

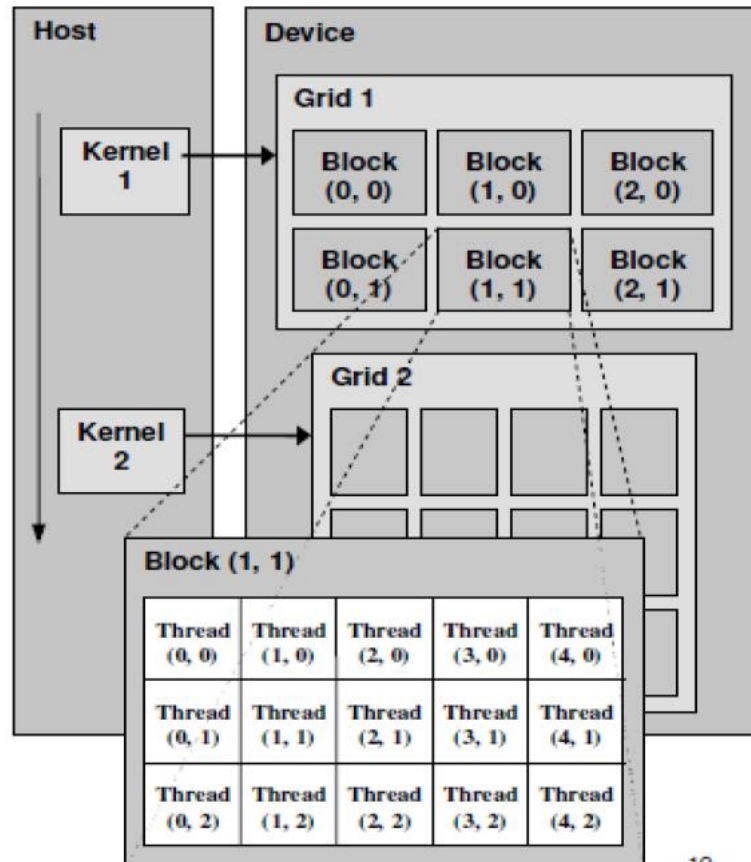
# Introduction to GPU Programming

# Anatomy of a GPU Application

- Host side
- Device side
- CUDA programming model

# CUDA Programming Model

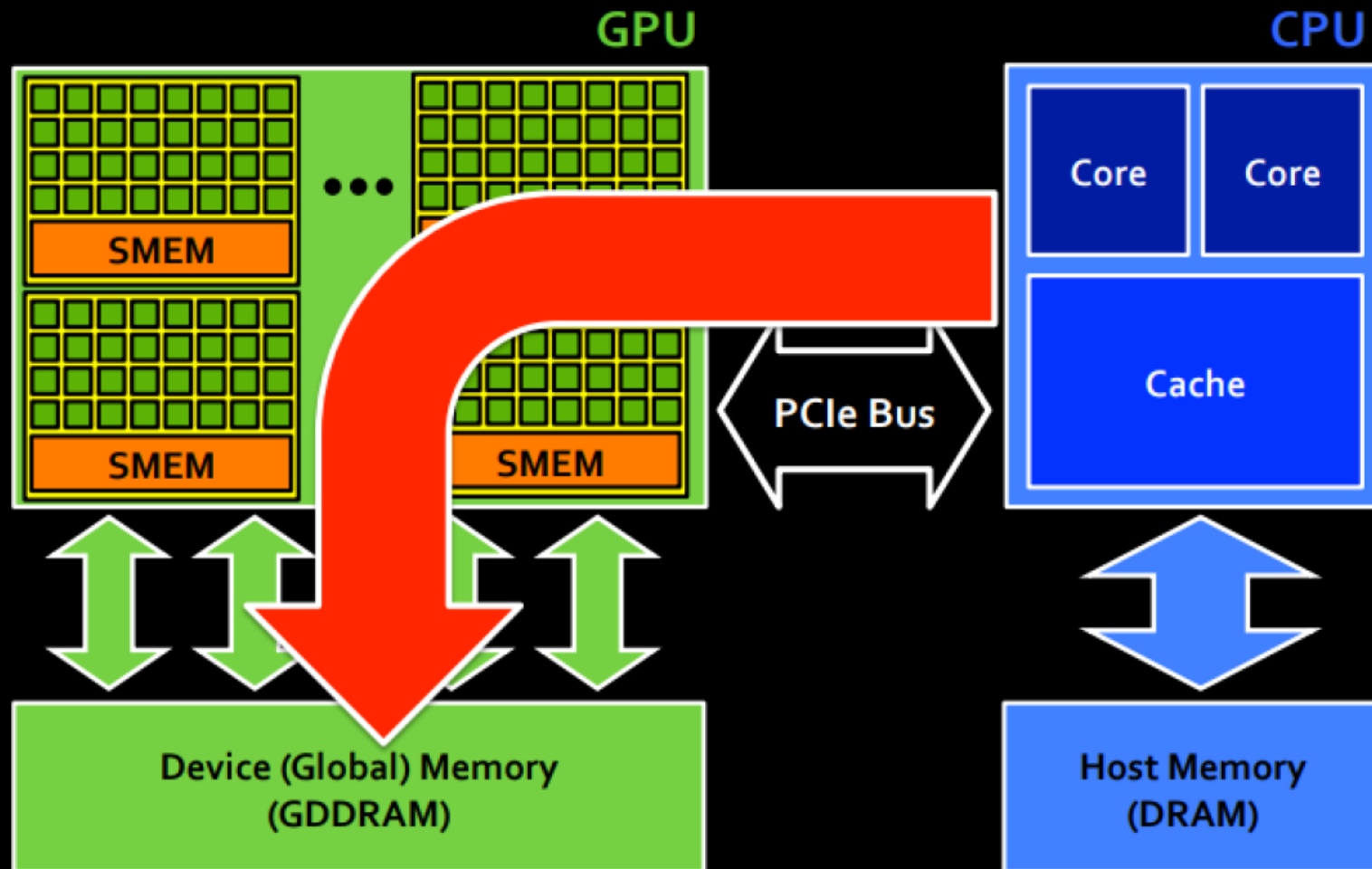
*Device = GPU*  
*Host = CPU*  
*Kernel =*  
*function that*  
*runs on the*  
*device*



