

Introduction to Scientific Programming using GPGPU and CUDA



Day 1

Sergio Orlandini
s.orlandini@cineca.it

Mario Tacconi
m.tacconi@cineca.it

Agenda

First Day:

- Introduction to GPGPU
- CUDA Model for GPGPU
- CUDA GPU architectures
- Other GPGPU approaches

--- lunch ---

- Error Checking
- Measuring Performances
- Hands on

Second Day:

- GPU Memory Hierarchy
- Concurrency
- CPU-GPU Interaction
- Working with multi-GPU
- Hands on

--- lunch---

- CUDA-Toolkit
- CUDA Enabled Libraries
- Hands on

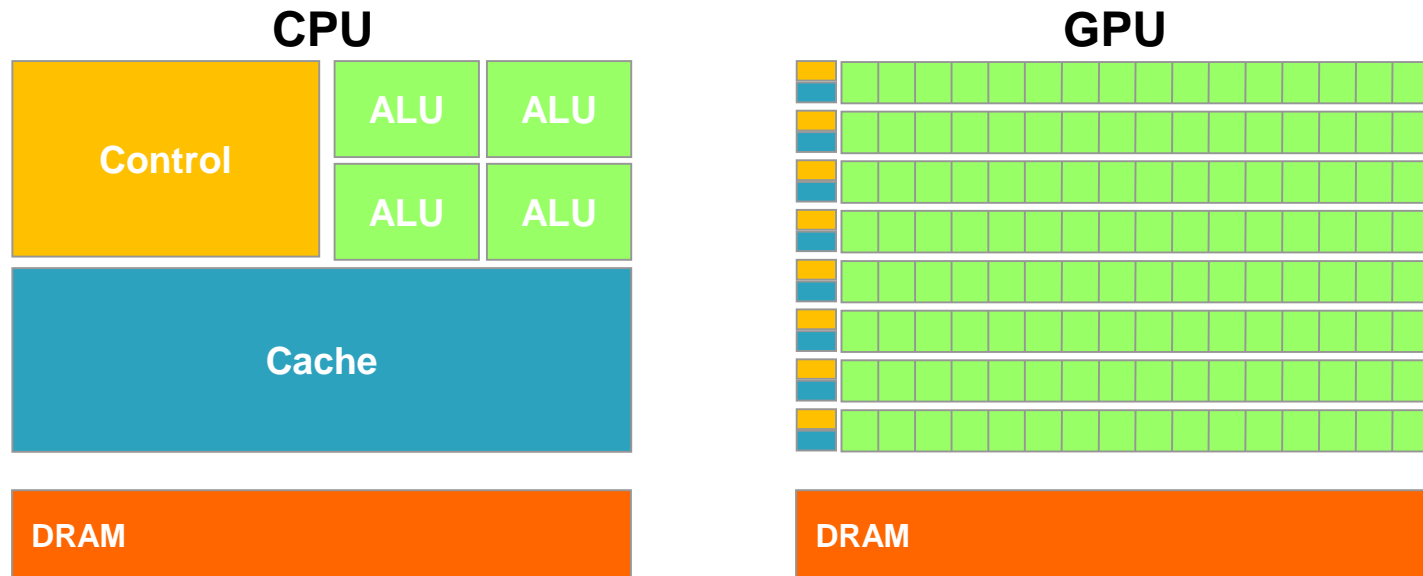
What is a GPU

- **Graphics Processor Unit**
 - a device equipped with an highly parallel microprocessor (*many-core*) and a private memory with very high bandwidth
- born in response to the growing demand for high definition 3D rendering graphic applications



CPU vs GPU Architectures

- GPU hardware is specialized for problems which can be classified as *intense data-parallel computations*
 - the same set of operation is executed many times in parallel on different data
 - designed such that more transistors are devoted to data processing rather than data caching and flow control



"The GPU devotes more transistors to Data Processing"
(NVIDIA CUDA Programming Guide)

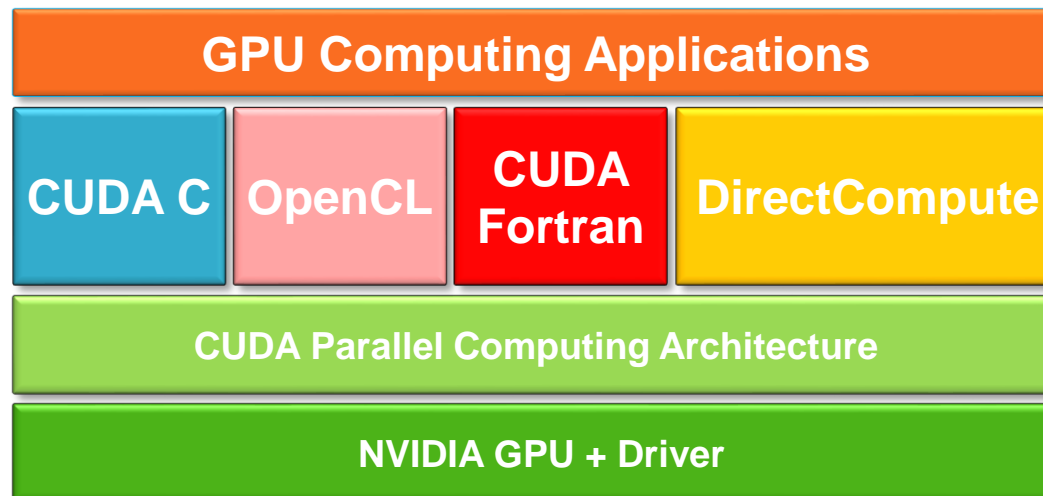
GPGPU (General Purpose GPU) and GPU computing

- many applications that process large data sets can use a data-parallel programming model to speed up the computations
- many algorithms outside the field of image rendering are accelerated by data-parallel processing
- ... so why not using GPU power for applications out of the 3D graphics domain?
- many attempts were made by brave programmers and researchers in order to force GPU APIs to treat their scientific data as pixel or vertex in order to be computed by the GPU.
- not many survived, still the era of GPGPU computing was just begun ...

A General-Purpose Parallel Computing Architecture

Compute Unified Device Architecture (CUDA)

- a general purpose parallel computing platform and programming model that easy GPU programming, which provides:
 - a new hierarchical multi-threaded programming paradigm
 - a new architecture instruction set called PTX (Parallel Thread eXecution)
 - a small set of syntax extensions to higher level programming languages (C, Fortran) to express thread parallelism within a familiar programming environment
 - A complete collection of development tools to compile, debug and profile CUDA programs.



There is more than CUDA to program a GPU

- OpenCL (Open Computing Language):
a standard open-source programming model developed by major brands of hardware manufacturers (Apple, Intel, AMD/ATI, nVIDIA).
 - provides extensions to C/C++ and a developer toolkit
 - extensions for specific hardware (GPUs, FPGAs, MICs, etc)
 - low level API, verbose programs
- Compiler directives:
 - PGI Accelerator
 - OpenACC
 - new feature of OpenMP v4.x for accelerators
 - you hope your compiler understand what you want, and do a good job
- AMD solution:
 - ATI Stream (dead)
 - ArrayFire (array-based open source library for easy programmability)
 - OpenCL
- Microsoft DirectCompute
 - Included in the DirectX API (version ≥ 10)
 - support GPU directives and shaders programming

■ CUDA programming model

- Heterogeneous execution
- Writing a CUDA Kernels
- Thread Hierarchy

■ Getting started with CUDA programming:

- the Vector-Vector Add
- handling data transfers from CPU to GPU memory and back
- write and launch the CUDA program

