

# Memory Coalescing Techniques

## 1 Accessing Global and Shared Memory

- acceleration with graphics processing units
- memory coalescing to global memory
- avoiding bank conflicts in shared memory

## 2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

## 3 Avoiding Bank Conflicts

- data staging algorithms
- arrays of composite data
- computing consecutive powers

MCS 572 Lecture 32  
Introduction to Supercomputing  
Jan Vershelde, 31 March 2023

# Memory Coalescing Techniques

- 1 Accessing Global and Shared Memory
  - acceleration with graphics processing units
  - memory coalescing to global memory
  - avoiding bank conflicts in shared memory
- 2 Memory Coalescing Techniques
  - accessing global memory for a matrix
  - using shared memory for coalescing
- 3 Avoiding Bank Conflicts
  - data staging algorithms
  - arrays of composite data
  - computing consecutive powers

# acceleration with graphics processing units

Graphics Processing Units (GPUs) achieve teraflop performance: can execute a trillion floating-point operations per second.

Instruction level, data parallel algorithms are required:

- 1 blocks of threads execute the same instructions on different data,
- 2 many more threads than the number of cores must be launched, to keep the GPU fully occupied and achieve teraflop performance.

Blocks of threads are launched by the Central Processing Unit (CPU), called the host, and the device (GPU) accelerates the computations.

In this lecture, we look at how threads access memory, we define memory coalescing and bank conflicts.

Avoiding bank conflicts leads to better performing algorithms.

Data staging algorithms arrange data for memory coalescing.

# Memory Coalescing Techniques

## 1 Accessing Global and Shared Memory

- acceleration with graphics processing units
- **memory coalescing to global memory**
- avoiding bank conflicts in shared memory

## 2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

## 3 Avoiding Bank Conflicts

- data staging algorithms
- arrays of composite data
- computing consecutive powers

# dynamic random access memories (DRAMs)

Accessing data in the global memory is critical to the performance of a CUDA application.

In addition to tiling techniques utilizing shared memories we discuss memory coalescing techniques to move data efficiently from global memory into shared memory and registers.

Global memory is implemented with dynamic random access memories (DRAMs). Reading one DRAM is a very slow process.

Modern DRAMs use a parallel process:

Each time a location is accessed, many consecutive locations that includes the requested location are accessed.

If an application uses data from consecutive locations before moving on to other locations, the DRAMs work close to the advertised peak global memory bandwidth.

## memory coalescing

Recall that all threads in a warp execute the same instruction.

When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations.

The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations.

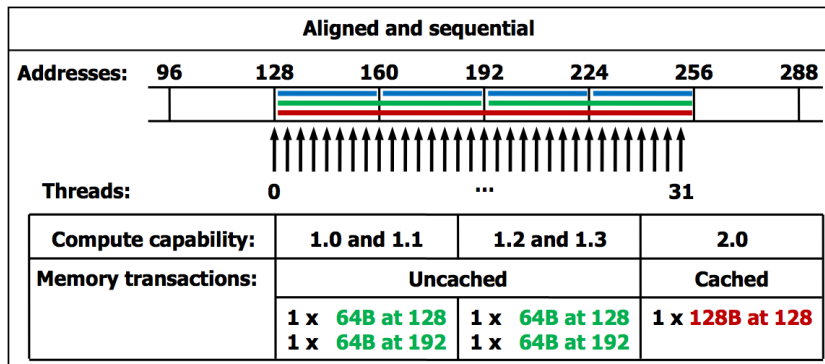
In this favorable case, the hardware *coalesces* all memory accesses into a consolidated access to consecutive DRAM locations.

### Definition (memory coalescing)

If, in a warp, thread 0 accesses location  $n$ , thread 1 accesses location  $n + 1$ ,  $\dots$  thread 31 accesses location  $n + 31$ , then all these accesses are *coalesced*, that is: combined into one single access.

The CUDA C Best Practices Guide gives a high priority recommendation to coalesced access to global memory.

# an example of a global memory access by a warp



from Figure G-1 of the *NVIDIA Programming Guide*.