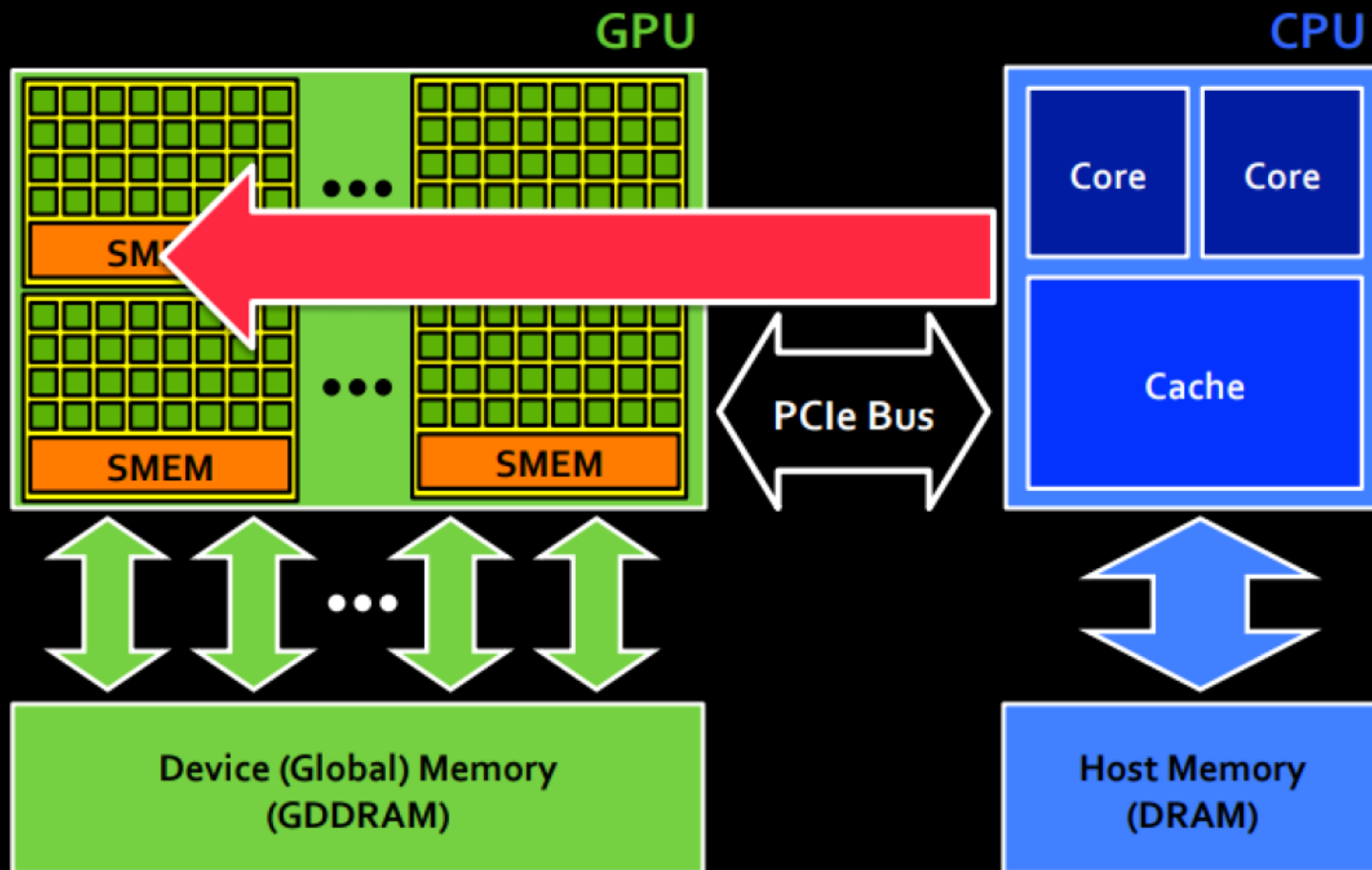
# Development: Basic Idea

1. Allocate equal size of memory for both host and device
2. Transfer data from host to device
3. Execute kernel to compute on data
4. Transfer data back to host