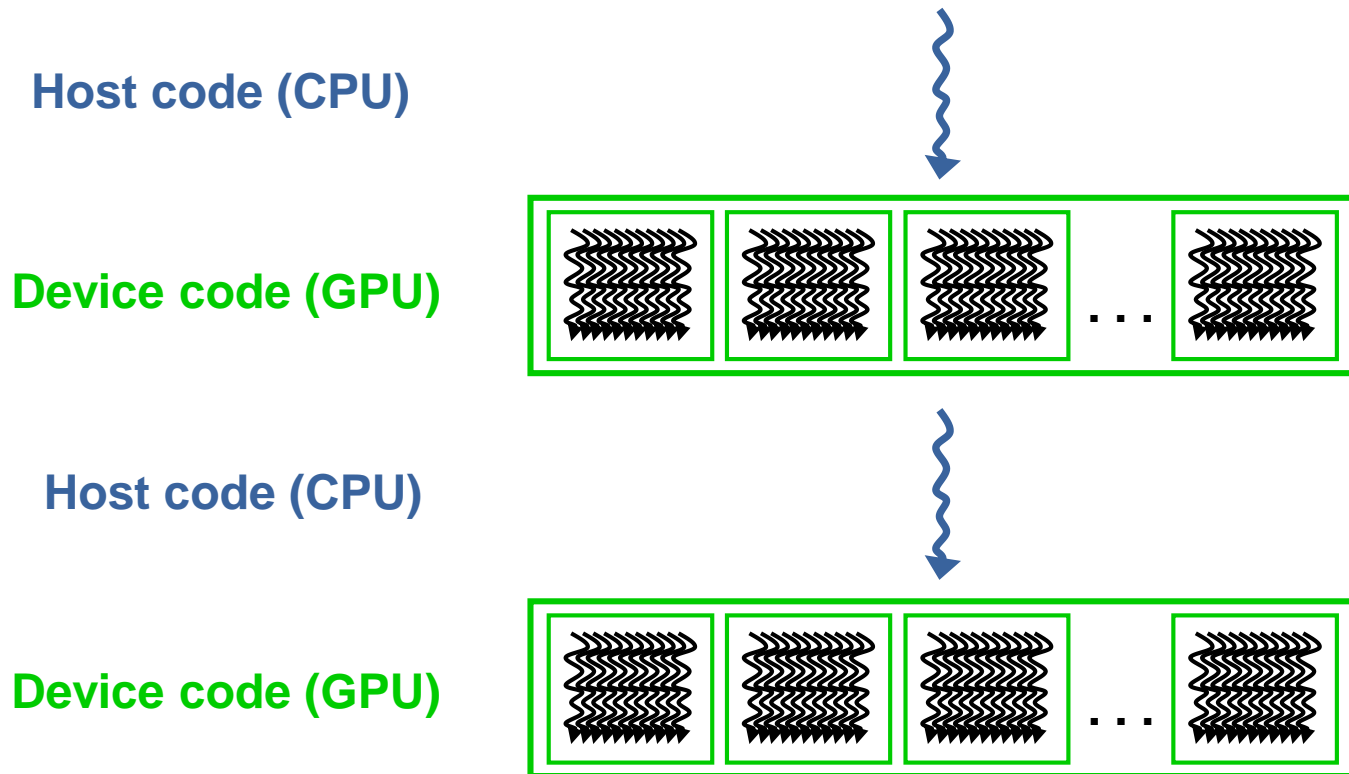


CUDA Programming Model

- GPU is seen as an auxiliary coprocessor with its own memory space
- *data-parallel, computational-intensive* portions of a program can be executed on the GPU
 - each *data-parallel* computational portion can be isolated into a function, called CUDA kernel, that is executed on the GPU
 - CUDA kernels are executed by many different threads in parallel
 - each thread can compute different data elements independently
 - the GPU parallelism is very close to the SPMD (Single Program Multiple Data) paradigm. Single Instruction Multiple Threads (SIMT) according to the Nvidia definition.
- GPU threads are extremely *light weight*
 - no penalty in case of a *context-switch* (each thread has its own registers)
 - the more are the threads *in flight*, the more the GPU hardware is able to hide memory or computational latencies, i.e better overall performances at executing the kernel function

CUDA Execution Model

- serial portions of a program, or those with low level of parallelism, keep running on the CPU (host)
- Data-parallel , computational intensive portions of the program are isolated into CUDA kernel function. The CUDA kernel are executed onto the GPU (device)



CUDA syntax extensions to the C language

CUDA defines a small set of extensions to the high level language as the C in order to define the kernels and to configure the kernel execution.

- A CUDA kernel function is defined using the `__global__` declaration
- when a CUDA kernel is called, it will be executed N times in parallel by N different CUDA threads on the device
- the number of CUDA threads that execute that kernel is specified using a new syntax, called kernel execution configuration
 - `cudaKernelFunction <<<...>>> (arg_1, arg_2, ..., arg_n)`
- each thread has a unique thread ID
 - the thread ID is accessible within the CUDA kernel through the built-in `threadIdx` variable
- the built-in variables **`threadIdx`** are a 3-component vector
 - use `.x`, `.y`, `.z` to access its components

A simple CUDA program

```
int main(int argc, char *argv[]) {
    int i;
    const int N = 1000;
    double u[N], v[N], z[N];

    initVector (u, N, 1.0);
    initVector (v, N, 2.0);
    initVector (z, N, 0.0);

    printVector (u, N);
    printVector (v, N);

    // z = u + v
    for (i=0; i<N; i++)
        z[i] = u[i] + v[i];

    printVector (z, N);

    return 0;
}
```

```
__global__
void gpuVectAdd( const double *u,
                  const double *v, double *z)
{ // use GPU thread id as index
  i = threadIdx.x;
  z[i] = u[i] + v[i];
}
```

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

```
    // z = u + v
    {
        // run on GPU using
        // 1 block of N threads in 1D
        gpuVectAdd <<<1,N>>> (u, v, z);
    }
```

```
    ...
}
```

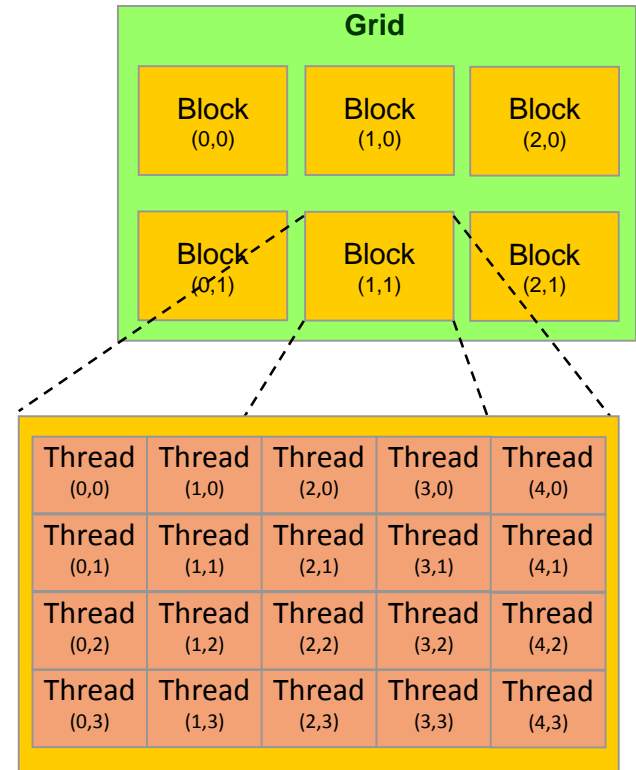
CUDA Threads

- threads are organized into blocks of threads
 - blocks can be 1D, 2D, 3D sized in threads
 - blocks can be organized into a 1D, 2D, 3D grid of blocks
 - each block of threads will be executed independently
 - no assumption is made on the blocks execution order
- each block has a unique block ID
 - the block ID is accessible within the CUDA kernel through the built-in **blockIdx** variable
- The built-in variable **blockIdx** is a 3-component vector
 - use .x, .y, .z to access its components