# aligned memory access for higher compute capability

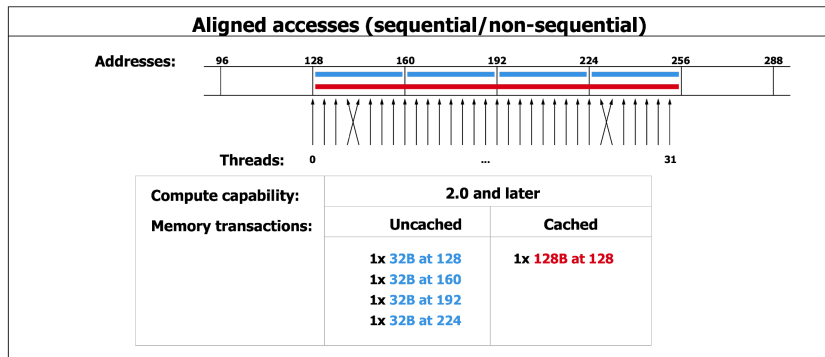


Figure 16 of the 2016 *NVIDIA Programming Guide*



# mis-aligned memory access

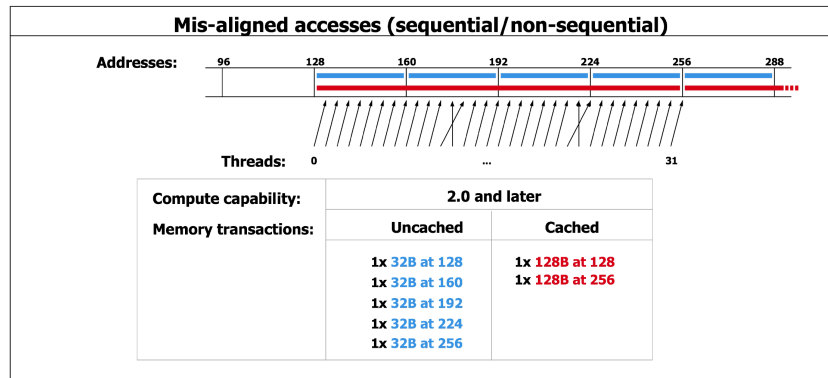


Figure 16 of the 2016 *NVIDIA Programming Guide*

## alignment in memory

In `/usr/local/cuda/include/vector_types.h`  
we find the definition of the type `double2` as

```
struct __device_builtin__ __builtin_align__(16) double2
{
    double x, y;
};
```

The `__align__(16)` causes the doubles in `double2` to be 16-byte or 128-bit aligned.

Using the `double2` type for the real and imaginary part of a complex number allows for coalesced memory access.

## exploring the effects of misaligned memory access

With a simple copy kernel we can explore what happens when access to global memory is misaligned:

```
__global__ void copyKernel
( float *output, float *input, int offset )
{
    int i = blockIdx.x*blockDim.x + threadIdx.x + offset;
    output[i] = input[i];
}
```

The bandwidth will decrease significantly for  $\text{offset} > 1$ .

# Memory Coalescing Techniques

## 1 Accessing Global and Shared Memory

- acceleration with graphics processing units
- memory coalescing to global memory
- **avoiding bank conflicts in shared memory**

## 2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

## 3 Avoiding Bank Conflicts

- data staging algorithms
- arrays of composite data
- computing consecutive powers

## shared memory and memory banks

Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e.: interleaved.

The bandwidth of shared memory is 32 bits per bank per clock cycle. Because shared memory is on chip, uncached shared memory latency is roughly  $100\times$  lower than global memory.

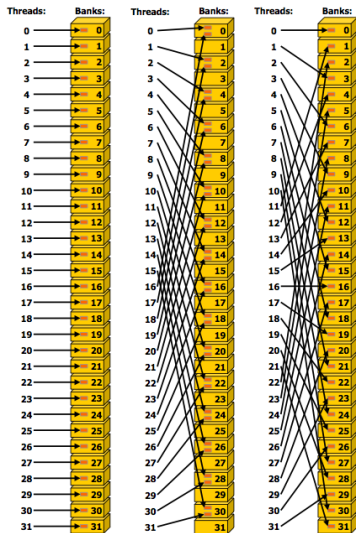
### Definition (bank conflict)

A *bank conflict* occurs if two or more threads access any bytes within *different* 32-bit words belonging to the *same* bank.

