

# Memory

Rupesh Nasre.

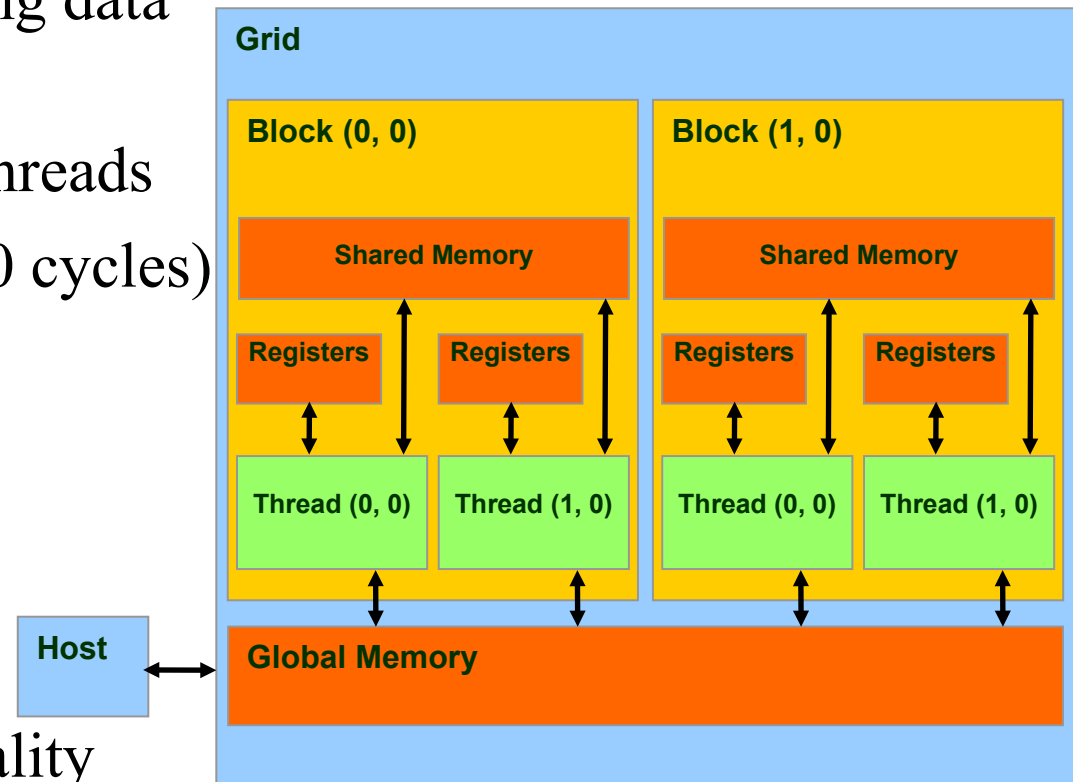
GPU Programming  
January 2024

# Agenda

- Computation
- **Memory**
- Synchronization
- Functions
- Support
- Streams
- Topics
- Case Study – Graphs