blockIdx:
block coordinates inside the grid

blockDim:
block dimensions in thread units

gridDim:
grid dimensions in block units



Simple 1D CUDA vector add

```
__global__
void gpuVectAdd( int N, const double *u, const double *v, double *z)
{
    // use GPU thread id as index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    // check out of border access
    if ( index < N ) {
        z[index] = u[index] + v[index];
    }
}

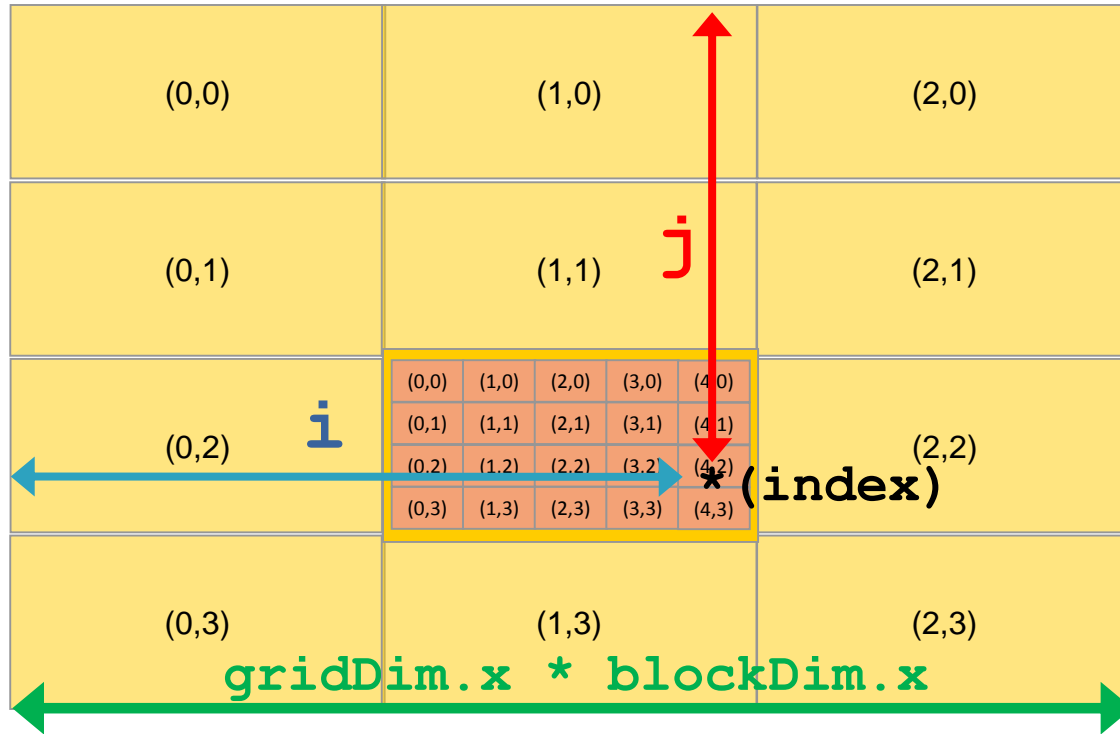
int main(int argc, char *argv[]) {
    ...

    // use 1D block threads
    dim3 blockSize = 512;

    // use 1D grid blocks
    dim3 gridSize = (N + blockSize-1) / blockSize.x;

    gpuVectAdd <<< gridSize,blockSize >>> (N, u, v, z);
    ...
}
```

Composing 2D CUDA Thread Indexing



threadIdx:
thread coordinates inside a block

blockIdx:
block coordinates inside the grid

blockDim:
block dimensions in thread units

gridDim:
grid dimensions in block units

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

```
j = blockIdx.y * blockDim.y + threadIdx.y;
```

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

2D array element-wise add (matrix add)

- As an example, the following code adds two matrices *A* and *B* of size $N \times N$ and stores the result into the matrix *C*

```
__global__ void matrixAdd(int N, const float *A, const float *B, float *C) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    int j = blockIdx.y * blockDim.y + threadIdx.y;  
  
    // matrix elements are organized in row major order in memory  
    int index = i * N + j;  
  
    C[index] = A[index] + B[index];  
}  
  
int main(int argc, char *argv[]) {  
    ...  
  
    // add NxN matrices on GPU using 1 block of NxN threads  
    matrixAdd <<< 1, N >>> (N, A, B, C);  
    ...  
}
```


Memory allocation on GPU device

- CUDA API provides 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 must be cast to `(void**)`!
 - The second parameter is the size in bytes of the buffer to be allocated
- `cudaFree(void* bufferPtr)`
 - It frees the storage space of the object

Memory Initialization on GPU device

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

It 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.

- it's like the standard library C `memset()` function
 - devPtr - Pointer to device memory
 - value - Value to set for each byte of specified memory
 - count - Size in bytes to set
- REM: to initialize an array of double (float, int, ...) to a specific value you need to execute a CUDA kernel.

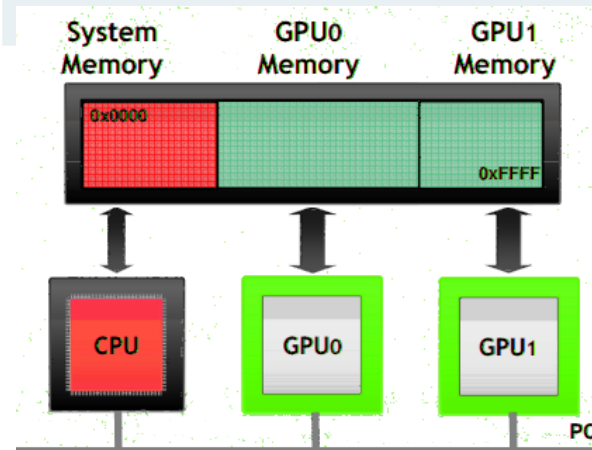
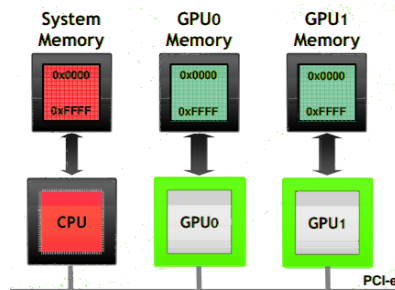
Memory copy between CPU and GPU

- `cudaMemcpy(void *dst, void *src, size_t size, direction)`
 - `dst`: destination buffer pointer
 - `src`: source buffer pointer
 - `size`: number of bytes to copy
 - `direction`: macro name which defines the direction of data copy
 - from CPU to GPU: `cudaMemcpyHostToDevice` (H2D)
 - from GPU to CPU: `cudaMemcpyDeviceToHost` (D2H)
 - on the same GPU: `cudaMemcpyDeviceToDevice`
 - the copy begins only after all previous kernel have finished
 - the copy is blocking: it prevents CPU control to proceed further in the program until last byte has been transferred
 - returns only after copy is complete

CUDA 4.x - Unified Virtual Addressing

- CUDA 4.0 introduces a unique virtual address space for memory (Unified Virtual Address) shared between GPU and HOST:
 - the actual memory type a data resides is automatically understood at runtime
 - greatly simplify programming model
 - allow simple addressing and transfer of data among GPU devices

Pre-UVA	UVA
A macro for each combination of source/destination	The system keeps track of the buffer location.
<code>cudaMemcpyHostToHost</code> <code>cudaMemcpyHostToDevice</code> <code>cudaMemcpyDeviceToHost</code> <code>cudaMemcpyDeviceToDevice</code>	<code>cudaMemcpyDefault</code>



CUDA 6.x - Unified Memory

- Unified Memory creates a pool of memory with an address space that is shared between the CPU and GPU. In other word, a block of Unified Memory is accessible to both the CPU and GPU by using the same pointer;
- the system automatically *migrates* data allocated in Unified Memory mode between the host and device memory
 - no need to explicitly declare device memory regions
 - no need to explicitly copy back and forth data between CPU and GPU devices
 - greatly simplifies programming and speeds up CUDA ports
- REM: it can result in performances degradation with respect to an explicit, finely tuned data transfer.