If two or more threads access any bytes within the same 32-bit word, then there is no bank conflict between these threads.

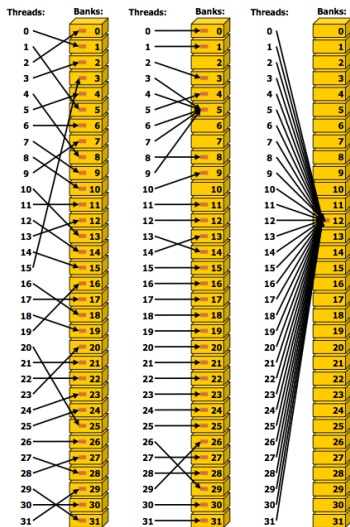
The CUDA C Best Practices Guide gives a medium priority recommendation to shared memory access without bank conflicts.

# examples of strided shared memory accesses



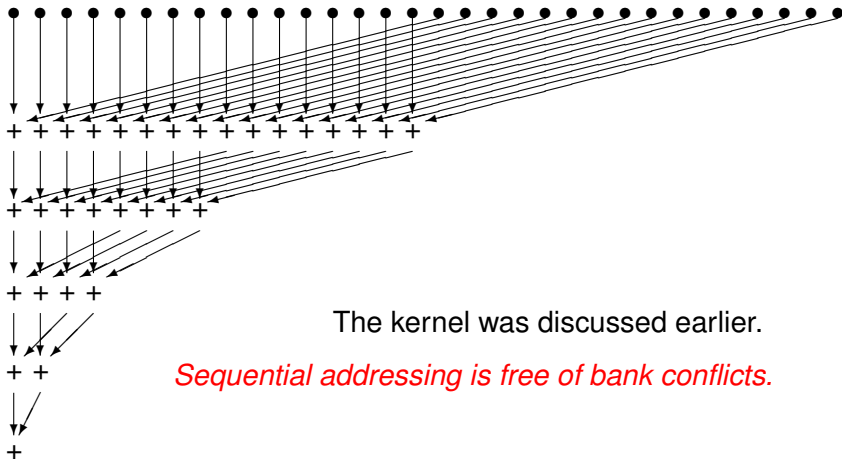
from Figure G-2 of the *NVIDIA Programming Guide*.

# irregular and colliding shared memory accesses



from Figure G-3 of the *NVIDIA Programming Guide*.

# computing all prefix sums



The kernel was discussed earlier.

*Sequential addressing is free of bank conflicts.*

Mark Harris. **Parallel Prefix Sum (Scan) with CUDA.**  
NVIDIA, April 2007.



# Memory Coalescing Techniques

- 1 Accessing Global and Shared Memory
  - acceleration with graphics processing units
  - memory coalescing to global memory
  - avoiding bank conflicts in shared memory

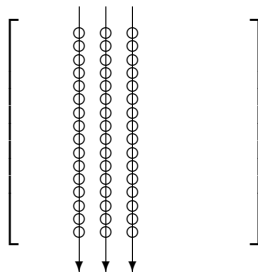
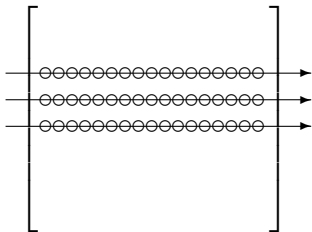
- 2 **Memory Coalescing Techniques**
  - **accessing global memory for a matrix**
  - using shared memory for coalescing

- 3 Avoiding Bank Conflicts
  - data staging algorithms
  - arrays of composite data
  - computing consecutive powers

# accessing the elements in a matrix

Consider two ways of accessing the elements in a matrix:

- 1 elements are accessed row after row; or
- 2 elements are accessed column after column.



# linear address system

Consider a 4-by-4 matrix:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

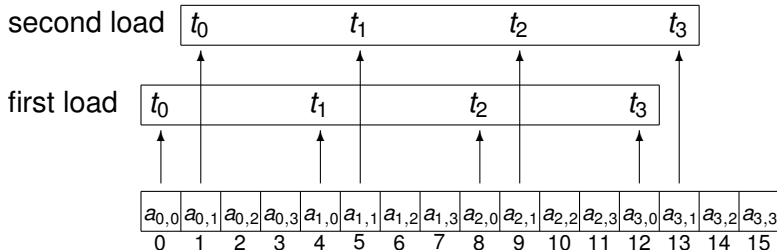
In C, the matrix is stored row wise as a one dimensional array.

