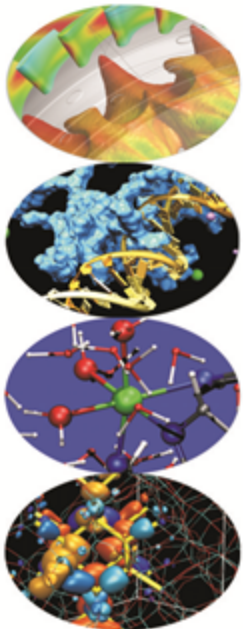
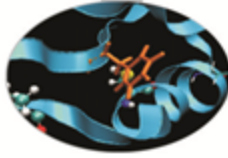


Introduction to GPGPUs and to CUDA programming model

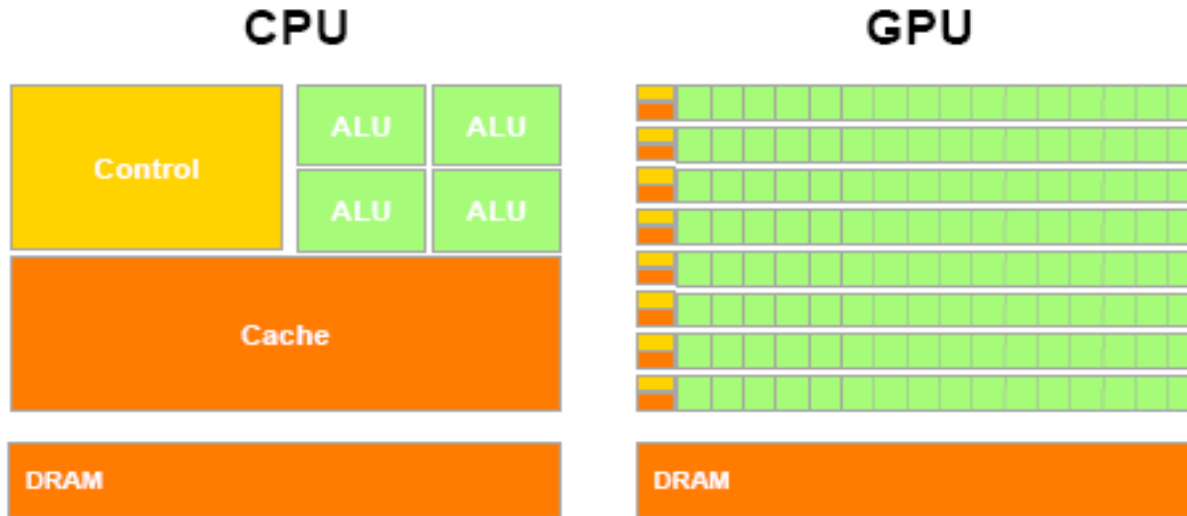
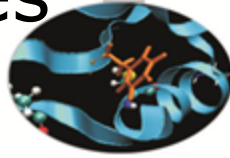


Agenda



- 🔑 GPGPU architecture
- 🔑 CUDA programming model
- 🔑 CUDA efficient programming
- 🔑 Debugging & profiling tools
- 🔑 CUDA libraries

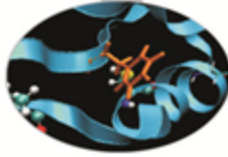
GPU vs CPU: different philosophies



- ⌚ Design of CPUs optimized for sequential code performance:
- ⌚ multi-core
- ⌚ sophisticated control logic unit
- ⌚ large cache memories to reduce access latencies

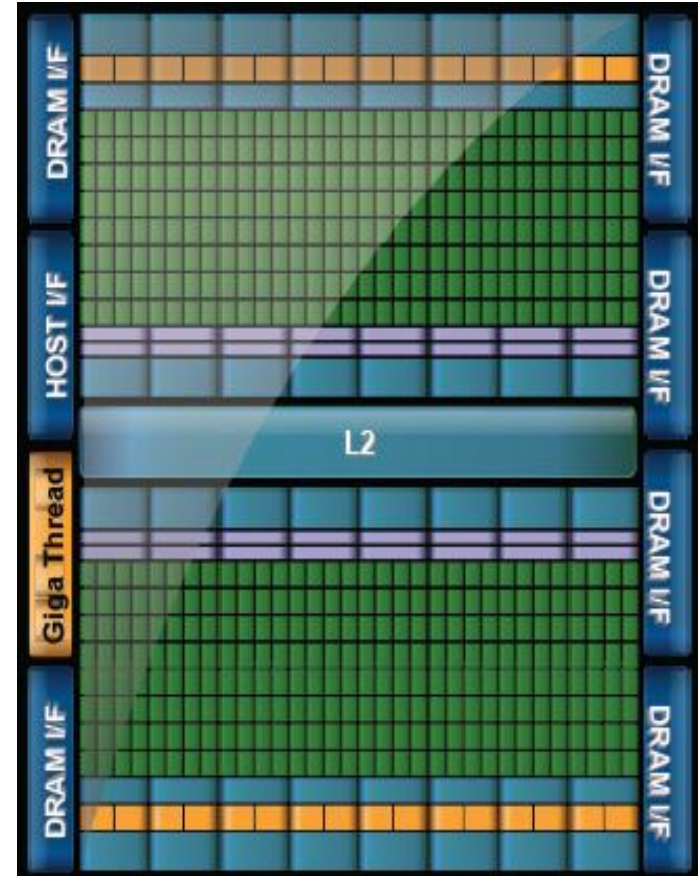
Design of GPUs optimized for the execution of large number of threads dedicated to floating-points calculations:

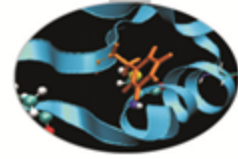
- ⌚ many-cores (several hundreds)
- ⌚ minimized the control logic in order to manage lightweight threads and maximize execution throughput
- ⌚ taking advantage of large number of threads to overcome long-latency memory accesses



Fermi architecture

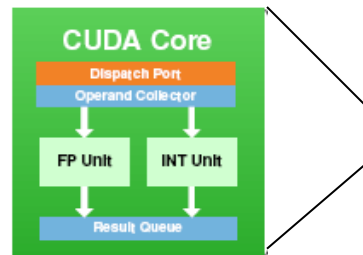
- 512 cores
(16 SM x 32 SP)
- first GPU architecture to support a true cache hierarchy:
 - L1 cache per SM
 - unified L2 caches (768 KB)
- Memory Bandwidth (GDDR5)
148 GB/s (ECC off)
- 6 GB of global memory
- 48KB of shared memory
- Concurrent Kernels execution
- support C++



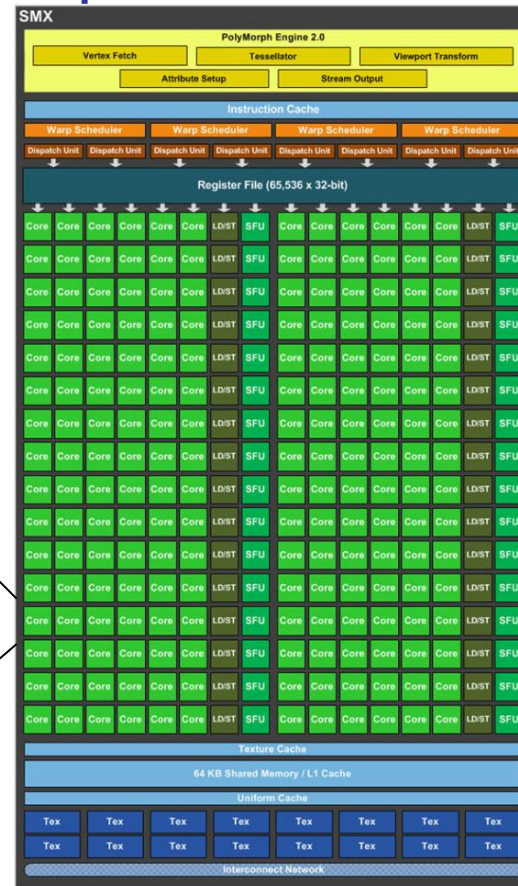


CUDA core architecture

- New IEEE 754-2008 floating point standard**
- Fused multiply-add (FMA) instruction for both single and double precision**
- Newly designed integer ALU optimized for 64-bit and extended precision operations**



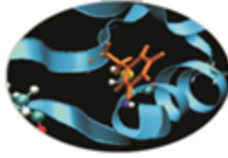
Kepler SMX



Fermi SM

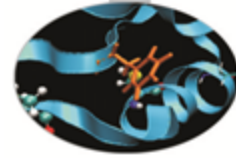


NVIDIA naming



- Mainstream & laptops: GeForce
 - Target: videogames and multi-media
- Workstation: Quadro
 - Target: graphic professionals who use CAD and 3D modeling applications
 - The surcharge is due to more memory and especially the specific drivers for accelerating applications
- GPGPU: Tesla
 - Target: High Performance Computing

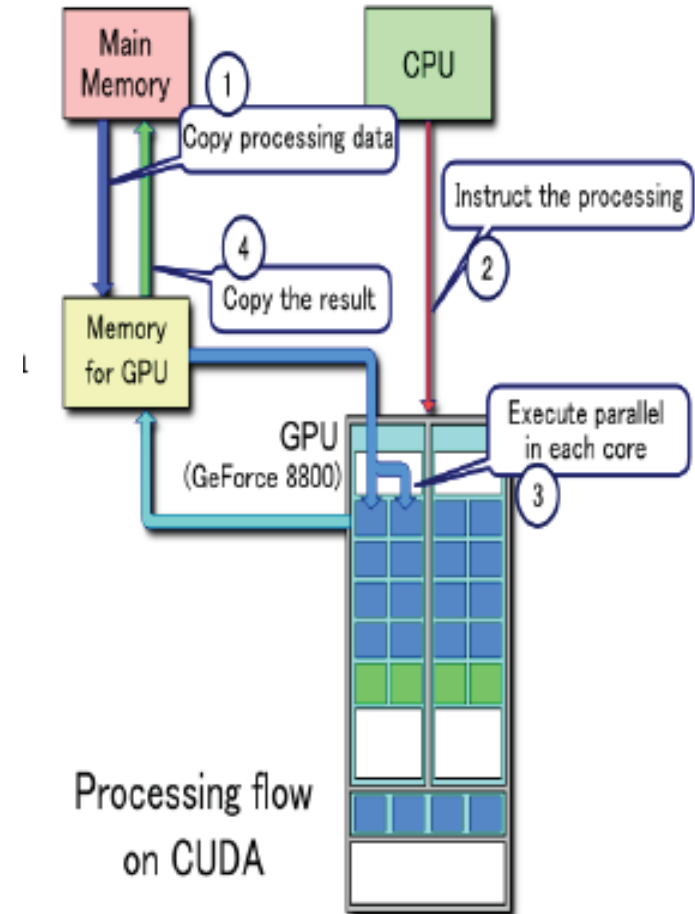
There cannot be a GPU without a CPU

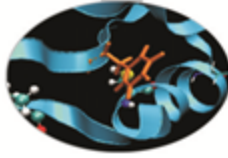


GPUs are designed as numeric computing engines, therefore they will not perform well on other tasks.

Applications should use both CPUs and GPUs, where the latter is exploited as a coprocessor in order to speed up numerically intensive sections of the code by a massive fine grained parallelism.

CUDA programming model introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.