Sample code using CUDA Unified Memory

CPU code

```
void sortfile (FILE *fp, int N) {  
    char *data;  
  
    data = (char *) malloc (N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data)  
}
```

GPU code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, compare);  
  
    qsort<<< ... >>> (data, N, 1, compare);  
  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

Three steps for a CUDA porting

1. identify data-parallel, computational intensive portions
 1. isolate them into functions (CUDA kernels candidates)
 2. identify involved data to be moved between CPU and GPU
2. translate identified CUDA kernel candidates into real CUDA kernels
 1. choose the appropriate thread index map to access data
 2. change code so that each thread acts on its own data
3. modify code in order to manage memory and kernel calls
 1. allocate memory on the device
 2. transfer needed data from host to device memory
 3. insert calls to CUDA kernel with execution configuration syntax
 4. transfer resulting data from device to host memory

Vector Sum

1. identify data-parallel computational intensive portions

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

    const int N = 1000;
    double u[N], v[N], z[N];

    initVector (u, N, 1.0);
    initVector (v, N, 2.0);
    initVector (z, N, 0.0);

    printVector (u, N);
    printVector (v, N);

    // z = u + v
    for (i=0; i<N; i++)
        z[i] = u[i] + v[i];

    printVector (z, N);

    return 0;
}
```

```
program vectoradd
integer :: i
integer, parameter :: N=1000
real(kind(0.0d0)), dimension(N) :: u, v, z

call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)

call printVector (u, N)
call printVector (v, N)

! z = u + v
do i = 1,N
    z(i) = u(i) + v(i)
end do

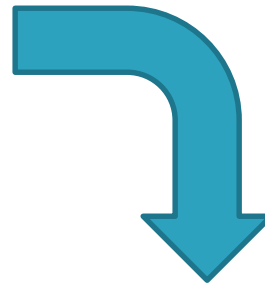
call printVector (z, N)

end program
```


- each *thread* execute the same kernel, but act on different data:
 - turn the loop into a CUDA kernel function
 - map each CUDA *thread* onto a unique index to access data
 - let each *thread* retrieve, compute and store its own data using the unique address
 - prevent out of border access to data if data is not a multiple of thread block size

```
const int N = 1000;
double u[N], v[N], z[N];

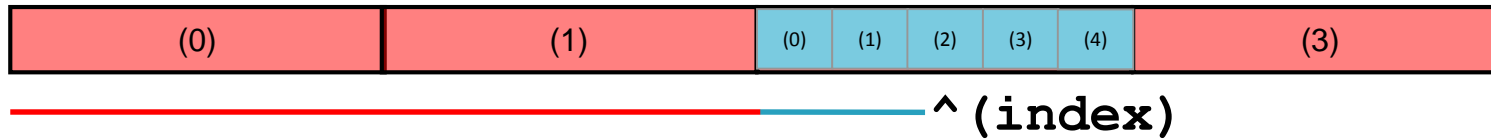
// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```



```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier for each GPU thread
    int index = blockIdx.x * blockDim.x + threadIdx.x ;
    if (index < N)
        z[index] = u[index] + v[index];
}
```

Vector Sum

2. translate the identified data-parallel portions into CUDA kernels



```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index = blockIdx.x * blockDim.x + threadIdx.x ;
    if (index < N)
        z[index] = u[index] + v[index];
}
```

The **__global__** qualifier declares a CUDA kernel

CUDA kernels are special C functions:

- can be called from host only
- must be called using the *execution configuration* syntax
- the return type must be *void*
- they are asynchronous: control is returned immediately to the host code
- an explicit synchronization is needed in order to be sure that a CUDA kernel has completed the execution

```
module vector_algebra_cuda
use cudafor
contains
attributes(global) subroutine gpuVectAdd (N, u, v, z)
  implicit none
  integer, intent(in), value :: N
  real, intent(in) :: u(N), v(N)
  real, intent(inout) :: z(N)
  integer :: i

  i = ( blockIdx%x - 1 ) * blockDim%x + threadIdx%x

  if (i .gt. N) return

  z(i) = u(i) + v(i)
end subroutine
end module vector_algebra_cuda
```

```
attributes(global) subroutine gpuVectAdd (N, u, v, z)
    ...
end subroutine

program vectorAdd
use cudafor
implicit none
interface
    attributes(global) subroutine gpuVectAdd (N, u, v, z)
        integer, intent(in), value :: N
        real, intent(in) :: u(N), v(N)
        real, intent(inout) :: z(N)
        integer :: i
    end subroutine
end interface
    ...
end program vectorAdd
```

If the kernels are not defined within a module, then an explicit interface must be provided for each kernel you want to launch within a program unit.

- **CUDA C API:** `cudaMalloc(void **p, size_t size)`
 - allocates size bytes of GPU global memory
 - p is a valid device memory address (i.e. SEGV if you dereference p on the host)

```
double *u_dev, *v_dev, *z_dev;  
  
cudaMalloc((void **)&u_dev, N * sizeof(double));  
cudaMalloc((void **)&v_dev, N * sizeof(double));  
cudaMalloc((void **)&z_dev, N * sizeof(double));
```

- in CUDA Fortran the attribute **device** needs to be used while declaring a GPU array. The array can be allocated by using the Fortran statement **allocate**:

```
real(kind(0.0d0)), device, allocatable, dimension(:, :) :: u_dev, v_dev, z_dev  
  
allocate( u_dev(N), v_dev(N), z_dev(N) )
```

```
double *u_dev;  
  
cudaMalloc((void **) &u_dev, N*sizeof(double));
```

- `&u_dev`
 - `u_dev` it's a variable defined on the *host* memory
 - `u_dev` contains an address of the *device* memory
 - C pass arguments to function by value
 - we need to pass `u_dev` by reference to let its value be modified by the `cudaMalloc` function
 - this has nothing to do with CUDA, it's a C common idiom
 - if you don't understand this, probably you are not ready for this course
- `(void **)` is a cast to force `cudaMalloc` to handle pointer to memory of any kind
 - again, if you don't understand this...

■ CUDA C API:

```
cudaMemcpy(void *dst, void *src, size_t size, direction)
```

- copy size bytes from the src to dst buffer

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);  
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

■ in CUDA Fortran you can use the array syntax

```
u_dev = u ; v_dev = v
```

Insert calls to CUDA kernels using the execution configuration syntax:

kernelCUDA<<<numBlocks, numThreads>>> (...)

specifying the thread/block hierarchy you want to apply:

- **numBlocks**: specify grid size in terms of thread blocks along each dimension
- **numThreads**: specify the block size in terms of threads along each dimension

```
dim3 numThreads(32);  
dim3 numBlocks( ( N + numThreads - 1 ) / numThreads.x );  
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
```

```
type(dim3) :: numBlocks, numThreads  
numThreads = dim3( 32, 1, 1 )  
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )  
call gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev )
```


Vector Sum: the complete CUDA code

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void **)&u_dev, N * sizeof(double));
cudaMalloc((void **)&v_dev, N * sizeof(double));
cudaMalloc((void **)&z_dev, N * sizeof(double));

cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);

dim3 numThreads( 256); // 128-512 are good choices
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

```
real(kind(0.0d0)), device, allocatable, dimension(:,:) :: u_dev, v_dev, z_dev
type(dim3) :: numBlocks, numThreads
allocate( u_dev(N), v_dev(N), z_dev(N) )
u_dev = u; v_dev = v

numThreads = dim3( 256, 1, 1 ) ! 128-512 are good choices
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )
z = z_dev
```

- CUDA Architecture
 - FERMI and KEPLER generation
 - more on CUDA Execution Model
- Compiling a CUDA program
 - PTX, cubin, what's inside
 - computing capability
- other GPGPU approaches
 - OpenCL
 - OpenACC
 - Intel Xeon Phi (MIC) coprocessor

