

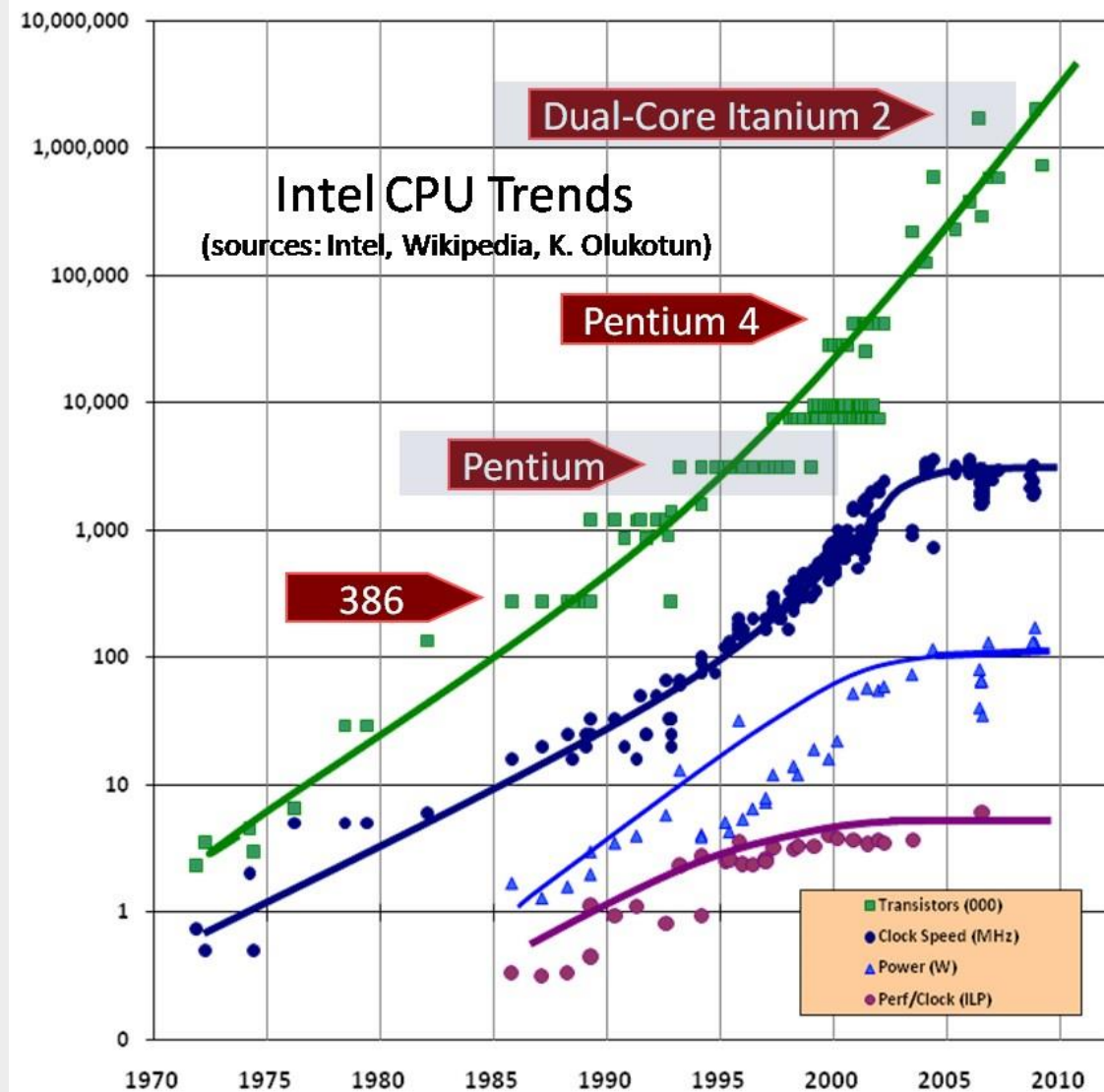


# Parallel Programming with CUDA. Basics

Katerina Bolgova, PhD  
eScience Research Institute & HPC Department

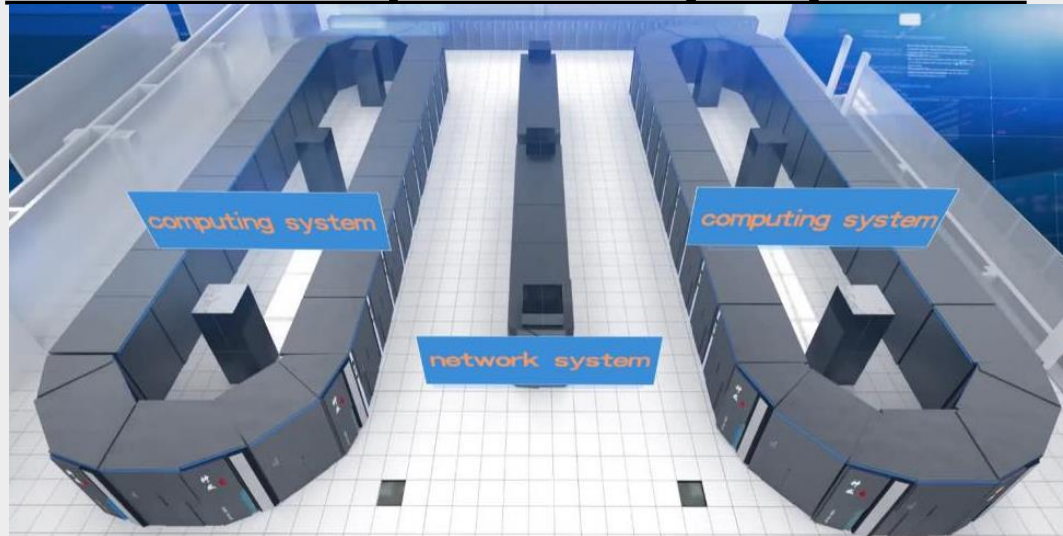
# Back to PP concepts...

What is this picture about?



# Top 500 supercomputers (November, 2017):

## # 1 "Sunway TaihuLight", China



Total Cores	Power (kW)
10,649,600	15,371.00

Rmax (TFlops)	Rpeak (TFlops)
93,014.6	125,435.9

## #227 "Lomonosov", Russia

Total Cores	Power (kW)
78,660	2,800.00

Rmax (TFlops)	Rpeak (TFlops)
901.9	1,700.2



# Back to PP concepts...

2.3 PFlops



7.0  
Megawatts

7000 homes



7.0  
Megawatts

Traditional CPUs are  
not economically feasible

**Moore's law** isn't true any longer:  
It's impossible to make reliable low-  
cost processors that run  
significantly faster



# Back to PP concepts...

- Parallel Computing / Memory Architectures

Distributed Memory

Shared Memory

Hybrid Shared-Distributed  
Memory

**And**

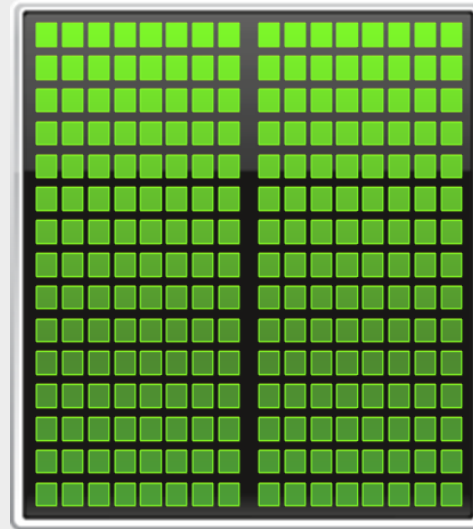
heterogeneous computing

**CPU**



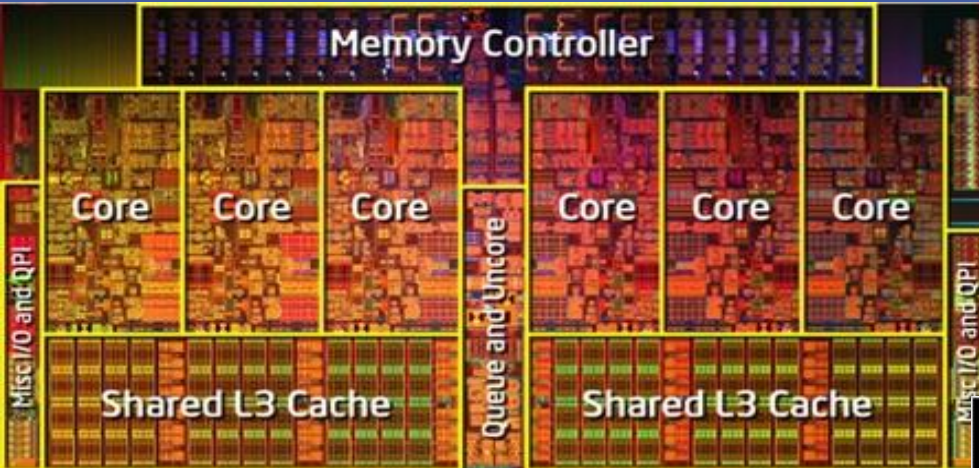
+

**GPU**





# Back to PP concepts...



## Intel Core i7

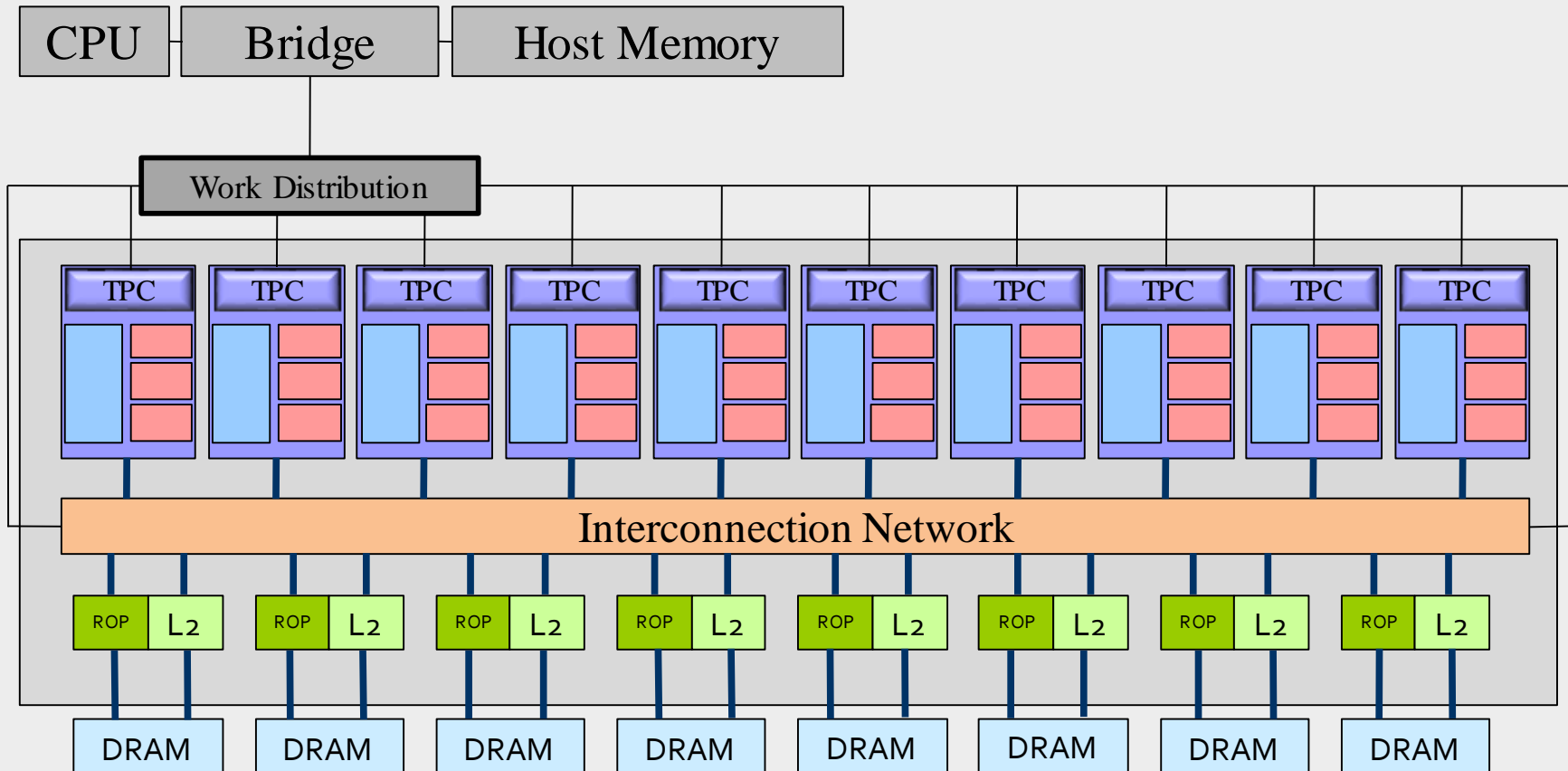
## Fermi



# Introduction

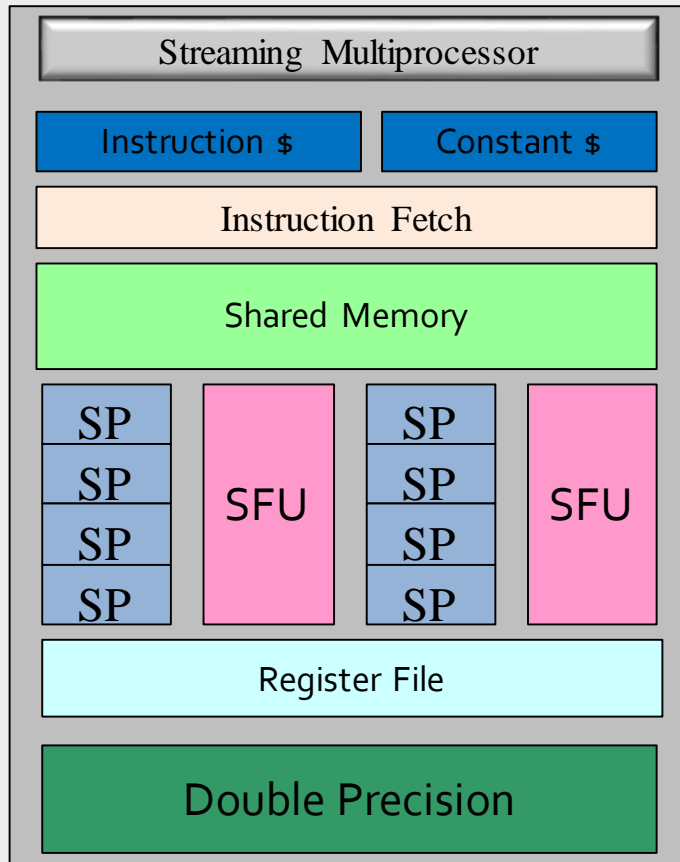
NVIDIA history: <http://www.nvidia.ru/object/corporate-timeline-ru.html>