# first access

Threads  $t_0$ ,  $t_1$ ,  $t_2$ , and  $t_3$  access the elements on the first two columns:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

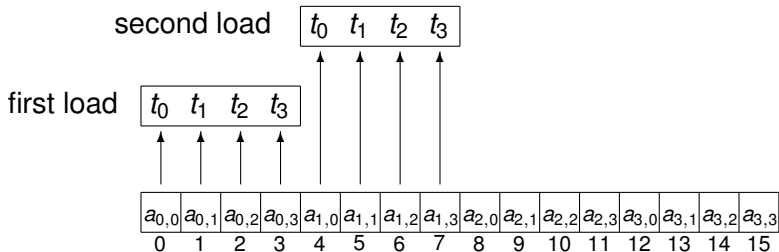
Two vertical arrows point down from the first and second columns of the table.



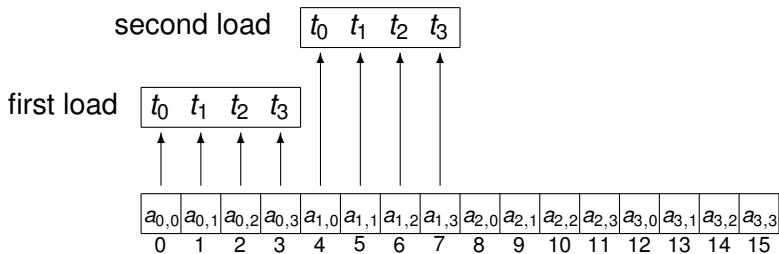
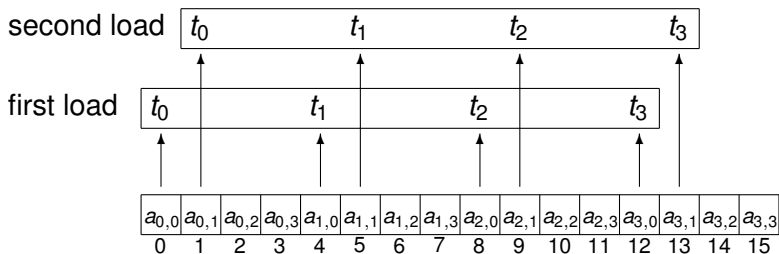
## second access

Four threads  $t_0$ ,  $t_1$ ,  $t_2$ , and  $t_3$  access elements on the first two rows:

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	→
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	→
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	



# uncoalesced versus coalesced access



# Memory Coalescing Techniques

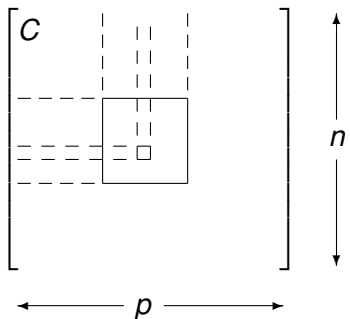
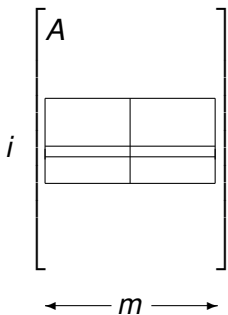
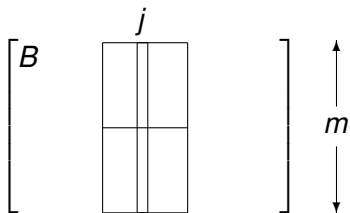
- 1 Accessing Global and Shared Memory
  - acceleration with graphics processing units
  - memory coalescing to global memory
  - avoiding bank conflicts in shared memory

- 2 **Memory Coalescing Techniques**
  - accessing global memory for a matrix
  - **using shared memory for coalescing**

- 3 Avoiding Bank Conflicts
  - data staging algorithms
  - arrays of composite data
  - computing consecutive powers

# tiling matrix-matrix multiplication

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$$





## tiling matrix multiplication with shared memory

For  $C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$ ,  $A \in \mathbb{R}^{n \times m}$ ,  $B \in \mathbb{R}^{m \times p}$ ,  $A_{i,k}$ ,  $B_{k,j}$ ,  $C_{i,j} \in \mathbb{R}^{w \times w}$ ,

every warp reads one tile  $A_{i,k}$  of  $A$  and one tile  $B_{k,j}$  of  $B$ : every thread in the warp reads one element of  $A_{i,k}$  and one element of  $B_{k,j}$ .