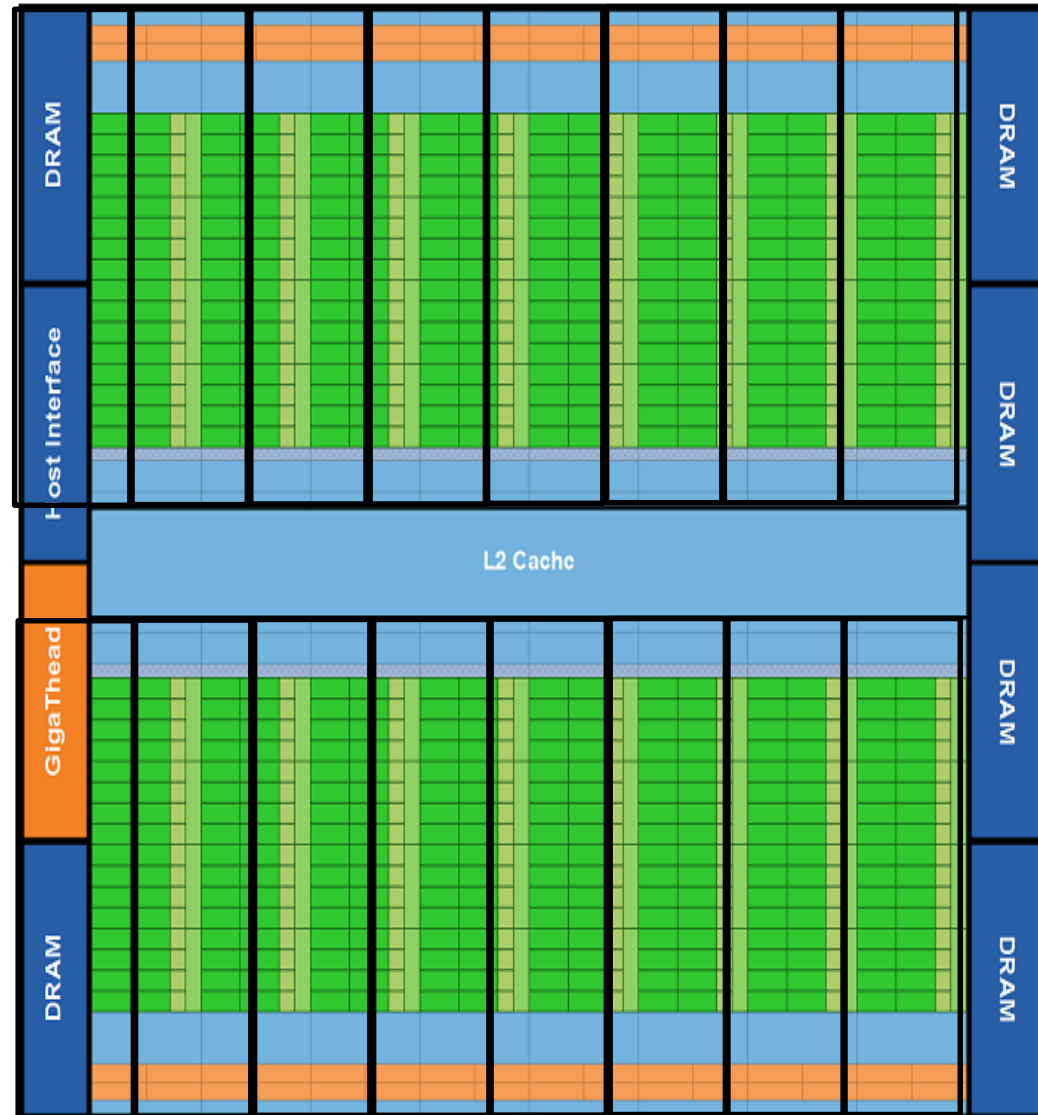


NVIDIA Architectures naming

- Desktop & laptop computers: GeForce
 - gaming, multimedia
- Mobile devices: Tegra
 - SOC for tablets and smartphones
- Workstation: Quadro
 - professional graphics applications such as CAD, modeling 3D, animation and visual effects
- GPGPU: Tesla
 - High Performance Computing

NVIDIA Fermi Architecture

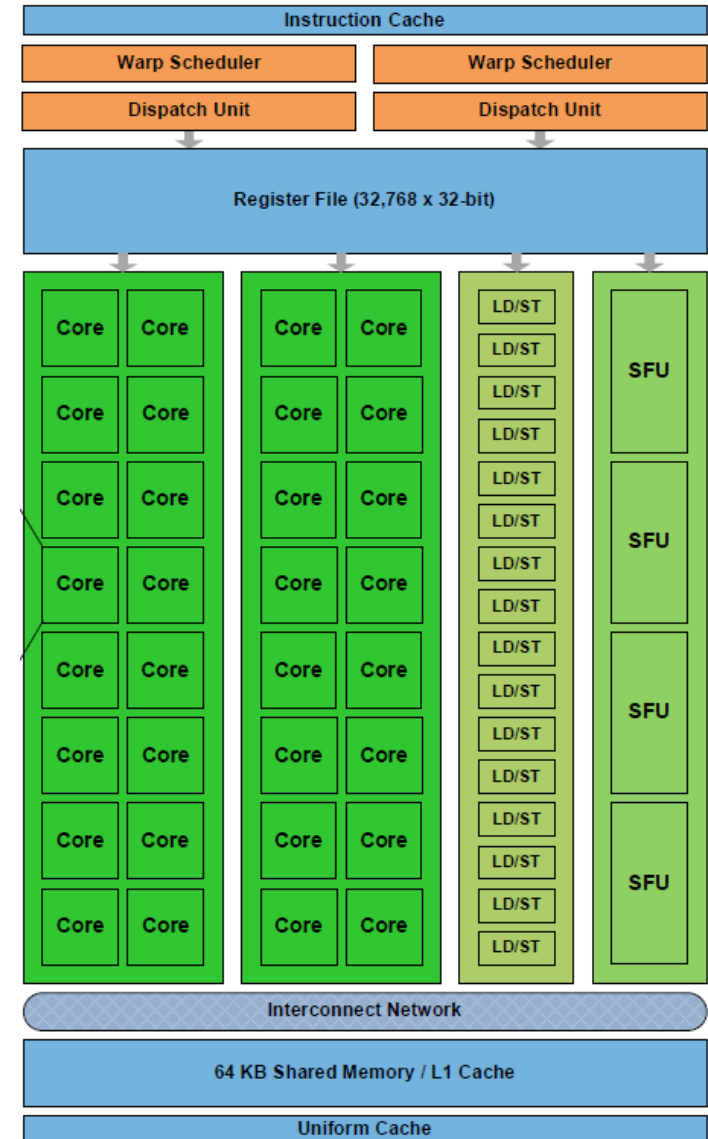
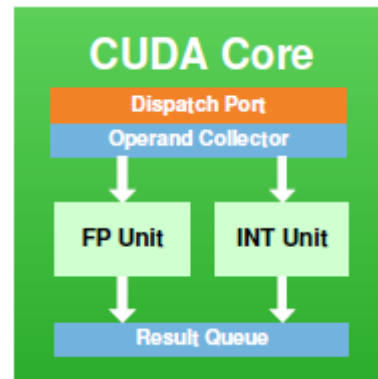
- 16 Streaming Multiprocessors (SM)
- 4-6 GB global memory with ECC
- first model with a cache hierarchy:
 - L1 (16-48KB) per SM
 - L2 (768KB) shared among all SM
- 2 independent controllers for data transfer from/to host through PCI-Express
- Global thread scheduler (GigaThread global scheduler) which manage and distribute thread blocks to be processed on SM resources



Fermi Streaming Multiprocessor (SM)