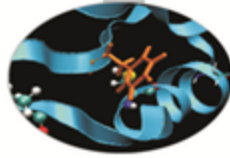
Compute **U**nified **D**evice **A**rchitecture:

- ⌚ extends ANSI C language with minimal extensions
- ⌚ provides application programming interface (API) to manage host and device components

CUDA program:

- ⌚ Serial sections of the code are performed by CPU (**host**)
- ⌚ The parallel ones (that exhibit rich amount of *data parallelism*) are performed by GPU (**device**) in the SIMD mode as **CUDA kernels**.
- ⌚ host and device have separate memory spaces: programmers need to transfer data between CPU and GPU in a manner similar to “one-sided” message passing.

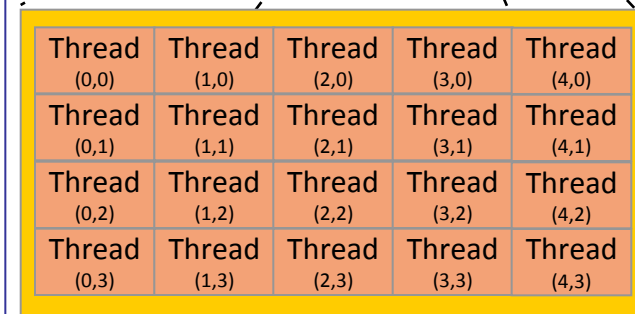
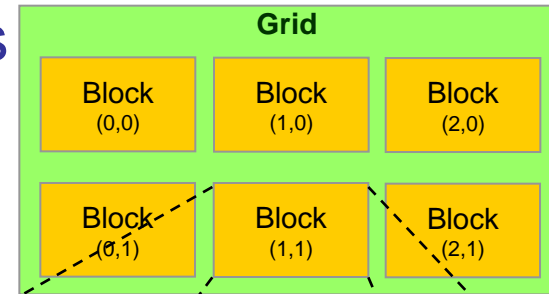
CUDA threads organization



A kernel is executed as a **grid** of many parallel threads.

They are organized into a two-level hierarchy:

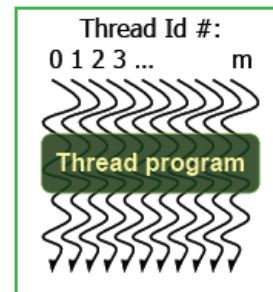
- 🔑 a grid is organized as up to 3-dim *array of thread blocks*
- 🔑 each block is organized into up to 3-dim *array of threads*
- 🔑 all blocks have the same number of threads
- 🔑 organized in the same manner.



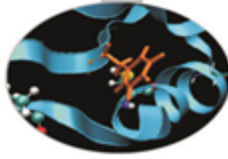
Block of threads:

set of concurrently executing threads that can *cooperate* among themselves through

- 🔑 barrier synchronization, by using the function `__syncthreads ()` ;
- 🔑 shared memory.



CUDA threads organization



Because all threads in a grid execute the same code, they rely on unique coordinates assigned to them by the CUDA runtime system as built-in preinitialized variables

- Block ID up to 3 dimensions:

(`blockIdx.x`, `blockIdx.y`, `blockIdx.z`)

- Thread ID within the block up to 3 dimensions:

(`threadIdx.x`, `threadIdx.y`, `threadIdx.z`)

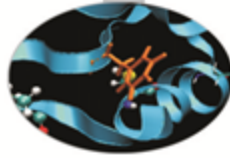
The exact organization of a grid is determined by the execution configuration provided at kernel launch.

Two additional variables of type `dim3` (C struct with 3 unsigned integer fields) are declared:

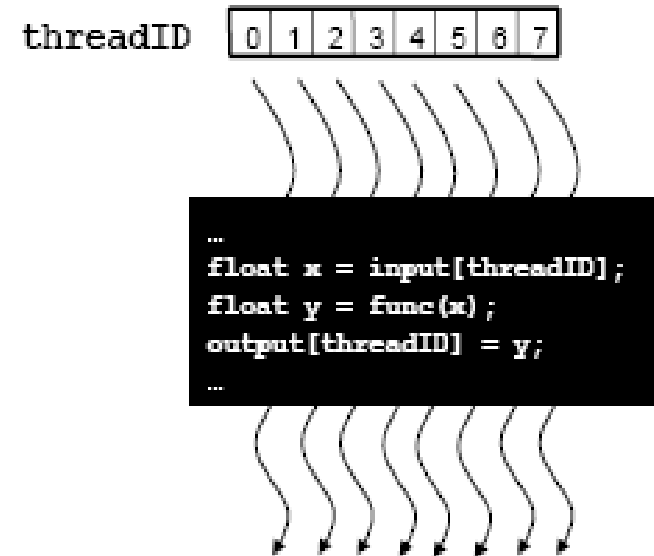
- `gridDim` → dimensions of the grid in terms of number of blocks

- `blockDim` → dimensions of the block in terms of number of threads

Thread ID computation



The built-in variables are used to compute the global ID of the thread, in order to determine the area of data that it is designed to work on.



1D:

$\text{int id} = \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x};$

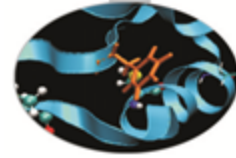
2D:

$\text{int iy} = \text{blockDim.y} * \text{blockIdx.y} + \text{threadIdx.y};$

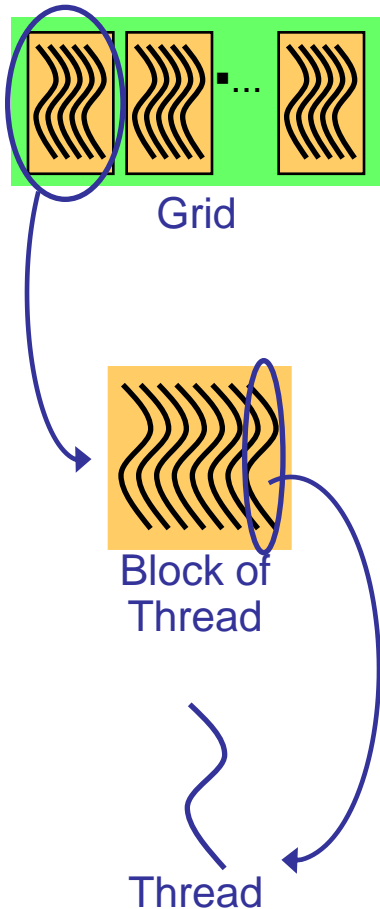
$\text{int ix} = \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x};$

$\text{int id} = \text{iy} * \text{dimx} + \text{ix};$

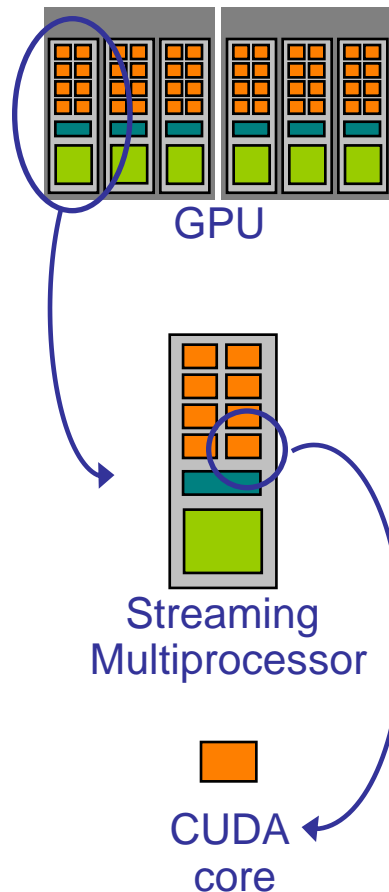
Threads execution model



Software



Hardware

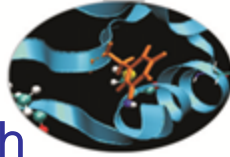


CUDA's hierarchy of threads/memories maps to the hierarchy of processors on the GPU:

- a GPU executes one or more kernel grids;
- a streaming multiprocessor (SM) executes one or more thread blocks;
- a streaming processor (SP) in the SM executes threads.

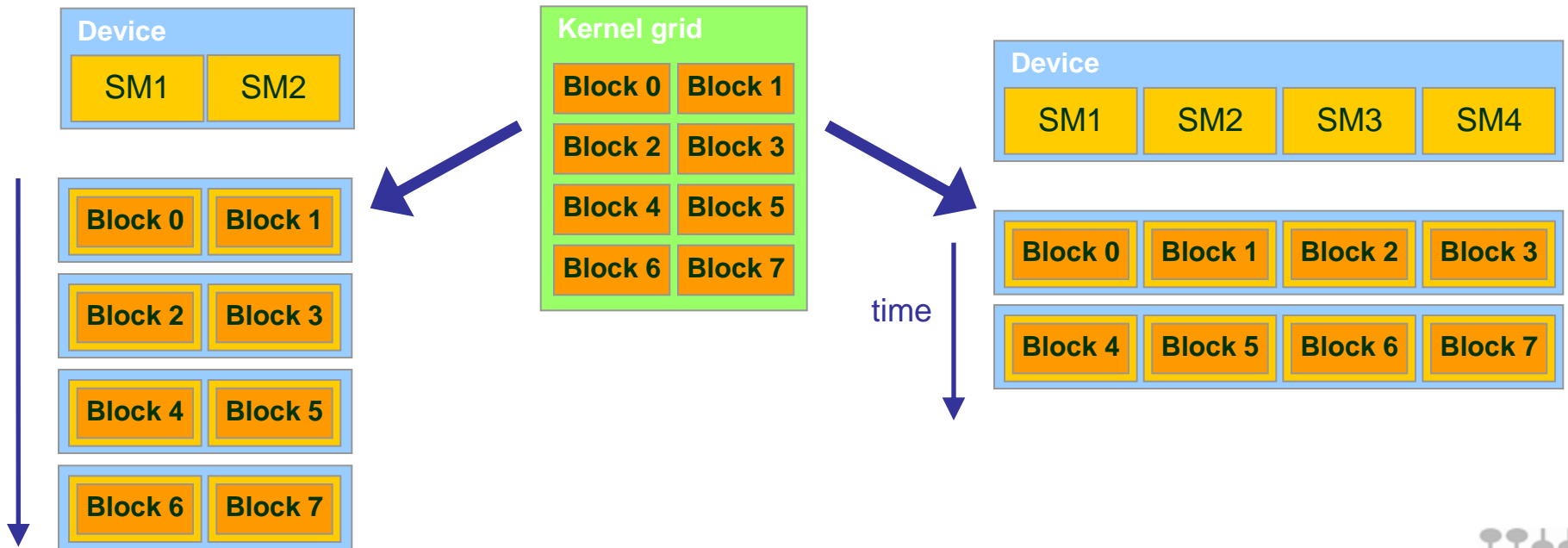
A maximum number of blocks can be assigned to each SM (8 for Fermi, 16 for Kepler)
The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.

Transparent scalability

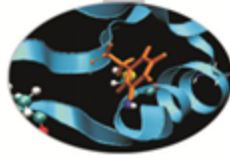


By not allowing threads in different blocks to synchronize with each other, CUDA runtime system can execute blocks in any order relative to each other.

This flexibility enables to execute the same application code on hardware with different numbers of SM (*transparent scalability*).