## Tesla 10 Architecture

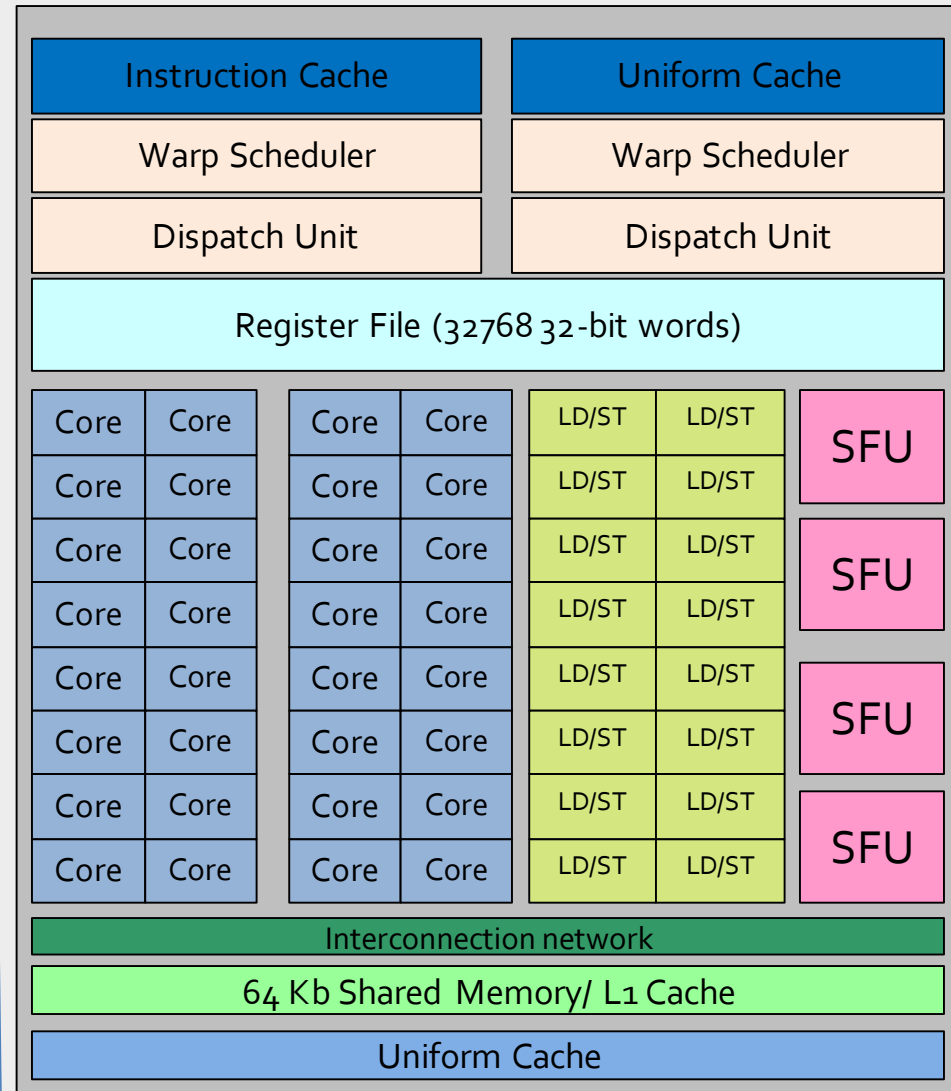


# Introduction

## Tesla 10 Architecture (Streaming multiprocessor, SM)



## Tesla 20 Architecture (SM)

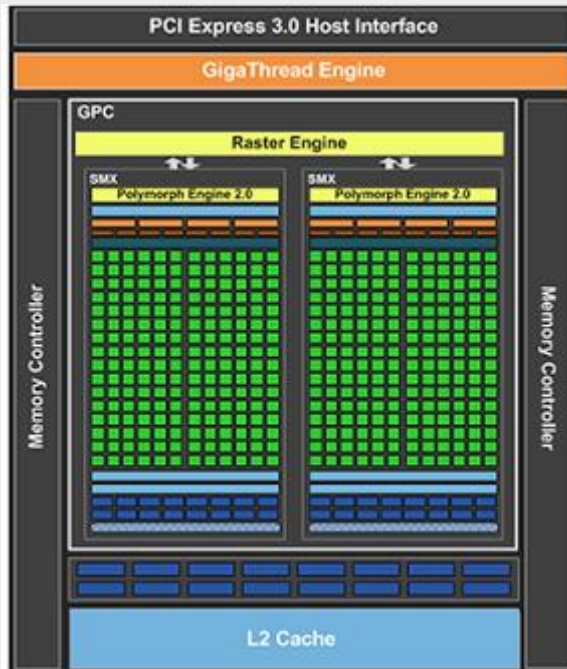




# Introduction

## NVIDIA accelerators:

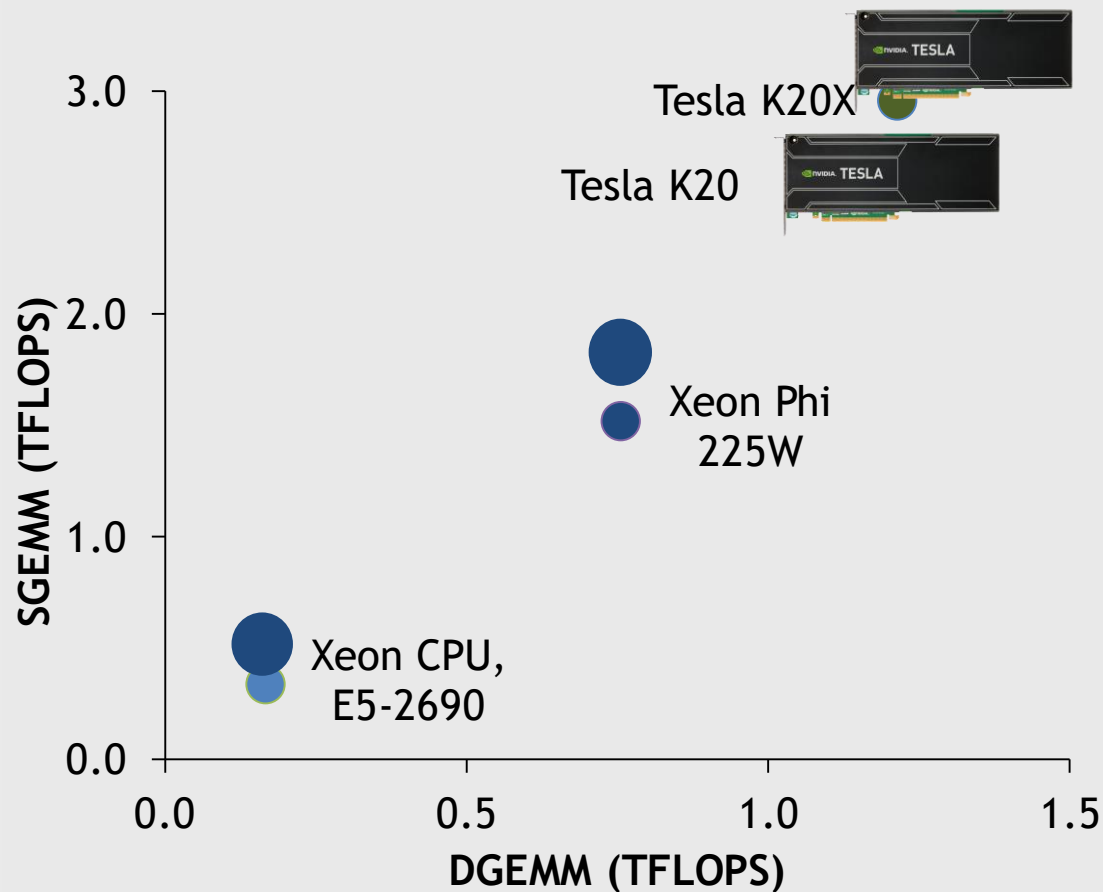
- Fermi
- Kepler
- Maxwell
- Pascal
- Volta



# CPU vs. GPU

Here was a video. You can find it by link:

<https://www.youtube.com/watch?v=-P28LKWTzrl>



Tesla K20X vs Xeon CPU

8x Faster SGEMM

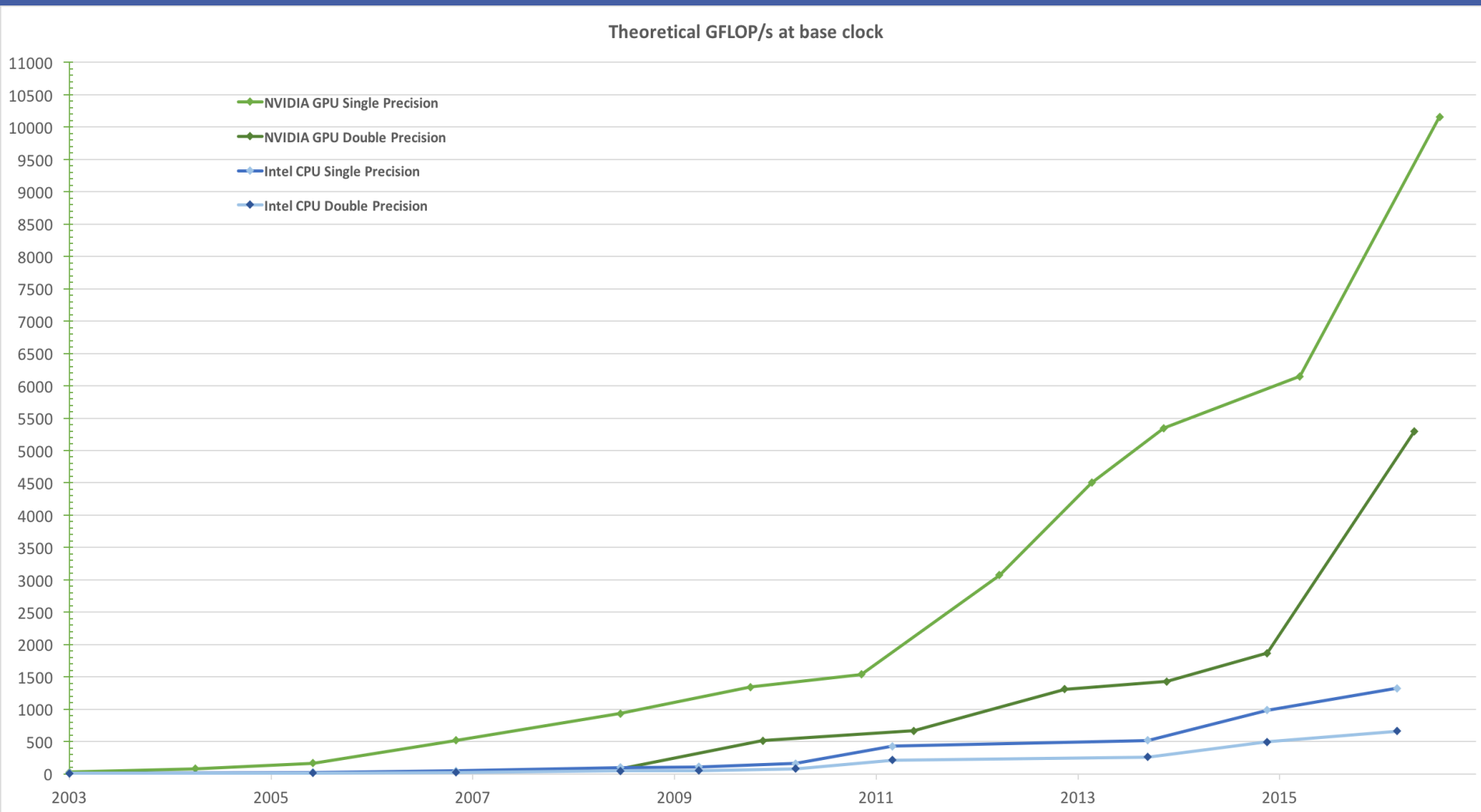
6x Faster DGEMM

Tesla K20X vs Xeon Phi

90% Faster SGEMM

60% Faster DGEMM





\* CUDA C Programming Guide



## General-Purpose computing on Graphics Processing Units:

- became practical and popular after 2001, with the advent of both programmable *shaders* and *floating point* support on graphics processors