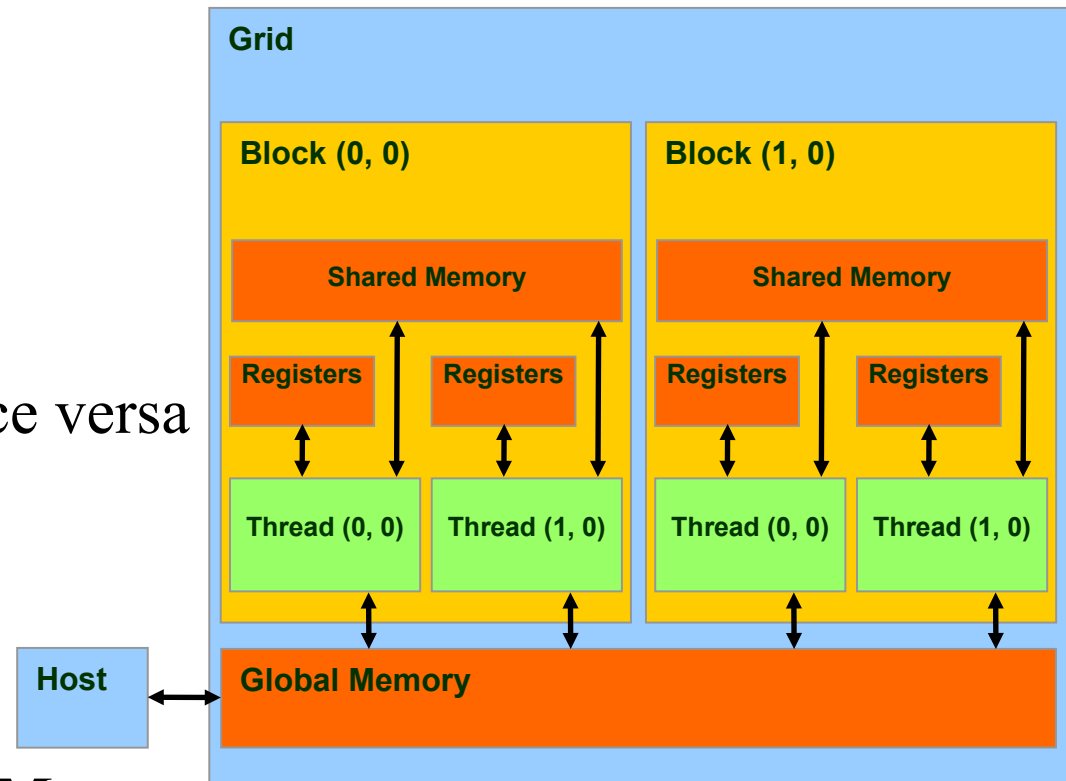
# CUDA Memory Model Overview

- **Global / Video memory**
  - Main means of communicating data between **host** and **device**
  - Contents visible to all GPU threads
  - Long latency access (400-800 cycles)
  - Throughput ~200 GBPS
- **Texture Memory**
  - Read-only (12 KB)
  - ~800 GBPS
  - Optimized for 2D spatial locality
- **Constant Memory**
  - Read-only (64 KB)



# CUDA Memory Model Overview

- **L2 Cache**
  - 768 KB
  - Shared among SMs
  - Fast atomics
- **L1 / Shared Memory**
  - Configurable 64 KB per SM
  - 16 KB shared+48 KB L1 or vice versa
  - Low latency (20-30 cycles)
  - High bandwidth (~1 TBPS)
- **Registers**
  - 32 K in number, unified, **per SM**
  - ~Max. 21 registers per thread
  - Very high bandwidth (~8 TBPS)



The numbers are typical values.