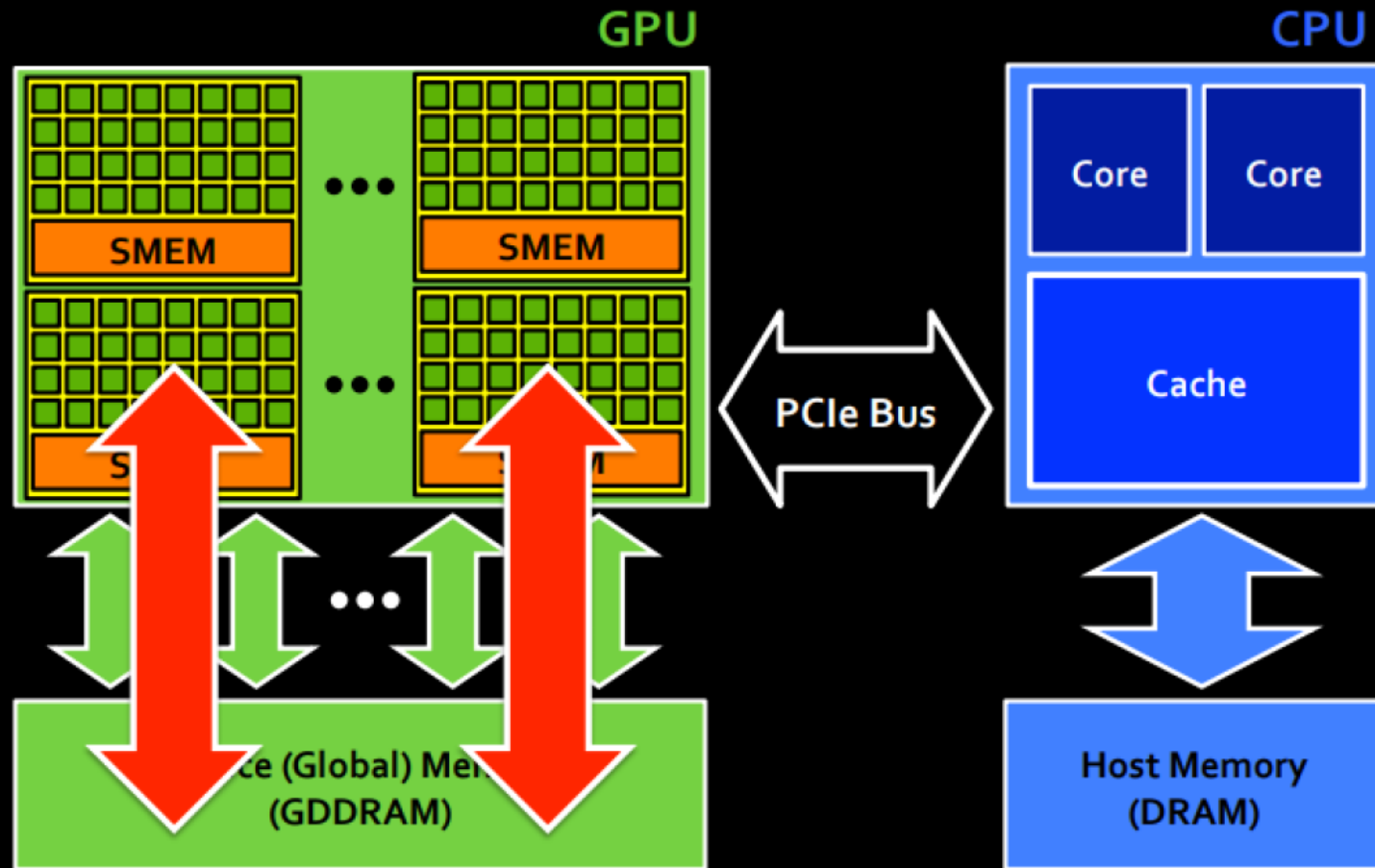
## Step 1 – copy data to GPU memory



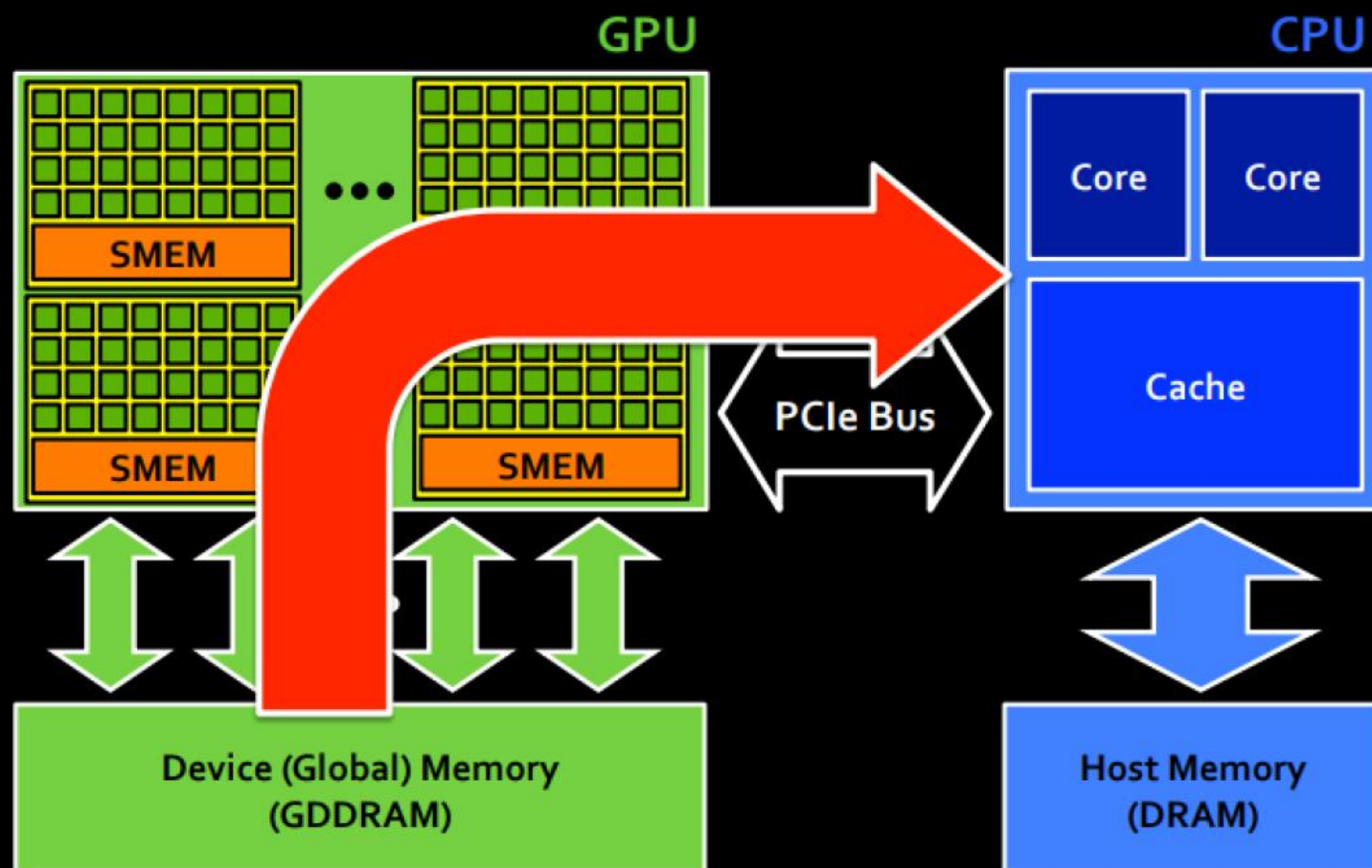
## Step 2 – launch kernel on GPU



## Step 3 – execute kernel on GPU



## Step 4 – copy data to CPU memory

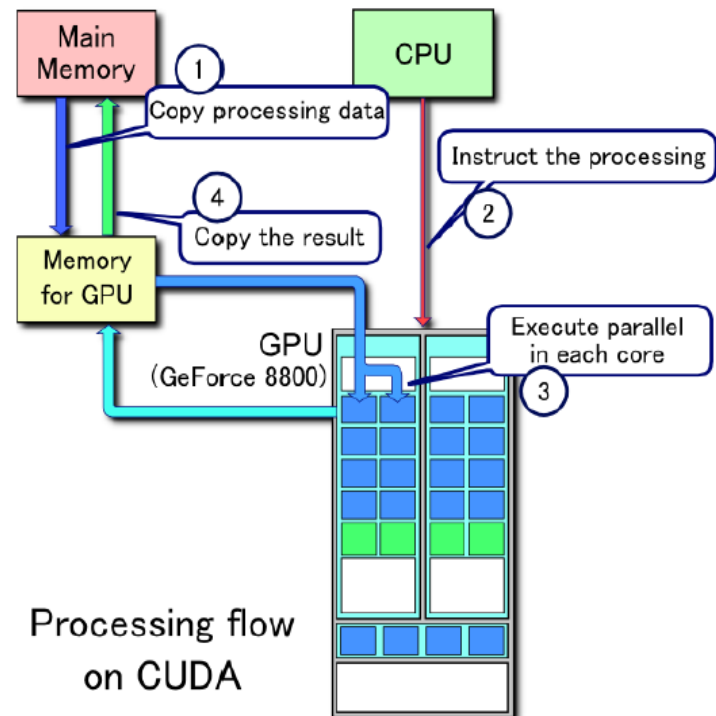




# Processing Flow

## Processing Flow of CUDA:

- Copy data from main mem to GPU mem.
- CPU instructs the process to GPU.
- GPU execute parallel in each core.
- Copy the result from GPU mem to main mem.



# CPU-Only Version

```
void vecAdd(int N, float* A, float* B, float* C) {  
    for (int i = 0; i < N; i++) C[i] = A[i] + B[i];  
}
```

Computational kernel

```
int main(int argc, char **argv)  
{  
    int N = 16384; // default vector size
```

```
    float *A = (float*)malloc(N * sizeof(float));  
    float *B = (float*)malloc(N * sizeof(float));  
    float *C = (float*)malloc(N * sizeof(float));
```

Memory allocation

```
    vecAdd(N, A, B, C); // call compute kernel
```

Kernel invocation

```
    free(A); free(B); free(C);
```

Memory de-allocation

```
}
```

# Adding GPU support

```
int main(int argc, char **argv)
{
    int N = 16384; // default vector size
```

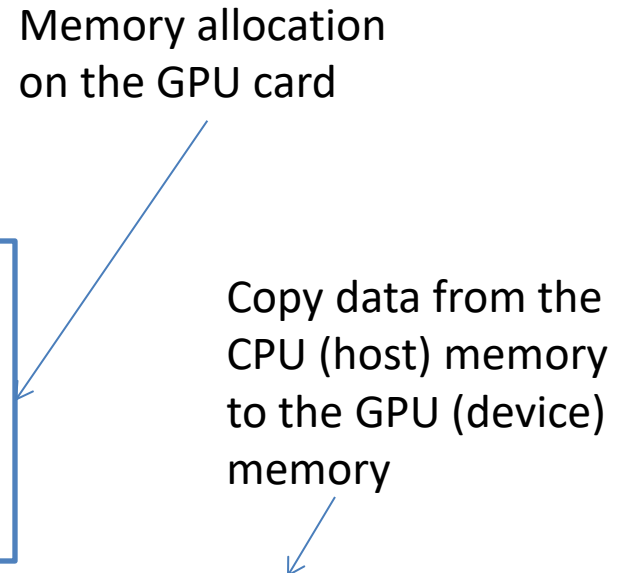
```
float *A = (float*)malloc(N * sizeof(float));
float *B = (float*)malloc(N * sizeof(float));
float *C = (float*)malloc(N * sizeof(float));
```

```
float *devPtrA, *devPtrB, *devPtrC;
```

```
cudaMalloc((void**)&devPtrA, N * sizeof(float));
cudaMalloc((void**)&devPtrB, N * sizeof(float));
cudaMalloc((void**)&devPtrC, N * sizeof(float));
```

```
cudaMemcpy(devPtrA, A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(devPtrB, B, N * sizeof(float), cudaMemcpyHostToDevice);
```

Memory allocation  
on the GPU card



Copy data from the  
CPU (host) memory  
to the GPU (device)  
memory

# Adding GPU support

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);
```

Kernel invocation

```
cudaMemcpy(C, devPtrC, N * sizeof(float), cudaMemcpyDeviceToHost);
```

```
cudaFree(devPtrA);  
cudaFree(devPtrB);  
cudaFree(devPtrC);
```

Copy results from  
device memory to  
the host memory

```
free(A); free(B); free(C);
```

Device memory  
de-allocation

```
}
```

# GPU Kernel

- **CPU version**

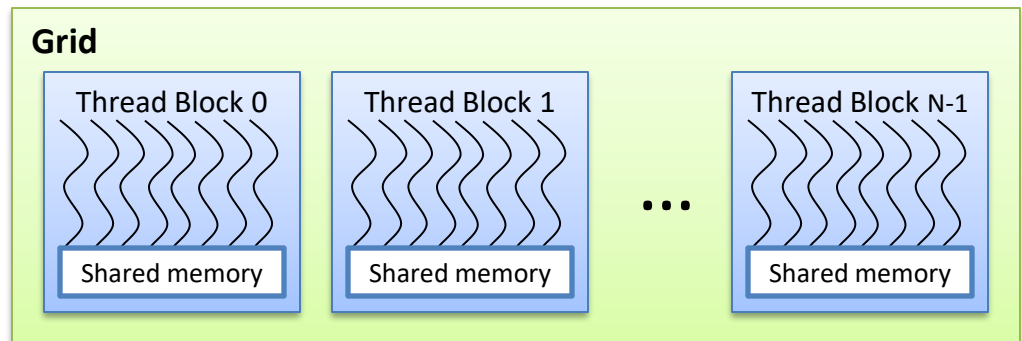
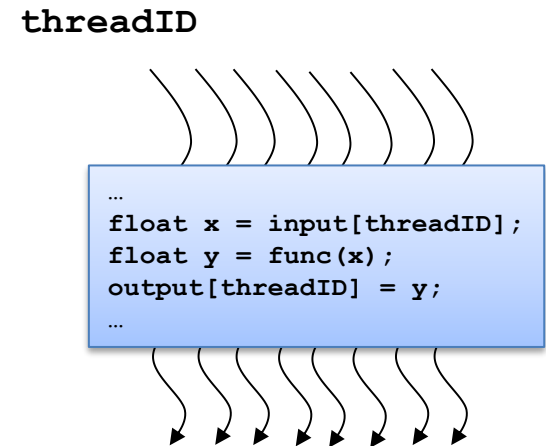
```
void vecAdd(int N, float* A, float* B, float* C)
{
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