Launching a kernel



A kernel must be called from the host with the following syntax:

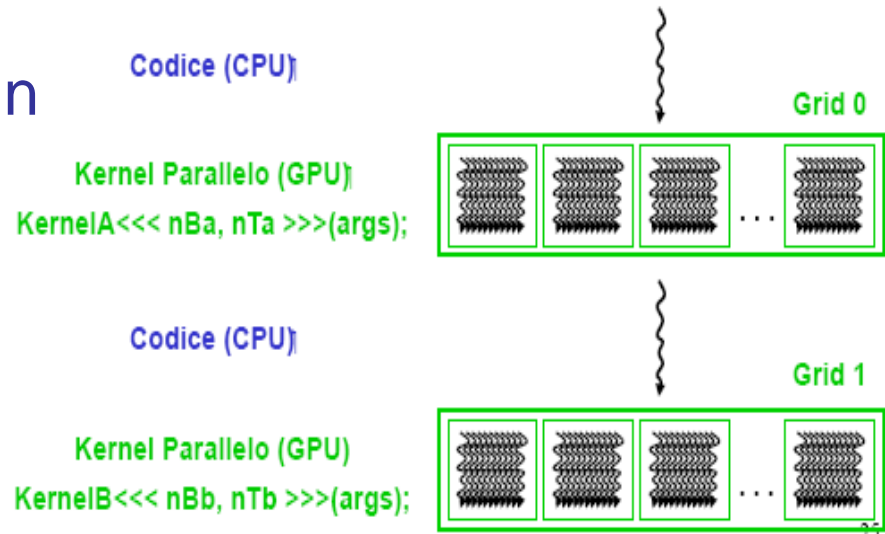
```

__global__ void KernelFunc(...);
dim3 gridDim(100, 50); // 5000 thread blocks
dim3 blockDim(8, 8, 4); // 256 threads per block

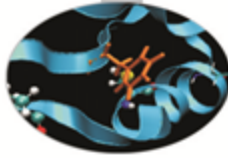
//call the kernel
KernelFunc<<< gridDim, blockDim >>>(<arguments>);
  
```

Typical CUDA grids contain thousands to millions of threads.

All kernel calls are asynchronous!



Kernel example



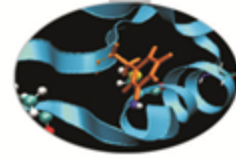
CPU code:

```
void increment_cpu(float* a, float b, int n){
    for (idx=0; idx<n; ++idx)
        a[idx]+=b;
}
int main(void){
    //...
    increment_cpu(h_a,h_b,16);
}
```

GPU code:

```
__global__ increment_gpu(float* a, float b, int n){
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    if (idx < n)
        a[idx]+=b;
}
int main(void){
    //...
    increment_gpu<<<blocks,threads>>>(d_a,d_b,16);
}
```

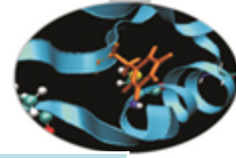
CUDA Function modifiers



CUDA extends C function declarations with three qualifier keywords.

Function declaration	Executed on the	Only callable from the
<code>__device__</code> (<i>device functions</i>)	device	device
<code>__global__</code> (<i>kernel function</i>)	device	host
<code>__host__</code> (<i>host functions</i>)	host	host

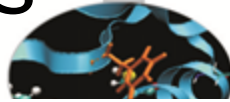
CUDA variable qualifiers



Variable declaration	memory	lifetime	scope
Automatic scalar variables	register	kernel	thread
Automatic array variables <code>__device__ __local__</code>	local	kernel	thread
<code>__device__ __shared__</code>	shared	kernel	block
<code>__device__</code>	global	application	grid
<code>__device__ __constant__</code>	constant	application	grid

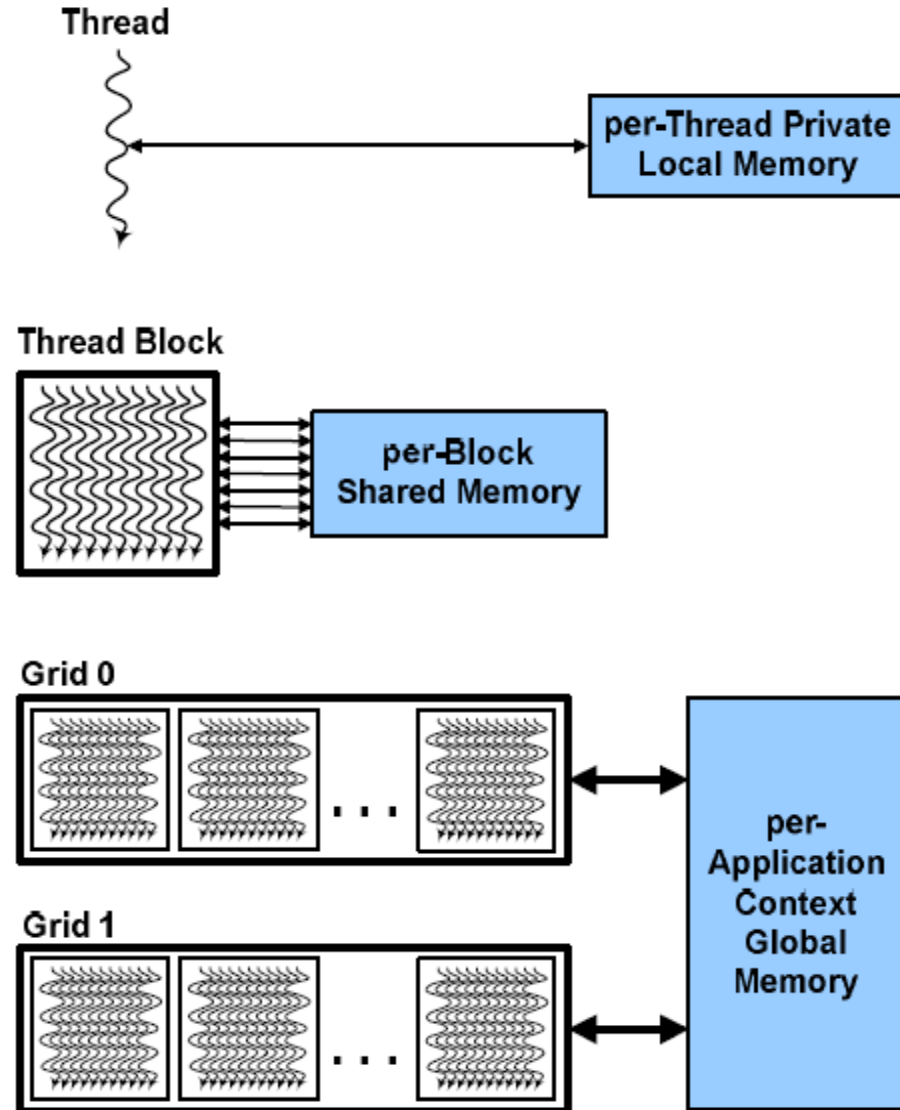
- † Global variables are often used to pass information from one kernel to another.
- † Constant variables are often used for providing input values to kernel functions.

Hierarchy of device memories

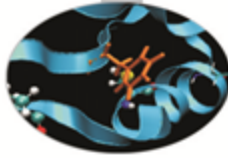


CUDA's hierarchy of threads maps to a hierarchy of memories on the GPU:

- Each thread has some **registers**, used to hold automatic scalar variables declared in kernel and device functions, and a **per-thread private memory space** used for register spills, function calls, and C automatic array variables
- Each thread block has a **per-block shared memory space** used for inter-thread communication, data sharing, and result sharing in parallel algorithms
- Grids of thread blocks share results in **global memory space**



CUDA device memory model

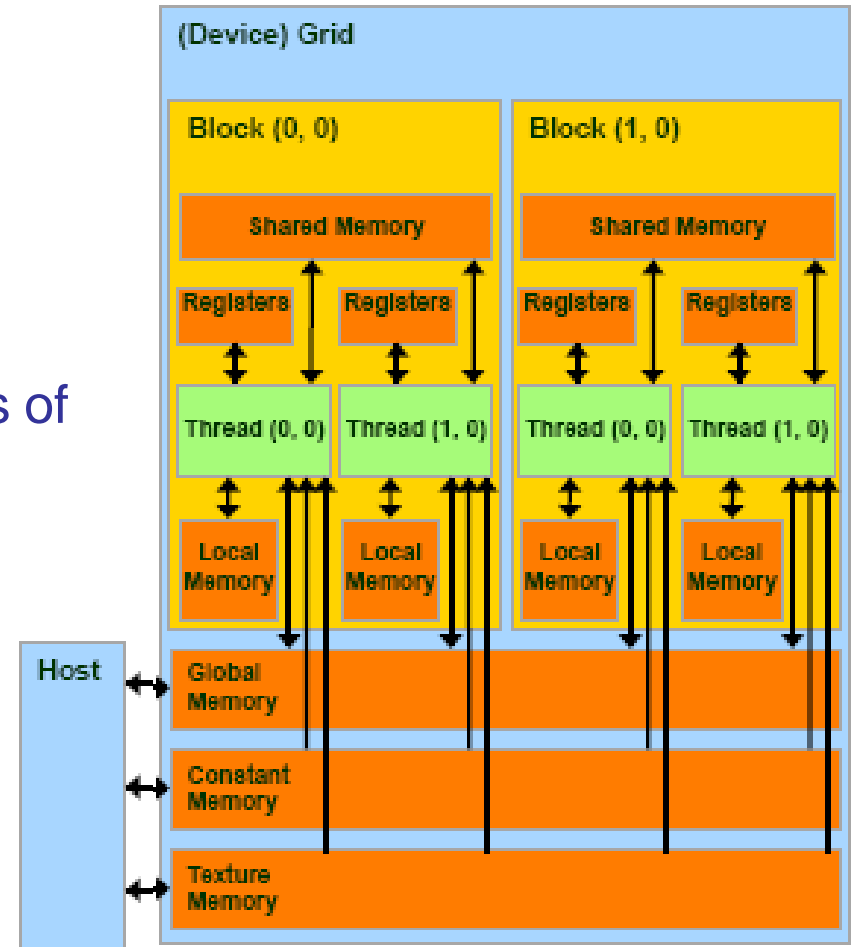


on-chip memories:

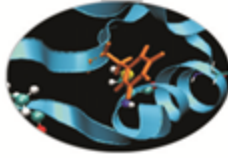
- ⌚ registers (~8KB) → SP
- ⌚ shared memory (~16KB) → SM
- ⌚ they can be accessed at very high speed in a highly parallel manner.

per-grid memories:

- ⌚ global memory (~4GB)
 - ⌚ long access latencies (hundreds of clock cycles)
 - ⌚ finite access bandwidth
- ⌚ constant memory (~64KB)
 - ⌚ read only
 - ⌚ short-latency (cached) and high bandwidth when all threads simultaneously access the same location
- ⌚ texture memory (read only)
- ⌚ CPU can transfer data to/from all per-grid memories.



Local memory is implemented as part of the global memory, therefore has a long access latencies too.



Static modality

inside the kernel:

```
__shared__ float f[100];
```

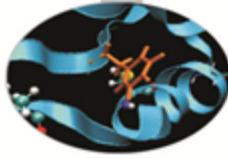
Dynamic modality

in the execution configuration of the kernel,
define the number of bytes to be allocated per
block in the shared memory :

```
kernel<<<DimGrid, DimBlock, SharedMemBytes>>>(...);
```

while inside the kernel:

```
extern __shared__ float f[ ];
```



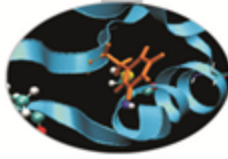
CUDA API functions to manage data allocation on the device global memory:

cudaMalloc(void** bufferPtr, size_t n)

- ‡ It allocates a buffer into the device global memory
- ‡ The first parameter is the address of a generic pointer variable that must point to the allocated buffer
 - ‡ it should be cast to (void**)!
- ‡ The second parameter is the size of the buffer to be allocated, in terms of bytes

cudaFree(void* bufferPtr)

- ‡ It frees the storage space of the object



```
cudaMemset(void* devPtr, int value, size_t count)
```

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte of the `int value` converted to unsigned char.

