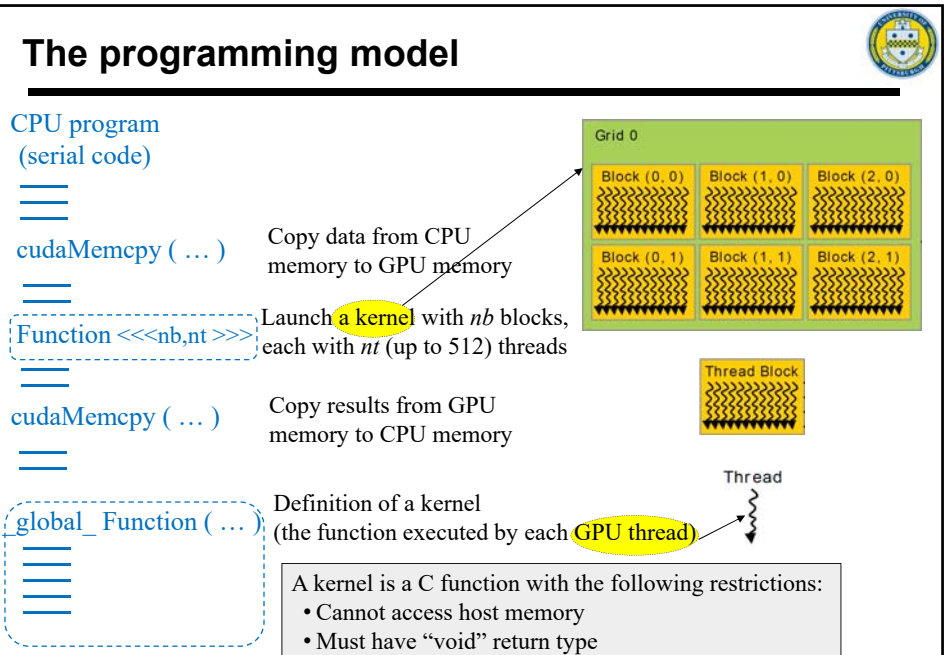
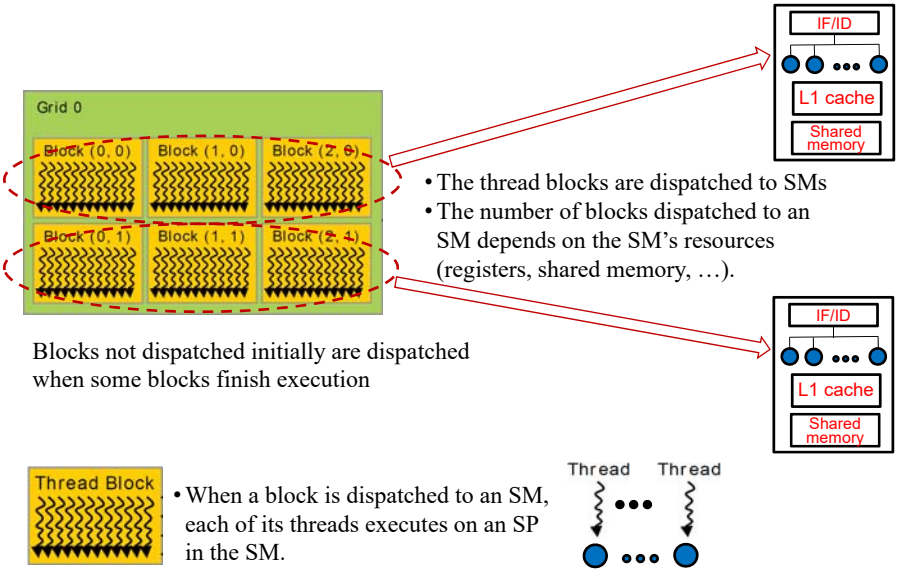
# The programming model




A kernel is a C function with the following restrictions:
- Cannot access host memory
- Must have "void" return type
- No variable number of arguments or static variables
- No recursion



# The execution model



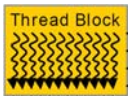

- The thread blocks are dispatched to SMs
- The number of blocks dispatched to an SM depends on the SM's resources (registers, shared memory, …).

Blocks not dispatched initially are dispatched when some blocks finish execution

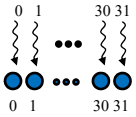



- When a block is dispatched to an SM, each of its threads executes on an SP in the SM.

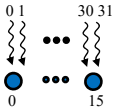
## The execution model



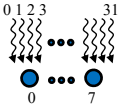

- Each block (up to 512 threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.
- The 32 threads of a warp execute in SIMD mode on the SM.
- The (up to 16) warps of a block are "coarse grain multithreaded"
- Depending on the number of SPs per SM:




➢ If 32 SP per SM → one thread of a warp executes on one SP (32 lanes of execution, one thread per lane)




➢ If 16 SP per SM → every two threads of a warp are time multiplexed (fine grain multithreading) on one SP (16 lanes of execution, 2 threads per lane)

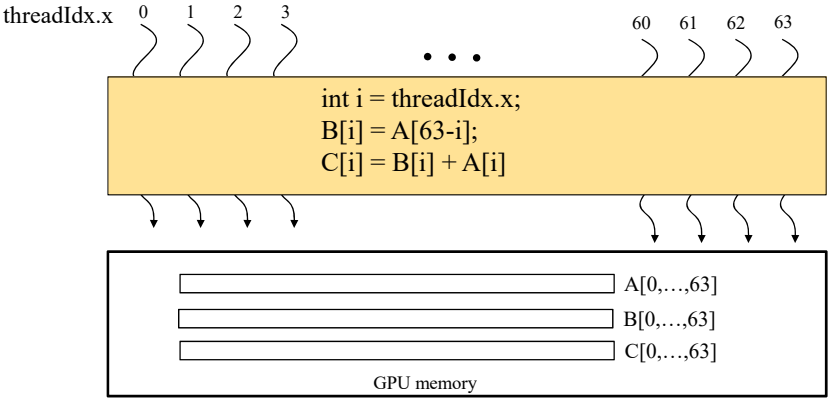



➢ If 8 SP per SM → every four threads of a warp are time multiplexed (fine grain multithreading) on one SP (8 lanes of execution, 4 threads per lane)