## Most popular implementations:

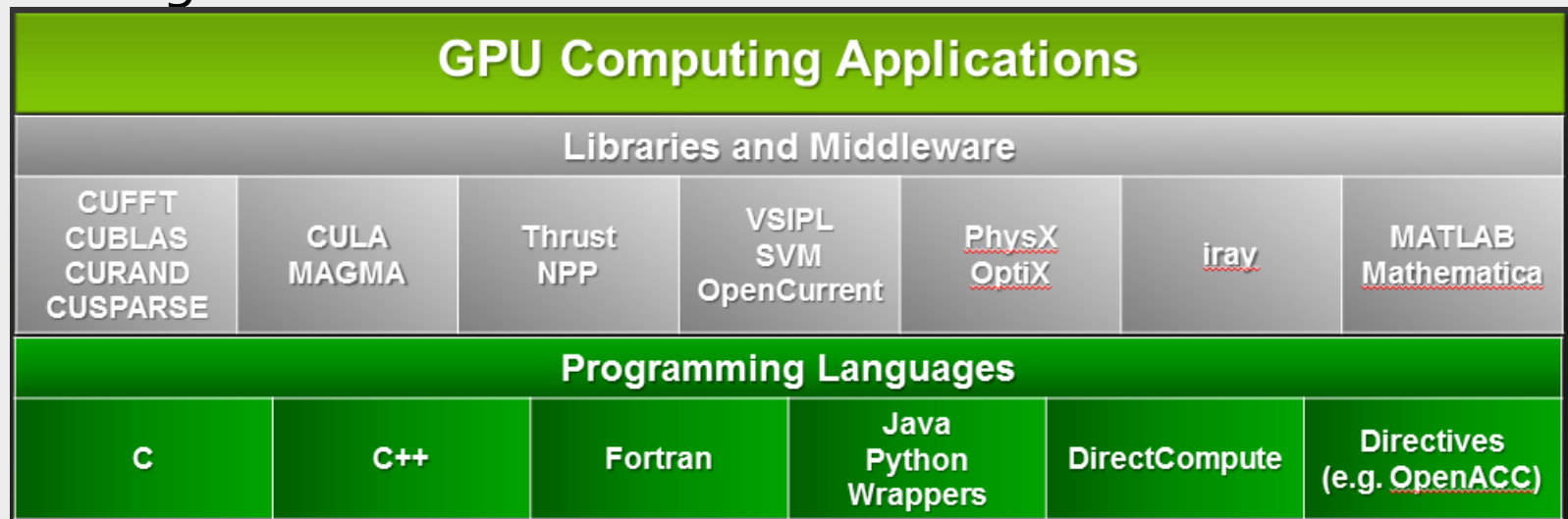
- OpenCL
- DirectCompute
- CUDA



# Compute Unified Device Architecture

**CUDA®** - a General-Purpose Parallel Computing Platform and Programming Model:

- was introduced by NVIDIA in 2006
- expose GPU computing for general purpose
- is designed to support various languages and application programming interfaces



- CUDA C/C++ - Small set of extensions to enable heterogeneous programming



# Compute Unified Device Architecture

Programming  
Approaches

Libraries

“Drop-in”  
Acceleration

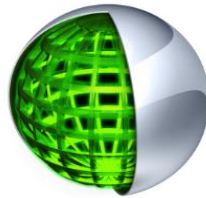
OpenACC  
Directives

Easily Accelerate  
Apps

Programming  
Languages

Maximum Flexibility

Development  
Environment



Nsight IDE  
Linux, Mac and Windows  
GPU Debugging and  
Profiling

CUDA-GDB  
debugger  
NVIDIA Visual  
Profiler

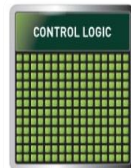
Open Compiler  
Tool Chain



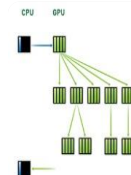
Enables compiling new languages to CUDA  
platform, and CUDA languages to other  
architectures

Hardware  
Capabilities

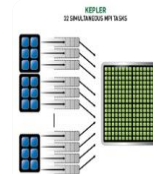
SMX



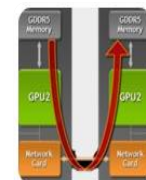
Dynamic  
Parallelism



HyperQ



GPUDirect





# Compute Unified Device Architecture

The CUDA Toolkit:

- is free
- available at <https://developer.nvidia.com>
- includes a compiler, math libraries and tools for debugging and optimizing the performance of applications

There are a lot of code samples, programming guides, user manuals, API references and other documentation. Most useful of them:

- CUDA Programming Guide,  
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- CUDA C Best Practices Guide,  
<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>



# Heterogeneous Computing

- Terminology:
  - **Host** The CPU and its memory (host memory)
  - **Device** The GPU and its memory (device memory)



Host



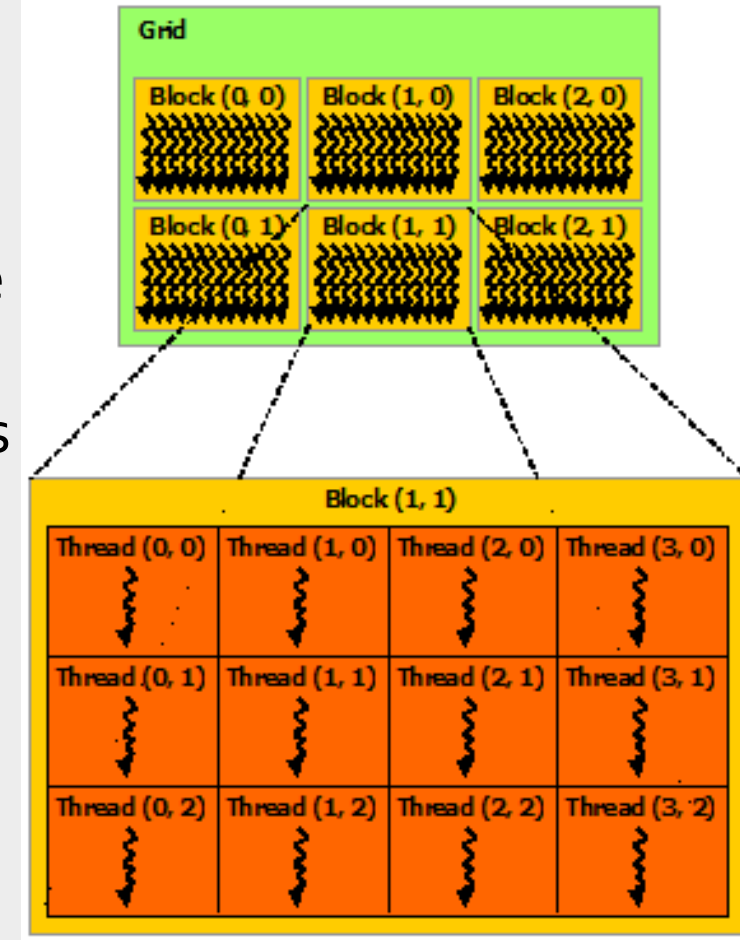
Device



# CUDA. Programming Model

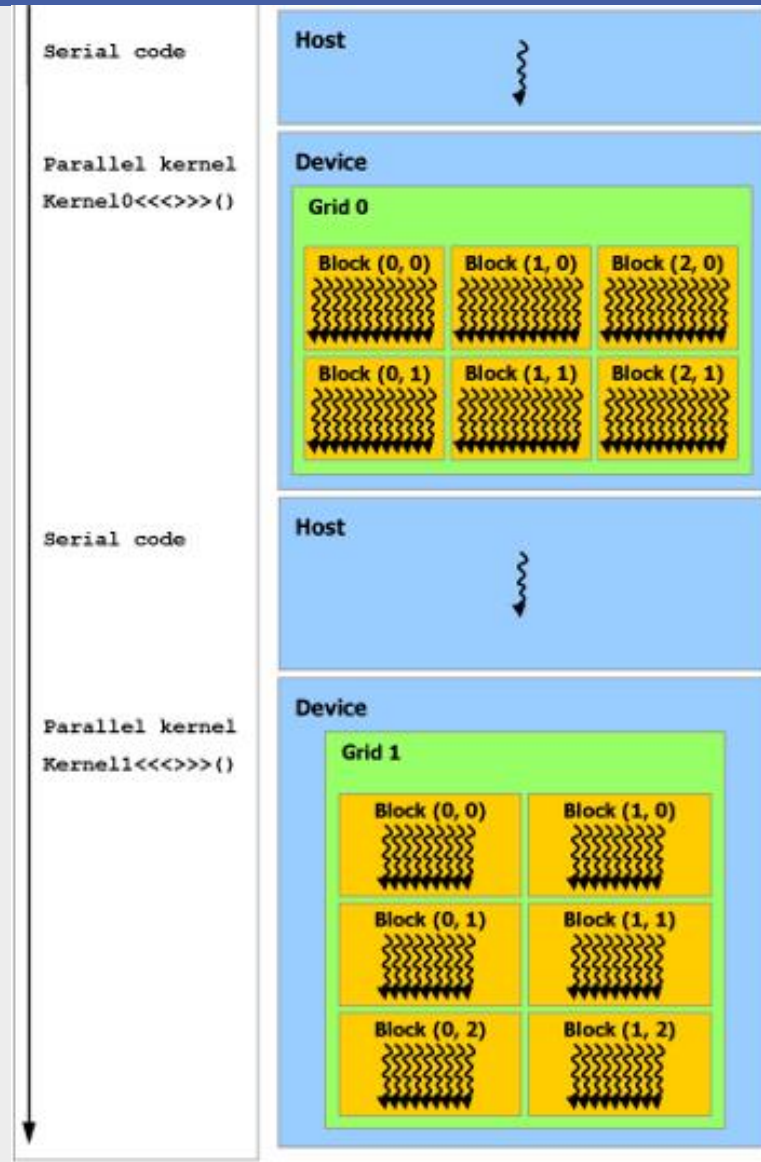
## Basic concepts:

- **Host:** The CPU and its memory (host memory). Serial code is executed here.
- **Device:** The GPU and its memory (device memory). Parallel code is executed here.
- **Kernel:** function that is executed N times in parallel
- kernels are executed by N different CUDA **threads**. Each thread has its own unique ID (threadIdx)
- Threads form a 1/2/3-dimensional **block**
- Blocks are organized into 1/2/3-dimensional **grid**

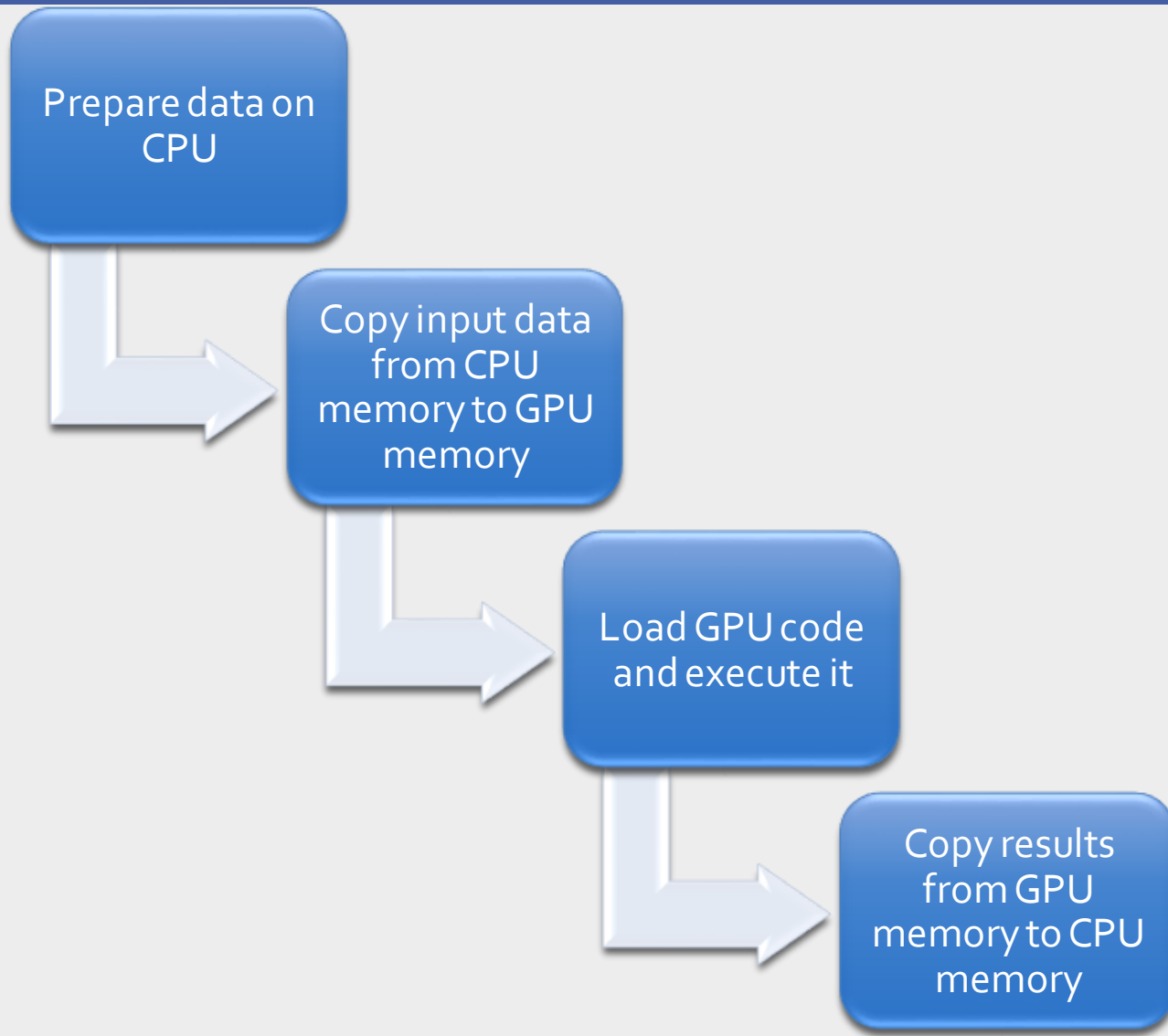


# CUDA. Programming Model

- Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series
- There is a limit to the number of threads per block
- Threads within a block can cooperate by sharing data
- Kernel=grid



# Simple Processing Flow



# “Hello world” with GPU

```
__global__ void mykernel(void) {  
    }  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that runs on the device.
- Triple angle brackets (`mykernel<<<1,1>>>()`) mark a call from host code to device code (also called a “kernel launch”).
- *nvcc* separates source code into host and device components:
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler (`gcc`, `cl.exe`)

That's 'minimum' that needs to execute a function on the GPU





# CUDA runtime API. Basics

- The **Compute Capability** describes the features supported by a CUDA hardware.
- Build-in structure ***cudaDeviceProp*** contains all information about device Compute Capability

GPU	Compute Capability
GeForce GTX 980 M	5.2
GeForce GTX 770	5.0
Tesla K40	3.5
Tesla C2050/C2070	2.0
GeForce GTX 480M	2.0
GeForce 8800 GT	1.1
GeForce 8800 GTX	1.0



# CUDA runtime API. Basics