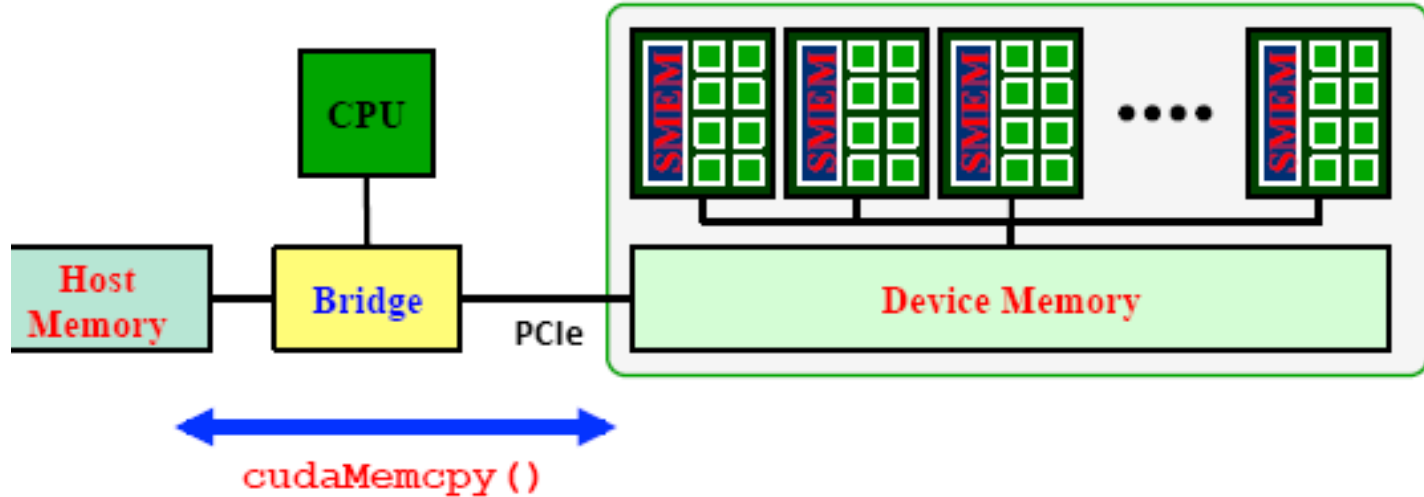
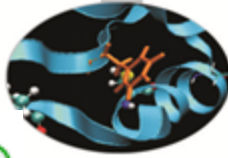
CUDA version of the C `memset()` function.

devPtr - Pointer to device memory

value - Value to set for each byte of specified memory

count - Size in bytes to set

Data transfer CPU-GPU



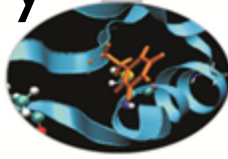
API **blocking** functions for data transfer between memories:

```
cudaMemcpy(dM, M, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(M, dM, size, cudaMemcpyDeviceToHost);
```

↙
↑
↖
↖

Destination source data number of bytes symbolic constant indicating the direction



```
cudaMemcpyToSymbol(const char * symbol,  
                  const void * src,  
                  size_t count,  
                  size_t offset,  
                  enum cudaMemcpyKind kind)
```

symbol - symbol destination on device, it can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space.

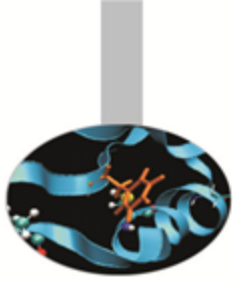
src - source memory address

count - size in bytes to copy

offset - offset from start of symbol in bytes

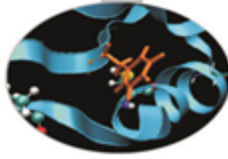
kind - type of transfer, it can be either

cudaMemcpyHostToDevice or
cudaMemcpyDeviceToDevice



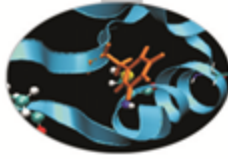
Device management

- Application can query and select GPUs
 - `cudaGetDeviceCount(int *count)`
 - `cudaSetDevice(int device)`
 - `cudaGetDevice(int *device)`
 - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
- Multiple threads can share a device
- A single thread can manage multiple devices
 - `cudaSetDevice(i)` to select current device
 - `cudaMemcpy(...)` for peer-to-peer copies



Device management (sample code)

```
int cudadevice;  
  
struct cudaDeviceProp prop;  
  
cudaGetDevice( &cudadevice );  
  
cudaGetDeviceProperties (&prop, cudadevice);  
  
mpc=prop.multiProcessorCount;  
  
mtpb=prop.maxThreadsPerBlock;  
  
shmsize=prop.sharedMemPerBlock;  
  
printf("Device %d: number of multiprocessors %d\n , max number of threads per  
    block %d\n, shared memory per block %d\n", cudadevice, mpc, mtpb, shmsize);
```



All runtime functions return an error code of type:

`cudaError_t`.

No error is indicated as `cudaSuccess`.

```
char* cudaGetErrorString(cudaError_t code)
```

returns a string describing the error:

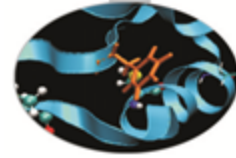
For asynchronous functions (i.e. kernels, asynchronous copies) the only way to check for errors just after the call is to synchronize: `cudaDeviceSynchronize()`

Then the following function returns the code of the last error:

```
cudaError_t cudaGetLastError()
```

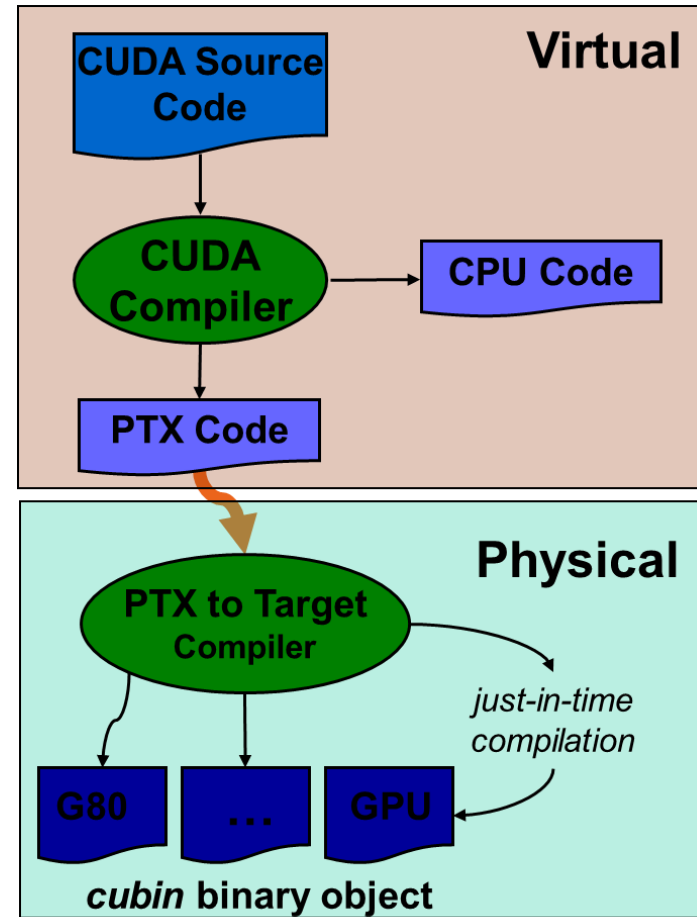
```
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

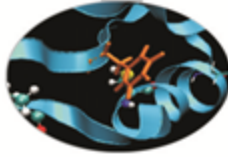
NVIDIA C compiler



nvcc front-end for compilation:

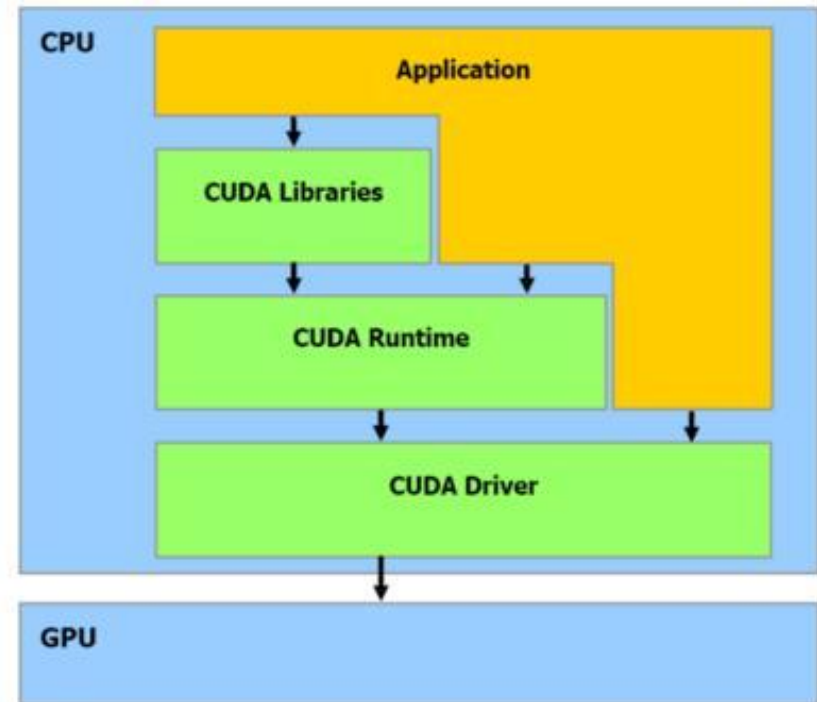
- ✦ separates GPU code from CPU code
- ✦ CPU code -> C/C++ compiler (Microsoft Visual C/C++, GCC, ecc.)
- ✦ GPU code is converted in an intermediate assembly language: PTX, then in binary form (the *cubin* object)
- ✦ link all executables

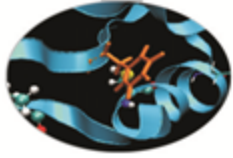




CUDA Driver Vs Runtime API

- † CUDA is composed of two APIs:
 - ‡ the CUDA runtime API
 - ‡ the CUDA driver API
- † They are mutually exclusive
- † Runtime API:
 - ‡ easier to program
 - ‡ it eases device code management: it's where the C-for-CUDA language lives
- † Driver API:
 - ‡ requires more code: no syntax sugar for the kernel launch, for example
 - ‡ finer control over the device especially in multithreaded application
 - ‡ doesn't need nvcc to compile the host code.

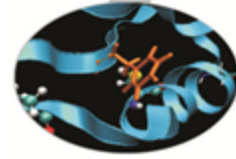




CUDA Driver API

- The driver API is implemented in the **nvcuda** dynamic library. All its entry points are prefixed with **cu**.
- It is a handle-based, imperative API: most objects are referenced by opaque handles that may be specified to functions to manipulate the objects.
- The driver API must be initialized with **cuInit()** before any function from the driver API is called. A **CUDA context** must then be created that is attached to a specific device and made current to the calling host thread.
- Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code**.
- Kernels are launched using API entry points.

**by the way, any application that wants to run on future device architectures must load PTX, not binary code



Vector add: driver Vs runtime API

// driver API

// initialize CUDA

```
err = cuInit(0);  
err = cuDeviceGet(&device, 0);  
err = cuCtxCreate(&context, 0, device);
```

// setup device memory