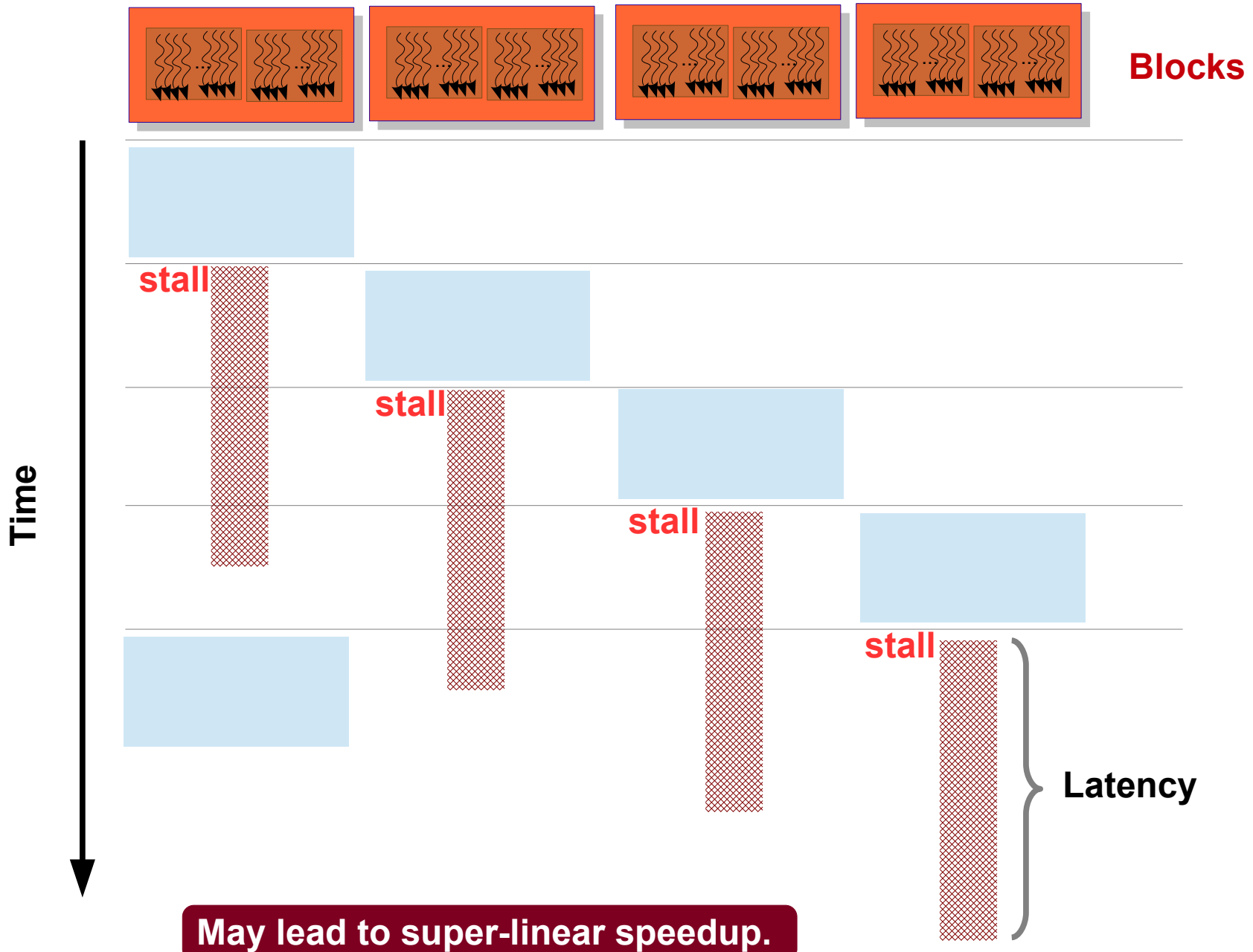
# Bandwidth

- Big (wide) data bus rather than fast data bus
- Parallel data transfer
- Techniques to improve bandwidth:
  - Share / reuse data
  - Data compression
  - Recompute than store + fetch

# Latency

- Latency is time required for I/O.
- Latency should be minimized; ideally zero.
  - Processor should have data available in no time.
  - In practice, memory I/O becomes the bottleneck.
- Latency can be reduced using caches.
  - CPU: Registers, L1, L2, L3, L4, RAM
  - GPUs have small L1 and L2, and many threads.
- Latency hiding on GPUs is done by exploiting massive multi-threading.

# Latency Hiding



# Locality

- Locality is important on GPUs also.
- All threads in a thread-block access their L1 cache.
  - This cache on Pascal GPU is 64 KB.
  - It can be configured as 48 KB L1 + 16 KB scratchpad (or 16 KB L1 + 48 KB scratchpad or 32 KB + 32 KB).
- Programmer can help exploit locality.
- In the GPU setting, another form of spacial locality is critical.



# Locality

## Spatial

If  $a[i]$  is accessed,  $a[i+k]$  would also be accessed.

```
for (i = 0; i < N; ++i)
    a[i] = 0;
```

## Temporal

If  $a[i]$  is accessed **now**, it would be accessed soon **again**.

```
for (i = 0; i < N; ++i) {
    a[i] = i;
    a[i] += N;
    b[i] = a[i] * a[i];
}
```

The localities are applicable on both CPU as well as GPU. But more applicable on CPUs.

# Classwork

- Check how the localities are in the following matrix multiplication programs (on CPU).

```
for (i = 0; i < M; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < P; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

```
for (i = 0; i < M; ++i)
  for (k = 0; k < P; ++k)
    for (j = 0; j < N; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Times taken for  $(M, N, P) = (1024, 1024, 1024)$  are 9.5 seconds and 4.7 seconds.

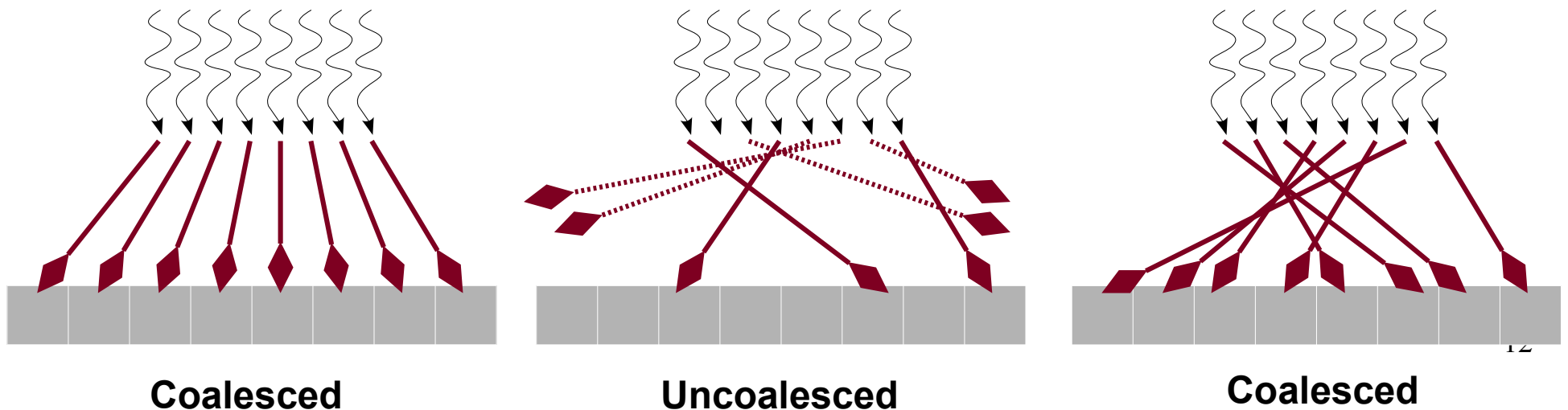
What happens on a GPU?