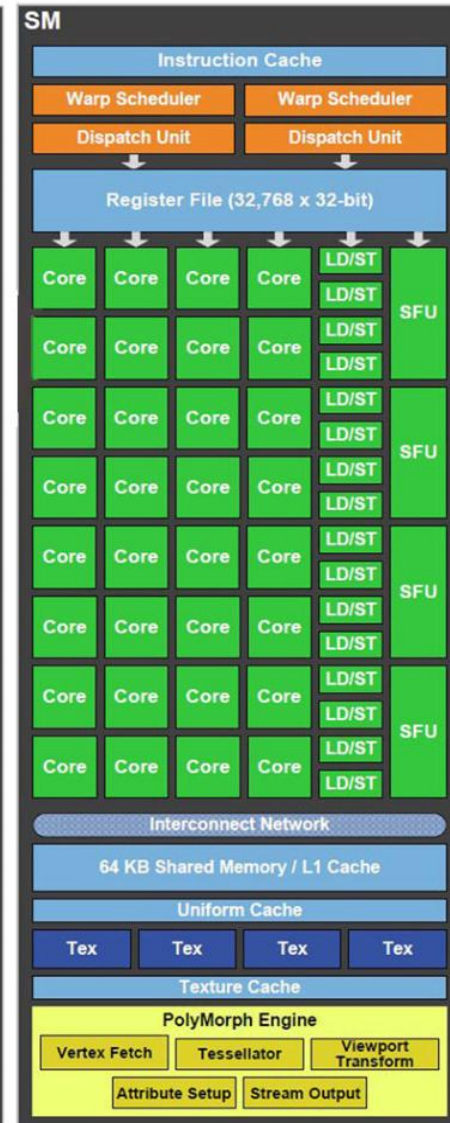
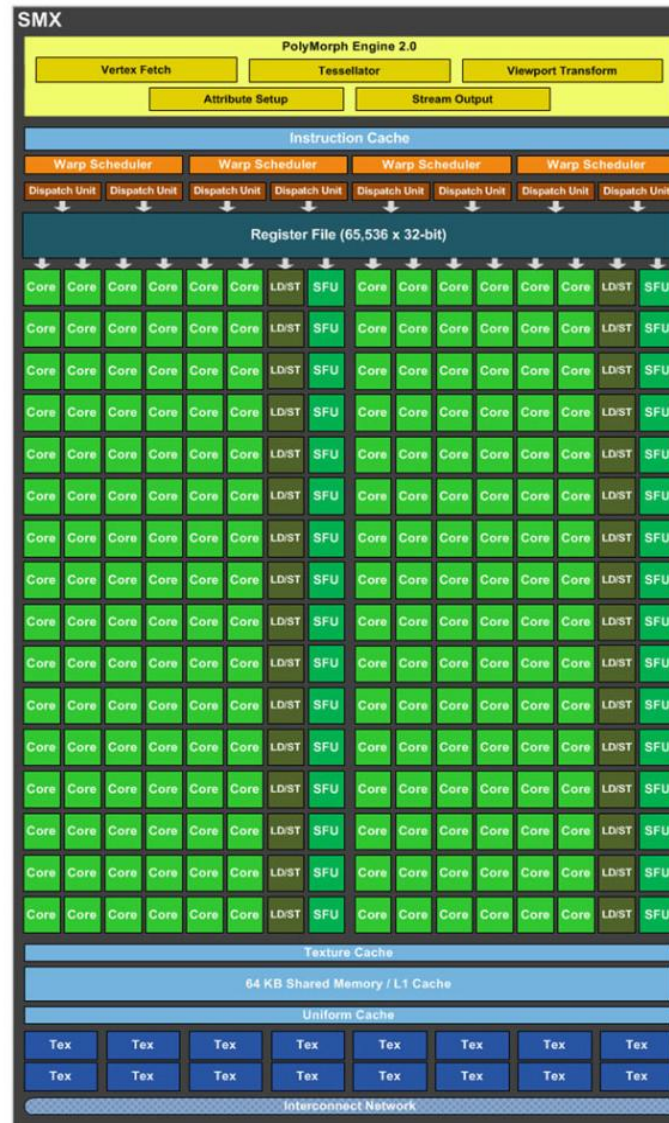
Streaming Multiprocessor sports:

- 32/48 CUDA cores with an arithmetic logic unit (ALU) and a floating point unit (FPU) fully *pipelined*
- floating point operations are fully IEEE 754-2008 a 32-bit e a 64-bit
 - **fused multiply-add** (FMA) for both single and double precision
- 32768 registers (32-bit)
- 64KB configurable L1 shared-memory/cache
 - 48-16KB or 16-48KB shared/L1 cache
- 16 load/store units
- 4 Special Function Unit (SFU) to handle transcendental mathematical functions (sin, sqrt, recp-sqrt,..)



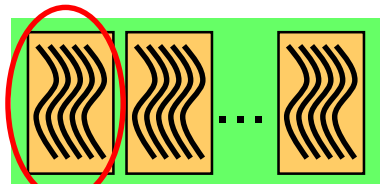
NVIDIA Kepler Architecture

- x3 performance/watt with respect to FERMI
 - 28nm lithography
- 192 CUDA cores
- 4 warp scheduler (2 dispatcher)
 - 2 independent instruction/warp
- standard IEEE 754-2008
- 65536 registers per SM (32-bit)
- 32 load/store units
- 32 Special Function Unit
- 1534KB L2 cache (x2 vs Fermi)
- 64KB shared-memory/cache + 48KB read-only L1 cache
- 16 texture units (x4 vs Fermi)



more on the CUDA Execution Model

Software



Griglia



Blocco di Thread



Thread

Hardware



GPU



Streaming Multiprocessor



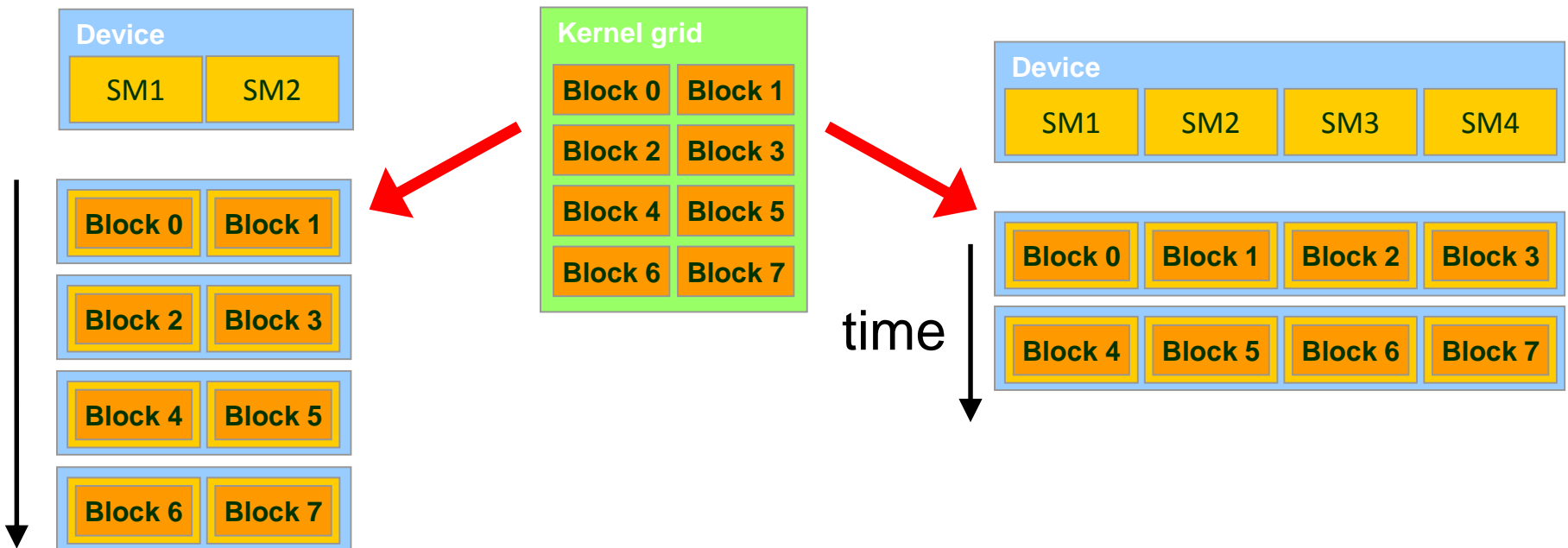
CUDA core

when a CUDA kernel is invoked:

- each thread block is assigned to a SM in a round-robin mode
 - a maximum number of blocks can be assigned to each SM, depending on hardware generation and on how many resources each block needs to be executed (registers, shared memory, etc)
 - 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.
 - once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed
 - each block execution is independent from the other (no synchronization is possible among them)
- threads of each block are partitioned into warps of 32 *threads* each, so to map each thread with a unique consecutive thread index in the block, starting from index 0.
- the scheduler selects for execution a warp from one of the residing blocks in each SM.
- A warp executes one common instruction at a time
 - each CUDA core takes care of one thread in the warp
 - fully efficient when all threads agree on their execution path