## All threads execute the same code

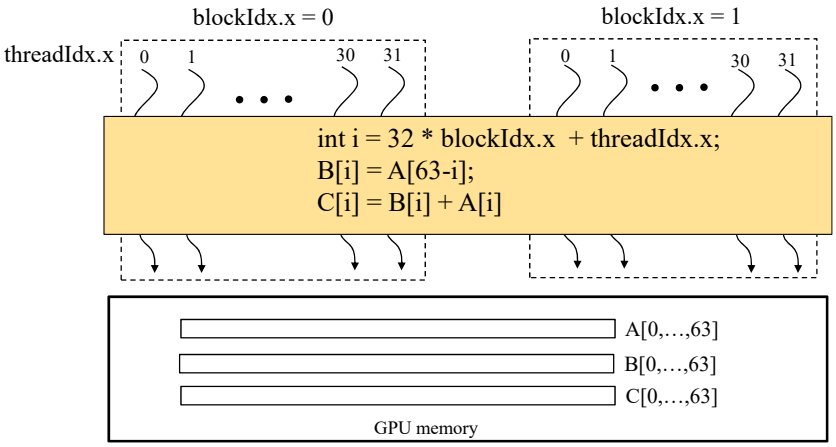
Assume one block with 64 threads – launched using **Kernel <<<1, 64>>>**




- Each thread in a thread block has a unique "thread index" → threadIdx.x
- The same sequence of instructions can apply to different data

## Blocks of threads

Assume two block with 32 threads each – launched using **Kernel <<<2, 32>>>**

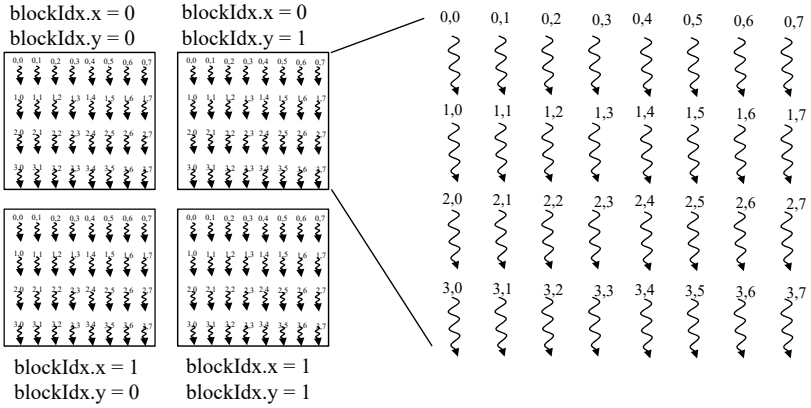



- Each thread block has a unique "block index" → blockIdx.x
- Each thread has a unique threadIdx.x within its own block
- Can compute a global index from the blockIdx.x and threadIdx.x



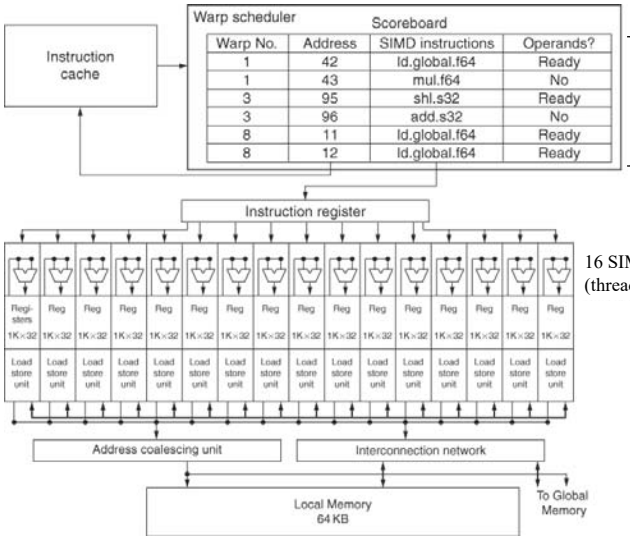
## Two-dimensions grids and blocks

Can launch a 2x2 array of blocks, each consisting of 4x8 array of threads using **Kernel <<<(2,2), (4,8)>>>**




- Each block has two indices (blockIdx.x, blockIdx.y)
- Each thread in a thread block has two indices (threadIdx.x, threadIdx.y)

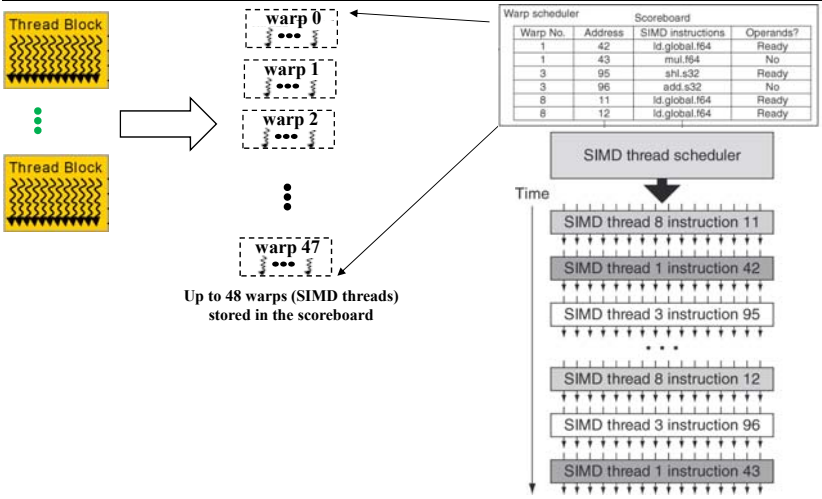
# Scheduling warps on SMs




**Block diagram of a 16-lanes SM:** A scoreboard keeps track of the current PCs (instruction address) of up to NW independent threads of SIMD instructions (NW warps).
**NW** = 48 in older GPUs and 64 for more recent GPUs