# Memory Coalescing

- If *warp threads* access words from the same block of 32 words, their memory requests are clubbed into one.
  - That is, the memory requests are **coalesced**.
- **Without coalescing**, each load / store instruction would required one memory cycle.
  - A warp would require 32 memory cycles.
  - The throughput would significantly reduce.
  - GPU would be useful only for compute-heavy kernels.

# Memory Coalescing

- If *warp threads* access words from the same block of 32 words, their memory requests are clubbed into one.
  - That is, the memory requests are coalesced.
- This can be effectively achieved for regular programs (such as dense matrix operations).



# Memory Coalescing

C  
P  
U

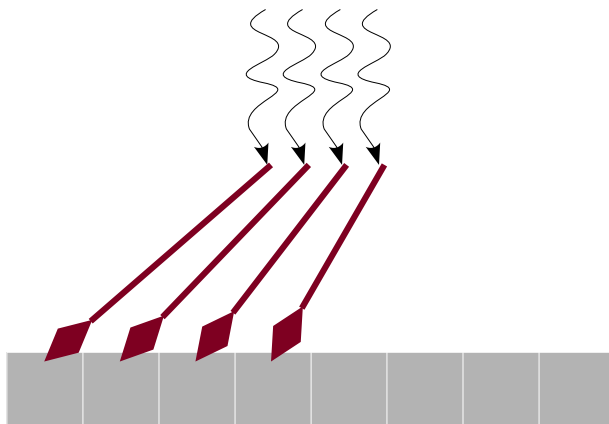
- Each thread should access consecutive elements of a chunk (strided).
- Array of Structures (AoS) has a better locality.

- A chunk should be accessed by consecutive threads (coalesced).
- Structure of Arrays (SoA) has a better performance.

G  
P  
U

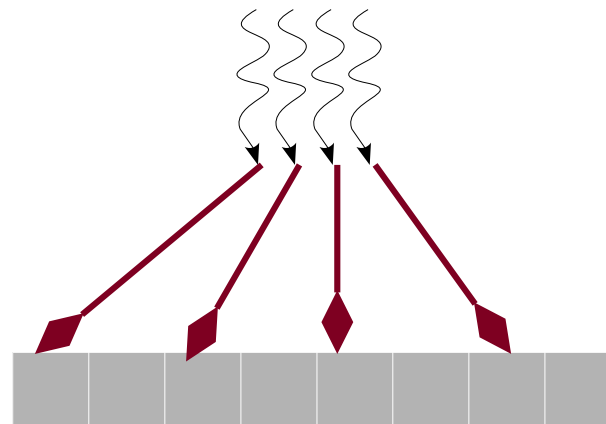
```
start = id * chunksize;  
end = start + chunksize;  
for (ii = start; ii < end; ++ii)
```

... **a[id]** ...



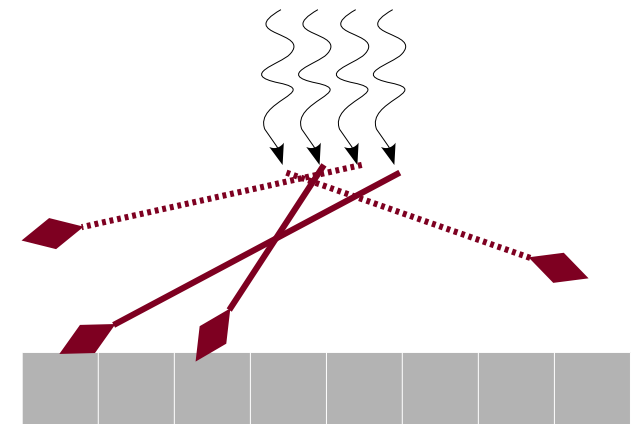
Coalesced

... **a[ii]** ...



Strided

... **a[input[id]]** ...



Random

# AoS versus SoA

```
struct node {  
    int a;  
    double b;  
    char c;  
};  
struct node allnodes[N];
```

**Expectation:** When a thread accesses an attribute of a node, *it also accesses other attributes of the same node.*

Better locality (on CPU).

```
struct node {  
    int alla[N];  
    double allb[N];  
    char allc[N];  
};
```

**Expectation:** When a thread accesses an attribute of a node, *its neighboring thread accesses the same attribute of the next node.*

Better coalescing (on GPU).

**Classwork: Write code for the two types using cudaMemcpy.**  
(note that all arrays would be pointers)

# AoS versus SoA

```
struct nodeAOS {  
    int a;  
    double b;  
    char c;  
} *allnodesAOS;
```

```
__global__ void dkernelsaos(struct nodeAOS *allnodesAOS) {  
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;  
    allnodesAOS[id].a = id;  
    allnodesAOS[id].b = 0.0;  
    allnodesAOS[id].c = 'c';  
}
```

```
struct nodeSOA {  
    int *a;  
    double *b;  
    char *c;  
} allnodesSOA;
```

```
__global__ void dkernelsoa(int *a, double *b, char *c) {  
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;  
    a[id] = id;  
    b[id] = 0.0;  
    c[id] = 'd';  
}
```

**AOS time = 61 units, SOA time = 22 units**

# Classwork

- Copy a linked-list from CPU to GPU.
  - Each node contains roll number, name, facad.
  - Try a single pass through the list, without knowing the number of nodes a priori.