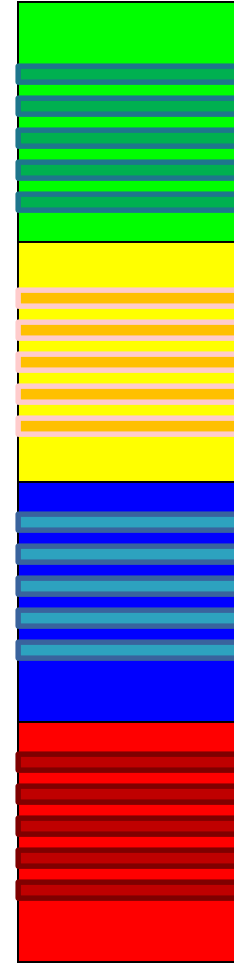
Transparent Scalability

- 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



Resources per Thread Block

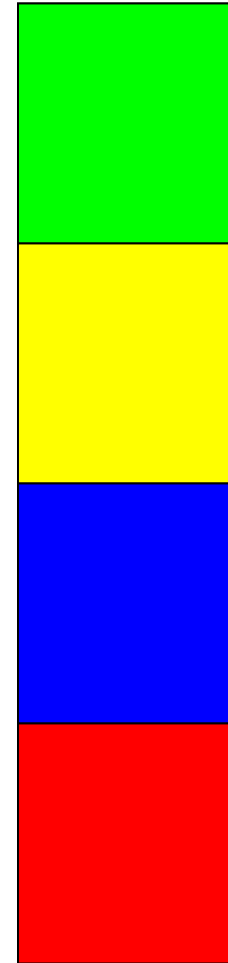
- Each CUDA kernel needs a specific amount of resources to run
- Once blocks are assigned to the SM, registers are assigned to each thread block, depending on kernel required resources
- Once assigned, registers will belong to that thread until the thread block complete its work
- so that each thread can access only its own assigned registers
- allow for zero-overload schedule when context switching among different warp execution



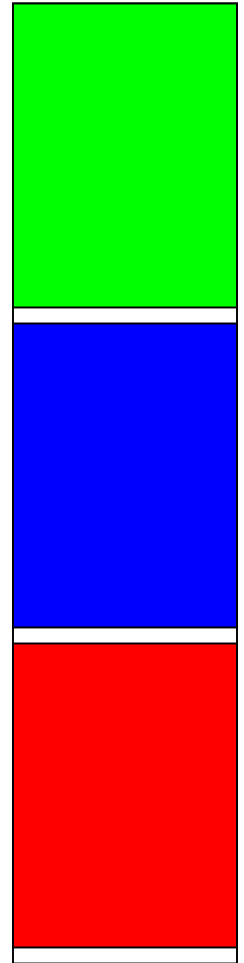
Assigning Thread Blocks to SM

- Let's provide an example of block assignment on a SM:
 - Fermi architecture: 32768 register per SM
 - CUDA kernel grid with 32x8 thread blocks
 - CUDA kernel needs 30 registers
- How many thread blocks can host a single SM?
 - each block requires $30 \times 32 \times 8 = 7680$ registers
 - $32768 / 7680 = 4$ blocks + "reminder"
 - only 4 blocks can be hosted (out of 8)
- What happen if we modify the kernel a little bit, moving to an implementation which requires 33 registers?
 - each block now requires $33 \times 32 \times 8 = 8448$ registers
 - $32768 / 8448 = 3$ blocks + "reminder"
 - only 3 blocks! (out of 8)
 - 25% reduction of potential parallelism

4 blocks



3 blocks

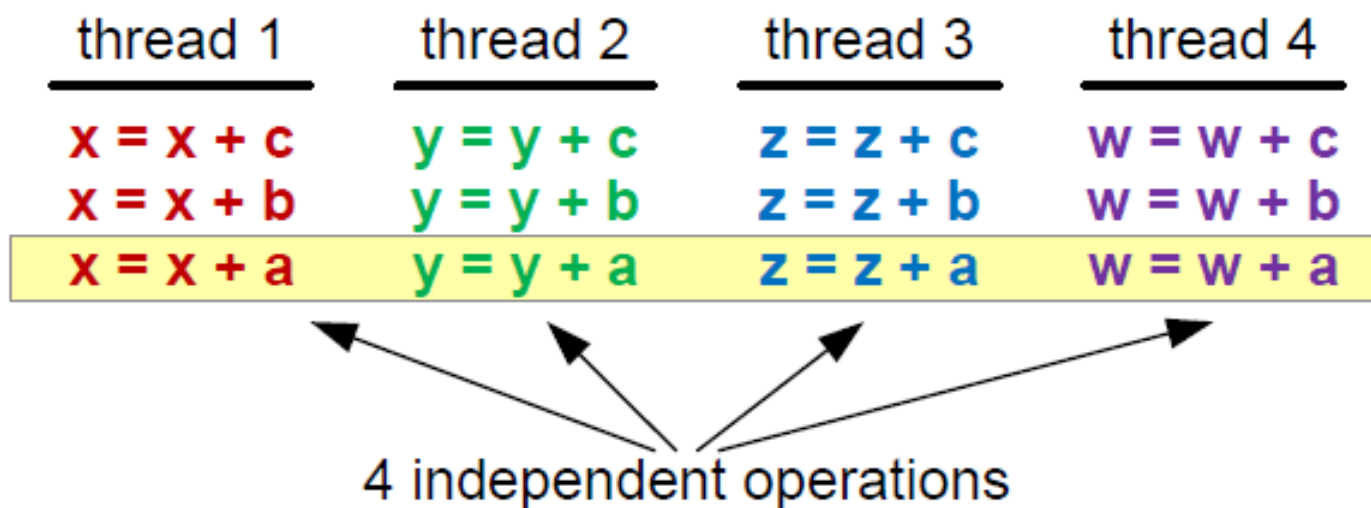


Hiding Latencies

- What is latency?
 - the number of clock cycles needed to complete an instruction
 - ... that is, the number of cycles I need to wait for before another **dependent operation** can start
 - arithmetic latency (~ 18-24 cycles)
 - memory access latency (~ 400-800 cycles)
- We cannot discard latencies (it's an hardware design effect), but we can lesser their effect and hide them.
 - saturating computational pipelines in computational bound problems
 - saturating bandwidth in memory bound problems
- We can organize our code so to provide the scheduler a sufficient number of **independent operations**, so that the more the warp are available, the more content-switch can hide latencies and proceed with other useful operations
- There are two possible ways and paradigms to use (can be combined too!)
 - Thread-Level Parallelism (TLP)
 - Instruction-Level Parallelism (ILP)

Thread-Level Parallelism (TLP)

- Strive for high SM occupancy: that is try to provide as much threads per SM as possible, so to easy the scheduler find a warp ready to execute, while the others are still busy
- This kind of approach is effective when there is a low level of independet operations per CUDA kernels



Instruction-Level Parallelism (ILP)

- Strive for multiple independent operations inside you CUDA kernel: that is, let your kernel act on more than one data
- this will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- note: the scheduler will not select a new warp untill there are eligible instructions ready to execute on the current warp