```
err = cuMemAlloc(&d_a, sizeof(int) * N);  
err = cuMemAlloc(&d_b, sizeof(int) * N);  
err = cuMemAlloc(&d_c, sizeof(int) * N);
```

// copy arrays to device

```
err = cuMemcpyHtoD(d_a, a, sizeof(int) * N);  
err = cuMemcpyHtoD(d_b, b, sizeof(int) * N);
```

// prepare kernel launch

```
kernelArgs[0] = &d_a;  
kernelArgs[1] = &d_b;  
kernelArgs[2] = &d_c;
```

// load device code (PTX or cubin. PTX here)

```
err = cuModuleLoad(&module, module_file);  
err = cuModuleGetFunction(&function, module, kernel_name);
```

// execute the kernel over the <N,1> grid

```
err = cuLaunchKernel(function, N, 1, 1, // Nx1x1 blocks  
                    1, 1, 1, // 1x1x1 threads  
                    0, 0, kernelArgs, 0);
```

// runtime API

// setup device memory

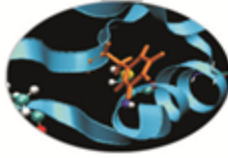
```
err = cudaMalloc((void**)&d_a, sizeof(int) * N);  
err = cudaMalloc((void**)&d_b, sizeof(int) * N);  
err = cudaMalloc((void**)&d_c, sizeof(int) * N);
```

// copy arrays to device

```
err=cudaMemcpy(d_a, a, sizeof(int) * N, cudaMemcpyHostToDevice);  
err=cudaMemcpy(d_b, b, sizeof(int) * N, cudaMemcpyHostToDevice);
```

// launch kernel over the <N, 1> grid

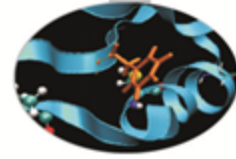
```
matSum<<<N,1>>>(d_a, d_b, d_c); // yum, syntax sugar!
```



The Open Computing Language: OpenCL

- † OpenCL is an open standard for cross-platform, parallel programming of modern processors. i.e, multi core CPU and GPGPU . OpenCL is a low-level C API (but C++ bindings are also available)
- † it can be used to program heterogeneous computer architecture (multicore CPU + accelerator, OCL slogan: '*program once, run everywhere*')
- † it can be used to program NVIDIA GPU, AMD GPU or even Imagination Technology GPU (i.e. you don't need to get married with NVIDIA GeForce/Tesla/Quadro products)
- † So, how does the OpenCL framework look like?
 - ‡ it supports the data parallel programming paradigm
 - ‡ it has its dialects: a **CUDA grid** translates into a **NDRange**, a **warp** becomes a **wavefront** and so on...
 - ‡ From a programmer point of view: it very closely resembles the CUDA driver API

Vector add: OCL Host Code



// initialize OpenCL

```

err = clGetPlatformIDs(1, &cpPlatform, NULL);
err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
queue = clCreateCommandQueue(context, device_id, 0, &err);
  
```

// setup device memory

```

d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, &err);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, &err);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, &err);
  
```

// copy array to the device

```

err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, bytes, h_a, 0, 0, 0, &err);
err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0, bytes, h_b, 0, 0, 0, &err);
  
```

// prepare kernel launch

```

err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
  
```

// load, *COMPILE* and *LINK* device code

```

program = clCreateProgramWithSource(context, 1,
                                     (const char **) & kernelSource, NULL, &err);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "vecAdd", &err);
  
```

// Execute the kernel over the NDRange

```

err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global,
                              0, NULL, NULL);
  
```

// initialize CUDA

```

err = cuInit(0);
err = cuDeviceGet(&device, 0);
err = cuCtxCreate(&context, 0, device);
  
```

// setup device memory

```

err = cuMemAlloc(&d_a, sizeof(int) * N);
err = cuMemAlloc(&d_b, sizeof(int) * N);
err = cuMemAlloc(&d_c, sizeof(int) * N);
  
```

// copy arrays to the device

```

err = cuMemcpyHtoD(d_a, a, sizeof(int) * N);
err = cuMemcpyHtoD(d_b, b, sizeof(int) * N);
  
```

// prepare kernel launch

```

kernelArgs[0] = &d_a;
kernelArgs[1] = &d_b;
kernelArgs[2] = &d_c;
  
```

// load device code (PTX or cubin. PTX here)

```

err = cuModuleLoad(&module, module_file);
err = cuModuleGetFunction(&function, module, kernel_name);
  
```

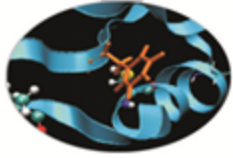
// execute the kernel over the <N,1> grid

```

err = cuLaunchKernel(function, N, 1, 1, // Nx1x1 blocks
                    1, 1, 1, // 1x1x1 threads
                    0, 0, kernelArgs, 0);
  
```

There ain't such thing as a stand-alone OpenCL compiler

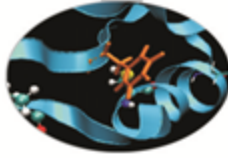
Vector add: OCL Device Code



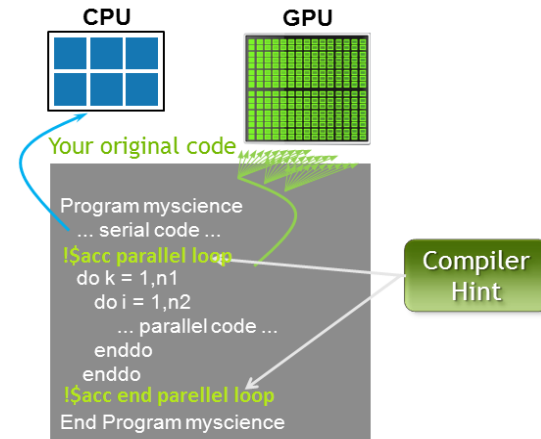
```
// OpenCL kernel. Each work item takes care of one element of c
const char *kernelSource =
"__kernel void vecAdd( __global double *a,
"                    __global double *b,
"                    __global double *c)
"{
" //Get our global thread ID
" int id = get_global_id(0);
"
" //Make sure we do not go out of bounds
" if (id < n)
"     c[id] = a[id] + b[id];
"}
"\n" ;
```

- † The kernel code, again, looks very similar to the CUDA counter part
- † Indeed, OpenCL/CUDA similarities code are so strong that a source-to-source translator is available (CU2CL)
- † **But** there is not such a thing as an OpenCL compiler:
 - † the compilation and linking of the kernel has to be done at runtime
 - † An OpenCL kernel is a string within an OpenCL host code

OpenACC

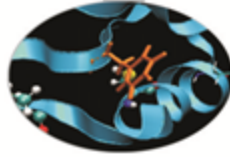


- OpenACC is a open parallel programming standard designed to easily take advantage of the heterogeneous CPU/GPU computing systems.
- OpenACC allows parallel programmers to provide simple hints, known as “directives,” to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself.
- OpenACC 1.0 (<http://www.openacc-standard.org>)
- Implementations available from PGI, Cray, and CAPS
- Will be rolled into OpenMP 4.0



Key Advantages:

- High-Level: No involvement of OpenCL, CUDA, etc.
- Single source: Compile the same program for accelerators or serial (NO separate GPU code).
- Portable: Supports GPU accelerators and co-processors from multiple vendors, current and future versions.



```
pgcc -acc -ta=nvidia -Minfo=accel saxpy.c
```

saxpy:

- 3, Generating present_or_copyin(x[0:n])
Generating present_or_copy(y[0:n])
Generating compute capability 1.0 binary
Generating compute capability 2.0 binary

- 4, **Loop is parallelizable**
Accelerator kernel generated

Compiler was able to parallelize

- 4, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
CC 1.0 : 8 registers; 48 shared, 0 constant, 0 local memory bytes
CC 2.0 : 12 registers; 0 shared, 64 constant, 0 local memory bytes

```
int main(){

    int N = 1<<10;

    float *x, *y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f; y[i] = 1.0f;
    }

    saxpy(N, 1.0f, x, y);

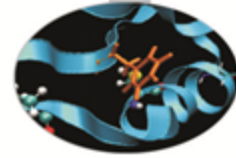
    return 0;
}
```

```
void saxpy (int n, float a,
            float *x, float *restrict y)
{

    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];