# Scheduling warps on SMs



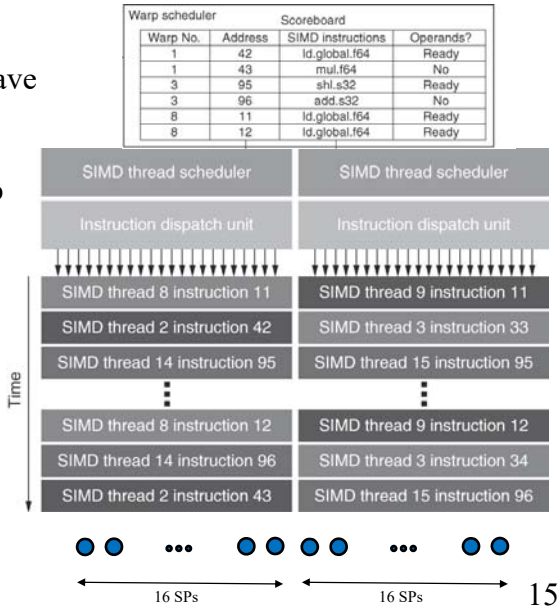

**Scheduling threads of SIMD instructions (warps):** The scheduler selects a ready instruction from some warp and issues it synchronously to all the SIMD lanes. Because warps are independent, the scheduler may select an instruction from a different warp at every issue.

## SMs with multiple warp schedulers

Newer SMs have large number of SPs and may have multiple warp schedulers.

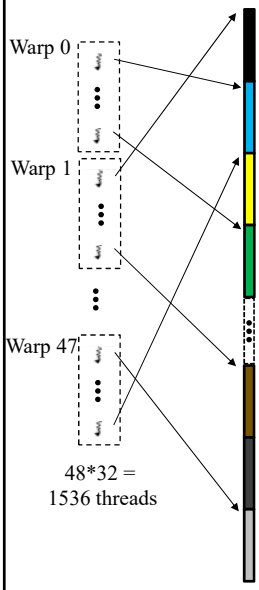
**Example: SM with Dual warp Scheduler.**

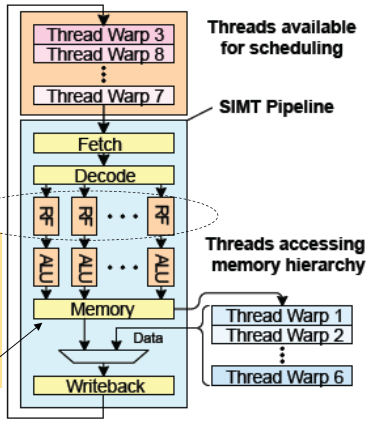
- Each SM has 32 SPs (cores) and is divided into two groups of 16 lanes each.
- One warp scheduler is responsible for scheduling a warp on one of the two 16-lanes groups.
- A warp is issued to each 16-lanes group every two cycles.






## Sharing the resources of an SM



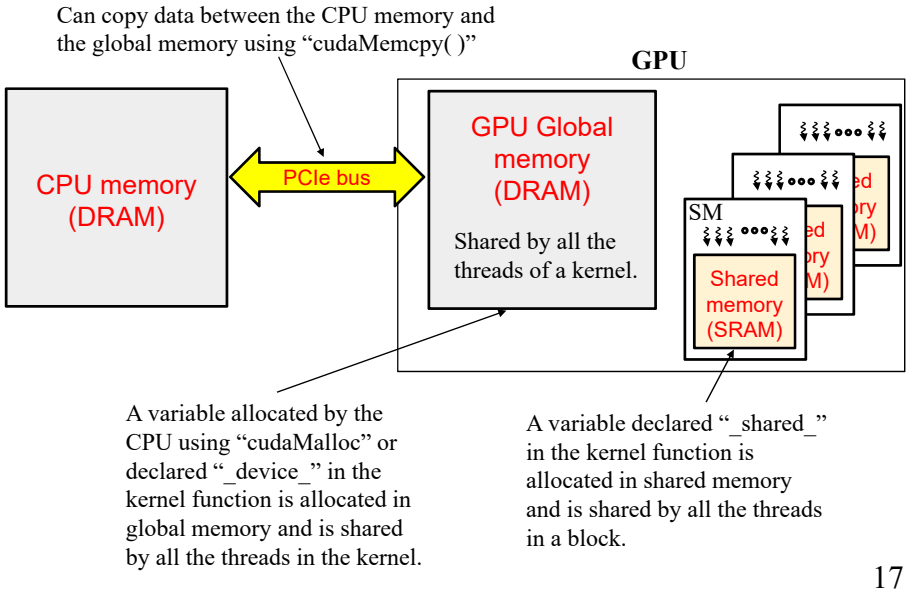

**Register file**

A single large register file (ex. 16K registers) is partitioned among the threads of the dispatched blocks

**Shared memory**

A single SRAM (ex. 16KB) is partitioned among the dispatched blocks. All the threads of a block can access the partition assigned to that block

## The memory architecture






## Getting data into GPU memory

cudaMalloc (void **pointer, size_t nbytes); /* malloc in GPU global memory */

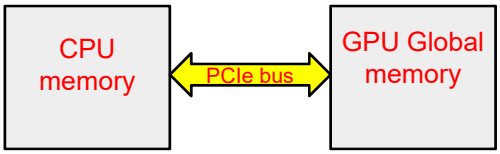
cudaMemset (void **pointer, int value, size_t count);

cudaMemcpy(void *dest, void *src, size_t nbytes, enum cudaMemcopyKind dir)

cudaFree(void **pointer) ;