- **GPU version**

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

# CUDA Programming Model

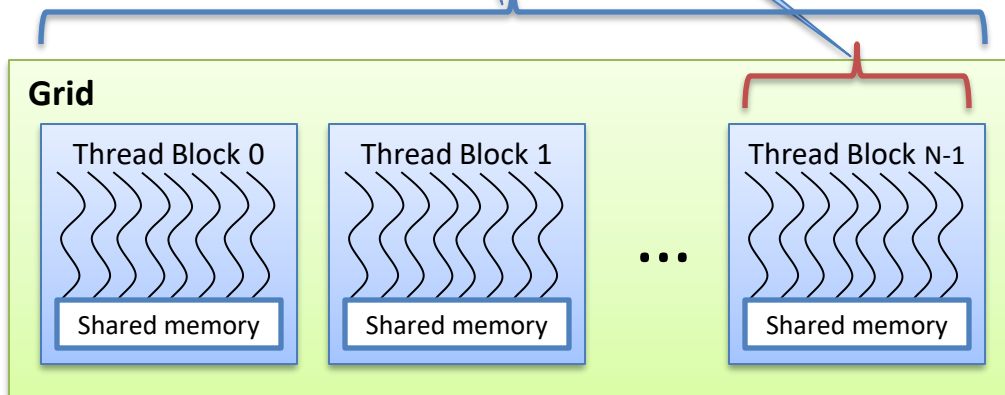
- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions
- Threads are arranged as a grid of thread blocks
  - Threads within a block have access to a segment of shared memory



# Kernel Invocation Syntax

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```

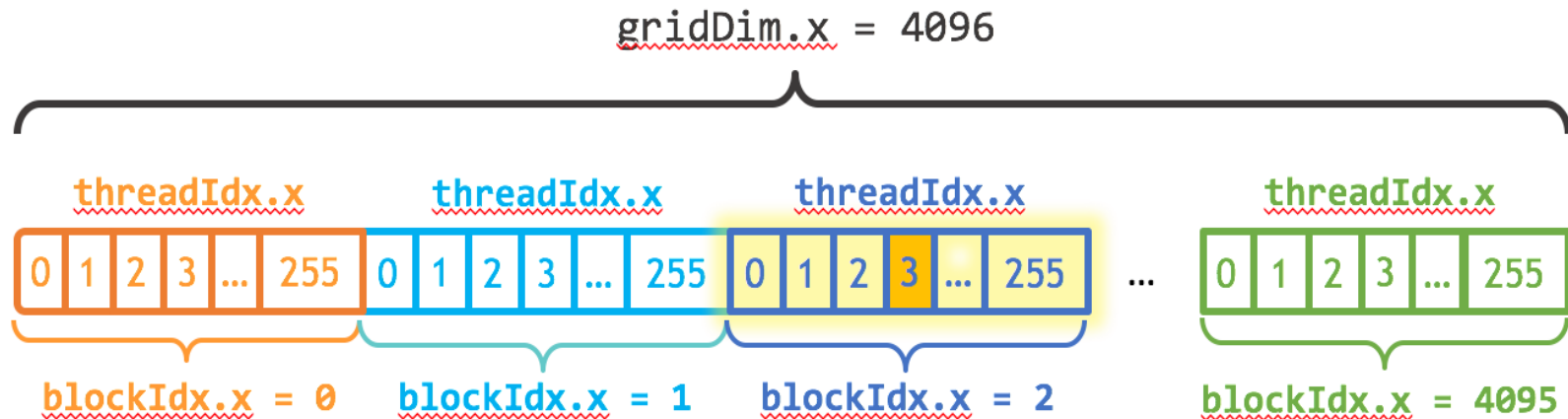


```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

block ID within a grid

number of threads per block

thread ID within a thread block



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

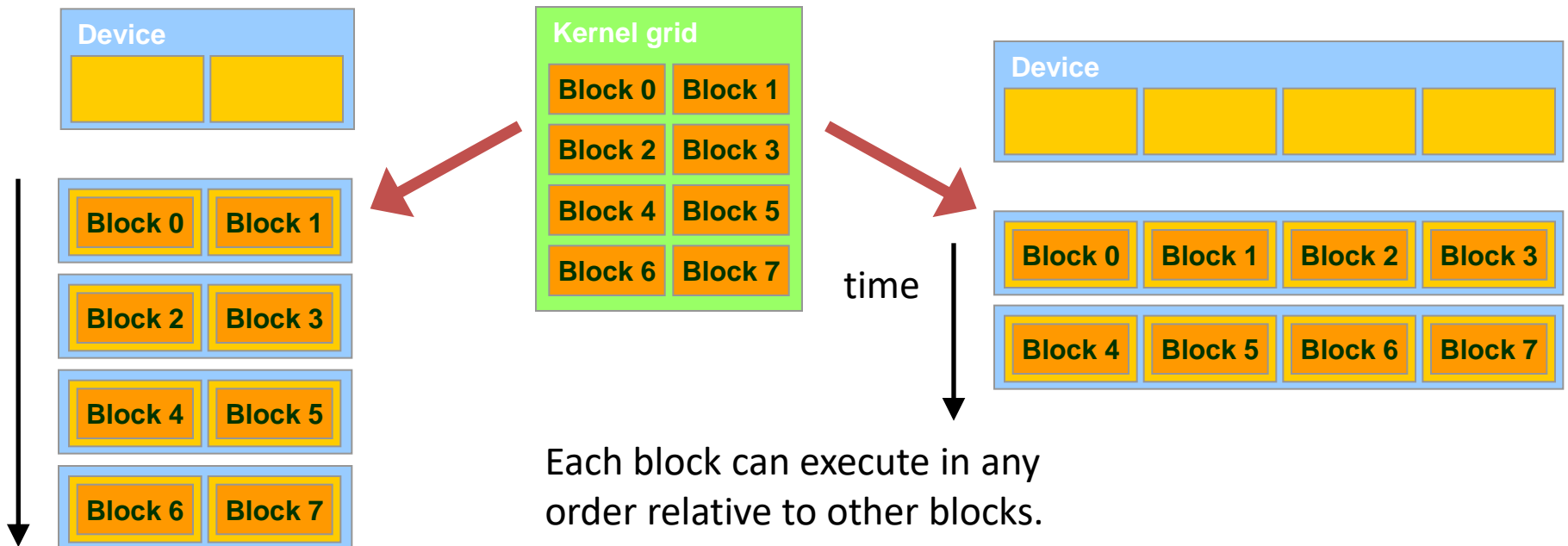
CUDA provides `gridDim.x`, which contains the number of blocks in the grid, `blockIdx.x`, which contains the index of the current thread block in the grid.

total number of threads in the grid (`blockDim.x * gridDim.x`).



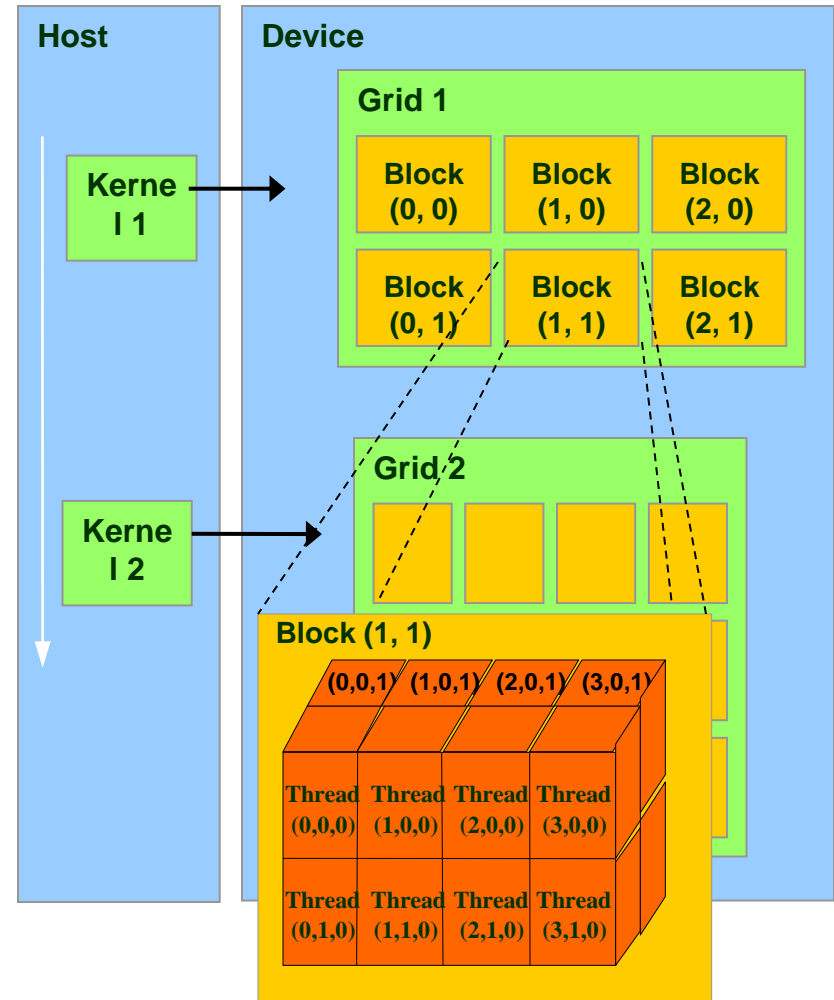
# Mapping Threads to the Hardware

- Blocks of threads are transparently assigned to SMs
  - A block of threads executes on one SM & does not migrate
  - Several blocks can reside concurrently on one SM



# CUDA Programming Model

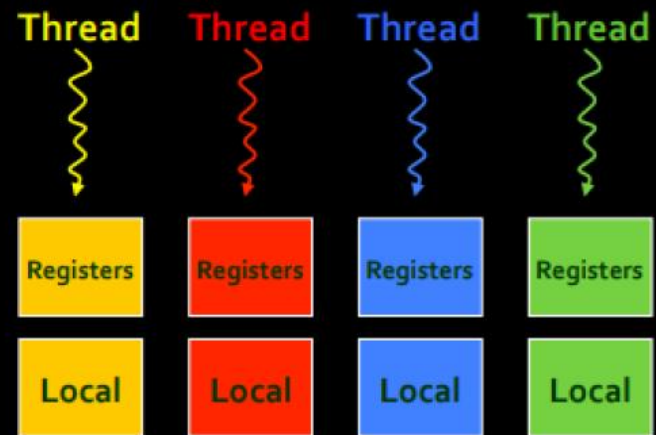
- A kernel is executed as a **grid of thread blocks**
  - Grid of blocks can be 1 or 2-dimensional
  - Thread blocks can be 1, 2, or 3-dimensional
- Different kernels can have different grid/block configuration
- Threads from the same block have access to a shared memory and their execution can be synchronized



# CUDA Memory Hierarchy



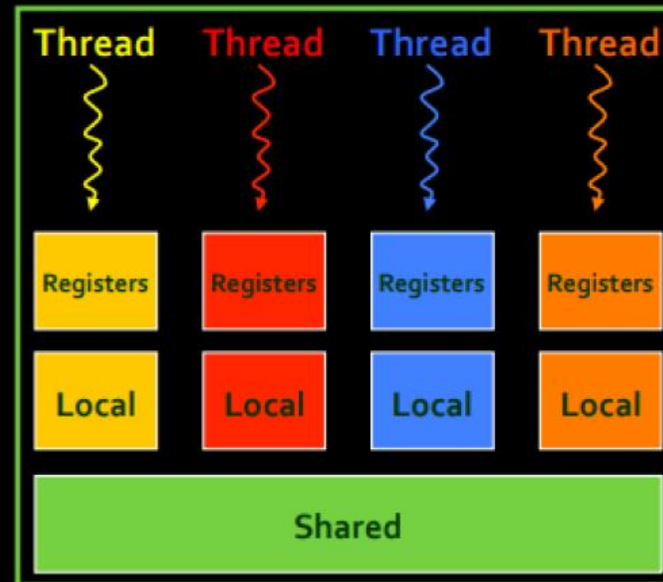
- Thread
  - Registers
  - Local memory





# CUDA Memory Hierarchy

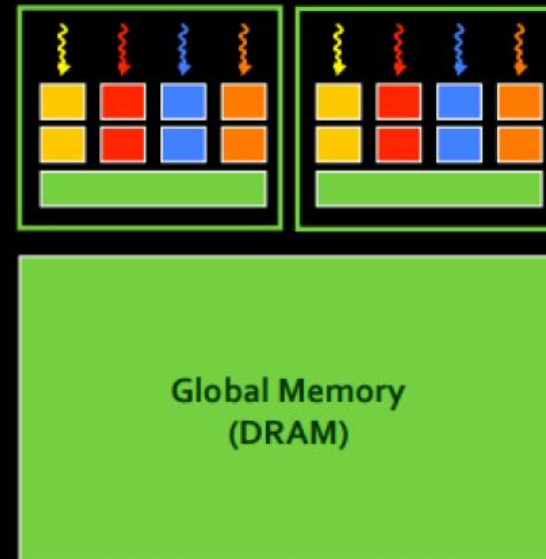
- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



# CUDA Memory Hierarchy



- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global Memory



# Programming model

- The key primitive in a CUDA program is called “launching a kernel”. Here a “**kernel**” is simply some C function that we would like to run on the GPU very many times.
- Computation is organized hierarchically in “**blocks**” of “**threads**”: we will have to specify the *number of blocks* and the number of *threads per block*.
- Here is a simple example of a complete CUDA program that does nothing — but in a massively parallel manner. We ask the GPU to launch 100 blocks of threads, each with 128 threads:

