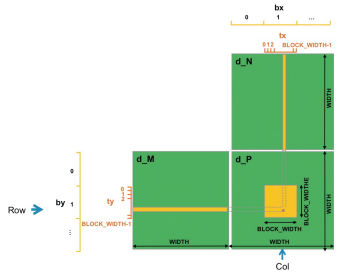



**FIGURE 4.6**
Matrix multiplication using multiple blocks by tiling d_P.

**FIGURE 4.7**
A simple matrix–matrix multiplication kernel using one thread to compute each d_P element.

```
#define BLOCK_WIDTH 16

// Setup the execution configuration
    int NumBlocks = Width/BLOCK_WIDTH;
    if (Width % BLOCK_WIDTH) NumBlocks++;
    dim3 dimGrid(NumBlocks, NumBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads!
    matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

used for automating → search for best BLOCK_WIDTH by iterating its value and compile and run for hardware of interest

2D

2D

no of blocks

**FIGURE 4.8**
Host code for launching the matrixMulKernel() using a compile-time constant BLOCK_WIDTH to set up its configuration parameters.

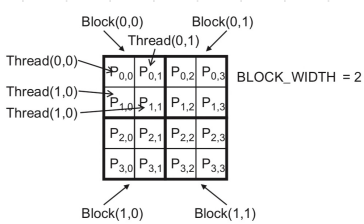



**FIGURE 4.9**
A small execution example of matrixMulKernel().

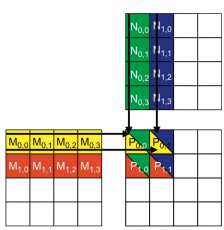


**FIGURE 4.10**
Matrix multiplication actions of one thread block. For readability, d_M, d_N, and d_P are shown as M, N, and P.

4.4

# Synchronization

↳ to coordinate the execution of multiple threads

## Barrier synchronization

↳ threads in same block coordinate their parallel activities

__syncthreads() → all threads in a block will wait till all threads reach calling location

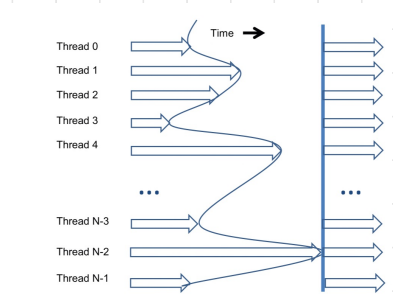



**FIGURE 4.11**
An example execution timing of barrier synchronization.

### ↳ If conditions

↳ If .....

__syncthreads() → either all thread execute this path

else

..... → or all threads execute this path

↳ $t_1$ goes in if, $t_2$ goes in else

then they would be waiting at diff barrier synchronization points and end up waiting for each other FOREVER

### ↳ Resources

↳ all threads in a block are assigned execution resources as a unit at the barrier

↳ avoids long waiting times, forever wait

proximity with each other to avoid excessively long waiting times. In fact, one needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever. CUDA runtime systems

### ↳ Transparent scalibility

↳ multiple blocks executing simultaneously

↳ this is possible by not allowing threads in diff blocks to perform barrier synchronization so the blocks can be executed in any order, as they dont need to wait for each other

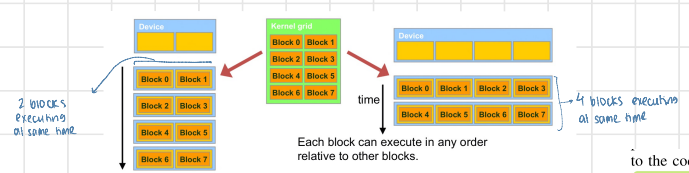



**FIGURE 4.12**
Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

to the code. The ability to execute the same application code on hardware with a different number of execution resources is referred to as *transparent scalability*, which reduces the burden on application developers and

43

# Assigning Resources to Blocks

↳ threads are assigned to execution resources on a block by block basis

↳ the execution resources are organised into streaming resources (SM)

## SM

↳ multiple thread blocks assigned to each SM

↳ each device has limits on no of thread blocks assigned to each SM

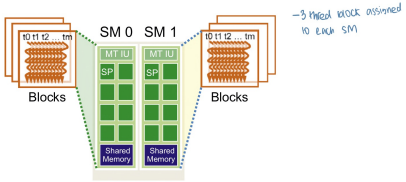
If insufficient amount of resources → of any type

↳ cuda runtime automatically reduces

no of treds assigned to each SM

until their combined resource falls under the limit

each SM until their combined resource usage falls under the limit. With a limited numbers of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device. Most grids contain many more




→ 3 thread block assigned to each SM

**FIGURE 4.13**
Thread block assignment to SMs.

## Runtime system

↳ maintains

↳ list of blocks that need to execute

↳ assigns new blocks to SM → when old ones are executed

# Querying Device Properties

↳ to find out amount of resources available → no of SM in a device, no of threads that can be assigned to each SM

↳ host code queries the properties of device avl in the system

↳ modern PCs can have 2 or more CUDA devices because they have integrated GPUs

↳ CUDA application doesn't perform well on these integrated devices

which is why query is used

// decides if device has sufficient resources, capabilities

```
int dev_count;
cudaGetDeviceCount(&dev_count);
```
→ gets no of avl CUDA devices in system

represents properties of a cuda device

```
cudaDeviceProp dev_prop;
for (i=0; i< dev_count ; i++)
    { cudaGetDeviceProperties(&dev_prop, i);
```
→ iterate through avl devices and query their properties

→ gives properties of device

# Cuda Device Properties

dev_prop.maxThreadPerBlock → max no of threads allowed in a block in the gived device

dev_prop.MultiProcessorCount → no of SM in the device

dev_prop.clockrate → clock freauency of device

→ their combination indicates hardware execution capacity of device

dev_prop.maxThreadsDim[0] → x dimention

dev_prop.maxThreadsDim[1] → y dimention

dev_prop.maxThreadsDim[2] → z dimention

→ max no of threads for each dimention

dev_prop.maxGridSize[0] → x dimention

dev_prop.maxGridSize[1] → y dimention

dev_prop.maxGridSize[2] → z dimention

→ check wether grid can have enough threads to handle entire data set OR some kind of iteration is needed

ishma hafeez notes

```
map (string key, string value)
    for each word w in value:
Emit Intermediate (w, "1")
```

```
Reduce (string key, Iterator value)
    result = 0;
    for each v in value
        result += ParseInt (v);
        Emit(AsString(result));
```