```
int main ( int argc, char * argv [] )
{
    int            deviceCount;
    cudaDeviceProp devProp;
    cudaGetDeviceCount ( &deviceCount );
    printf          ( "Found %d devices\n", deviceCount );
    for ( int device = 0; device < deviceCount; device++ )
    {
        cudaGetDeviceProperties ( &devProp, device );
        printf ( "Device %d\n", device );
        printf ( "Compute capability      : %d.%d\n", devProp.major, devProp.minor );
        printf ( "Name                    : %s\n", devProp.name );
        printf ( "Total Global Memory      : %d\n", devProp.totalGlobalMem );
        printf ( "Shared memory per block: %d\n", devProp.sharedMemPerBlock );
        printf ( "Registers per block      : %d\n", devProp.regsPerBlock );
        printf ( "Warp size                : %d\n", devProp.warpSize );
        printf ( "Max threads per block   : %d\n", devProp.maxThreadsPerBlock );
        printf ( "Total constant memory   : %d\n", devProp.totalConstMem );
        printf ( "Multiprocessor Count    : %d\n", devProp.devProp.mrItiProcessorCount );
    }
    return 0;
}
```

Demo



- It consists of a minimal set of extensions to the C language and a runtime library.
- Extensions allow programmers to define a kernel as a C function and use some new syntax to specify the grid and block dimension each time the function is called:
  - Function Type Qualifiers
  - Variable Type Qualifiers
  - Built-in Vector Types
  - Built-in Variables



# CUDA C. Function Type Qualifiers

Qualifier	Executed at	Called from
<code>__device__</code>	device	device
<code>__global__</code>	device	host
<code>__host__</code>	host	host

- The `__global__` qualifier declares a function as being a kernel.
- `__global__` functions must have void return type
- A call to a `__global__` function is asynchronous
- The `__device__` and `__host__` qualifiers can be used together however, in which case the function is compiled for both the host and the device.
- The `__global__` and `__host__` qualifiers cannot be used together
- It is equivalent to declare a function with only the `__host__` qualifier or to declare it without any of qualifiers



# CUDA C. Built-in Vector Types

- Vector types are derived from the basic integer and floating-point types:
  - (u)char, (u)int, (u)short, (u)long, float could have 1-/2-/3-/4-dimensions
  - longlong, double could have 1-/2-dimensions
  - the 1st, 2nd, 3rd, and 4th components are accessible through the fields x, y, z, and w, respectively
- They all come with a constructor function: `make_<type name>`

Example:

```
float4 b = make_float4 ( 1, 2, 5, 3 );
```

- The type ***dim3*** is an integer vector type based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1.



# CUDA C. Built-in Variables

Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device

- `dim3` **`gridDim`** — contains the dimensions of the grid,
- `uint3` **`blockIdx`** — contains the block index within the grid,
- `dim3` **`blockDim`** — contains the dimensions of the block,
- `uint3` **`threadIdx`** — contains the thread index within the block,





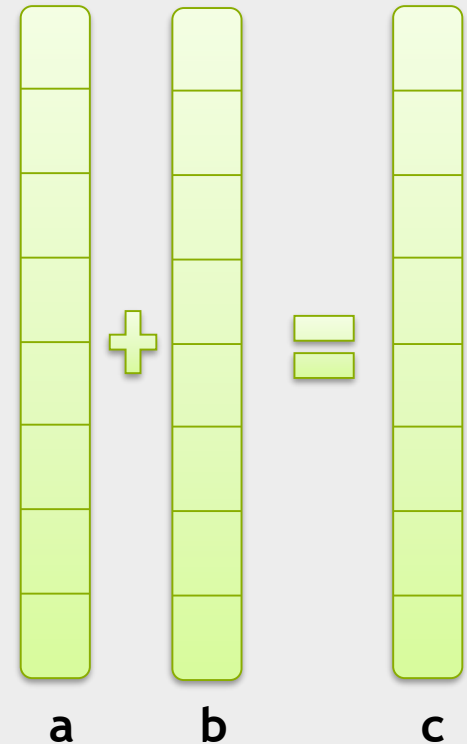
# CUDA C/C++ Programming

## Simple Example: vector addition

- $C_i = a_i + b_i$
- But before let's start by adding two integers :

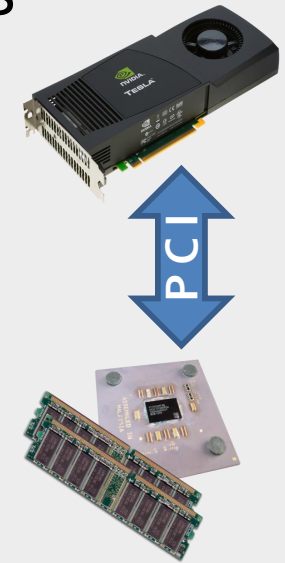
```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Note: that we use pointers for the variables
- `add()` runs on the device, so ***a***, ***b*** and ***c*** must point to device memory
- We need to allocate memory on the GPU



# Memory Management

- Host and device memory are separate entities
  - **Device** pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - **Host** pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



# Memory Management

```
cudaError_t cudaMalloc      ( void ** devPtr, size_t size );
```

```
cudaError_t cudaFree       ( void * devPtr );
```

```
cudaError_t cudaMemcpy     ( void * dst, const void * src,  
                              size_t count, enum cudaMemcpyKind kind );
```

```
cudaError_t cudaMemcpyAsync ( void * dst, const void * src,  
                                size_t count,  
                                enum cudaMemcpyKind kind,  
                                cudaStream_t stream );
```

```
cudaError_t cudaMemset     ( void * dst, int value, size_t count );
```

```
cudaError_t cudaMallocHost ( void ** devPtr, size_t size );
```



# CUDA C/C++ Programming

```
int main() {  
    int a, b, c;           // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;  
  
    // Copy inputs to device  
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);  
  
    // Launch add() kernel on GPU  
    add<<<1,1>>>(d_a, d_b, d_c);  
  
    // Copy result back to host  
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);  
  
    // Cleanup  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

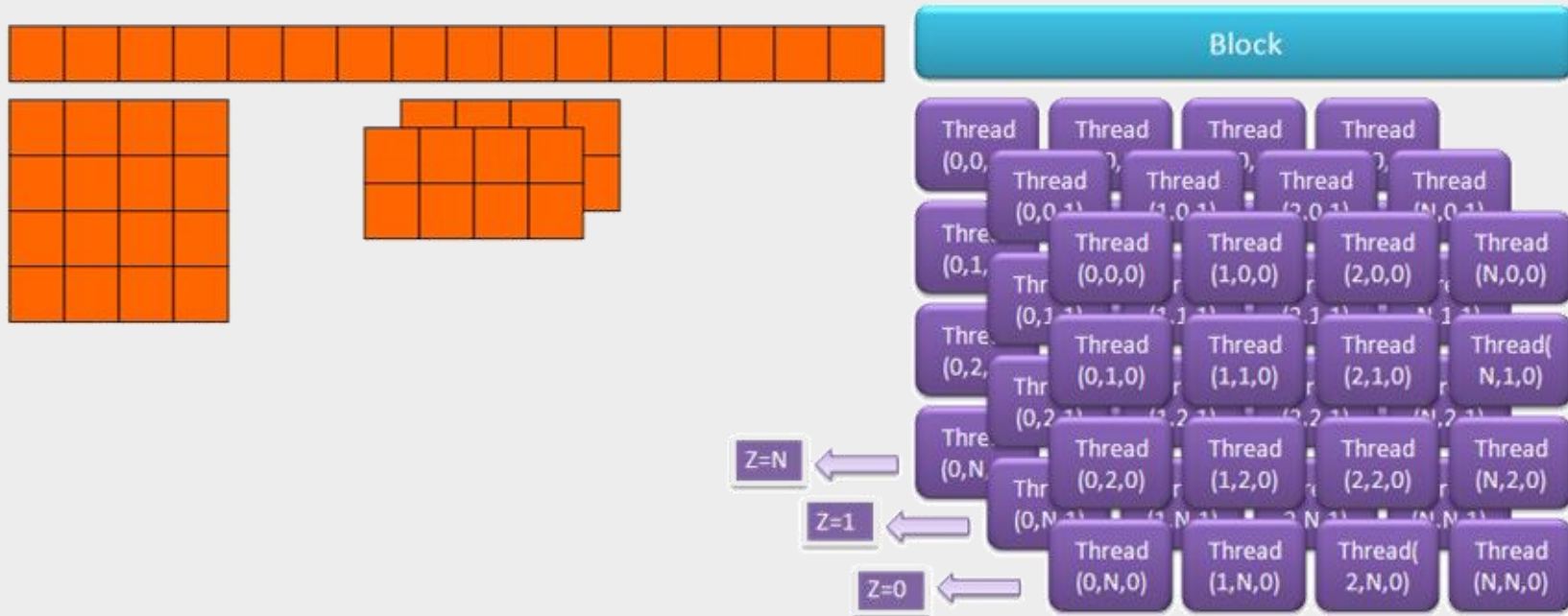


# CUDA C/C++. Kernel Launch

```
Kernel_name<<<grid, block, mem, stream>>>( params ),
```

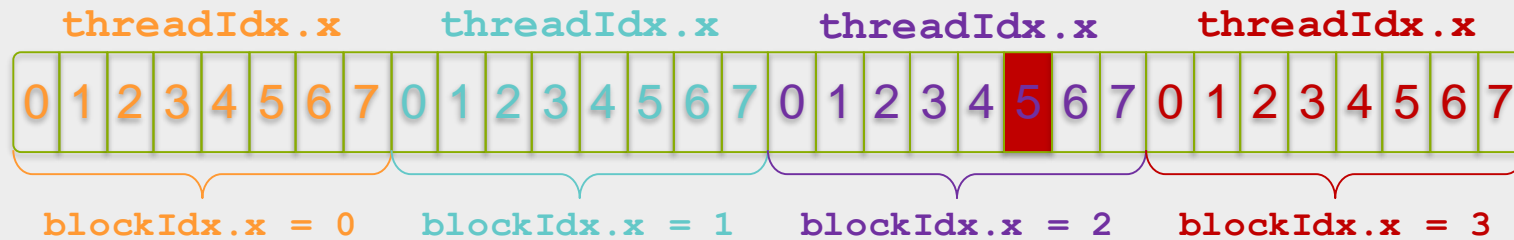
Where:

- dim3 **grid** - grid size (number of block);
- dim3 **block** - block size (thread number per block).
- size\_t **mem** - amount of shared memory per block;
- cudaStream\_t **stream** - CUDA stream number.



# CUDA C. Indexing Arrays with Blocks and Threads

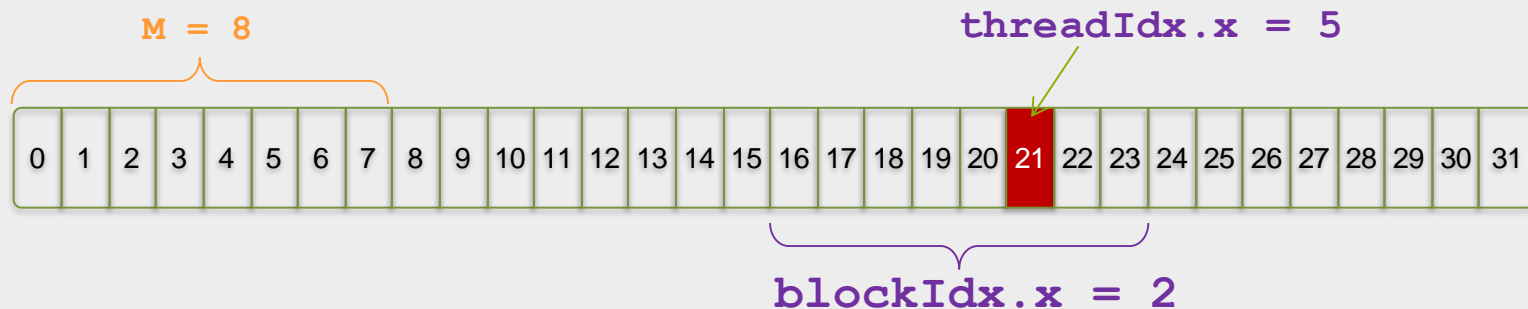
- Consider indexing an array with one element per thread (8 threads/block)



- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

- Which thread will operate on the red element?



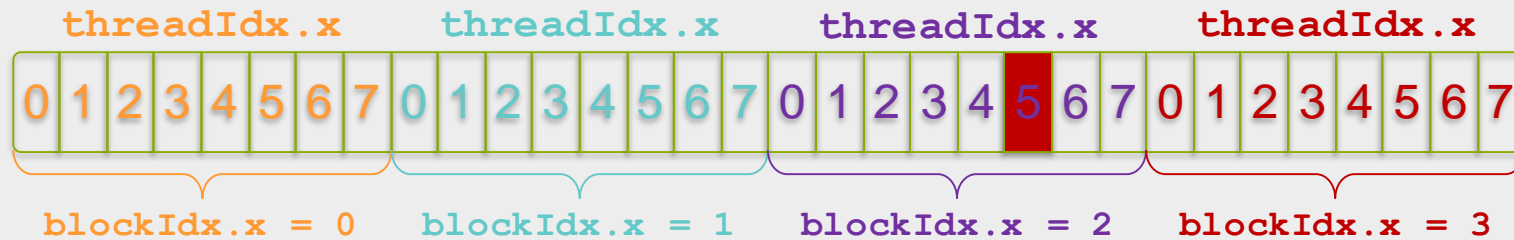
```
int index = threadIdx.x + blockIdx.x * M;  
          =           5      +           2      * 8;  
          = 21;
```



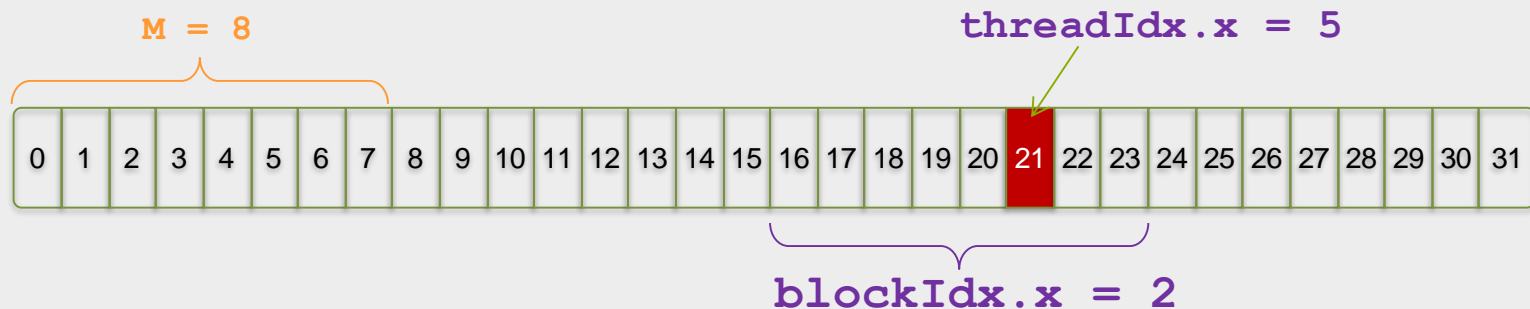


# CUDA C. Indexing Arrays with Blocks and Threads

- Consider indexing an array with one element per thread (8 threads/block)



- With M threads/block a unique index for each thread is given by:  
`int index = threadIdx.x + blockIdx.x * blockDim.x;`
- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * blockDim.x;  
          =           5      +           2      * 8;  
          = 21;
```



# CUDA C. Vector Addition

```
__global__ void vector_add(float *a, float *b, float *c)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];}

int main() {
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);

    vector_add<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



# CUDA runtime API .Time Measurement

**Event** in CUDA is **cudaEvent\_t** object that is used for indicate “the point” of some event. It’s used for event management.