```
__device__ void foo(int i, int j) {}

__global__ void mykernel() {
    int i = blockIdx.x;
    int j = threadIdx.x;
    foo(i, j);
}

int main() {
    mykernel<<<100, 128>>>();
    cudaDeviceSynchronize();
}
```

What will happen is that the GPU will run the following 12800 function calls in some order, most likely in a parallel fashion:

$$\begin{aligned} &f(0, 0), f(0, 1), f(0, 2), \dots, f(0, 127), \\ &f(1, 0), f(1, 1), f(1, 2), \dots, f(1, 127), \\ &\qquad\qquad\qquad \dots, \\ &f(99, 0), f(99, 1), f(99, 2), \dots, f(99, 127). \end{aligned}$$

# Multidimensional grids and blocks

- Often it is convenient to index blocks and threads with 2-dimensional or 3-dimensional indexes. Here is an example in which we create  $20 \times 5$  blocks, each with  $16 \times 8$  threads:

```
__device__ void foo(int iy, int ix, int jy, int jx) {}

__global__ void mykernel() {
    int ix = blockIdx.x;
    int iy = blockIdx.y;
    int jx = threadIdx.x;
    int jy = threadIdx.y;
    foo(iy, ix, jy, jx);
}

int main() {
    dim3 dimGrid(20, 5);
    dim3 dimBlock(16, 8);
    mykernel<<<dimGrid, dimBlock>>>();
    cudaDeviceSynchronize();
}
```



The end result is similar to the parallelized version of the following loop:

```
for (int iy = 0; iy < 5; ++iy) {  
    for (int ix = 0; ix < 20; ++ix) {  
        for (int jy = 0; jy < 8; ++jy) {  
            for (int jx = 0; jx < 16; ++jx) {  
                foo(iy, ix, jy, jx);  
            }  
        }  
    }  
}
```

Internally, the GPU will divide each block in smaller units of work, called “*warps*”.

Each warp consists of 32 *threads*.

This is one of the reasons why it makes sense to pick a nice round number such as 64, 128, or 256 threads per block.

Very small values make our code inefficient.

Very large values do not work at all: the GPU will simply refuse to launch the kernel if we try to ask for more than 1024 threads per block.

# Porting matrix multiplier to GPU

```

int main(int argc, char* argv[])
{
    int N = 1024;

    struct timeval t1, t2, ta, tb;
    long msec1, msec2;
    float flop, mflop, gflop;

    float *a = (float *)malloc(N*N*sizeof(float));
    float *b = (float *)malloc(N*N*sizeof(float));
    float *c = (float *)malloc(N*N*sizeof(float));

    minit(a, b, c, N);

    gettimeofday(&t1, NULL);
    mmult(a, b, c, N); // a = b * c
    gettimeofday(&t2, NULL);

    mprint(a, N, 5);

    free(a);
    free(b);
    free(c);

    msec1 = t1.tv_sec * 1000000 + t1.tv_usec;
    msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
    msec2 -= msec1;
    flop = N*N*N*2.0f;
    mflop = flop / msec2;
    gflop = mflop / 1000.0f;
    printf("msec = %10ld  GFLOPS = %.3f\n", msec2, gflop);
}

```

```

// a = b * c
void mmult(float *a, float *b, float *c, int N)
{
    for (int j = 0; j < N; j++)
        for (int k = 0; k < N; k++)
            for (int i = 0; i < N; i++)
                a[i+j*N] += b[i+k*N]*c[k+j*N];
}

void minit(float *a, float *b, float *c, int N)
{
    for (int j = 0; j < N; j++)
        for (int i = 0; i < N; i++) {
            a[i+N*j] = 0.0f;
            b[i+N*j] = 1.0f;
            c[i+N*j] = 1.0f;
        }
}

void mprint(float *a, int N, int M)
{
    int i, j;

    for (int j = 0; j < M; j++)
    {
        for (int i = 0; i < M; i++)
            printf("%.2f ", a[i+N*j]);
        printf("...\n");
    }
    printf("...\n");
}

```

# Matrix Representation in Memory

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    for (k = 0; k < n; ++k)
      a[i+n*j] += b[i+n*k] * c[k+n*j];
```