# Shared Memory

- Programmable L1 cache / Scratchpad memory
- Accessible only in a thread block
- Useful for repeated small data or coordination

```
__shared__ float a[N];  
__shared__ unsigned s;
```

```
a[id] = id;  
if (id == 0) s = 1;
```

# Classwork

- You are given a 1024x1024 integer matrix  $M$ .
- Each row is assigned to a thread block.
- Each thread is assigned a matrix element  $M[i][j]$ .
- It changes  $M[i][j]$  to  $M[i][j] + M[i][j+1]$  (where possible).
- Exploit shared memory.

# Shared Memory

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE    1024

__global__ void dkernel() {
    __shared__ unsigned s;

    if (threadIdx.x == 0) s = 0;

    if (threadIdx.x == 1) s += 1;

    if (threadIdx.x == 100) s += 2;

    if (threadIdx.x == 0) printf("s=%d\n", s);
}
int main() {
    dkernel<<<1, BLOCKSIZE>>>();
    cudaDeviceSynchronize();
}
```

s=3

# Shared Memory

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE    1024

__global__ void dkernel() {
    __shared__ unsigned s;

    if (threadIdx.x == 0) s = 0;

    if (threadIdx.x == 1) s += 1;

    if (threadIdx.x == 100) s += 2;

    if (threadIdx.x == 0) printf("s=%d\n", s);
}
int main() {
    dkernel<<<2, BLOCKSIZE>>>();
    cudaDeviceSynchronize();
}
```

s=3  
s=3

# Shared Memory

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE    1024

__global__ void dkernel() {
    __shared__ unsigned s;

    if (threadIdx.x == 0) s = 0;

    if (threadIdx.x == 1) s += 1;

    if (threadIdx.x == 100) s += 2;

    if (threadIdx.x == 0) printf("s=%d\n", s);
}
int main() {
    int i;
    for (i = 0; i < 10; ++i) {
        dkernel<<<2, BLOCKSIZE>>>();
        cudaDeviceSynchronize();
    }
}
```

s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=1  
s=3  
s=3  
s=3

# Shared Memory

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE    1024

__global__ void dkernel() {
    __shared__ unsigned s;

    if (threadIdx.x == 0) s = 0;
    __syncthreads(); // barrier across threads in a block
    if (threadIdx.x == 1) s += 1;
    __syncthreads();
    if (threadIdx.x == 100) s += 2;
    __syncthreads();
    if (threadIdx.x == 0) printf("s=%d\n", s);
}

int main() {
    int i;
    for (i = 0; i < 10; ++i) {
        dkernel<<<2, BLOCKSIZE>>>();
        cudaDeviceSynchronize();
    }
}
```

This one is redundant.

s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3  
s=3

# What is the output of this program?

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE    ...

__global__ void dkernel() {
    __shared__ char str[BLOCKSIZE+1];
    str[threadIdx.x] = 'A' + (threadIdx.x + blockIdx.x) % BLOCKSIZE;
    if (threadIdx.x == 0) {
        str[BLOCKSIZE] = '\0';
    }

    if (threadIdx.x == 0) {
        printf("%d: %s\n", blockIdx.x, str);
    }
}

int main() {
    dkernel<<<10, BLOCKSIZE>>>();
    cudaDeviceSynchronize();
}
```

What is the bug in this code?

# What is the output of this program?

```
#include <stdio.h>
#include <cuda.h>

#define BLOCKSIZE      ...

__global__ void dkernel() {
    __shared__ char str[BLOCKSIZE+1];
    str[threadIdx.x] = 'A' + (threadIdx.x + blockIdx.x) % BLOCKSIZE;
    if (threadIdx.x == 0) {
        str[BLOCKSIZE] = '\0';
    }
    __syncthreads(); // barrier across threads in a block
    if (threadIdx.x == 0) {
        printf("%d: %s\n", blockIdx.x, str);
    }
}

int main() {
    dkernel<<<10, BLOCKSIZE>>>();
    cudaDeviceSynchronize();
}
```

This is redundant if  
BLOCKSIZE <= 32.



# L1 versus Shared

- On CPU:
  - `cudaDeviceSetCacheConfig(kernelname, param);`
  - *kernelname* is the name of your kernel.
  - *param* is {`cudaFuncCachePreferNone`, `L1`, `Shared`}.
  - 3.x onward, one may also configure it as 32KB L1 + 32KB Shared. This is achieved using `cudaFuncCachePreferEqual`.

# L1 versus Shared