The number of threads equals  $w$ , the width of one tile, and threads are identified with  $tx = \text{threadIdx.x}$  and  $ty = \text{threadIdx.y}$ .

The  $by = \text{blockIdx.y}$  and  $bx = \text{blockIdx.x}$  correspond respectively to the first and the second index of each tile, so we have  $row = by * w + ty$  and  $col = bx * w + tx$ .

Row wise access to  $A$  uses  $A[row * m + (k * w + tx)]$ . For  $B$ :  
 $B[(k * w + ty) * m + col] = B[(k * w + ty) * m + bx * w + tx]$ .

Adjacent threads in a warp have adjacent  $tx$  values so we have coalesced access also to  $B$ .

## tiled matrix multiplication kernel

```
__global__ void mul ( float *A, float *B, float *C, int m )
{
    __shared__ float As[w][w];
    __shared__ float Bs[w][w];
    int bx = blockIdx.x;          int by = blockIdx.y;
    int tx = threadIdx.x;        int ty = threadIdx.y;
    int col = bx*w + tx;         int row = by*w + ty;
    float Cv = 0.0;
    for(int k=0; k<m/w; k++)
    {
        As[ty][tx] = A[row*m + (k*w + tx)];
        Bs[ty][tx] = B[(k*w + ty)*m + col];
        __syncthreads();
        for(int ell=0; ell<w; ell++)
            Cv += As[ty][ell]*Bs[ell][tx];
        C[row][col] = Cv;
    }
}
```

# Memory Coalescing Techniques

## 1 Accessing Global and Shared Memory

- acceleration with graphics processing units
- memory coalescing to global memory
- avoiding bank conflicts in shared memory

## 2 Memory Coalescing Techniques

- accessing global memory for a matrix
- using shared memory for coalescing

## 3 Avoiding Bank Conflicts

- **data staging algorithms**
- arrays of composite data
- computing consecutive powers

# data staging algorithms

All threads in the same warp execute the same instruction.

When retrieving/storing data from global memory, one instruction in a kernel defines the retrieval/storage of 32 data elements.

With memory coalescing, retrieving/storing 32 data elements requires as much time as retrieving/storing one data element.

## Definition (data staging algorithm)

A *data staging algorithm* arranges the data for memory coalescing.

Arranging data involves positioning the input and output data so that adjacent data elements are accessed by adjacent threads.

# Memory Coalescing Techniques

- 1 Accessing Global and Shared Memory
  - acceleration with graphics processing units
  - memory coalescing to global memory
  - avoiding bank conflicts in shared memory
- 2 Memory Coalescing Techniques
  - accessing global memory for a matrix
  - using shared memory for coalescing
- 3 Avoiding Bank Conflicts
  - data staging algorithms
  - **arrays of composite data**
  - computing consecutive powers

## arrays of composite data

Consider an array of complex numbers and/or multiple doubles.

The elements of such arrays are composite.

- Every complex number has a real and imaginary part.
- These parts can be one double, or a multiple double.
- A quad double consists of a most significant double, the second most, third most, fourth most significant double.

Using the straightforward representation will lead to bank conflicts.

Instead of an array of complex doubles, use two arrays:

- 1 one array with the real doubles,
- 2 another array with the imaginary doubles.

An array of complex quad doubles is stored in 8 arrays.

# Memory Coalescing Techniques

- 1 Accessing Global and Shared Memory
  - acceleration with graphics processing units
  - memory coalescing to global memory
  - avoiding bank conflicts in shared memory
- 2 Memory Coalescing Techniques
  - accessing global memory for a matrix
  - using shared memory for coalescing
- 3 Avoiding Bank Conflicts
  - data staging algorithms
  - arrays of composite data
  - **computing consecutive powers**

## consecutive powers

Consider the following problem:

Input :  $x_0, x_1, x_2, \dots, x_{31}$ , all of type float.