```
cudaError_t cudaEventCreate      ( cudaEvent_t * );  
cudaError_t cudaEventRecord      ( cudaEvent_t *, cudaStream_t );  
cudaError_t cudaEventQuery       ( cudaEvent_t );  
cudaError_t cudaEventSynchronize ( cudaEvent_t );  
cudaError_t cudaEventElapsedTime ( float * time, cudaEvent_t st,  
                                     cudaEvent_t sp );  
cudaError_t cudaEventDestroy     ( cudaEvent_t );
```



# CUDA C. Vector Addition

```
int main() {  
...  
    cudaEvent_t start, stop;  
    cudaEventCreate(&start);  
    cudaEventCreate(&stop);  
    float gpuTime;  
...  
    cudaEventRecord ( start, 0 );  
...  
    vector_add<<<numBlocks, threadsPerBlock>>>(A, B, C);  
...  
    cudaEventRecord ( stop, 0);  
    cudaEventSynchronize (stop) ;  
    cudaEventElapsedTime (&gpuTime, start, stop );  
    printf("time spent executing by the GPU: %.2f  
           milliseconds\n", gpuTime );  
  
    cudaEventDestroy (start);  
    cudaEventDestroy (stop);  
...  
}
```

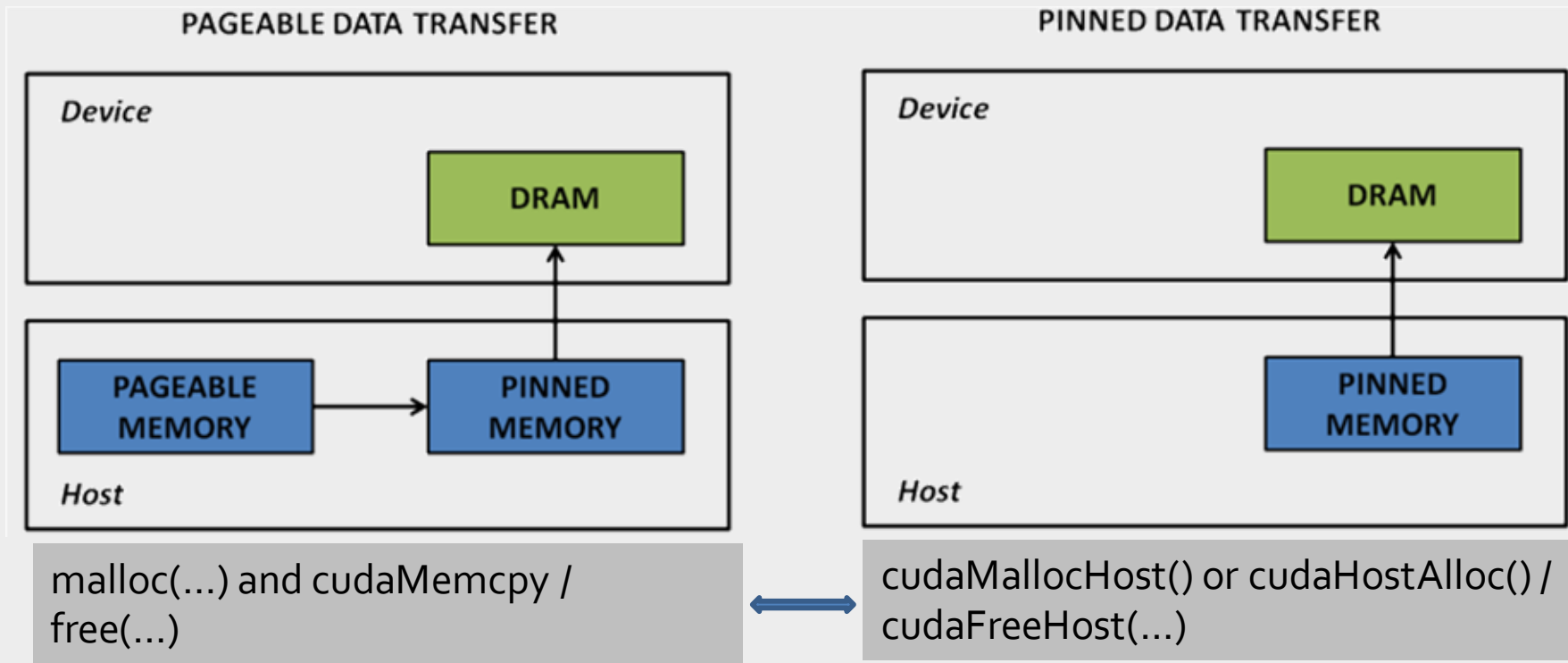


Demo



# CUDA C. Pinned-memory

- Host (CPU) data allocations are pageable by default.
- the CUDA driver must first allocate a temporary page-locked, or “pinned”, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory



\* How to Optimize Data Transfers in CUDA C/C++:

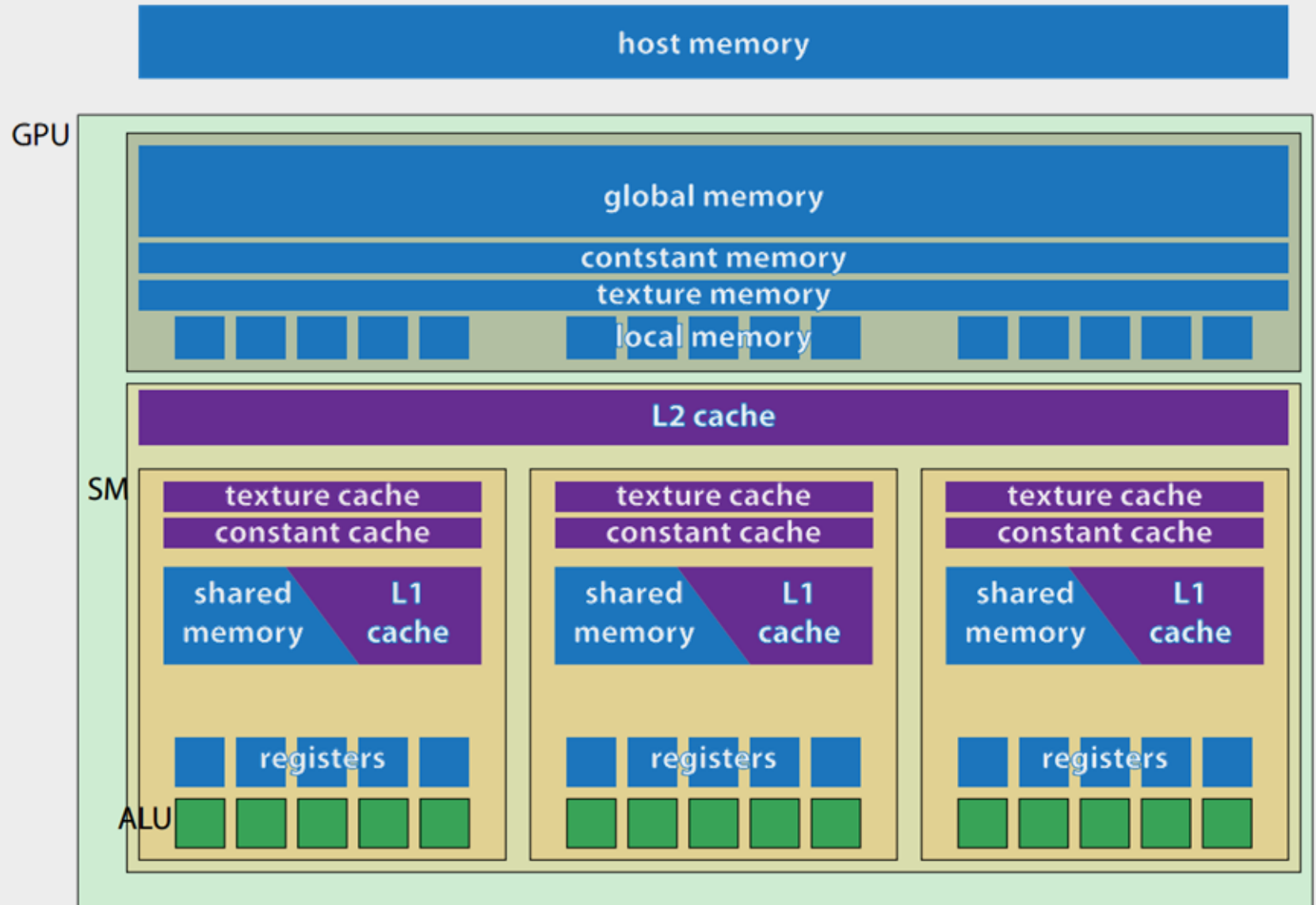
<https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/>

# CUDA. Memory Hierarchy

Memory Type	Access	Level	Speed	Localization
Registers	R/W	Per-thread	High	SM
Local	R/W	Per-thread	Low	DRAM
Shared	R/W	Per-block	High	SM
Global	R/W	Per-grid	Low	DRAM
Constant	R/O	Per-grid	High	DRAM
Texture	R/O	Per-grid	High	DRAM



# CUDA. Memory Hierarchy



# CUDA. Memory Hierarchy

Memory Type	Access	Level	Speed	Localization
Registers	R/W	Per-thread	High	SM
Local	R/W	Per-thread	Low	DRAM
Shared	R/W	Per-block	High	SM
Global	R/W	Per-grid	Low	DRAM
Constant	R/O	Per-grid	High	DRAM
Texture	R/O	Per-grid	High	DRAM

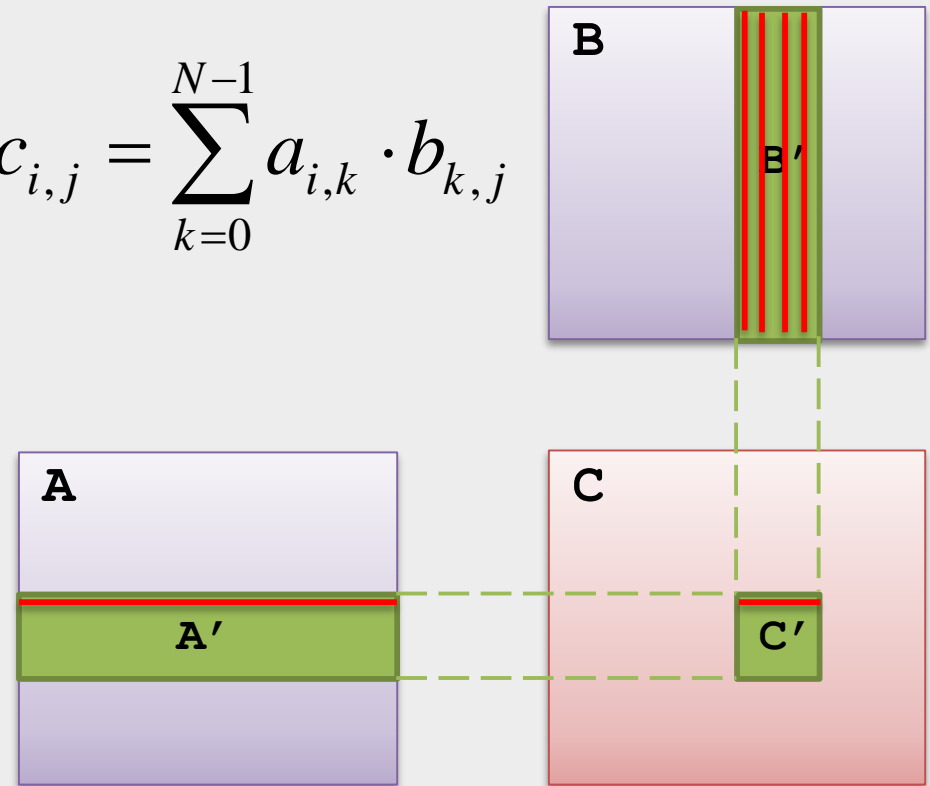




# Sample. Matrix Multiplication

- Consider 2 matrices  $A$  и  $B$  with  $(N*N)$  size
- Each block calculates a part of result matrix  $C$
- Each thread in block calculates 1 element in this part
- To simplify:
  - $N = 2048$
  - $Block\_size = 32$
  - Each matrix is 1D array

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j}$$



# Matrix Multiplication. Kernel