}
```

OpenAcc performance



CPU: Intel Xeon X5680
 6 Cores @ 3.33GHz
 GPU: NVIDIA Tesla M2070

Example: Laplace equation in 2D

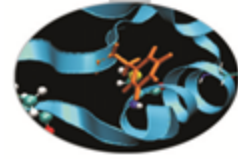
Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	13.65	2.9x

SpeedUp vs 1 CPU core

SpeedUp vs 6 CPU cores

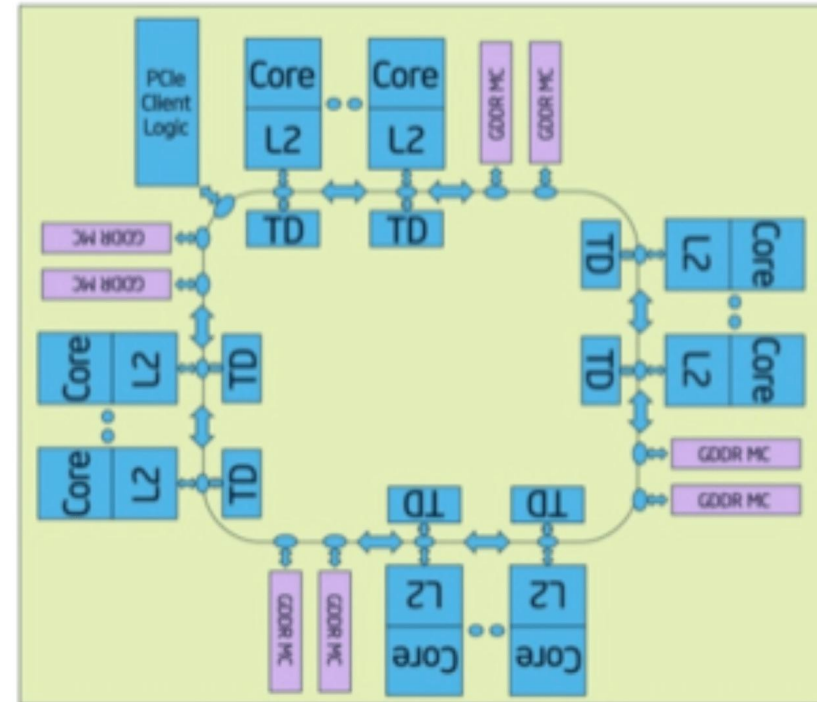
Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU

SpeedUp = 4x

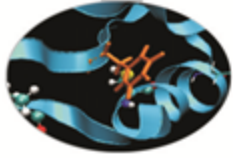


Many Integrated Core (MIC): the Intel answer to GPGPU computing

- The Intel Xeon Phi KNC (Knight Corner*) is a (60+1)-x86-core SMP chip. 60 cores are available for computation, 1 is reserved for the system.
- each core has a 512-bit wide SSE vector unit
- all cores are connected by a bi-directional 512 bit ring bus
- 512 KB of L2 cache and 32KB of I/D L1 (each core)
- A Xeon Phi KNC is packaged into a PCIe add-on card together with 8 GB of GDDR5 dedicated ram (theoretical peak perf. : 352 GB/s. actual peak with ECC: 200 GB/s)
- Intel claims: 'Up to **1 teraflops** of double precision performance'

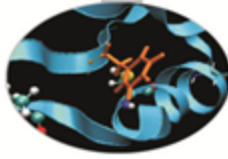


**Knight Corner* is the codename of the first-gen Intel MIC architecture processor. Second-gen MIC codename will be *Knight Landing*.



Intel Xeon Phi: programming model

- Familiar OpenMP (or pthreads), MPI programming model
 - no new language or new parallel programming paradigm to learn: what you already know about parallel programming is basically all that you need to start programming a Xeon Phi processor.
- OpenCL support to the MIC architecture is on its way
 - can help the porting of CUDA application. CUDA→OpenCL MIC with the CU2CL source-to-source translator, for example.
- Porting an existing OpenMP/MPI application onto a Xeon Phi processor can be as easy as to recompile the application with a couple of new MIC-specific pragmas and compiler flag activated,
 - *but* to make that application running at full speed on the Xeon Phi chip a little more effort is probably needed.



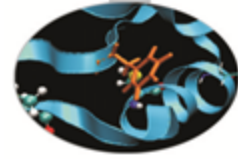
Intel Xeon Phi: accessing modes

† Offload mode:

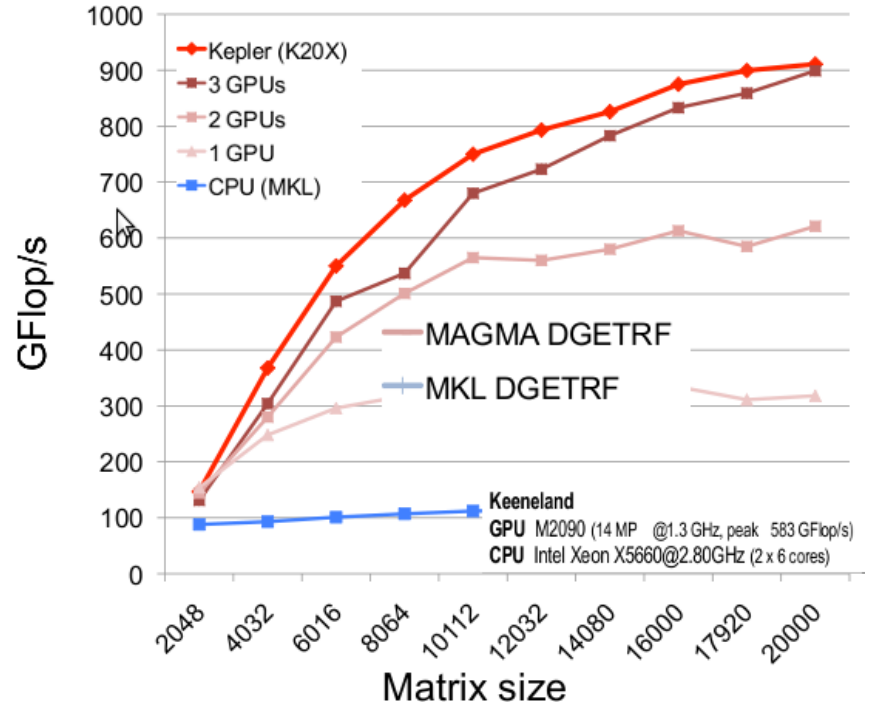
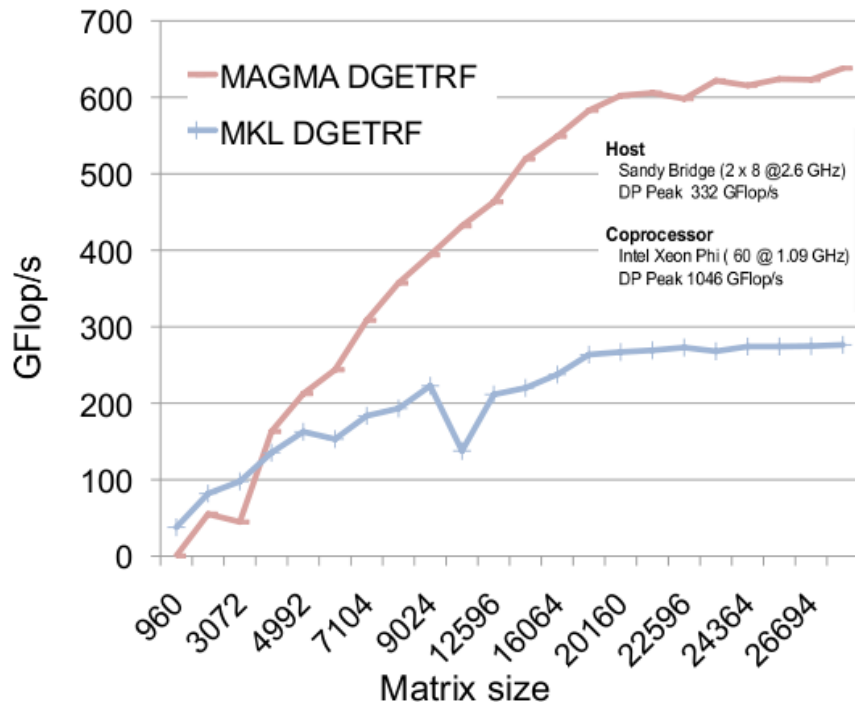
- ‡ using pragmas to augment existing codes so they offload work from the host processor to the Intel Xeon Phi coprocessors(s)
- ‡ Accessing the coprocessor as an accelerator through optimized libraries such as the Intel MKL (Math Kernel Library)

† Native mode:

- ‡ Recompiling source code to run the entire application directly on coprocessor as a separate many-core Linux SMP compute node
- ‡ Using each coprocessor as a node in an MPI cluster or, alternatively, as a device containing a cluster of MPI nodes



Intel Xeon Phi Vs Nvidia K20. MAGMA LU factorization



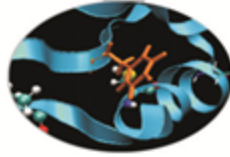
these two images are taken from the presentations:

http://icl.cs.utk.edu/projectsfiles/magma/pubs/25-MAGMA_1.3_SC12.pdf

http://icl.cs.utk.edu/projectsfiles/magma/pubs/24-MAGMA_MIC_03.pdf

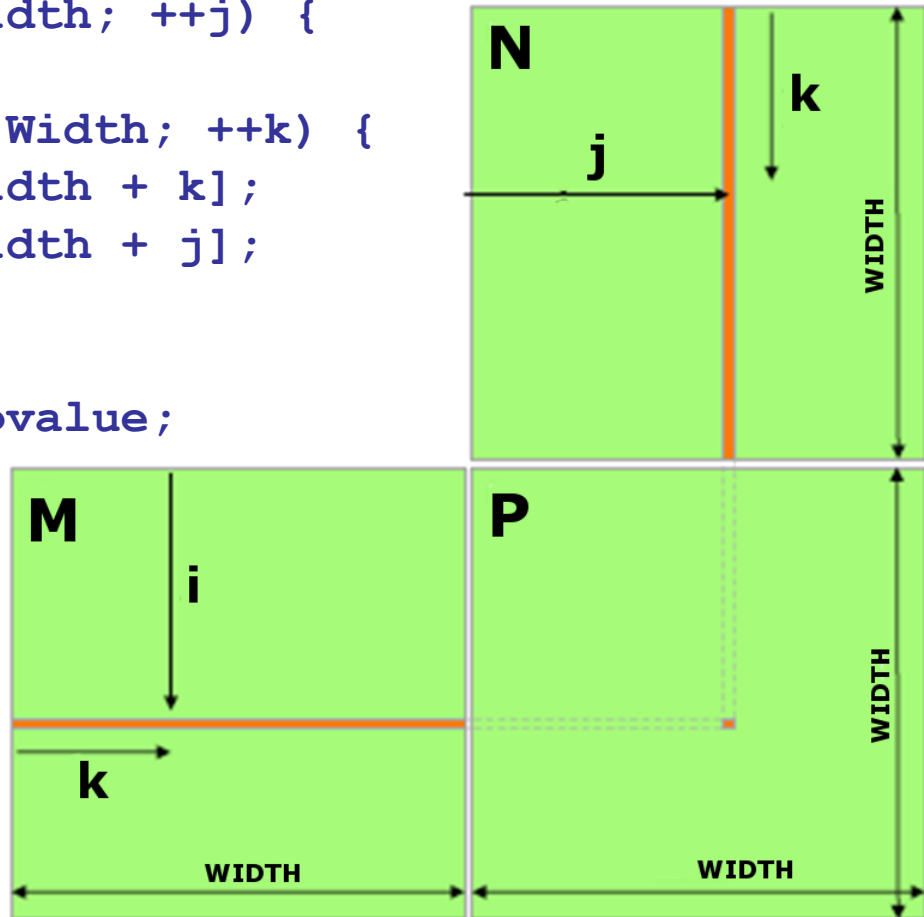
authors: ICL-group@University of Tennessee. The ICL-group is actively developing the MAGMA library which is a world-class performance open source Linear Algebra library for multicore+accelerator computer architecture: <http://icl.cs.utk.edu/magma/index.html>

Matrix-Matrix multiplication example



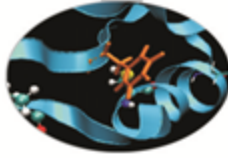
```
void MatrixMulOnHost(float* M, float* N, float* P,
                    int Width) {
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            float pvalue = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * Width + k];
                float b = N[k * Width + j];
                pvalue += a * b;
            }
            P[i * Width + j] = pvalue;
        }
    }
}
```

$$P = M * N$$



CUDA parallelization: each thread computes an element of P

Matrix-Matrix multiplication *device code*

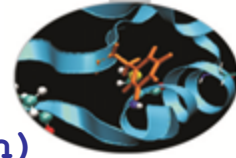


```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int
width)
{
    // 2D thread ID
    int col = threadIdx.x;
    int row = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the
    // thread
    float Pvalue = 0;
    for (int k=0; k < width; k++)
        Pvalue += Md[row * width + k] * Nd[k * width + col];

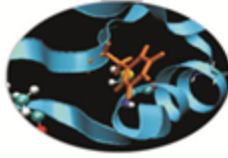
    // write the matrix to device memory
    // (each thread writes one element)
    Pd[row * width + col] = Pvalue;
}
```

Matrix-Matrix multiplication *host code*



```
void MatrixMultiplication(float* M, float *N, float *P, int width)
{
    size_t size = width*width*sizeof(float);
    float* Md, Nd, Pd;
    // allocate M, N and P on the device
    cudaMalloc((void**)&Md, size);
    cudaMalloc((void**)&Nd, size);
    cudaMalloc((void**)&Pd, size);
    // transfer M and N to the device memory
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // kernel invocation
    dim3 gridDim(1,1);
    dim3 blockDim(width,width);
    MNKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);
    // transfer P from the device to the host
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Matrix-Matrix multiplication example