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice




Notes:

- cudaMemcpy() blocks CPU thread until copy is complete
- cudaMemcpy() does not start copying until previous CUDA calls complete

# Data Movement Example

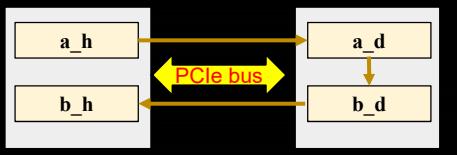
```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   GPUcomp<<<1, 14>>>(a_d, b_d, N) ;
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



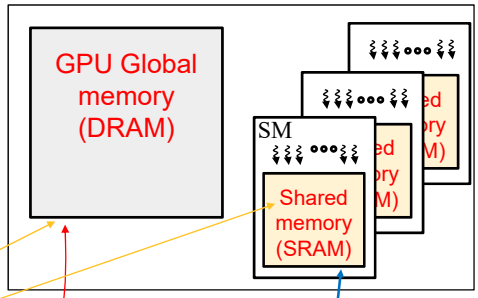

```
_global_ void GPUcomp(*a,*b,N)
{
  int i = threadIdx.x ;
  if( i < N) b(i) = a(i) ;
}
```

# Variable qualifiers in CUDA kernel functions

```
int main (void) {
  ___
  ___
kernel <<<16,64 >>>
  ___
  ___
  ___
  ___
  ___
}

_global_  void kernel ( … )
{
   int i ;
   _device_  float y[1024] ;
   _shared_  int x[64] ;
   ___
   ___
   x[threadIdx.x] = …
   … = y[blockIdx.x]
}
```

One copy of i is allocated per thread – either in a register or in global memory (if run out of registers)




y[] (1024 elements) is allocated in global memory and can be accessed by any thread (instead of using cudaMalloc)

One copy of the array x[] (64 elements) is allocated in shared memory for each thread block and can be accessed by each thread in that block

## Function qualifiers

- The functional qualifier "**_global_**" is used to designate a kernel
  - Called from Host and executed on Device
  - Must return "void"
- The functional qualifier "**_device_**" designates a function
  - Called from Device and executed on Device
  - Cannot be called from Host
- The functional qualifier "**_host_**" designates a function
  - Called from Host and executed on Host (default)

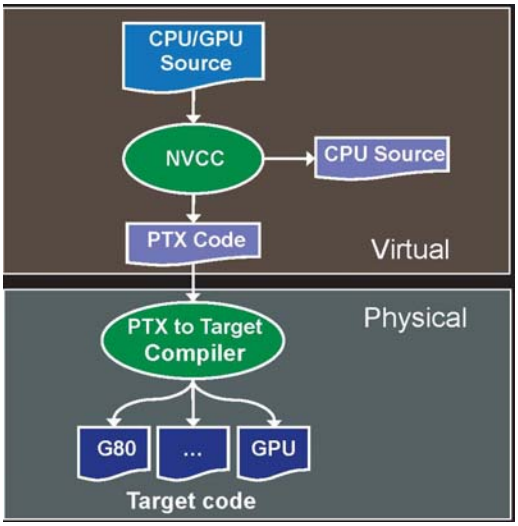
"**_host_**" and "**_device_**" can be combined to generate code that can execute on both Host and Device



## Compiling

nvcc outputs either
- C code for the CPU
- PTX or object code for GPU

An executable requires linking to:
- "cudart" run time library
- "cuda" code library



## Launching a kernel

- A kernel is launched using the syntax:

  kernel<<<dim3 dG, dim3 dB>>> (…) ;

  dim3 is a predefined data type

  parameters

- Execution configuration:
  - dG → dimension and size of grids in blocks
  - 3-dimensions, dG.x, dG.y and dG.z (default dG.y = dG.z =1)
  - Number of blocks in the launched grid = dG.x * dG.y * dG.z
  - dB → dimension and size of blocks in threads
  - 3-dimensions, dB.x, dB.y and dB.z (default dB.y = dB.z = 1)
  - Number of threads per block = dB.x * dB.y * dB.z

Note: kernel launch is non-blocking → control returns to CPU immediately



## Launching a kernel

```
dim3 grid, block;
grid.x = 2 ; grid.y = 4;
block.x = 4 ; block.y = 16 ;
kernel <<<grid, block>>>(…);
```

gridDim.x = 2
gridDim.y = 4
blockDim.x = 4
blockDim.y = 16
blockIdx.x = 0,1
blockIdx.y = 0, 1, 2, 3
threadIdx.x = 0, … ,3
threadIdx.y = 0, … , 15

In different blocks

In different threads

```
dim3 grid(2,4), block(4,16);
kernel <<<grid, block>>>(…);
```

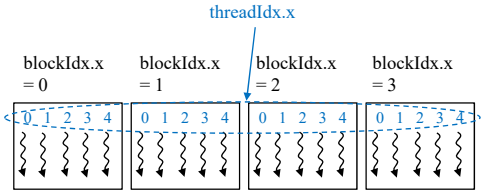
```
kernel <<<8, 512>>>(…);
```

gridDim.x = 8
blockDim.x = 512
blockIdx.x = 0, …, 7
threadIdx.x = 0, … ,511

gridDim, blockDim, blockIdx and threadIdx are built in variables of type dim3 → accessible by the kernel function.

## Example:

```
void main ()
{  cudaMalloc (int* &a, 20*sizeof(int));
   cudaMalloc (int* &b, 20*sizeof(int));
   cudaMalloc (int* &c, 20*sizeof(int));
 …
   kernel<<<4,5>>>(a, b, c) ;
   …
}
```




```
_global_ void kernel(int *a, *b, *c)
{ int i = blockIdx.x * blockDim.x + threadIdx.x ;
  a[i] = i ;
  b[i] = blockIdx.x;
  c[i] = threadIdx.x;
 }
```

*NOTE: Each block will consist of one warp – only 5 threads in the warp will do useful work and the other 27 threads will execute no-ops.*

Global Memory

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a[] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| b[] | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| c[] | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |



## Example: increment the elements of an array

**C program (on CPU)**