```
#define BLOCK_SIZE 32
#define N 2048
__global__ void matMult ( float * a, float * b, int n,
float * c )
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float sum = 0.0f;
    int ia = n * BLOCK_SIZE * by + n * ty;
    int ib = BLOCK_SIZE * bx + tx;
    int ic = ia + ib;

    for (int k = 0; k < n; k++)
        sum += a[ia + k] * b[ib + k*n];
    c[ic] = sum;
}
```



# Matrix Multiplication. Main ()

```
    int numBytes = N * N * sizeof(float); //define memory size
// allocate host memory
    float * h_A = (float*)malloc(N * N * sizeof(float));
    float * h_B = (float*)malloc(N * N * sizeof(float));
    float * h_C = (float*)malloc(N * N * sizeof(float));
/*init matrix*/ ...
//assign variable for device
    float * d_A;    float * d_B;        float * d_C;
// allocate device memory
    cudaMalloc((void**)&d_A, numBytes);
    cudaMalloc((void**)&d_B, numBytes);
    cudaMalloc((void**)&d_C, numBytes);
// set kernel launch configuration
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 blocks(N / BLOCK_SIZE, N / BLOCK_SIZE);
/*...*/
//copy data from host to device
    cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
```



# Matrix Multiplication. Main ()

```
//kernel launch
    matMult_Global_mem<<<blocks,threads>>> (d_A, d_B, N, d_C);
//copy data from device to host
    cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
/*...*/
//memory free
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```



# Matrix Multiplication. Global memory

- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*

- *GPU code:*
  - *Name : GeForce GTX 960*
  - *Compute capability : 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6



- **BUT:**
  - Each element (thread)
    - $2 \cdot N$  arithmetic operations
    - $2 \cdot N$  accesses to global memory
  - *Global Memory has a **low** access speed (200-400 clock cycles)*



# CUDA. Shared memory

- is much faster than global memory
- is allocated per thread block, so all threads in the block have access to the same shared memory.
- Threads can access data in shared memory loaded from global memory by other threads within the same thread block
- When sharing data between threads, we need to be careful to avoid race conditions

```
__shared__ int dSt[32];  
__shared__ float dSum;
```

```
extern __shared__ int dDyn[];
```

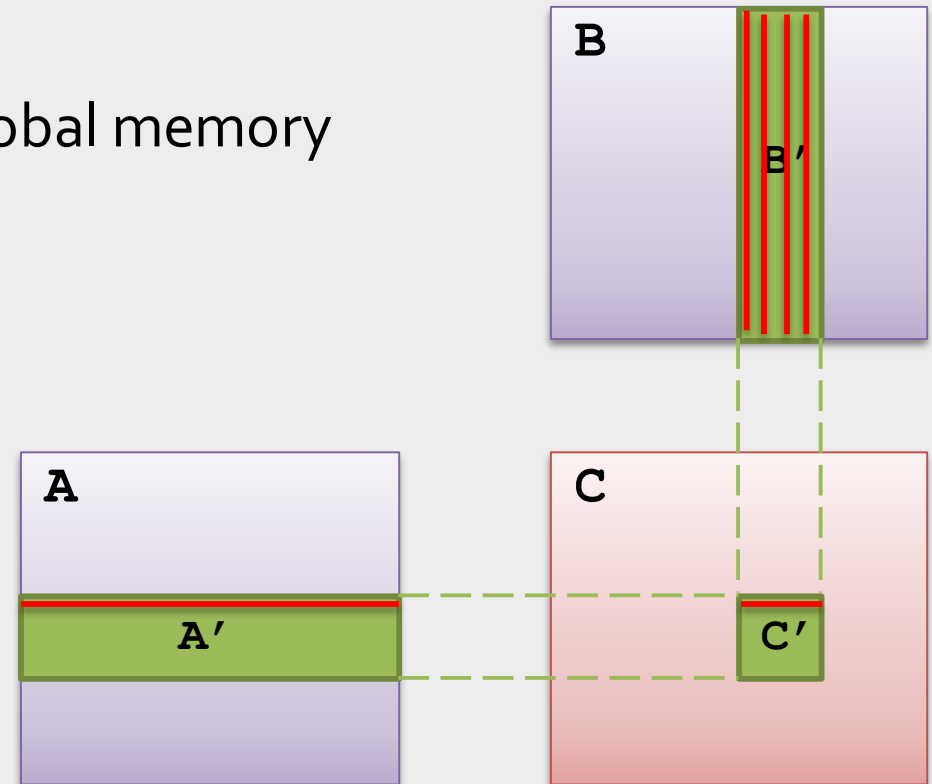
- **Usage pattern**

```
/*Load data from Global memory*/  
__syncthreads()  
/* some calculations */  
__syncthreads()  
/* write result to global memory */
```



# Matrix Multiplication

- During the calculation of the matrix  $C'$  is constantly used the same elements from matrices  $A$  and  $B$ 
  - Repeatedly read from the global memory
- These reusable elements form stripes in the matrices  $A$  and  $B$
- The size of this strip is  $N \times 32$  and even one such strip is not placed in the shared-memory



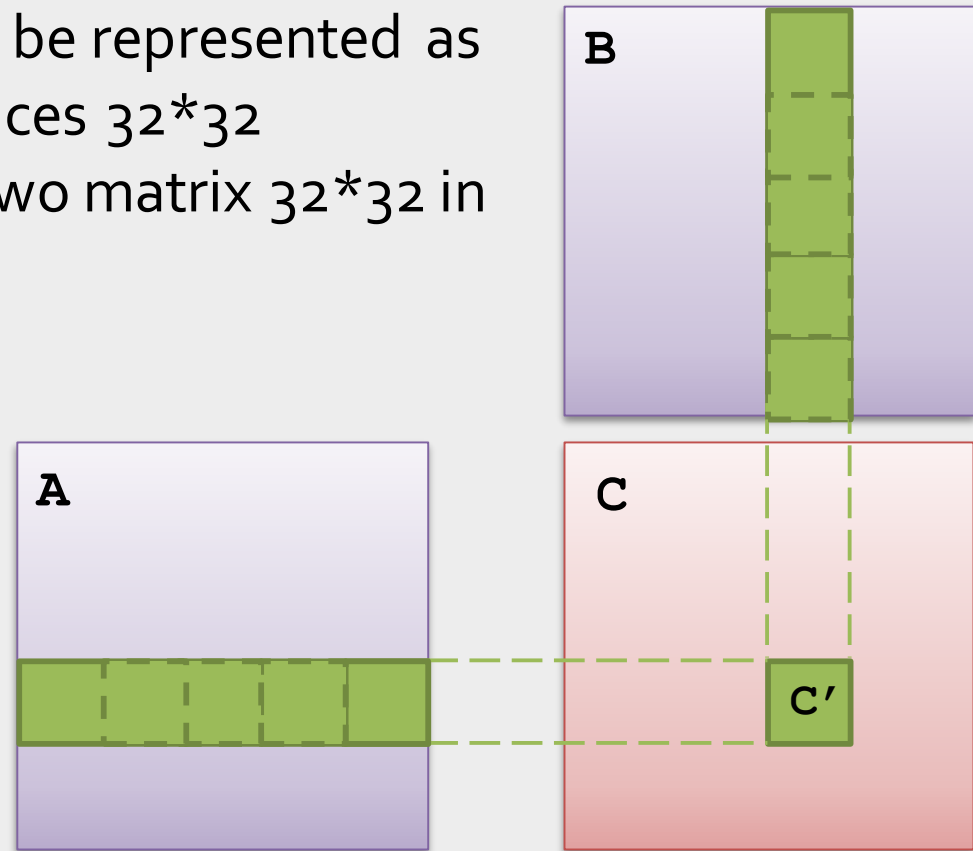
$$32 * 2048 * \text{sizeof(float)} = 256 \text{ Кбайт}$$



# Matrix Multiplication. Shared memory

- Divide each strip into square matrix ( $32 \times 32$ )
- Then required submatrix  $C'$  can be represented as a sum of products of such matrices  $32 \times 32$
- For calculations we need only two matrix  $32 \times 32$  in shared-memory

$$C' = A'_1 * B'_1 + \dots + A'_{N/32} * B'_{N/32}$$



**$4 \times 32 \times 32 \times \text{sizeof(float)} = 16 \text{ Кбайт}$**





# Matrix Multiplication. Shared memory. Kernel

```
__global__ void matMult_1 ( float * a, float * b, int n, float * c ) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
    int bBegin = BLOCK_SIZE * bx;
    int aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * n;
    float sum = 0.0f;

    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

    for ( int ia = aBegin, ib = bBegin; ia <= aEnd;
                                     ia += aStep, ib += bStep ){
        as [tx][ty] = a [ia + n * ty + tx];
        bs [tx][ty] = b [ib + n * ty + tx];
        __syncthreads ();
        for ( int k = 0; k < BLOCK_SIZE; k++ )
            sum += as [k][ty] * bs [tx][k];
        __syncthreads ();
    }
    c [aBegin + bBegin + n * ty + tx] = sum;
}
```



# Matrix Multiplication. Results

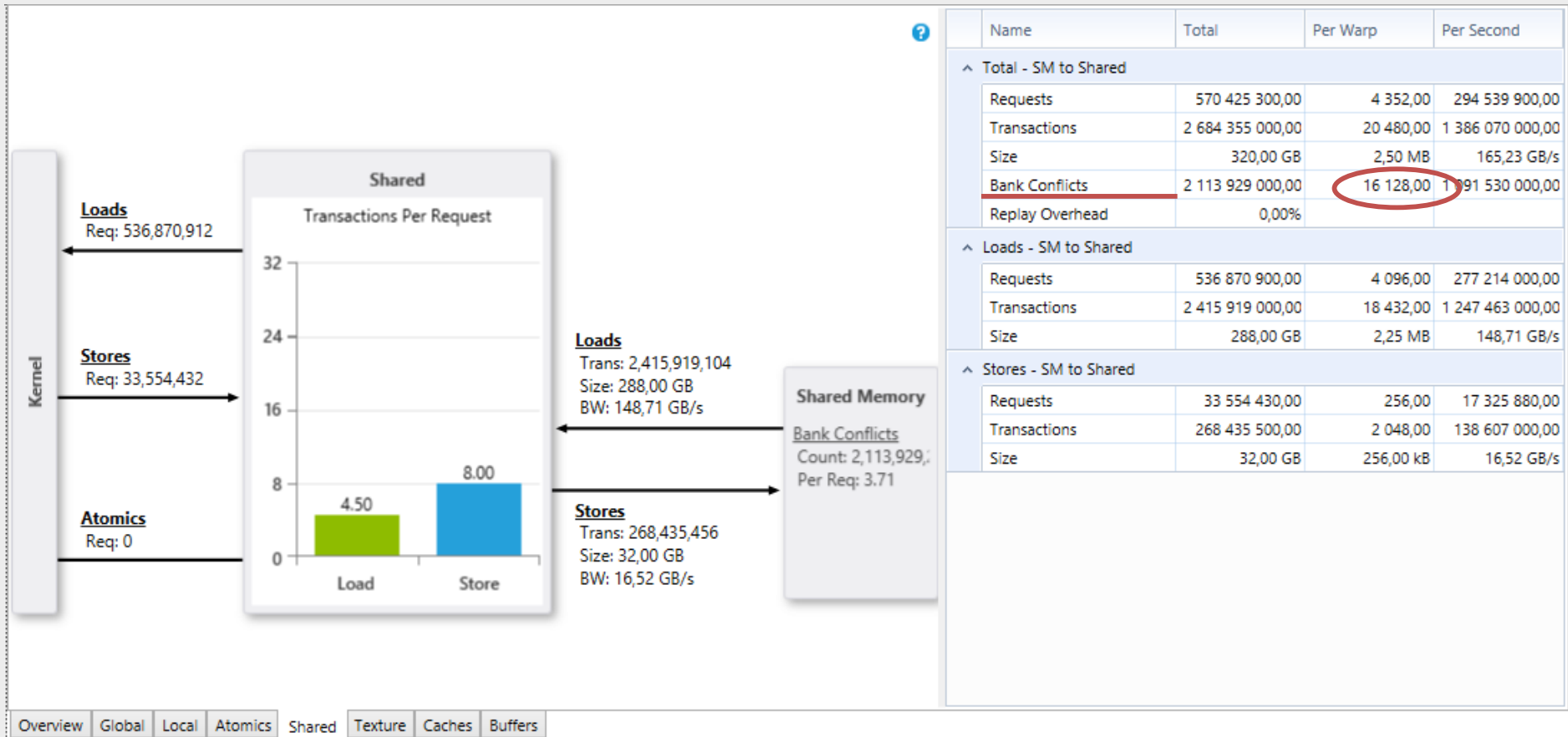
- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*
- GPU code:
  - *Name : GeForce GTX 960*
  - *Compute capability : 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6
Shared Memory	3151,4	22,14