- Matrices are stored in column-major order

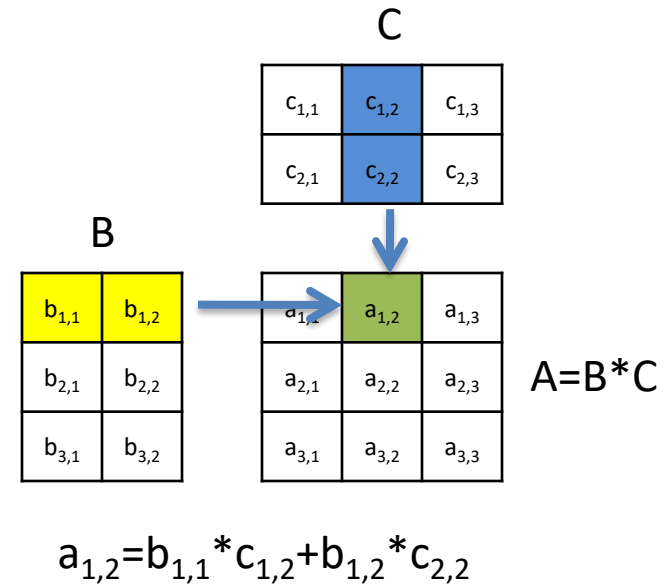


- For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

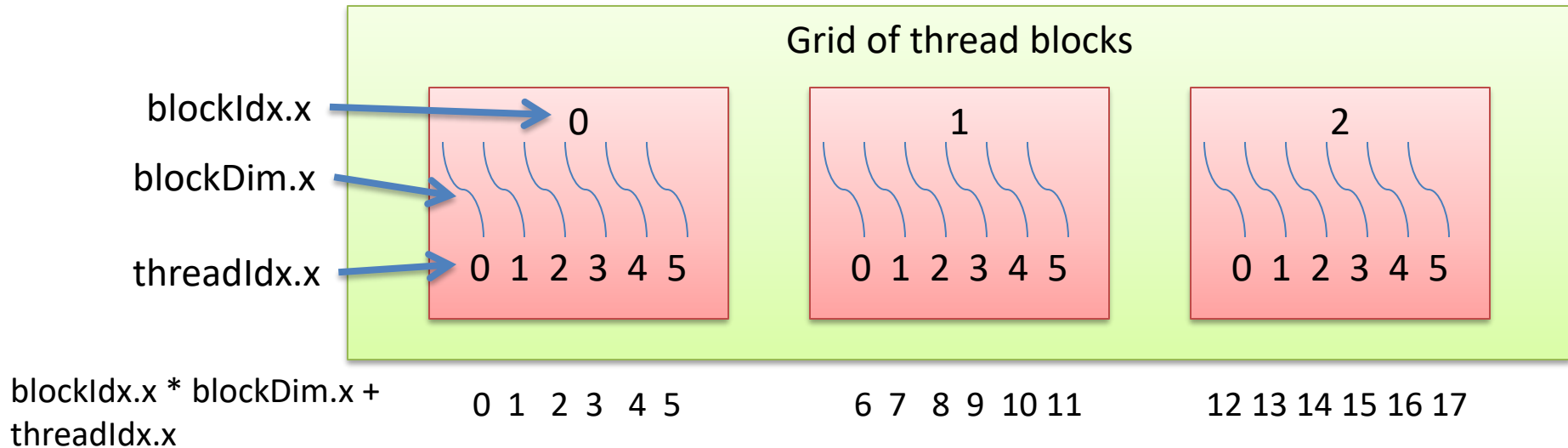
Map this code:

```

for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    for (k = 0; k < n; ++k)
      a[i+n*j] += b[i+n*k] * c[k+n*j];
  