```
void inc_cpu(int *a, int n)
{ int i ;
  for(i=0; i<n; i++)
        a[i] = a[i] + 1;
 }
```

```
void main ()
{
  …
  inc_cpu(a,n);
  …
}
```

**CUDA program (on CPU+GPU)**

```
_global_ void inc_gpu(int *A, int n)
{ int i = blockIdx.x * blockDim.x + threadIdx.x ;
  if( i < n )
        A[i] = A[i] + 1;
 }
```

```
void main ()
{ …
  blocksize = 64 ;
  // cudaMalloc array A[n]
  // cudaMemcpy data to A
  dim3 dimB (blocksize) ;
  dim3 dimG(ceil(n/blocksize));
  inc_gpu<<<dimG, dimB>>>(A, n) ;
  …
}
```

n/64 64

## Example: computing $y = ax + y$

**C program (on CPU)**

```
void saxpy_serial(int n, float a,
float *x, float *y)
{
  for(int i = 0; i<n; i++)
        y[i] = a * x[i] + y[i];
}
```

```
void main ()
{
  …
  saxpy_serial(n, 2.0, x, y);
  …
}
```

**CUDA program (on CPU+GPU)**

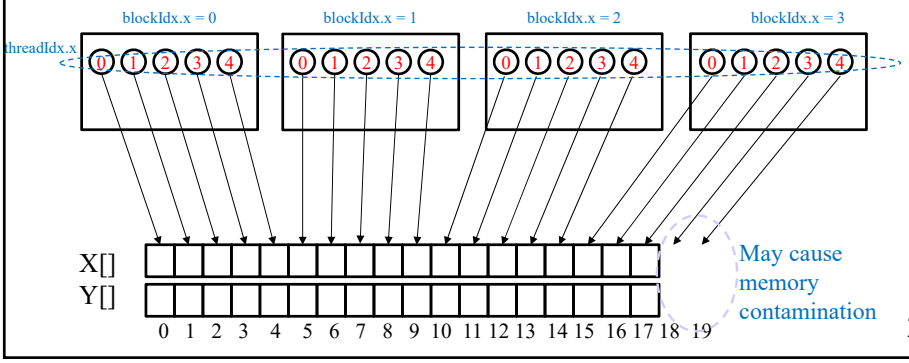
```
_global_void saxpy_gpu(int n, float a,
float *x, float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n ) y[i] = a * x[i] + y[i];
}
```