# CUDA Toolkit. Profiling

- Visual Profiler
- *NSight*

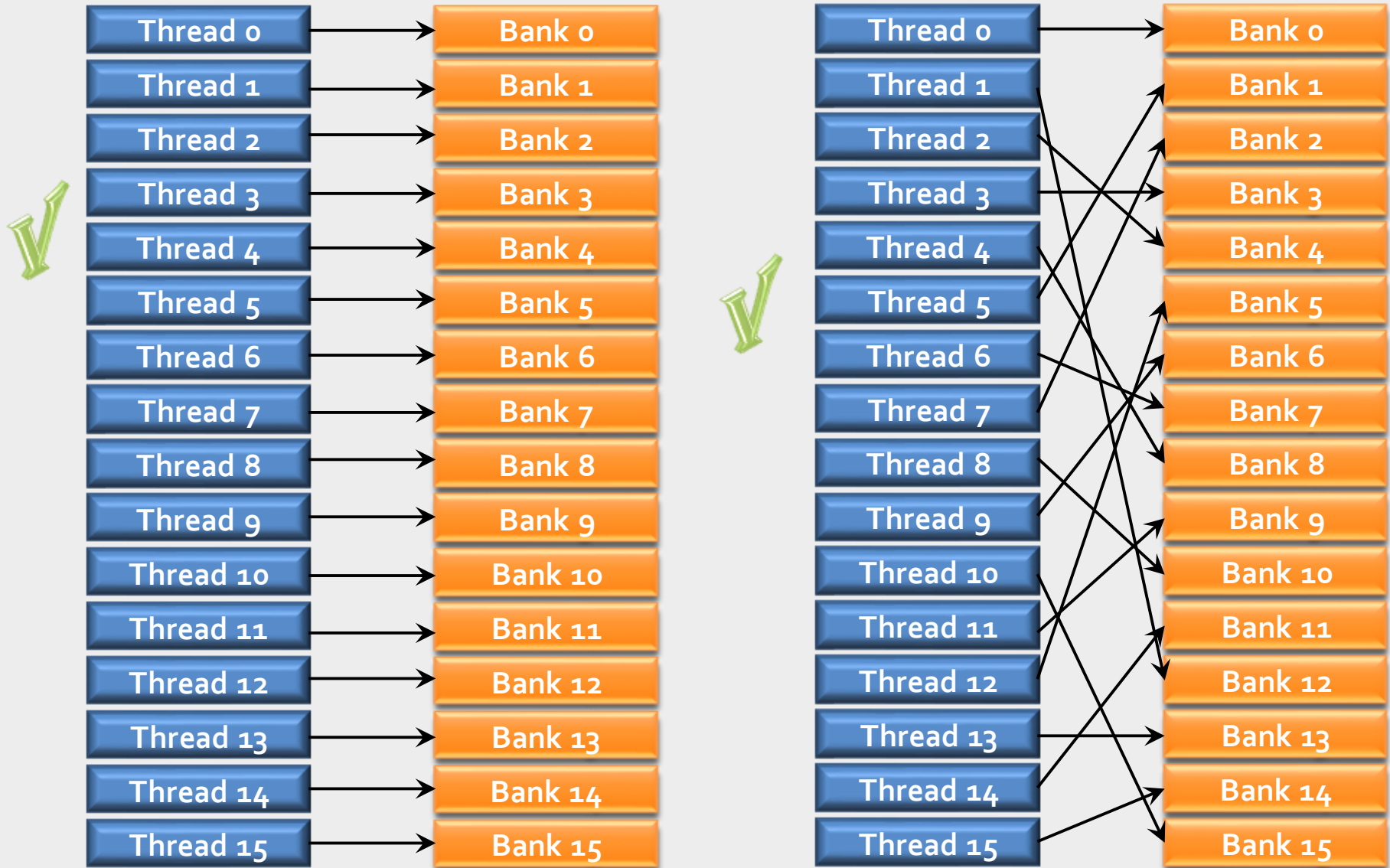


# Shared Memory. Bank conflicts

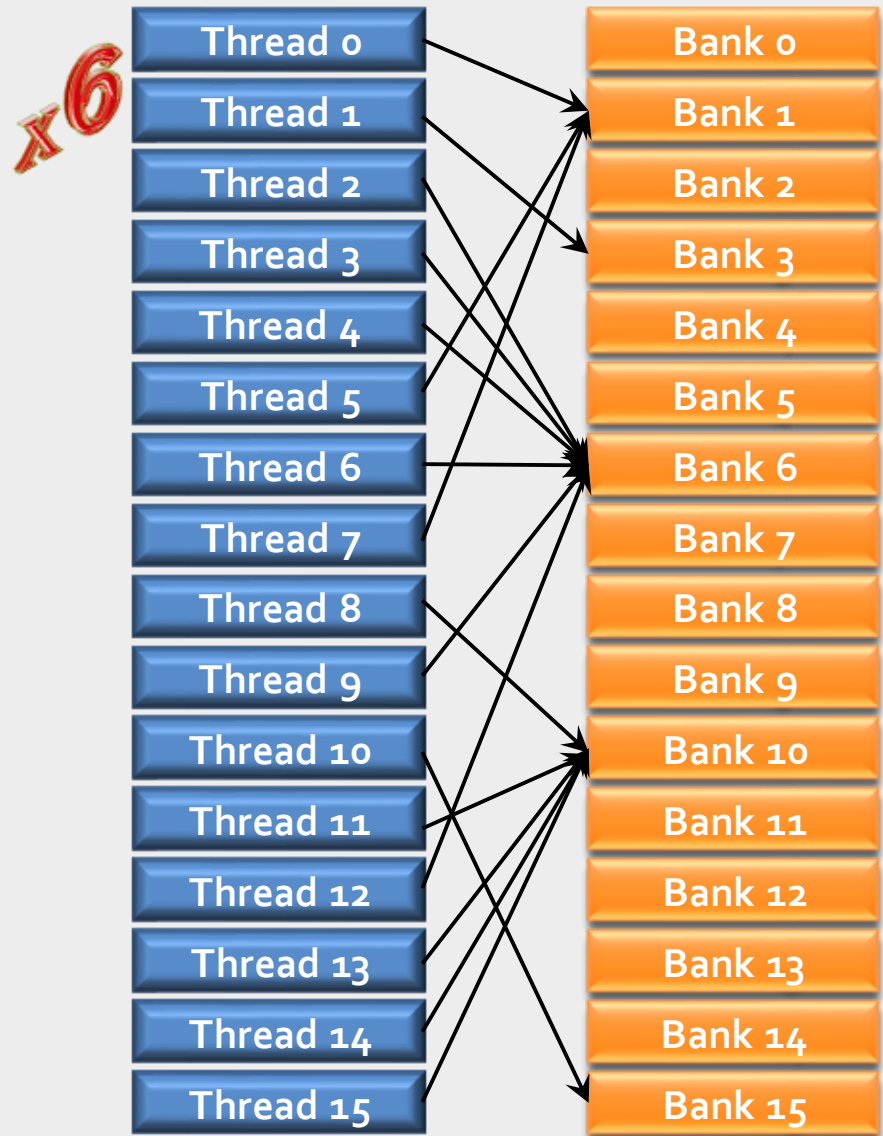
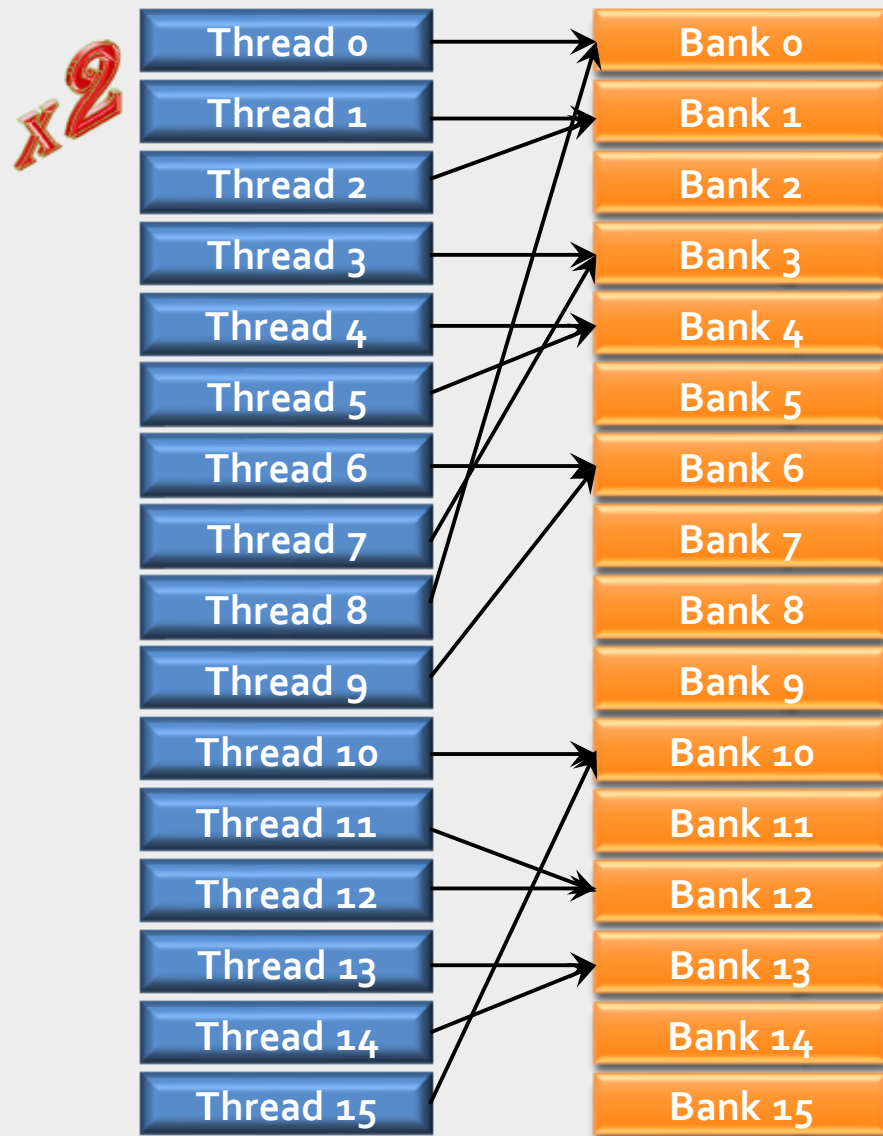
- shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously.
- Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle.
- If multiple threads' requested addresses map to the same memory bank, the accesses are serialized (Bank conflicts).



# Shared Memory. Bank conflicts

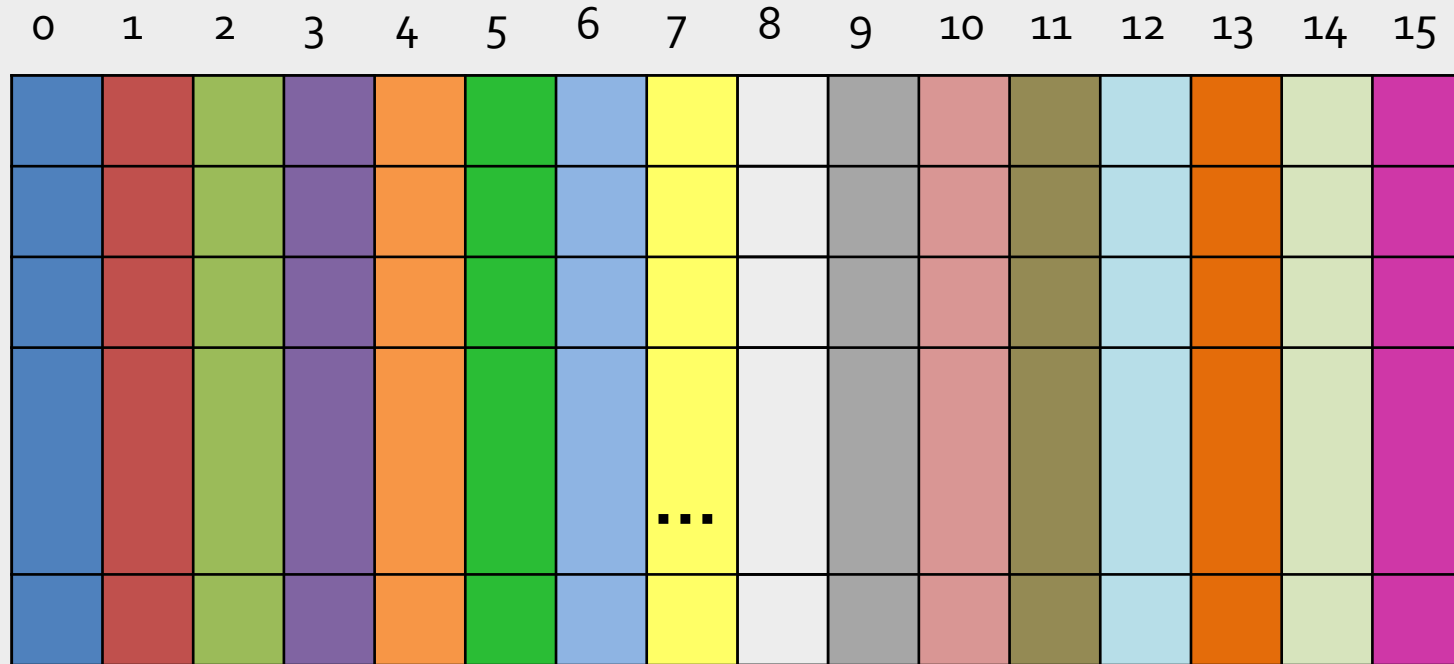


# Shared Memory. Bank conflicts



# Sample. Matrix Multiplication (16\*16)

- The multiplication of two matrices 16\*16, located in the shared memory
- Access to one matrix is by rows, to the another - by columns







































































- All of the elements of the row are distributed evenly across the 16 banks. There are no conflicts
- All the elements of a column are in one Bank. 16-th order bank conflict



# Sample. Matrix Multiplication (16\*16)

- Complete each line with one (fictitious) element

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
																
																
																
							...									
																