```
__global__ void dkernel() {  
    __shared__ unsigned data[BLOCKSIZE];  
    data[threadIdx.x] = threadIdx.x;  
}  
int main() {  
    cudaFuncSetCacheConfig(dkernel, cudaFuncCachePreferL1);  
    //cudaFuncSetCacheConfig(dkernel, cudaFuncCachePreferShared);  
    dkernel<<<1, BLOCKSIZE>>>();  
    cudaDeviceSynchronize();  
}
```

# Dynamic Shared Memory

- When the amount of shared memory required is unknown at compile-time, dynamic shared memory can be used.
- This is specified as the **third** parameter of kernel launch.

# Dynamic Shared Memory

```
#include <stdio.h>
#include <cuda.h>

__global__ void dynshared() {
    extern __shared__ int s[];

    s[threadIdx.x] = threadIdx.x;
    __syncthreads();

    if (threadIdx.x % 2) printf("%d\n", s[threadIdx.x]);
}

int main() {
    int n;
    scanf("%d", &n);
    dynshared<<<1, n, n * sizeof(int)>>>();
    cudaDeviceSynchronize();

    return 0;
}
```

# Shared Memory with Multiple Arrays

```
#include <stdio.h>
#include <cuda.h>

__global__ void dynshared(int sz, int n1) {
    extern __shared__ int s[];
    int *s1 = s;
    int *s2 = s + n1;
    if (threadIdx.x < n1) s1[threadIdx.x] = threadIdx.x;
    if (threadIdx.x < (sz - n1)) s2[threadIdx.x] = threadIdx.x * 100;
    __syncthreads();
    if (threadIdx.x < sz && threadIdx.x % 2) printf("%d\n", s1[threadIdx.x]);
}

int main() {
    int sz;
    scanf("%d", &sz);
    dynshared<<<1, 1024, sz * sizeof(int)>>>(sz, sz / 2);
    cudaDeviceSynchronize();