```
void main ()
{ …
  // cudaMalloc arrays X and Y
  // cudaMemcpy data to X and Y
  // blocksize = 256 ;
  int NB = (n + 255) / 256;
  saxpy_gpu<<<NB, 256>>>(n, 2.0, X, Y);
  // cudaMemcpy data from Y
}
```



## Example: computing $y = ax + y$

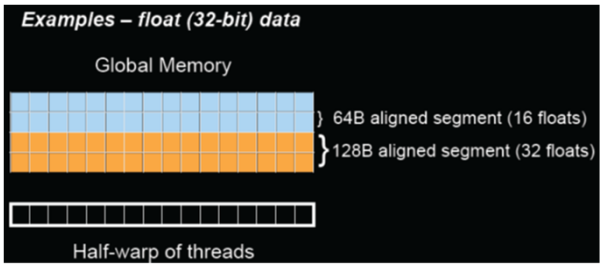
```
_global_void saxpy_gpu(int n, float a, float *X, float *Y)
{
  int i = blockIdx.x*blockDim.x +  threadIdx.x;
  if (i < n ) Y[i] = a * X[i] + Y[i];
}
…..
saxpy_gpu<<<4, 5>>>(18, 2.0, X, Y); /* X and Y are arrays of size 18 each
*/
```



## Global Memory



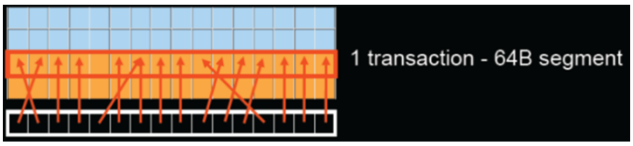
- Global memory is the off-chip DRAM memory
  - Accesses must go through interconnect and memory controller
- Many concurrent threads generate memory requests → coalescing is necessary
  - Combining memory accesses made by threads in a warp into fewer transactions – each memory transaction is for 64 bytes (16, 4-bytes words).
  - E.g. if each thread of a warp are accessing consecutive 4-byte sized locations in memory, send two 64-byte request to DRAM instead of 32 4-byte requests
- Coalescing is achieved for any pattern of addresses that fits into a segment of size 128B.

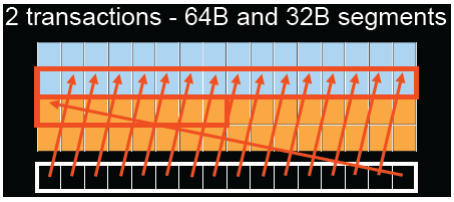


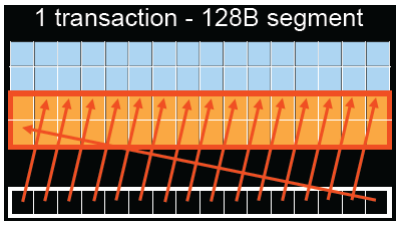



## Coalescing (cont.)





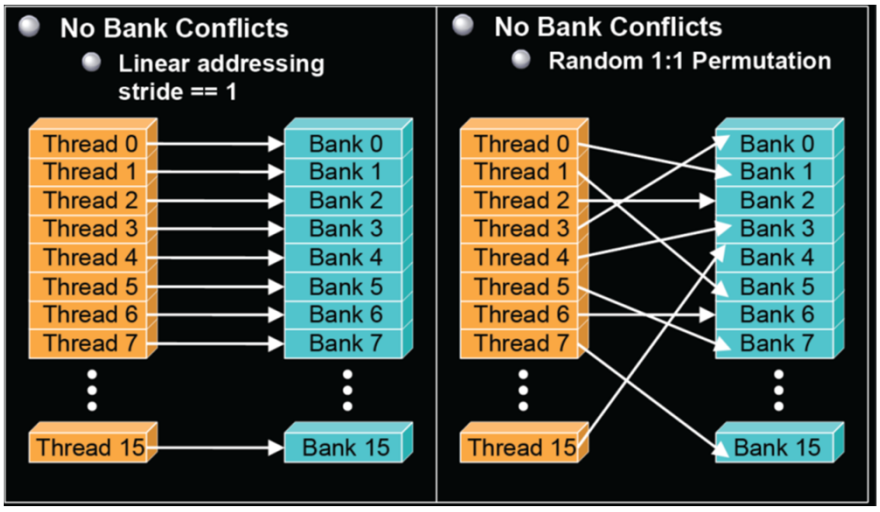

A warp may issue up to 32 memory accesses. They can be completed in
- Two transactions, each for 16 coalesced accesses (if perfectly coalesced)
- 32 separate transactions (if addresses cannot be coalesced)
- Fewer than 32 transactions (if partial coalescing is possible).

## Shared Memory

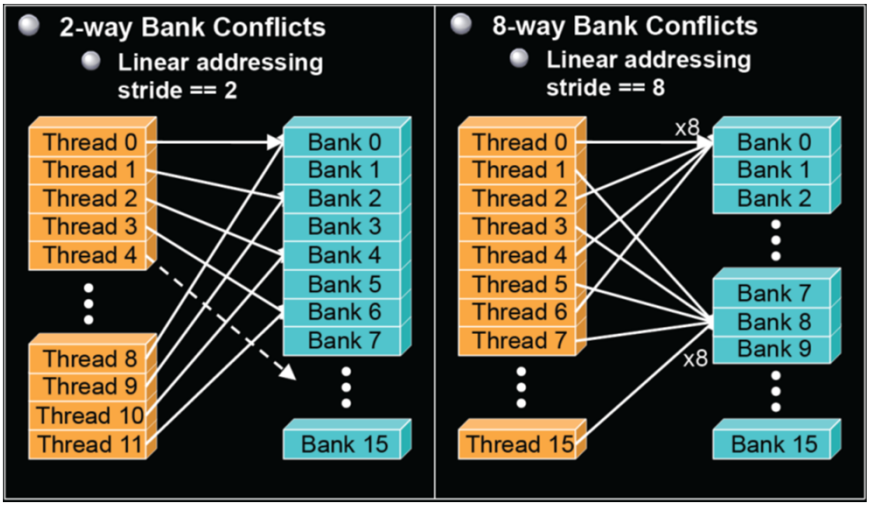
- A memory address subspace in each SM (at least 48KB in nvidia gpus)
  - As fast as register files if no bank conflict
  - May be used to reduce global memory traffic (called scratchpad)
- Managed by the code (programmer)
- Many threads accessing shared memory → Highly banked
  - Successive 32-bit words assigned to successive banks
- Each bank serves one address per cycle
  - Shared Memory can service as many simultaneous accesses as it has banks
- Multiple concurrent accesses to a bank result in a bank conflict (has to be serialize)




## Bank Addressing Example



## Bank Addressing Example (cont.)






## Synchronization

- **_synchthreads_()** ; a barrier for threads within a thread block
- Allowed in conditional code only if all condition is uniform in all the threads of a block
- Used to avoid hazards when using shared memory

To synchronize threads across different thread blocks, need to use **atomic operations** on variables in global memory

- Add, sub, min, max, …
- And, or, xor
- Increment, decrement
- Exchange, compare and swap

**cudaThreadSynchronize():** called in CPU code to block until all previously issued cuda calls complete.

## Maximize the use of shared memory

Shared memory is hundreds of times faster than global memory

To take advantage of shared memory

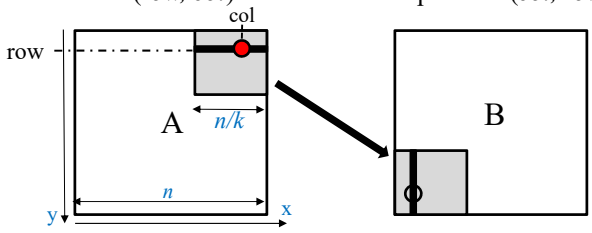
- Partition the data sets into subsets that fit into shared memory
- Handle each subset with one thread block
  - Load the subset from global memory to shared memory
  - _synchthread()
  - Perform the computation while data is in shared memory
  - _synchthread()
  - Copy results back from shared to global memory

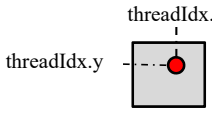
The code examples in the next slides are copied and modified from:
J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA"
Queue - GPU Computing Maganine, Volume 6 Issue 2, March/April 2008, Pages 40-53 .



## Transposing an nxn matrix

- *B[j][i] = A[i][j]* for *i=0,…, n-1* and *j=0,…,n-1*
- Partitions A into $k^2 = k \times k$ tiles, each with *n/k rows and n/k columns*
- Launch a 2-D grid of *(k,k)* thread blocks, each with *(n/k,n/k)* threads
- Assign (*threadIdx.x, threadIdx.y*) in (*blockIdx.x, blockIdx.y*) to copy element (*row, col*) of matrix *A* into position (*col, row*) of matrix *B*.




tiledim = *n/k*
*k = gridDim.x = gridDim.y*

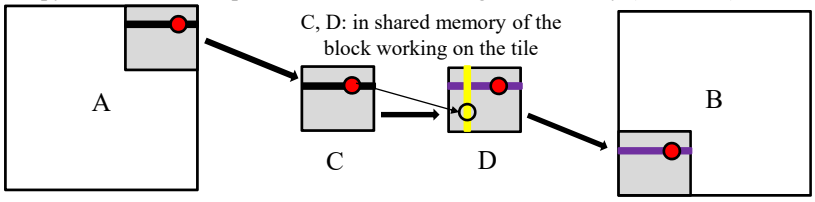
```
__global__ void transpose(float* A, float* B, int n)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col  = blockIdx.x * blockDim.x + threadIdx.x;
     B[col][row] = A[row][col];
}
```

lw Reg, A[row + n*col]  // coalesced
sw Reg, B[col + n*row]  // not coalesced

Naïve implementation exhibits non-coalesced access to global memory

# Transposing with coalesced memory access

- Each block copies the rows of its tile from global to shared memory (coalesced)
- Transpose in shared memory
- Copy rows of the transposed tile from shared to global memory (coalesced)

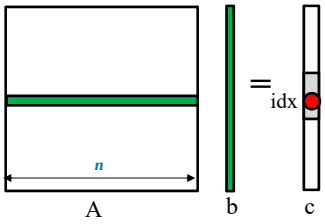