- All column elements are in different banks
- Actually, by slightly increasing the amount of memory completely got rid of the conflicts





# Matrix Multiplication. Shared memory. V.2

```
__global__ void matMult_2 ( float * a, float * b, int n, float * c ) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
    int bBegin = BLOCK_SIZE * bx;
    int aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * n;
    float sum = 0.0f;
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE+1];
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE+1];

    for ( int ia = aBegin, ib = bBegin; ia <= aEnd;
          ia += aStep, ib += bStep ){
        as [tx][ty] = a [ia + n * ty + tx];
        bs [tx][ty] = b [ib + n * ty + tx];
        __syncthreads ();
        for ( int k = 0; k < BLOCK_SIZE; k++ )
            sum += as [k][ty] * bs [tx][k];
        __syncthreads ();
    }
    c [aBegin + bBegin + n * ty + tx] = sum;
}
```



# Matrix Multiplication. Shared memory. v.3

```
__global__ void matMult_2 ( float * a, float * b, int n, float * c ) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int aBegin = n * BLOCK_SIZE * by;
    int aEnd = aBegin + n - 1;
    int bBegin = BLOCK_SIZE * bx;
    int aStep = BLOCK_SIZE, bStep = BLOCK_SIZE * n;
    float sum = 0.0f;
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

    for ( int ia = aBegin, ib = bBegin; ia <= aEnd;
                                     ia += aStep, ib += bStep ){
        as [ty][tx] = a [ia + n * ty + tx];
        bs [ty][tx] = b [ib + n * ty + tx];
        __syncthreads ();
        for ( int k = 0; k < BLOCK_SIZE; k++ )
            sum += as [ty][k] * bs [k][tx];
        __syncthreads ();
    }
    c [aBegin + bBegin + n * ty + tx] = sum;
}
```



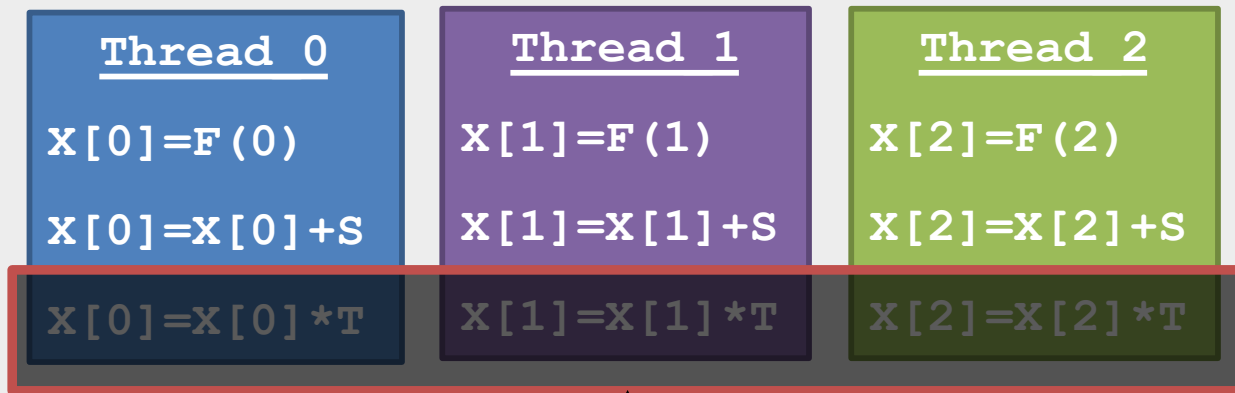
# Matrix Multiplication. Results

- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*
- GPU code:
  - *Name : GeForce GTX 960*
  - *Compute capability : 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6
Shared Memory 1	3151,4	22,14
Shared Memory 2/3	1652	42,5

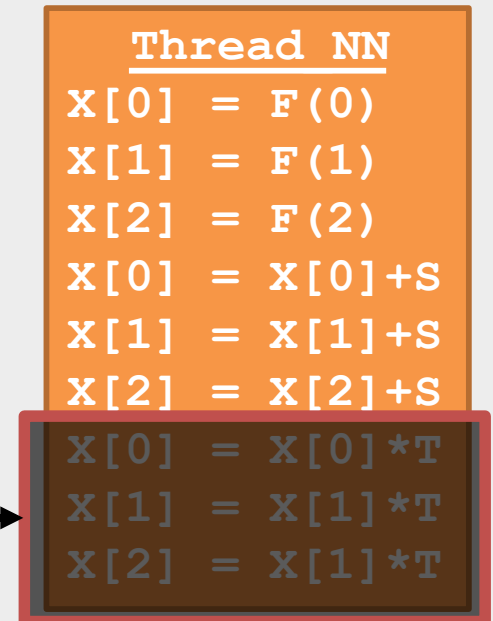


## Thread-Level Parallelism (TLP)



Independent operations

## Instruction-Level Parallelism (ILP)



# Matrix Multiplication. Shared memory. v.4

## Instead of lines in kernel (in v.3)

```
float sum = 0.0f;  
as [ty][tx] = a [ia + n * ty + tx];  
bs [ty][tx] = b [ib + n * ty + tx];  
sum += as [ty][k] * bs [k][tx];  
c [aBegin + bBegin + n * ty + tx] = sum;
```

## Following lines

```
float sum1 = 0.0f, sum2 = 0.0f;  
as[ty][tx] = a[ia + n * ty + tx];  
bs[ty][tx] = b[ib + n * ty + tx];  
as[ty+16][tx] = a[ia + n * (ty + 16) + tx];  
bs[ty+16][tx] = b[ib + n * (ty + 16) + tx];  
sum1 += as[ty][k] * bs[k][tx];  
sum2 += as[ty + 16][k] * bs[k][tx];  
c[aBegin + bBegin + n * ty + tx] = sum1;  
c[aBegin + bBegin + n * (ty + 16) + tx] = sum2;
```

## Add lines in 'main'

```
dim3 threads_4(BLOCK_SIZE, BLOCK_SIZE / 2);  
matMult_Shared_mem4 << <blocks, threads_4 >> > (d_A, d_B, N, d_C);
```



# Matrix Multiplication. Results

- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*
- GPU code:
  - *Name : GeForce GTX 960*
  - *Compute capability : 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6
Shared Memory 1	3151,4	22,14
Shared Memory 2/3	1652	42,5
Shared Memory 4	1421,5	49,1



# Matrix Multiplication. Shared memory. v.5

## Add following lines in kernel

```
float sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f, sum4 = 0.0f;
as[ty][tx] = a[ia + n * ty + tx];
bs[ty][tx] = b[ib + n * ty + tx];
as[ty + 8][tx] = a[ia + n * (ty + 8) + tx];
bs[ty + 8][tx] = b[ib + n * (ty + 8) + tx];
as[ty + 16][tx] = a[ia + n * (ty + 16) + tx];
bs[ty + 16][tx] = b[ib + n * (ty + 16) + tx];
as[ty + 24][tx] = a[ia + n * (ty + 24) + tx];
bs[ty + 24][tx] = b[ib + n * (ty + 24) + tx];
sum1 += as[ty][k] * bs[k][tx];
sum2 += as[ty + 8][k] * bs[k][tx];
sum3 += as[ty + 16][k] * bs[k][tx];
sum4 += as[ty + 24][k] * bs[k][tx];
c[aBegin + bBegin + n * ty + tx] = sum1;
c[aBegin + bBegin + n * (ty + 8) + tx] = sum2;
c[aBegin + bBegin + n * (ty + 16) + tx] = sum3;
c[aBegin + bBegin + n * (ty + 24) + tx] = sum4;
```

## Changes in 'main'

```
dim3 threads_5(BLOCK_SIZE, BLOCK_SIZE / 4);
matMult_Shared_mem4 << <blocks, threads_5 >> > (d_A, d_B, N, d_C);
```



# Matrix Multiplication. Results

- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*
- GPU code:
  - *Name : GeForce GTX 960*
  - *Compute capability : 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6
Shared Memory 1	3151,4	22,14
Shared Memory 2/3	1652	42,5
Shared Memory 4	1421,5	49,1
Shared Memory 5	1258,8	55,4





# 3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility



# GPU Programming Languages

**Numerical analytics** ▶

MATLAB, Mathematica, LabVIEW

**Fortran** ▶

OpenACC, CUDA Fortran

**C** ▶

OpenACC, CUDA C

**C++** ▶

Thrust, CUDA C++

**Python** ▶

PyCUDA, Copperhead

**F#** ▶

Alea.cuBase



# 3 Ways to Accelerate Applications

## Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

Programming  
Languages

Maximum  
Flexibility



# Libraries: Easy, High-Quality Acceleration

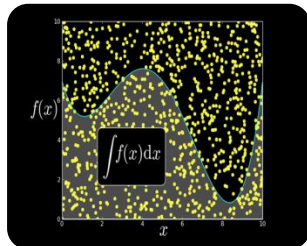
- **Easy of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts



# Some GPU-accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



Vector Signal  
Image Processing



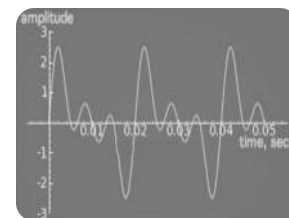
GPU Accelerated  
Linear Algebra



Matrix Algebra  
on GPU and  
Multicore



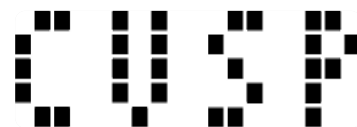
open source  
initiative



NVIDIA cuFFT



ArrayFire Matrix  
Computations



Sparse Linear  
Algebra



C++ STL  
Features for  
CUDA



# Matrix Multiplication. Results

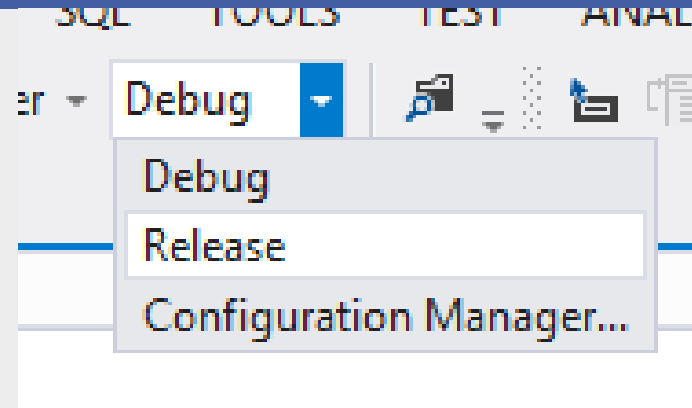
- Sequential code:
  - *CPU: Intel Core i7 58-20K 3,3 GHz*
  - *Dram: 16 GB*
- GPU code:
  - *Name : GeForce GTX 960*
  - *Compute capability: 5.2*

	Execution Time	Rate
Sequential code	69773,2	-
Global Memory	3230	21,6
Shared Memory 1	3151,4	22,14
Shared Memory 2/3	1652	42,5
Shared Memory 4	1421,5	49,1
Shared Memory 5	1258,8	55,4
CUBLAS	188	371



# Just one "but"...

- Sequential code:
  - CPU: Intel Core i7 58-20K 3,3 GHz*
  - Dram: 16 GB*
- GPU code:
  - Name : GeForce GTX 960*
  - Compute capability: 5.2*



	Execution Time (Debug)	Rate (Debug)	Execution Time (Release)	Rate (Release)
Sequential code	69773,2	-	35665,4	-
Global Memory	3230	21,6	188,2	189,5
Shared Memory 1	3151,4	22,14	179	199,2
Shared Memory 2/3	1652	42,5	80,7	442
Shared Memory 4	1421,5	49,1	45,7	780,4
Shared Memory 5	1258,8	55,4	37,3	956,2
CUBLAS	188	371	187,4	190,3



# 3 Ways to Accelerate Applications

## Applications

Libraries

“Drop-in”  
Acceleration

OpenACC  
Directives

Easily Accelerate  
Applications

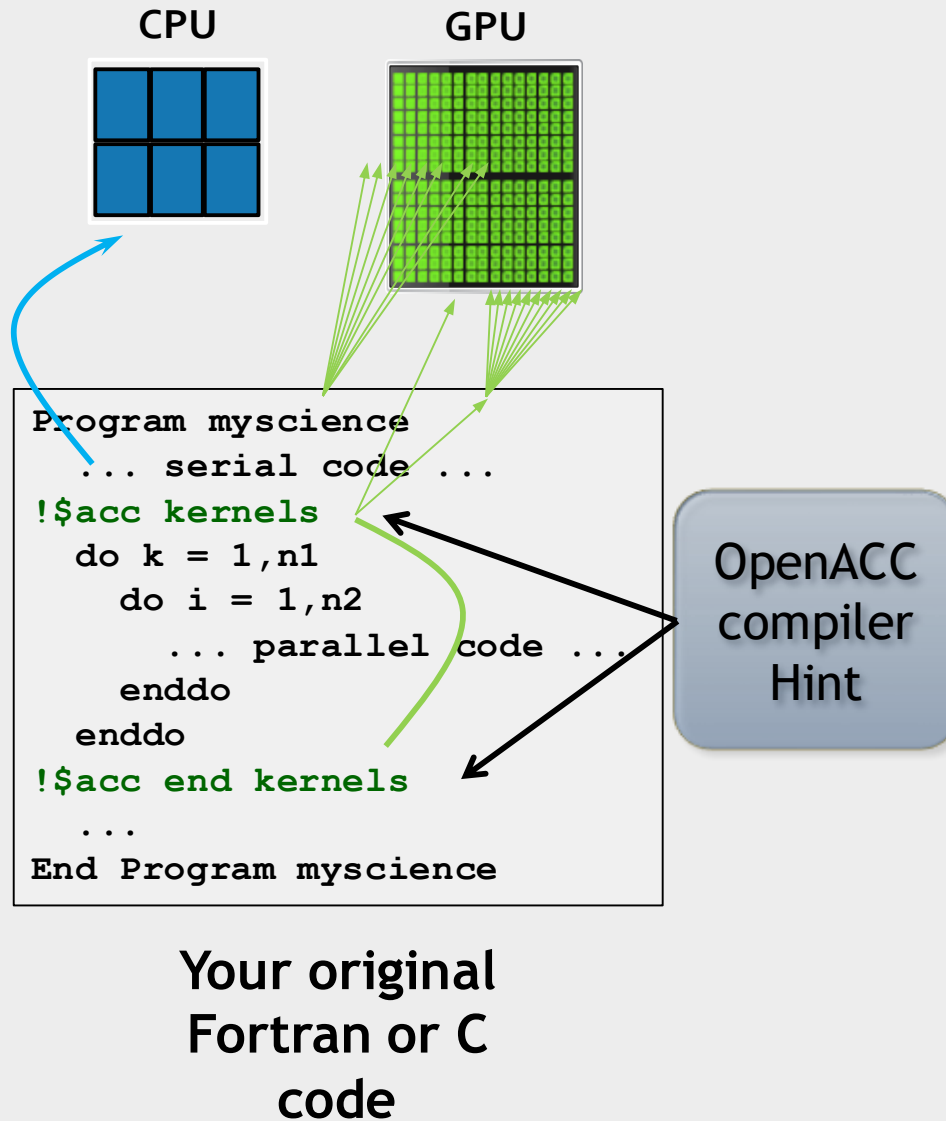
Programming  
Languages

Maximum  
Flexibility





# OpenACC Directives



Simple Compiler hints

Compiler Parallelizes  
code

Works on many-core  
GPUs & multicore CPUs



# OpenACC Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU



# Further read

- CUDA Programming Guide,  
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- CUDA C Best Practices Guide,  
<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>

...