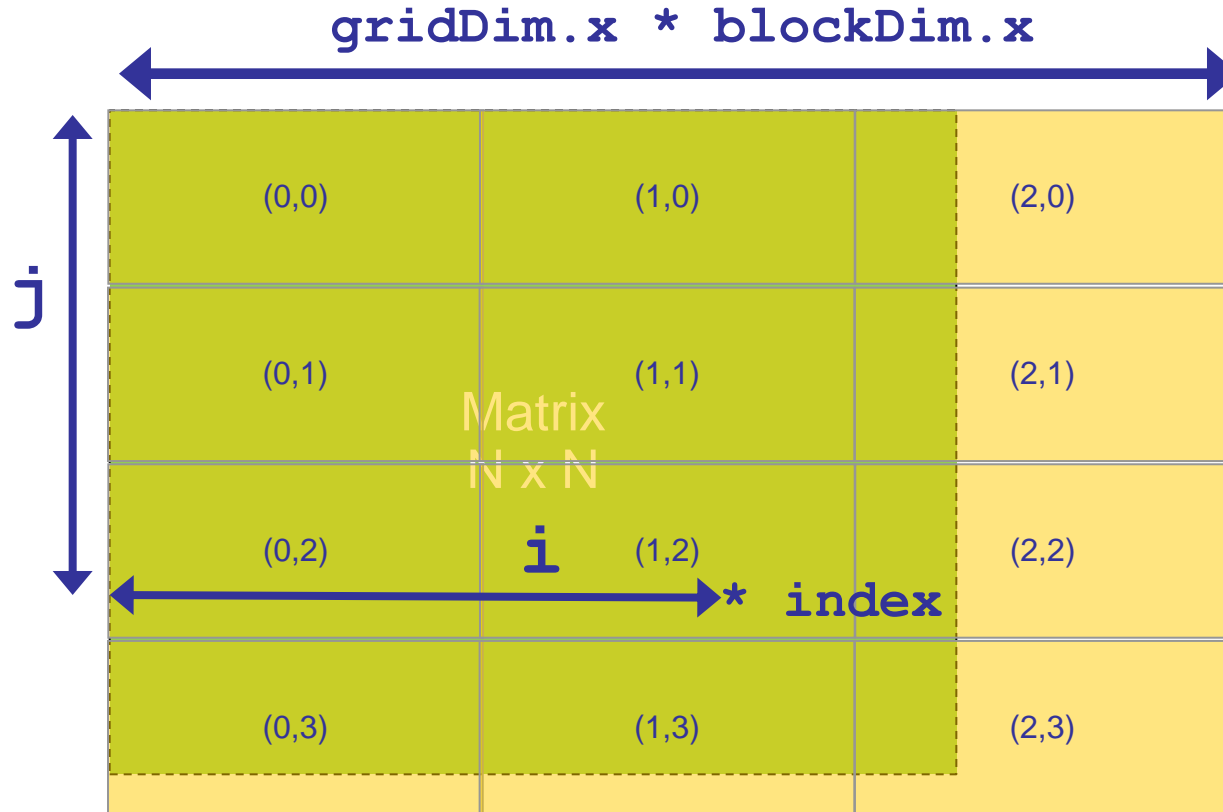
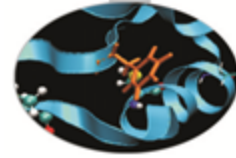


Limitation: a block can have up to 1024 threads (for Fermi and Kepler). Therefore the previous implementation can compute square matrices of order less or equal to 32.

Improvement:

- use more blocks by breaking matrix Pd into square tiles
- all elements of a tile are computed by a block of threads
- each thread still calculates one Pd element but it uses its *blockIdx* values to identify the tile that contains its element.

Matrix-Matrix multiplication example



```

i = blockIdx.x * blockDim.x + threadIdx.x;
j = blockIdx.y * blockDim.y + threadIdx.y;

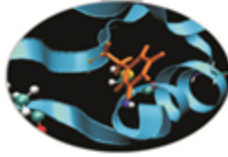
```

```

index = j * gridDim.x * blockDim.x + i;

```

Matrix-Matrix multiplication example



```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int
width)
{
    // 2D thread ID
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;

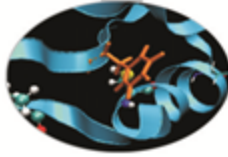
    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;
    for (int k=0; k < width; k++)
        Pvalue += Md[row * width + k] * Nd[k * width + col];

    Pd[row * width + col] = Pvalue;
}
```

Kernel invocation:

```
dim3 gridDim(width/TILE_WIDTH,width/TILE_WIDTH);
dim3 blockDim(TILE_WIDTH,TILE_WIDTH);
MNKernel<<<dimGrid, dimBlock>>>(Md,Nd,Pd,width);
```

Matrix-Matrix multiplication example



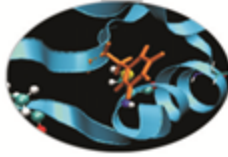
Which is the optimal dimension of the block (i.e. TILE_WIDTH)?

Knowing that each SM of a Fermi can have up to 1536 threads, we have

- $8 \times 8 = 64$ threads $\Rightarrow 1536/64 = 24$ blocks to fully occupy an SM; but we are limited to 8 blocks in each SM therefore we will end up with only $64 \times 8 = 512$ threads in each SM.
- $16 \times 16 = 256$ threads $\Rightarrow 1536/256 = 6$ blocks we will have full thread capacity in each SM.
- $32 \times 32 = 1024$ threads $\Rightarrow 1536/1024 = 1.5 \Rightarrow 1$ block.

 **TILE_WIDTH = 16**

Matrix-Matrix multiplication example

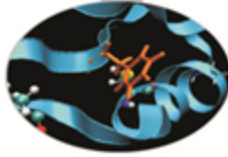


Which is the optimal dimension of the block (i.e. TILE_WIDTH)?

Knowing that each SM of a Kepler can have up to 2048 threads, we have

- $8 \times 8 = 64$ threads $\Rightarrow 2048/64 = 32$ blocks to fully occupy an SM; but we are limited to 16 blocks in each SM therefore we will end up with only $64 \times 16 = 1024$ threads in each SM.
- $16 \times 16 = 256$ threads $\Rightarrow 2048/256 = 8$ blocks we will have full thread capacity in each SM.
- $32 \times 32 = 1024$ threads $\Rightarrow 2048/1024 = 2$ blocks.

 **TILE_WIDTH = 16 or 32**

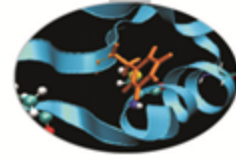


Although having many threads available for execution can theoretically tolerate long memory access latency, one can easily run into a situation where traffic congestion prevents all but few threads from making progress, thus making some SM idle!

A common **strategy for reducing global memory traffic** (i.e. increasing the number of floating-point operations performed for each access to the global memory) is to partition the data into subsets called *tiles* such that each tile fits into the shared memory and the kernel computations on these tiles can be done independently of each other.

In the simplest form, the tile dimensions equal those of the block.

Matrix-Matrix multiplication example



In the previous kernel:

thread(x,y) of block(0,0) access the elements of M_d row x and N_d column y from the global memory.

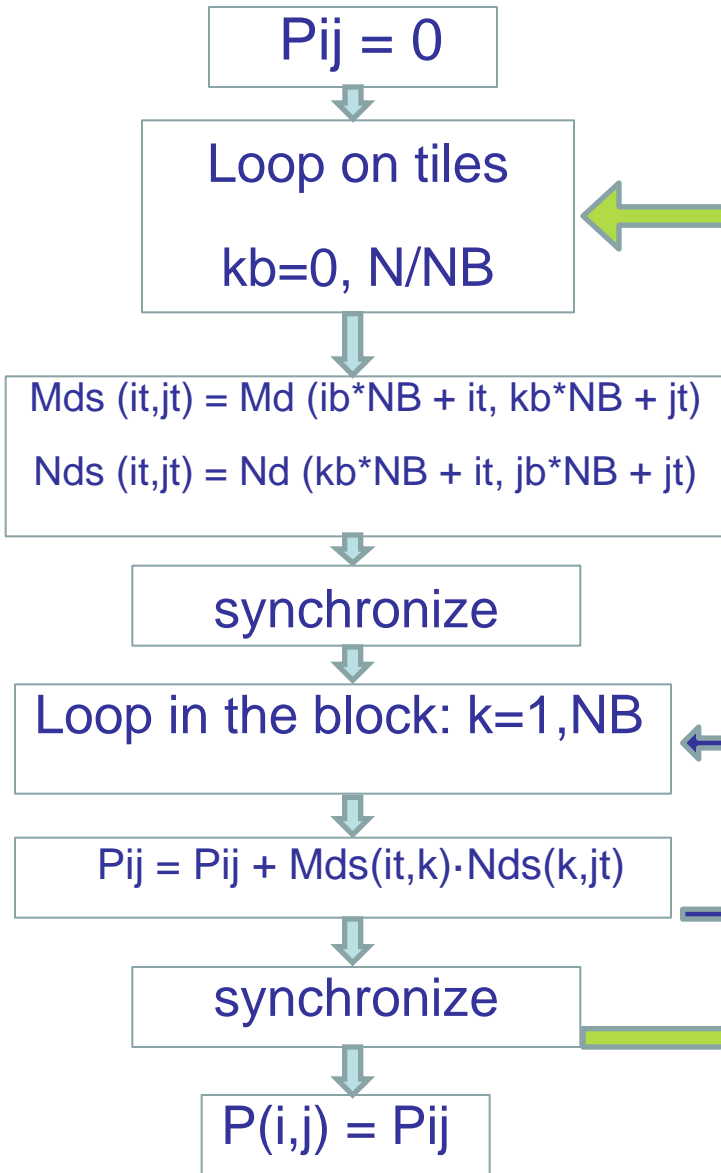
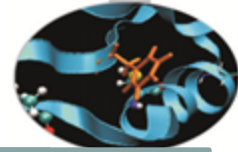
⇒ *thread(0,0) and thread(0,1) access the same M_d row 0*

What if these threads collaborate so that the elements of this row are only loaded from the global memory once? We can reduce the total number of accesses to the global memory by half!

Basic idea:

- † *to have the threads within a block collaboratively load M_d and N_d elements into the shared memory before they individually use these elements in their dot product calculation.*
- † *The dot product performed by each thread is now divided into phases: in each phase all threads in a block collaborate to load a tile of M_d and a tile of N_d into the shared memory and use these values to compute a partial product. The dot product would be performed in $\text{width}/\text{TILE_WIDTH}$ phases.*
- † *the reduction of the accesses to the global memory is by a factor of TILE_WIDTH .*