```
__global__ void transpose(float* A, float* B, int n)
{   _shared_ float C[tiledim][tiledim], D[tiledim][tiledim];  // defined parameter "tiledim" = n/k
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col  = blockIdx.x * blockDim.x + threadIdx.x;
    C[threadIdx.y][threadIdx.x] = A[row][col];                // coalesced load from A
    _synchthreads() ;                                         // complete loading before transposing
    D [threadIdx.x][threadIdx.y] = C[threadIdx.y][threadIdx.x] ;
    _synchthreads() ;                                         // complete transpose in shared memory
    int row = blockIdx.x * blockDim.y + threadIdx.y;          // note that blockDim.x = blockDim.y
    int col  = blockIdx.y * blockDim.x + threadIdx.x;
    B[row][col] = D[threadIdx.y][threadIdx.x]                 // coalesced storing to B
}
```



# Multiplication c = A * b of an *nxn* matrix, A, by a vector b

- Each element *c[i]* will be computed by one thread
- Partitions c into *k* parts, each with *n/k* elements
- Launch *k* thread blocks, each with *n/k* threads
- Thread *threadIdx.x* in block *blockIdx.x* computes *c[idx]* where *idx = blockIdx.x * blockDim.x + threadIdx.x*




```
__global__ void mv(float* A, float* b, float* c, int n)
{     int row = blockIdx.x * blockDim.x + threadIdx.x;
      if (row < n) {
            float temp = 0 ;
            for (int k = 0; k < n; k++)
                  temp   += A[row][k] * b[k] ;
      }
      c[row] = temp ;
}
```

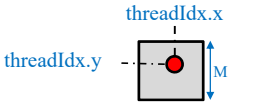
Can you make use of shared memory?

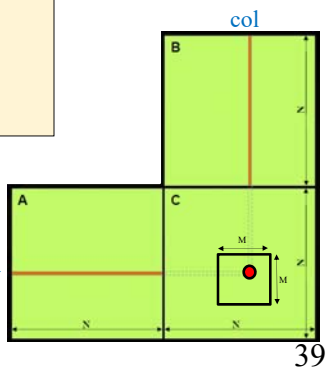
```
#define blocksize = 128;                          // or any other block size
n = 2048 ;                                        // or any other matrix size
int nblocks = (n + blocksize - 1) / blocksize;
mv<<<nblocks, blocksize>>>(A, b, c, n);
```

## Multiplication C=A*B of two *nxn* matrices.

- Partitions each matrix into $k^2 = k \times k$ tiles, each with *M rows and M columns (M= N/k)*
- Launch a 2-D grid of *(k,k)* thread blocks, each with *(M,M)* threads
- Assign thread (threadIdx.x, threadIdx.y) in block (blockIdx.x, blockIdx.y) to handle element (row, col) of the matrix C.

```
__global__ void mm_simple(float* C, float* A, float* B, int N)
{   int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col  = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    for (int k = 0; k < N; k++)
            sum += A[row][k] * B[k][col];
    C[row][col] = sum;
}
```




**In the main program:**

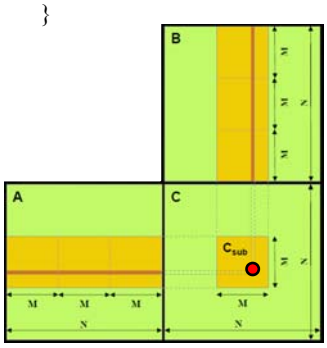
cudaMalloc d_A, d_B and d_C ;
cudaMemcopy to d_A, d_B ;
dim3 threads(tiledim, tiledim);  // tiledim = *M* = *N/k*
dim3 grid(n/tiledim, n/tiledim);
mm_simple <<< grid, threads >>>(d_C, d_A, d_B, *N*);
_synchtreads();
cudaMemcopy back d_C ;

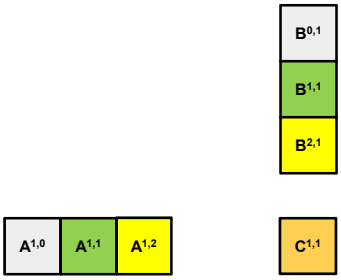


## Matrix-matrix multiplication using shared memory

- The thread that computes C[row][col] accesses row A[row][*] and column B{*][col] from the global memory
- Hence, each row of A and each column of B is accessed *M* times by different threads in the same block.
- To avoid repeated access to global memory, the block that computes a tile $C^{i,j}$, where $0 \le i < k$ and $0 \le j < k$ executes:

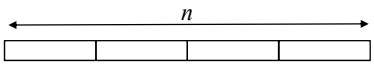
```
for q = 0, ... , k-1                    // k = 3 in the example shown */
    {  load tiles A^{i,q} and B^{q,j} into the shared memory
       C^{i,j} = C^{i,j}  + A^{i,q} * B^{q,j}     // accumulate the product of A^{i,q} * B^{q,j}
    }
```



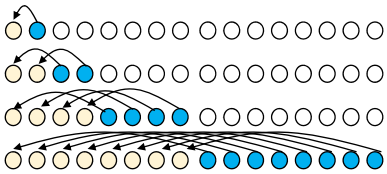
## Parallel reduction