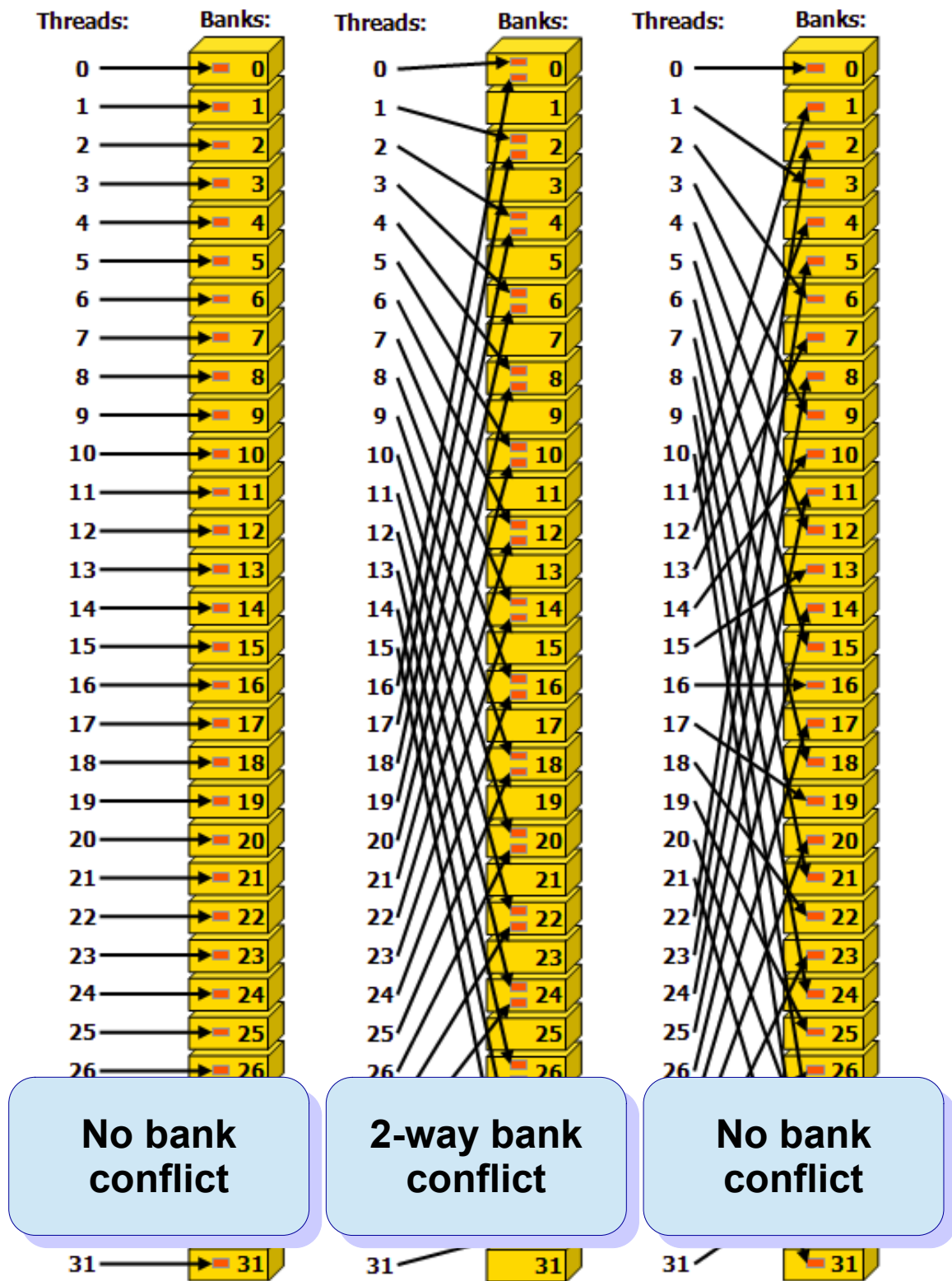
    return 0;
}
```

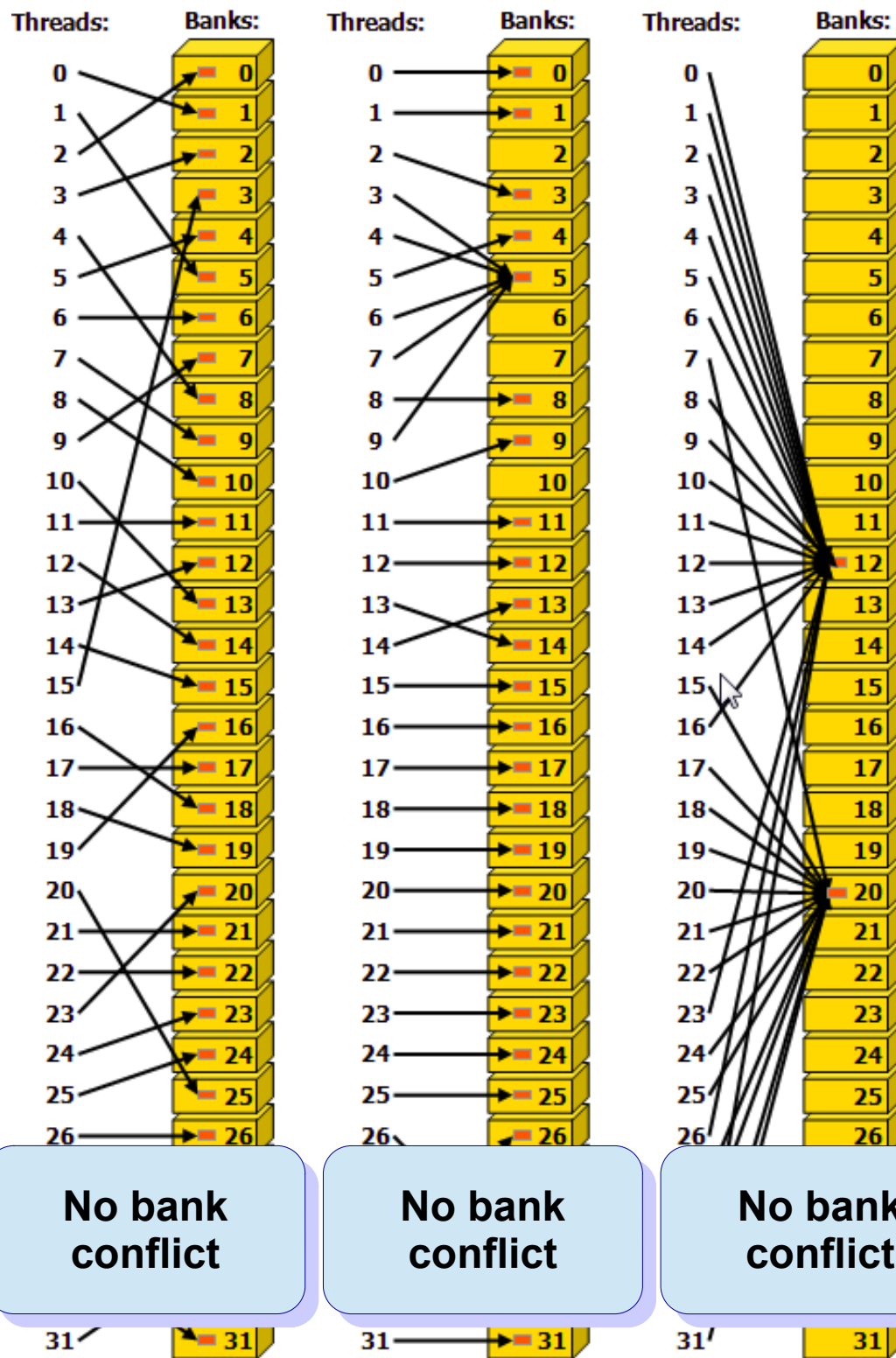
# Bank Conflicts

- Shared memory is organized into 32 banks.
- Accesses to the same bank are sequential.
- Consecutive words are stored in adjacent banks.
  - Useful for coalesced access.
- **Exception:** Warp accesses to the same word are not sequentialized.