```

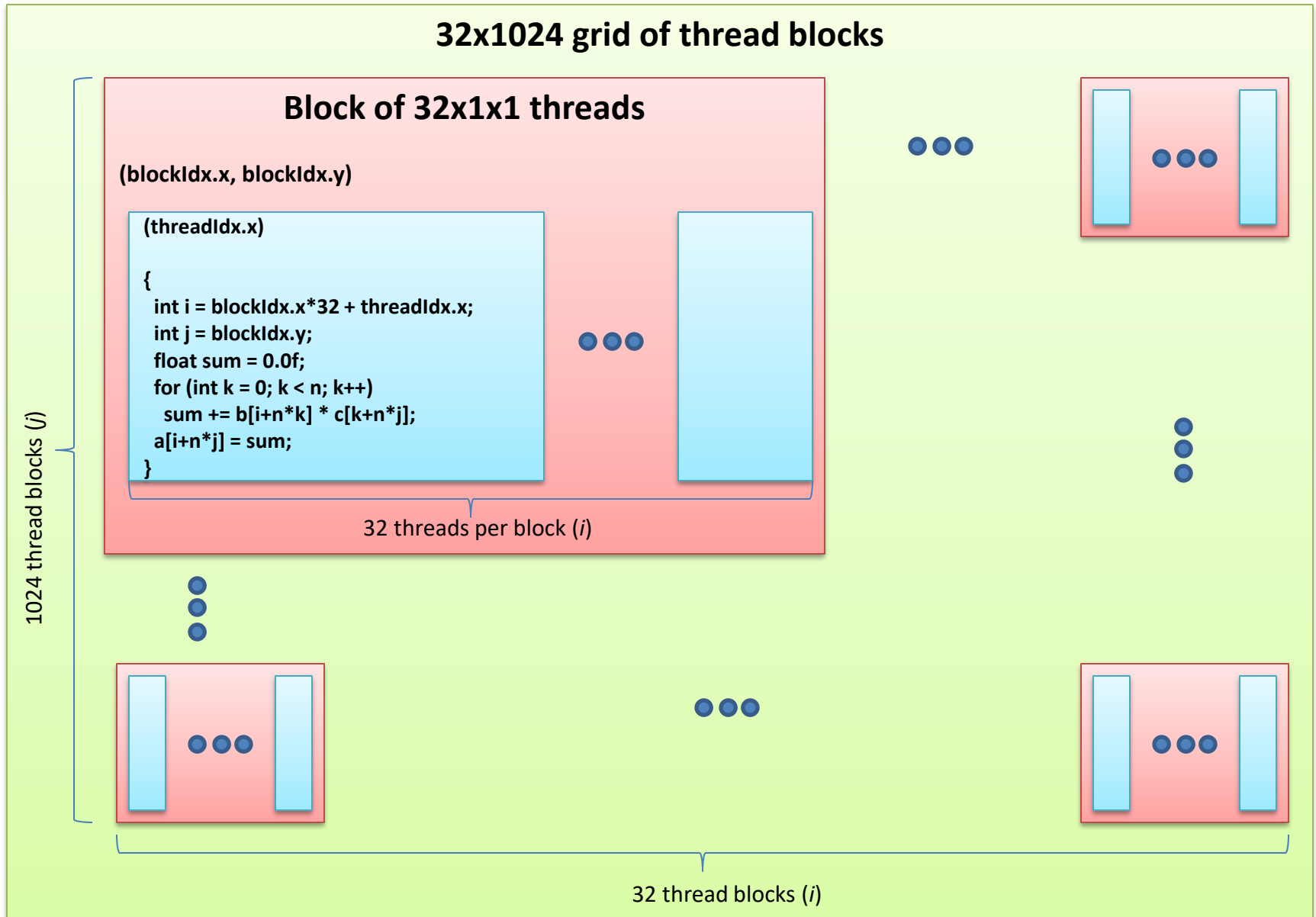


into this (logical) architecture:



```
dim3 grid(1024/32, 1024);
```

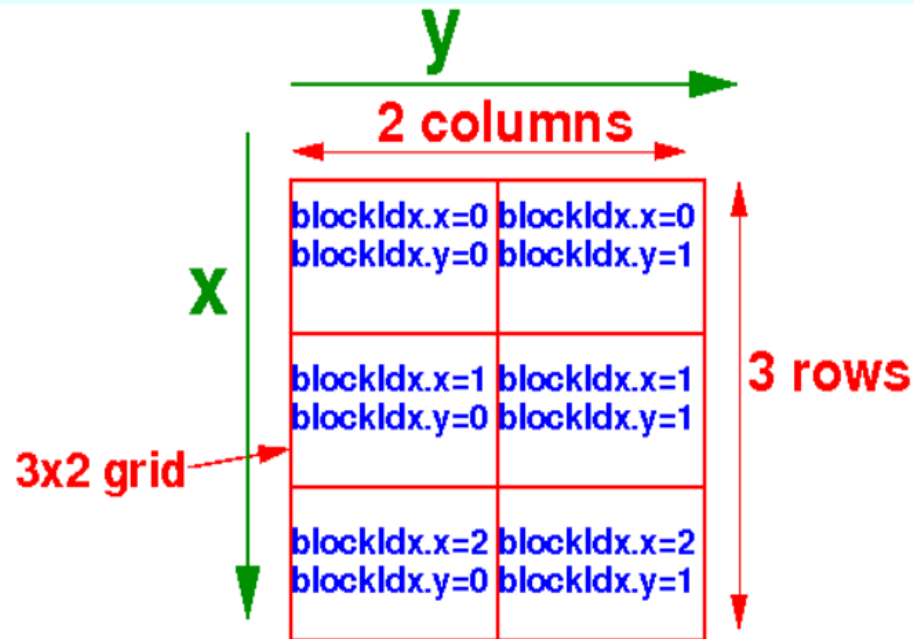
```
dim3 threads (32);
```



## A 2-dimensional grid shape

The values of the identifying variables of each thread block in the 3×2 grid shape:

- `dim3 gridShape = dim3( 3, 2 );`

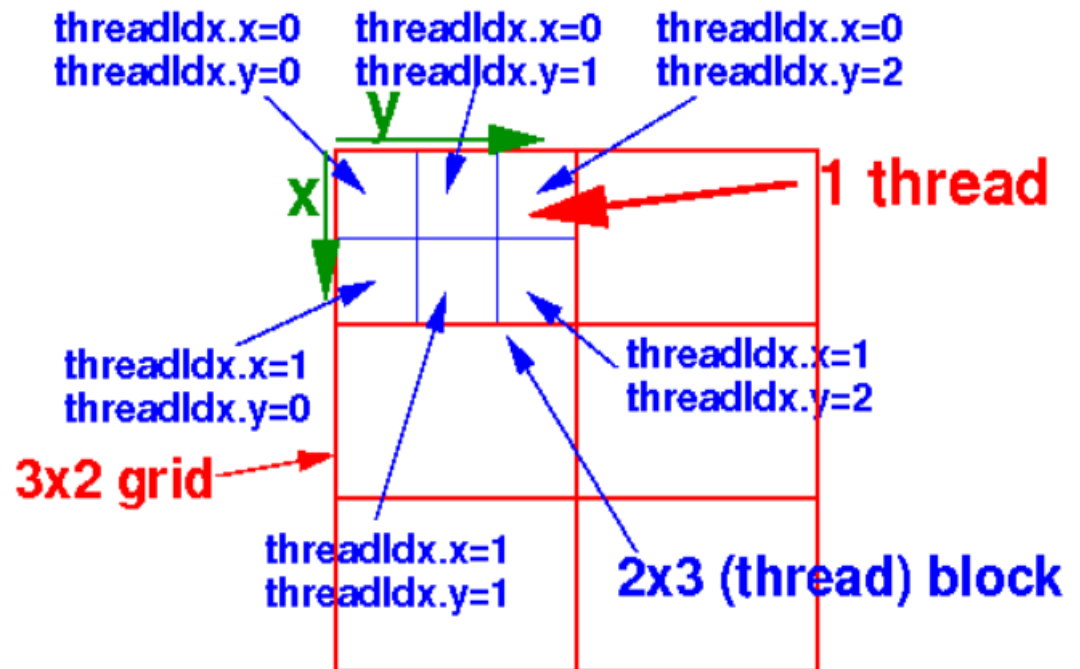




## A 2-dimensional (thread) block shape

The **values** of the **identifying variables** of each **thread** in the **2×3 thread block shape**:

- `dim3 blockShape = dim3( 2, 3 );`



# Kernel

## Original CPU kernel

```
void mmult(float *a, float *b, float *c, int N)
{
for (int i = 0; i < N; i++)
for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
        a[i+j*N] += b[i+k*N]*c[k+j*N];
}
```

## GPU Kernel

```
__global__
void mmult(float *a, float *b, float *c, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y;
    float sum = 0.0;

    for (int k = 0; k < N; k++)
        sum += b[i+N*k] * c[k+N*j];

    a[i+N*j] = sum;
}
```

```
dim3 dimGrid(32, 1024);
dim3 dimBlock(32);
mmult<<<dimGrid, dimBlock>>>(devPtrA, devPtrB, devPtrC, N);
```

```
int main(int argc, char* argv[])  
{
```

```
    int N = 1024;
```

```
    struct timeval t1, t2;
```

```
    long msec1, msec2;
```

```
    float flop, mflop, gflop;
```

```
    float *a = (float *)malloc(N*N*sizeof(float));
```

```
    float *b = (float *)malloc(N*N*sizeof(float));
```

```
    float *c = (float *)malloc(N*N*sizeof(float));
```

```
    minit(a, b, c, N);
```

```
    // allocate device memory
```

```
    float *devPtrA, *devPtrB, *devPtrC;
```

```
    cudaMalloc((void**)&devPtrA, N*N*sizeof(float));
```

```
    cudaMalloc((void**)&devPtrB, N*N*sizeof(float));
```

```
    cudaMalloc((void**)&devPtrC, N*N*sizeof(float));
```

```
    // copy input arrays to the device memory
```

```
    cudaMemcpy(devPtrB, b, N*N*sizeof(float), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(devPtrC, c, N*N*sizeof(float), cudaMemcpyHostToDevice);
```

```
gettimeofday(&t1, NULL);  
msec1 = t1.tv_sec * 1000000 + t1.tv_usec;
```

```
// define grid and thread block sizes  
dim3 dimGrid(32, 1024);  
dim3 dimBlock(32);
```

```
// launch GPU kernel  
mmult<<<dimGrid, dimBlock>>>(devPtrA, devPtrB, devPtrC, N);
```

```
// check for errors  
cudaError_t err = cudaGetLastError();  
if (cudaSuccess != err)  
{  
    fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err) );  
    exit(EXIT_FAILURE);  
}
```

```
// wait until GPU kernel is done  
cudaThreadSynchronize();
```

```
gettimeofday(&t2, NULL);  
msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
```

**// copy results to host**

**cudaMemcpy(a, devPtrA, N\*N\*sizeof(float), cudaMemcpyDeviceToHost);**

**mprint(a, N, 5);**

**// free device memory**

**cudaFree(devPtrA);**

**cudaFree(devPtrB);**

**cudaFree(devPtrC);**

**free(a);**

**free(b);**

**free(c);**

**msec2 -= msec1;**

**flop = N\*N\*N\*2.0f;**

**mflop = flop / msec2;**

**gflop = mflop / 1000.0f;**

**printf("msec = %10ld GFLOPS = %.3f\n", msec2, gflop);**

**}**

# More on CUDA Programming

- Language extensions
  - Function type qualifiers
  - Variable type qualifiers
  - Execution configuration
  - Built-in variables
- Common runtime components
  - Built-in vector types
- Device runtime components
  - Intrinsic functions
  - Synchronization and memory fencing functions
  - Atomic functions
- Host runtime components (runtime API only)
  - Device management
  - Memory management
  - Error handling
  - Debugging in the device emulation mode
- Exercise

# Function Type Qualifiers:

Function type qualifiers to specify whether a function executes on the host or on the device

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

`__device__` and `__global__` functions do not support recursion, cannot declare static variables inside their body, cannot have a variable number of arguments

`__device__` functions cannot have their address taken

`__host__` and `__device__` qualifiers can be used together, in which case the function is compiled for both

`__global__` and `__host__` qualifiers cannot be used together

`__global__` function must have void return type, its execution configuration must be specified, and the call is asynchronous

# Variable Type Qualifiers:

Variable type qualifiers to specify the memory location on the device

	Memory	Scope	Lifetime
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	Application
<code>volatile int GlobalVar or SharedVar;</code>			

`__shared__` and `__constant__` variables have implied static storage

`__device__`, `__shared__` and `__constant__` variables cannot be defined using `external` keyword

`__device__` and `__constant__` variables are only allowed at file scope

`__constant__` variables cannot be assigned to from the devices, they are initialized from the host only

`__shared__` variables cannot have an initialization as part of their declaration