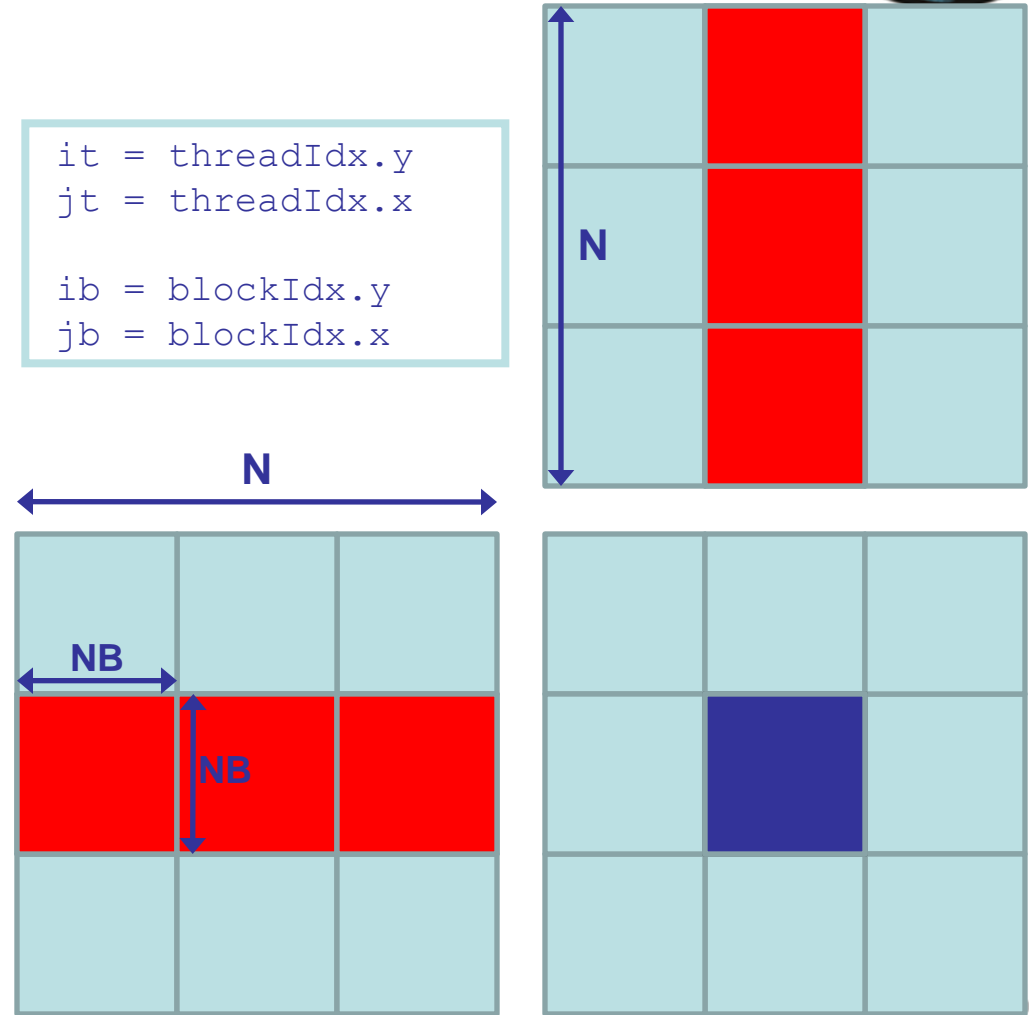
Matrix-Matrix multiplication example



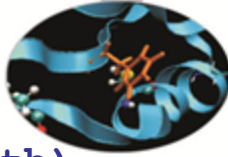
```

it = threadIdx.y
jt = threadIdx.x

ib = blockIdx.y
jb = blockIdx.x
  
```



Matrix-Matrix multiplication example

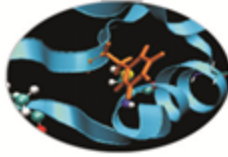


```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    // 2D thread ID
    int tx = threadIdx.x; int ty = threadIdx.y;
    int col = blockIdx.x*BlockDim.x + tx;
    int row = blockIdx.y*BlockDim.y + ty;

    float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
    for (int m=0; m < width/TILE_WIDTH; m++)
    { //collaborative loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[row*width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*width + col];
        __syncthreads();
        for (int k=0; k < TILE_WIDTH; k++)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }
    Pd[row * width + col] = Pvalue;
}
```

Memory as a limiting factor to parallelism



The limited amount of CUDA memory limits the number of threads that can simultaneously reside in the SM!

For the matrix multiplication example, shared memory can become a limiting factor:

$TILE_WIDTH = 16 \implies$ each block requires $16 \times 16 \times 4 = 1\text{KB}$ of storage for **Mds**
+ 1KB for **Nds**
 \implies 2KB of shared memory per block

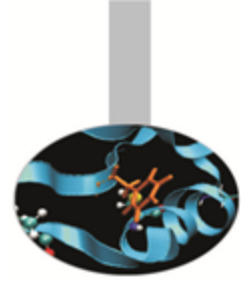
The 48KB shared memory allows 24 blocks to simultaneously reside in an SM. **OK!**

But the maximum number of threads per SM is 1536 (for Fermi)

 only $1536/256 = 8$ blocks are allowed in each SM
only $8 \times 2\text{KB} = 16\text{KB}$ of the shared memory will be used.

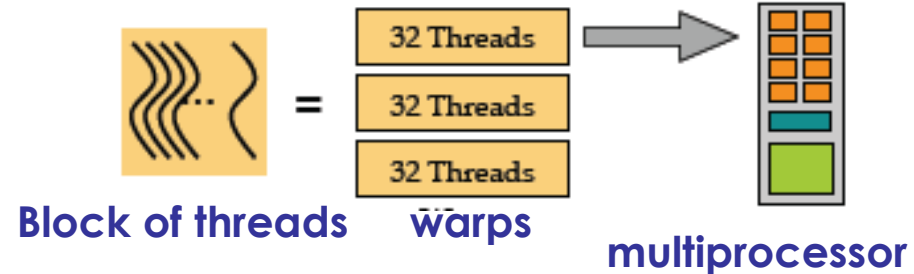
Hint: Use occupancy calculator

Thread scheduling



Once a block is assigned to a SM, it is further partitioned into 32-thread units called **warps**.

Warps are the *scheduling units in SM*: all threads in a same warp execute the same instruction when the warp is selected for execution (Single-Instruction, Multiple-Thread)

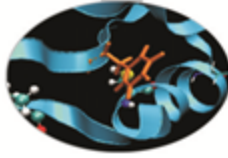


Threads often execute *long-latency operations*:

- global memory access
- pipelined floating point arithmetics
- branch instructions

*It is convenient to assign a large number of warps to each SM, because the long waiting time of some warp instructions is hidden by executing instructions from other warps. Therefore the selection of ready warps for execution does not introduce any idle time into the execution timeline (**zero-overhead thread scheduling**).*

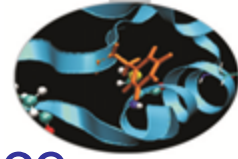
Control flow



- ⌚ The hardware executes an instruction for all threads in the same warp before moving to the next instruction (SIMT).
- ⌚ It works well when all threads within a warp follow the same control flow path when working their data.
- ⌚ When threads in the same warp follow different paths of control flow, we say that these threads *diverge* in their execution.
- ⌚ For an *if-then-else* construct the execution of the warp will require multiple passes through the divergent paths.

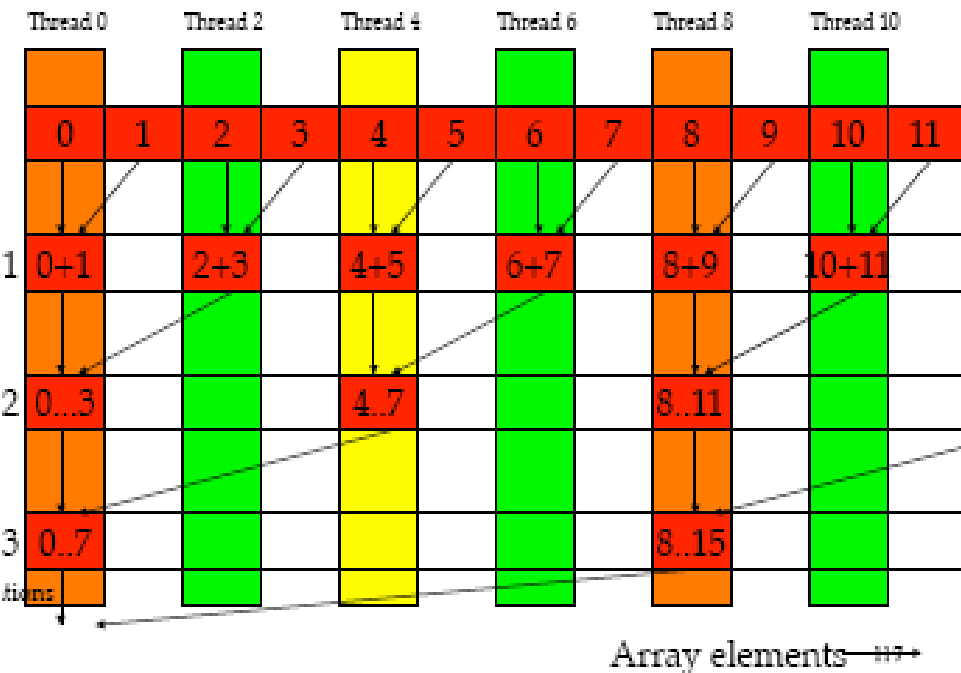
Try to avoid *warp divergence*

Vector reduction example (within a thread block)



An *if-then-else* construct can result in thread divergence when its decision condition is based on `threadIdx` values.

A sum reduction algorithm extracts a single value from an array of values in order to sum them. Within a block exploit the shared memory!



```

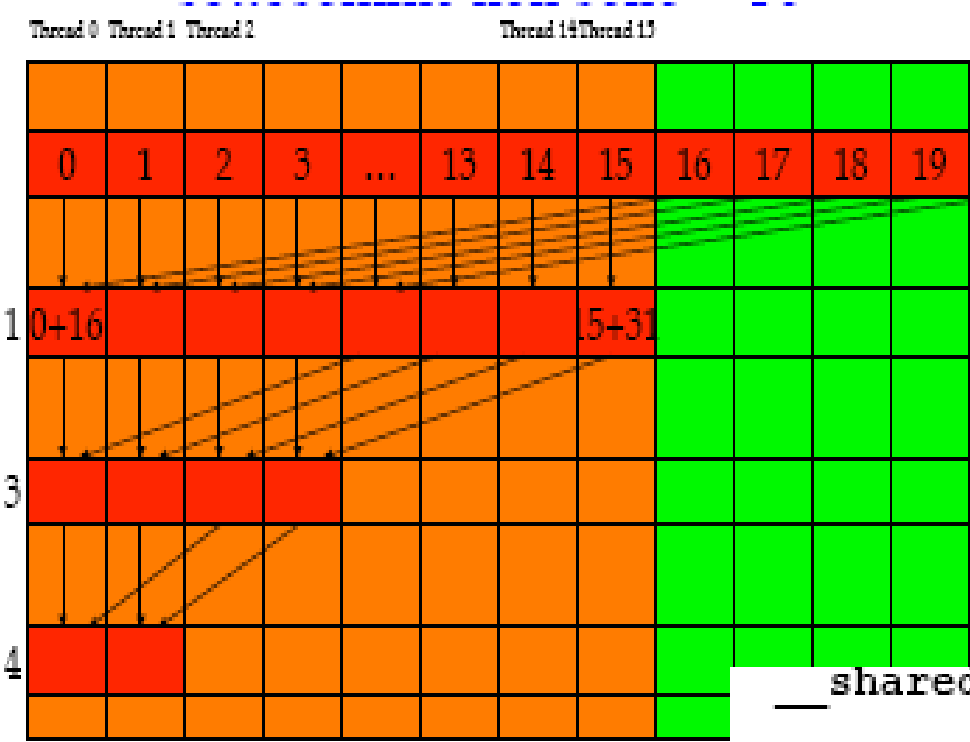
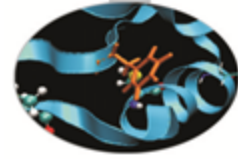
__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x; stride *= 2) {
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
  
```

There is thread divergence!



Vector reduction example



Instead of adding neighbor elements in the first round, add elements that are half a section away from each other and so on.

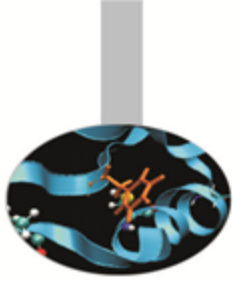
```

__shared__ float partialSum[]

unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1; stride >> 1) {
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}

```

No divergence until partial sums involve less than 32 elements (because of the warp size)



Reference

<http://developer.nvidia.com/cuda>

- 📌 CUDA Programming Guide
- 📌 CUDA Zone – tools, training, webinars and more

NVIDIA Books:

- 📌 “*Programming Massively Parallel Processors*”,
D.Kirk - W.W. Hwu
- 📌 “CUDA by example”, J.Sanders - E. Kandrot