```
_global_void reduce(int *input, int n, int *total_sum)
{
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  _shared_ int x[blocksize];
  x[tid] = input[i] ;  // load elements into shared memory
  _syncthreads();
  // Tree reduction over elements of the block.
  for(int half = blockDim.x/2; half > 0; half = half /2)
  {
    if(tid < half) x[tid] += x[tid + half];
    _syncthreads();
  }
  // Thread 0 adds the partial sum to the total sum
  if( tid == 0 ) atomicAdd(total_sum, x[tid]);
}
```




*input array*

*Partitioned the array into nblocks of "blocksize" elements each*



*Tree reduction within each block*

```
#define blocksize = 128;                    // or any other block size
n = 2048 ;                                  // or any other array size
int nblocks = n / blocksize;
reduce <<<nblocks, blocksize>>> (input, &total_sum);
```



## Static Vs dynamic shared arrays

- When declaring a shared array using "_shared_ float C[tiledim][tiledim]" , tiledim has to be statically defined. For example, in the matrix transpose example, the main should look like

```
#define tiledim 16 ;                    // blocksize = 16*16 = 256
n = 2048 ;                              // the dimension of the matrix
dim3 threads(tiledim, tiledim) ;
dim3 grid(n/tiledim, n/tiledim) ;       // assuming n is a multiple of 16
transpose<<<grid, threads>>>  (A, B, n) ;
```

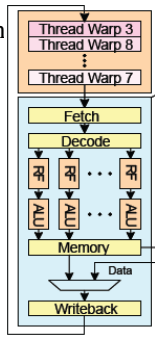
- Can declare an unsized shared array using "extern __shared__ float C[]" (note the [] ) and then at kernel launch time, use a third argument to specify the size of the shared array:

```
n = 2048 ;
tiledim =  *** ;                        // can determine dynamically
dim3 threads(tiledim, tiledim) ;
dim3 grid(n/tiledim, n/tiledim) ;
transpose<<<grid, threads, tiledim*tiledim*sizeof(float)>>>  (A, B, n) ;
```

## Dealing with data dependency

- Instructions in a warp are scheduled "in order"
- No pipeline forwarding → a dependent instruction cannot be issued
- Latency due to dependencies are hidden by issuing instructions from other warps (similar to multithreading – call it multi-warping)

```
add.f32        $f3, $f1, $f2
add.f32        $f5, $f3, $f4   <- Cannot be issued before $f3 is written back
ld.shared.f32  $f6, 0($r31)
add.f32        $f6, $f6, $f7
sw.shared.f32  $f6, 0($r31)
```




- Each SP is a 6-stage pipeline with no forwarding
- Two dependent instructions have to be separated by 6 other instructions
- Can completely hide the latency if the scheduler can issue instructions from 6 different warps(192 threads) . They may be form different blocks.



## Blocks per grid heuristics

The number of blocks should be larger than the number of SMs

**The number of threads per SM is limited by the number of registers per SM** – note that local variables not assigned to registers are stored in global memory. May use *–maxregcount = n* flag when compiling to restrict the number of registers per thread to *n*.

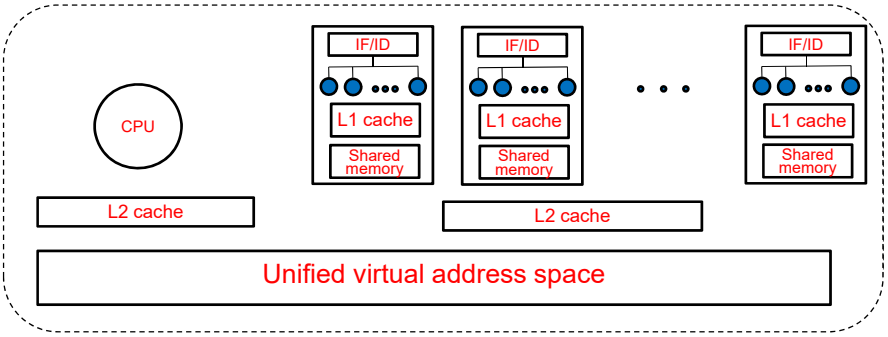
Example: If kernel uses 8 registers per thread, and register file is 16K registers per SM, then each SM can support at most 2048 threads)

**The number of Blocks per SM is determined by the shared memory declared in each block and the number of threads per block**.

Example: If 2KB shared memory is declared per block and each SM has 16KB of shared memory, then each SM can support at most 8 blocks).

Example: If each block has 256 threads (8 warps), and the GPU can support 48 warps per SM, then each SM can support at most 6 blocks.

## Unified CPU/GPU memory (in CUDA 6 and later)




- CPU and GPU share the same virtual memory (UVM)
- If physical memory is integrated (shared), then CPU and GPU use the same page table.
- If each of the CPU and the GPU has its own physical memory, pages are copied on demand (triggered by page faults).



## Unified CPU/GPU memory

CPU code (not using a GPU)

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 code with unified memory

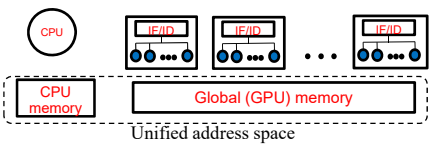
```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```




The pointer is passed as argument to the kernel → copying occurs on demand.

No need for
malloc(&cpudata, N) ;
cudaMalloc(&data, N);
cudaMemcpy(&cpudata, &data ..) ;