Output :  $x_0^2, x_0^3, x_0^4, \dots, x_0^{33}, x_1^2, x_1^3, x_1^4, \dots, x_1^{33}, x_2^2, x_2^3, x_2^4, \dots, x_2^{33},$   
 $\dots, x_{31}^2, x_{31}^3, x_{31}^4, \dots, x_{31}^{33}.$

This gives 32 threads in a warp 1,024 multiplications to do.

Assume the input and output resides in shared memory.

How to compute without bank conflicts?



## writing with stride

Observe the order of the output sequence:

Input :  $x_0, x_1, x_2, \dots, x_{31}$ , all of type float.

Output :  $x_0^2, x_0^3, x_0^4, \dots, x_0^{33}, x_1^2, x_1^3, x_1^4, \dots, x_1^{33}, x_2^2, x_2^3, x_2^4, \dots, x_2^{33},$   
 $\dots, x_{31}^2, x_{31}^3, x_{31}^4, \dots, x_{31}^{33}.$

If thread  $i$  computes  $x_i^2, x_i^3, x_i^4, \dots, x_i^{33}$ , then after the first step, all threads write  $x_0^2, x_1^2, x_2^2, \dots, x_{31}^2$  to shared memory.

If the stride is 32, all threads write into the same bank.

Instead of a simultaneous computation of 32 powers at once, the writing to shared memory will be serialized.

## changed order of storage

If we alter the order in the output sequence:

Input :  $x_0, x_1, x_2, \dots, x_{31}$ , all of type float.

Output :  $x_0^2, x_1^2, x_1^2, \dots, x_{31}^2, x_0^3, x_1^3, x_2^3, \dots, x_{31}^3, x_0^4, x_1^4, x_2^4, \dots, x_{31}^4,$   
 $\dots, x_0^{33}, x_1^{33}, x_2^{33}, \dots, x_{31}^{33}.$

After the first step, thread  $i$  writes  $x_i^2$  in adjacent memory, next to  $x_{i-1}^2$  (if  $i > 0$ ) and  $x_{i+1}^2$  (if  $i < 31$ ).

Without bank conflicts, the speedup will be close to 32.

## a basic Julia version

using CUDA

```
"""
```

```
function gpupwr32!(a, b)
```

raises the elements in the array a  
to the powers 2, 3, .., 33,  
writing the results in the array b.

```
"""
```

```
function gpupwr32!(a, b)
```

```
    i = threadIdx().x      # starts at 1
```

```
    idx = 1 + 32*(i-1)
```

```
    b[idx] = a[i]*a[i]
```

```
    idx = idx + 1
```

```
    for p=3:33
```

```
        b[idx] = a[i]*b[idx-1]
```

```
        idx = idx + 1
```

```
    end
```

```
    return nothing
```

```
end
```

## launching the kernel

```
dx = convert(Float32, 0.2/31)
x_h = [0.9f0 + (k-1)*dx for k=1:32]
y_h = [0.0f0 for k=1:32*32] # output
println("the input numbers : ", x_h)
x_d = CuArray(x_h)
y_d = CuArray(y_h)

# run with 32 threads

@cuda threads=32 gpupwr32!(x_d, y_d)
```

The complete code compares with the output of `cpupwr32!`.

## summary and exercises

Writing GPU accelerated code requires the application of *data staging algorithms*: the arrangement of data for memory coalescing.

We covered §6.2 in the book of Kirk & Hwu.

- 1 Run `copyKernel` for large enough arrays for zero `offset` and an `offset` equal to two. Measure the timings and deduce the differences in memory bandwidth between the two different values for `offset`.
- 2 Consider the kernel of `matrixMul` in the GPU computing SDK. Is the loading of the tiles into shared memory coalesced? Justify your answer.
- 3 Write a CUDA program for the computation of consecutive powers, using coalesced access of the values for the input elements. Compare the two orders of storing the output sequence in shared memory: once with and once without bank conflicts.