```
__global__ void bankNOconflict() {  
    __shared__ unsigned s[1024];  
    s[1 * threadIdx.x] = threadIdx.x;  
}
```

```
__global__ void bankconflict() {  
    __shared__ unsigned s[1024];  
    s[32 * threadIdx.x] = threadIdx.x;  
}
```







# Texture Memory

- Fast read-only memory
- Optimized for 2D spatial access
- Definition: `texture<float, 2, cudaReadModeElementType> tex;`
- In main: `cudaBindTextureToArray(tex, cuArray, ...);`
- In kernel: `... = tex2D(tex, ...);`

# Texture Memory

- Example from CUDA SDK

```
__global__ void transformKernel(float *output, int width, int height, float theta)
{
    unsigned x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = (float)x - (float)width / 2;
    float v = (float)y - (float)height / 2;
    float tu = (u * cosf(theta) - v * sinf(theta)) / width;
    float tv = (v * cosf(theta) + u * sinf(theta)) / height;

    output[y * width + x] = tex2D(tex, tu + 0.5, tv + 0.5);
}
```



# Constant Memory

- Read-only Memory
- 64KB on the GPU
- Definition: **\_\_constant\_\_** unsigned meta;
- Main: **cudaMemcpyToSymbol**(meta, &hmeta, sizeof(unsigned));
- Kernel: data[threadIdx.x] = meta[0];

# Constant Memory

```
#include <cuda.h>
#include <stdio.h>

__constant__ unsigned meta[1];

__global__ void dkernel(unsigned *data) {
    data[threadIdx.x] = meta[0];
}

__global__ void print(unsigned *data) {
    printf("%d %d\n", threadIdx.x, data[threadIdx.x]);
}

int main() {

    unsigned hmeta = 10;
    cudaMemcpyToSymbol(meta, &hmeta, sizeof(unsigned));
    unsigned *data;
    cudaMalloc(&data, 32 * sizeof(unsigned));
    dkernel<<<1, 32>>>(data);
    cudaDeviceSynchronize();
    print<<<1, 32>>>(data);
    cudaDeviceSynchronize();
    return 0;
}
```

# Compute Capability

- Version number: **Major.minor** (e.g., 6.2)
  - Features supported by the GPU hardware.
  - Used by the application at runtime (*-arch=sm\_62*).
- `__CUDA_ARCH__` is defined (e.g., 620) in device code.
- CUDA version is the software version (e.g., CUDA 10.1).

# Compute Capability

Major number	Architecture
1	Tesla
2	Fermi
3	Kepler
5	Maxwell
6	Pascal
7	Volta
8	Turing
9	Ampere
10	Lovelace
11	Hopper

# Compute Capability

Feature	2.x	3.0	3.5, 3.7, 5.0, 5.2	6.x	7.x	8.0
<b>Atomics int, float</b>	Yes	Yes	Yes	Yes	Yes	Yes
<b>warp-vote</b>	Yes	Yes	Yes	Yes	Yes	Yes
<b>__syncthreads</b>	Yes	Yes	Yes	Yes	Yes	Yes
<b>Unified memory</b>		Yes	Yes	Yes	Yes	Yes
<b>Dynamic parallelism</b>			Yes	Yes	Yes	Yes
<b>Atomics double</b>				Yes	Yes	Yes
<b>Tensor core</b>					Yes	Yes
<b>Hardware async copy</b>						Yes

# Classwork

- Write CUDA code for the following functionality.
  - Assume following data type, filled with some values.  
`struct Point { int x, y; } arr[N];`
  - Each thread should operate on 4 elements.
  - Find the average AVG of x values.
  - If a thread sees y value above the average, it replaces all 4 y values with AVG.
  - Otherwise, it adds y values to a global sum.
  - Host prints the number of elements set to AVG.



# CUDA, in a nutshell

- [Compute Unified Device Architecture](#). It is a hardware and software architecture.
- Enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, and other languages.
- A CUDA program calls parallel [kernels](#). A kernel executes in parallel across a set of parallel threads.
- The programmer or compiler organizes these threads in thread [blocks and grids](#) of thread blocks.
- The GPU instantiates a kernel program on a grid of parallel thread blocks.
- Each thread within a thread block executes an instance of the kernel, and has a [thread ID](#) within its thread block, program counter, registers, per-thread private memory, inputs, and output results.
- A thread block is a set of concurrently executing threads that can cooperate among themselves through [barrier synchronization and shared memory](#).
- A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, and write results to global memory.
- Each thread has a per-thread [private memory space](#) used for register spills, function calls, and C automatic array variables.
- Each thread block has a per-block [shared memory space](#) used for inter-thread communication, data sharing, and result sharing in parallel algorithms.