# Execution Configuration

Function declared as

```
__global__ void kernel(float* param);
```

must be called like this:

```
kernel<<<Dg, Db, Ns, S>>>(param);
```

where

- **Dg** (type dim3) specifies the dimension and size of the grid, such that  $Dg.x * Dg.y$  equals the number of blocks being launched;
- **Db** (type dim3) specifies the dimension and size of each block of threads, such that  $Db.x * Db.y * Db.z$  equals the number of threads per block;
- optional **Ns** (type size\_t) specifies the number of bytes of shared memory dynamically allocated per block for this call in addition to the statically allocated memory
- optional **S** (type cudaStream\_t) specifies the stream associated with this kernel call

# Built-in Variables

variable	Data type	description
<b>gridDim</b>	dim3	dimensions of the grid
<b>blockID</b>	uint3	block index within the grid
<b>blockDim</b>	dim3	dimensions of the block
<b>threadIdx</b>	uint3	thread index within the block
<b>warpSize</b>	int	warp size in threads

It is not allowed to take addresses of any of the built-in variables

It is not allowed to assign values to any of the built-in variables

# Built-in Vector Types

Vector types derived from basic integer and float types

- char1, char2, char3, char4
- uchar1, uchar2, uchar3, uchar4
- short1, short2, short3, short4
- ushort1, ushort2, ushort3, ushort4
- int1, int2, int3, int4
- uint1, uint2, uint3 (**dim3**), uint4
- long1, long2, long3, long4
- ulong1, ulong2, ulong3, ulong4
- longlong1, longlong2
- float1, float2, float3, float4
- double1, double2

They are all structures, like this:

```
typedef struct {  
    float x,y,z,w;  
} float4;
```

They all come with a constructor function in the form **make\_<type name>**, e.g.,

```
int2 make_int2(int x, int y);
```

# Intrinsic Functions

Supported on the device only

Start with `__`, as in `__sinf(x)`

End with

`_rn` (round-to-nearest-even rounding mode)

`_rz` (round-towards-zero rounding mode)

`_ru` (round-up rounding mode)

`_rd` (round-down rounding mode)

as in `__fadd_rn(x,y);`

There are mathematical (`__log10f(x)`), type conversion (`__int2float_rn(x)`), type casting (`__int_as_float(x)`), and bit manipulation (`__ffs(x)`) functions

# Memory Fencing Functions

- Memory fence functions can be used to enforce some ordering on memory accesses.

# Synchronization and Memory Fencing Functions

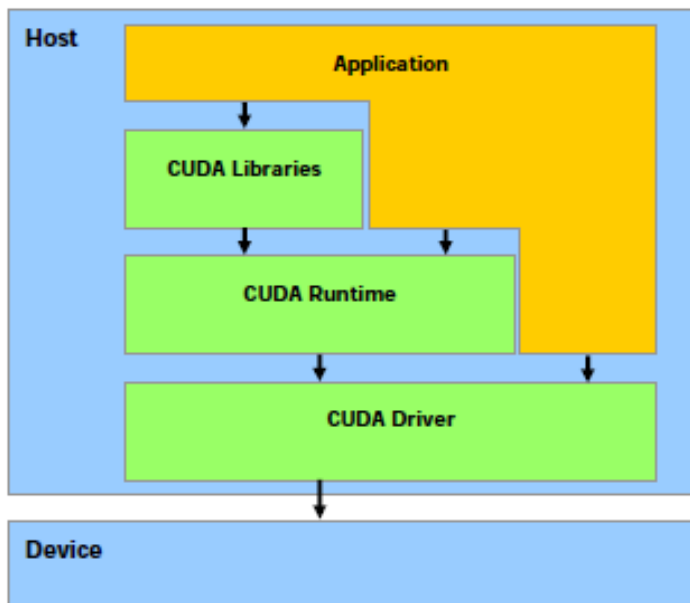
function	description
<code>void __threadfence()</code>	wait until all global and shared memory accesses made by the calling thread become visible to all threads in the <b>device</b> for global memory accesses and all threads in the thread block for shared memory accesses
<code>void __threadfence_block()</code>	Waits until all global and shared memory accesses made by the calling thread become visible to all threads in the <b>thread block</b>
<code>void __syncthreads()</code>	Waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads become visible to all threads in the block

# CUDA APIs

- higher-level API called the **CUDA runtime API**
  - `myKernel(unsigned char*)devPtr, width, <<<Dg, Db>>>(( height, pitch);`

- low-level API called the **CUDA driver API**

- `cuModuleLoad( &module, binfile );`
- `cuModuleGetFunction( &func, module, "mmkernel" );`
- ...
- `cuParamSetv( func, 0, &args, 48 );`
- `cuParamSetSize( func, 48 );`
- `cuFuncSetBlockShape( func, ts[0], ts[1], 1 );`
- `cuLaunchGrid( func, gs[0], gs[1] );`



# Device Management

function	description
<code>cudaGetDeviceCount()</code>	Returns the number of compute-capable devices
<code>cudaGetDeviceProperties()</code>	Returns information on the compute device
<code>cudaSetDevice()</code>	Sets device to be used for GPU execution
<code>cudaGetDevice()</code>	Returns the device currently being used
<code>cudaChooseDevice()</code>	Selects device that best matches given criteria



# Device Management Example

```
void cudaDeviceInit() {  
    int devCount, device;  
    cudaGetDeviceCount(&devCount);  
    if (devCount == 0) {  
        printf("No CUDA capable devices detected.\n");  
        exit(EXIT_FAILURE);  
    }  
    for (device=0; device < devCount; device++) {  
        cudaDeviceProp props;  
        cudaGetDeviceProperties(&props, device);  
        // If a device of compute capability >= 1.3 is found, use it  
        if (props.major > 1 || (props.major == 1 && props.minor >= 3)) break;  
    }  
    if (device == devCount) {  
        printf("No device above 1.2 compute capability detected.\n");  
        exit(EXIT_FAILURE);  
    }  
    else cudaSetDevice(device);  
}
```

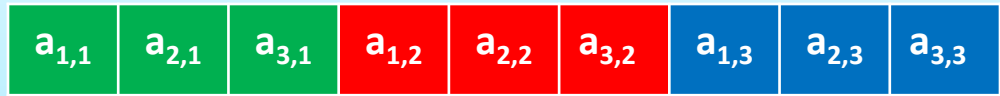
# Memory Management

function	description
<code>cudaMalloc()</code>	Allocates memory on the GPU
<code>cudaMallocPitch()</code>	Allocates memory on the GPU device for 2D arrays, may pad the allocated memory to ensure alignment requirements
<code>cudaFree()</code>	Frees the memory allocated on the GPU
<code>cudaMallocArray()</code>	Allocates an array on the GPU
<code>cudaFreeArray()</code>	Frees an array allocated on the GPU
<code>cudaMallocHost()</code>	Allocates page-locked memory on the host
<code>cudaFreeHost()</code>	Frees page-locked memory in the host

# More on Memory Alignment

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

```
cudaMalloc(&dev_a, m*n*sizeof(float));
```



Matrix columns are not aligned at 64-bit boundary

```
cudaMallocPitch(&dev_a, &n, n*sizeof(float), m);
```



Matrix columns are aligned at 64-bit boundary

$n$  is the allocated (aligned) size for the first dimension (the *pitch*), given the requested sizes of the two dimensions.

# Memory Management Example

```
cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);
```

```
myKernel<<<100, 192>>>(devPtr, pitch);
```

```
// device code
```

```
__global__ void myKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

# Memory Management

function	description
<code>cudaMemset()</code>	Initializes or sets GPU memory to a value
<code>cudaMemCpy()</code>	Copies data between host and the device
<code>cudaMemcpyToArray()</code>	
<code>cudaMemcpyFromArray()</code>	
<code>cudaMemcpyArrayToArray()</code>	
<code>cudaMemcpyToSymbol()</code>	
<code>cudaMemcpyFromSymbol()</code>	
<code>cudaGetSymbolAddress()</code>	Finds the address associated with a CUDA symbol
<code>cudaGetSymbolSize()</code>	Finds the size of the object associated with a CUDA symbol

# Error Handling

All CUDA runtime API functions return an error code. The runtime maintains an error variable for each host thread that is overwritten by the error code every time an error concurs.

function	description
<code>cudaGetLastError()</code>	Returns error variable and resets it to <code>cudaSuccess</code>
<code>cudaGetErrorString()</code>	Returns the message string from an error code

```
cudaError_t err = cudaGetLastError();
if (cudaSuccess != err) {
    fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
}
```