thread

$w = w + b$

$z = z + b$

$y = y + b$

$x = x + b$

$w = w + a$

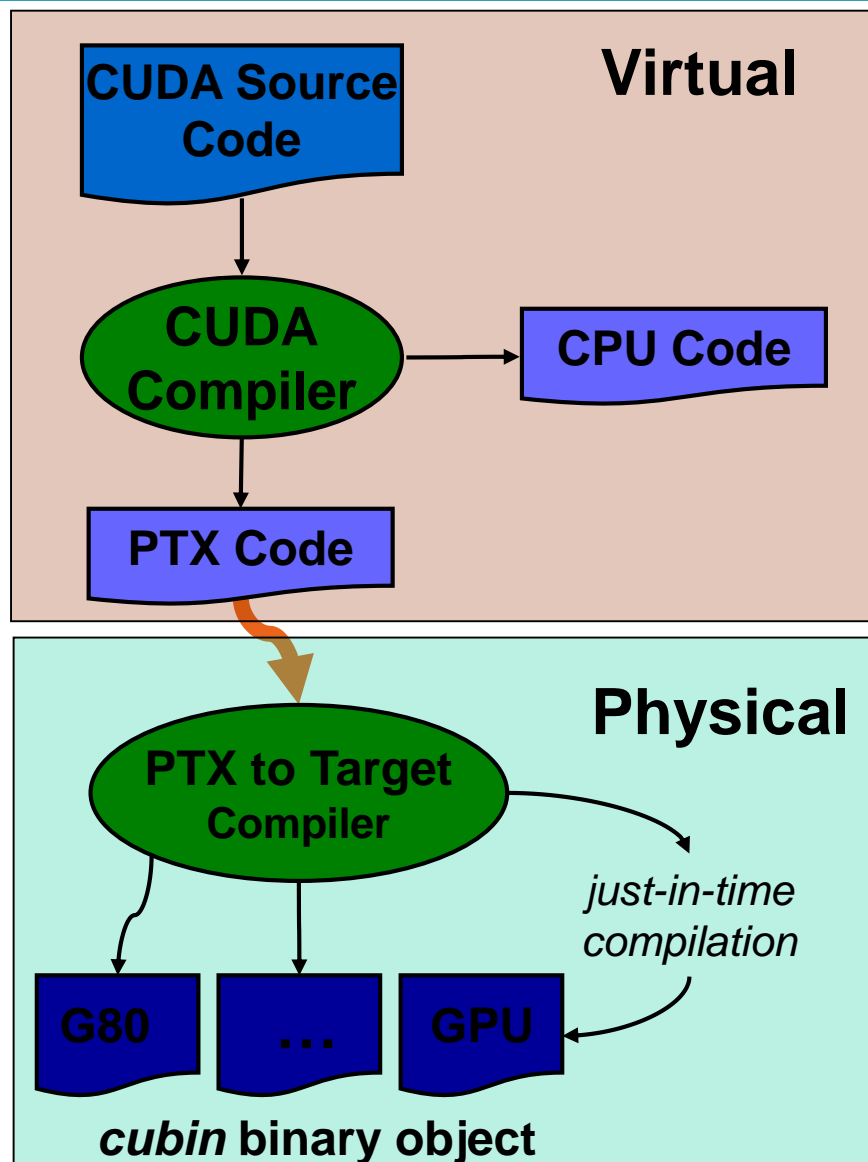
$z = z + a$

$y = y + a$

$x = x + a$

4 independent operations

CUDA Compilation Workflow



- each source file with CUDA extension should be compiled with a proper CUDA aware compiler
 - `nvcc` CUDA C (NVIDIA)
 - `pgf90 -Mcuda` CUDA Fortran (PGI)
- CUDA compiler processes the source code, separating device code from host code:
 - *host* is modified replacing CUDA extensions by the necessary CUDA C runtime functions calls
 - the resulting *host* code is output to a host compiler
 - *device* code is compiled into the PTX assembly form
- starting from the PTX assembly code you can:
 - generate one or more object forms (*cubin*) specialized for specific GPU architectures
 - generate an executable which include both PTC code and object code

Compute Capability

- the *compute capability* of a device describes its architecture
 - *registers, memory sizes, features and capabilities*
- the compute capability is identified by a code like “compute_Xy”
 - major number (X): identifies base line chipset architecture
 - minor number (y): identifies variants and releases of the base line chipset
- a compute capability select the set of usable PTX instructions

<i>compute capability</i>	<i>feature support</i>
compute_10	basic CUDA support
compute_13	improved memory accesses + double precision + atomics
compute_20	FERMI architecture caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion
compute_30	KEPLER K10 architecture (support only single precision)
compute_35	KEPLER K20, K20X, K40 architectures

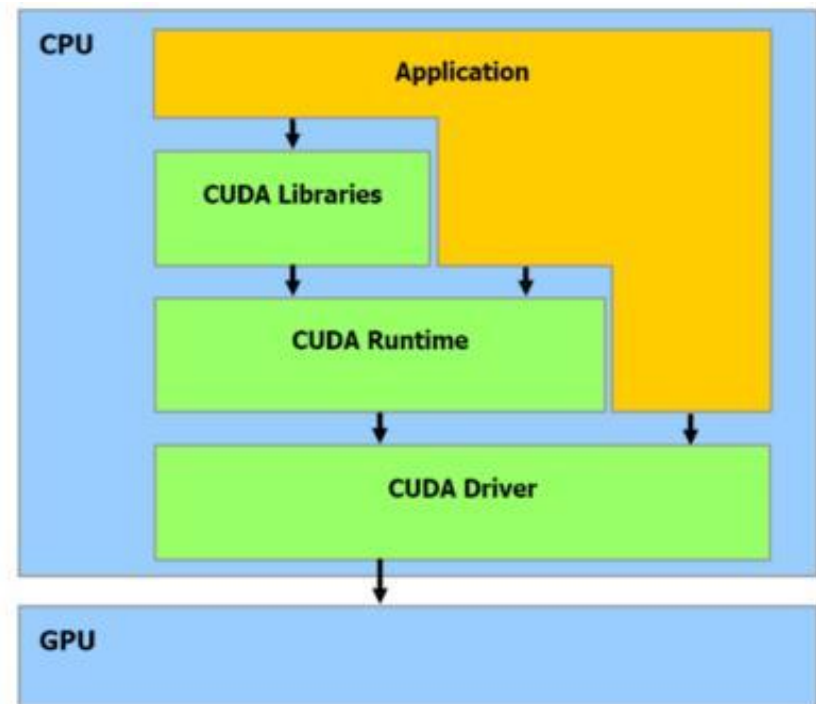
Capability: resources constraints



Technical Specifications	Compute Capability						
	1.0	1.1	1.2	1.3	2.x	3.0	3.5
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2 ³¹ -1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255
Maximum amount of shared memory per multiprocessor	16 KB				48 KB		
Number of shared memory banks	16				32		
Amount of local memory per thread	16 KB				512 KB		
Constant memory size	64 KB						
Cache working set per multiprocessor for constant memory	8 KB						
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB						
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536		

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); // CUDA C 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: very close to the CUDA driver API

Vector add: OpenCL 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);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL);
```

// copy array to the device

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

// 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 NRange

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

Vector add: OpenCL Device Code

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

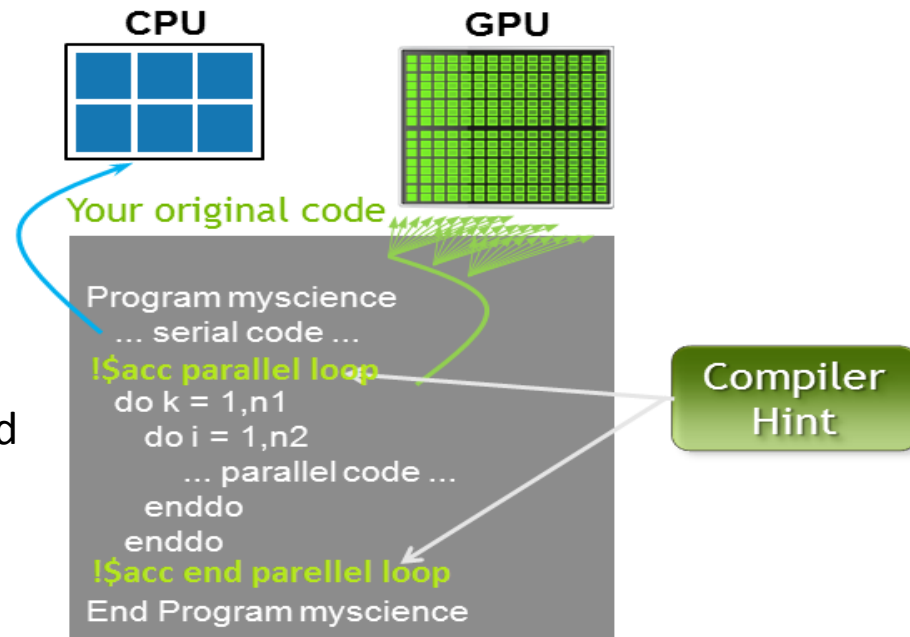
- OpenCL kernels are much like CUDA kernels, but ...
- they are strings loaded at need on the *host* code
- compiled and linked at run-time
- there are such many similarities between OpenCL/CUDA that source-to-source translator programs are growing or on the way (i.e: CU2CL)

OpenACC

- OpenACC is an open standard for parallel programming design to ease the access to heterogeneous architectures
- OpenACC let the programmer insert hints to the compiler (through directives) to identify the regions to be accelerated
- the compiler will interpret these directives and do its best to arrange code to run on any accelerator it can support.

Major advantages:

- High-Level: does require a very small set of instructions to be on the way, with respect to the learning curve required by OpenCL, CUDA, etc.
- Single source: source code remains the same for the serial/parallel/accelerated version
- Portable: provides a good level of support to many accelerators devices from many vendors
- most of these features are now part of the OpenMP v4.x revision



OpenACC: A Simple Example

```
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 is 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

Example: Laplace equation in 2D

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

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

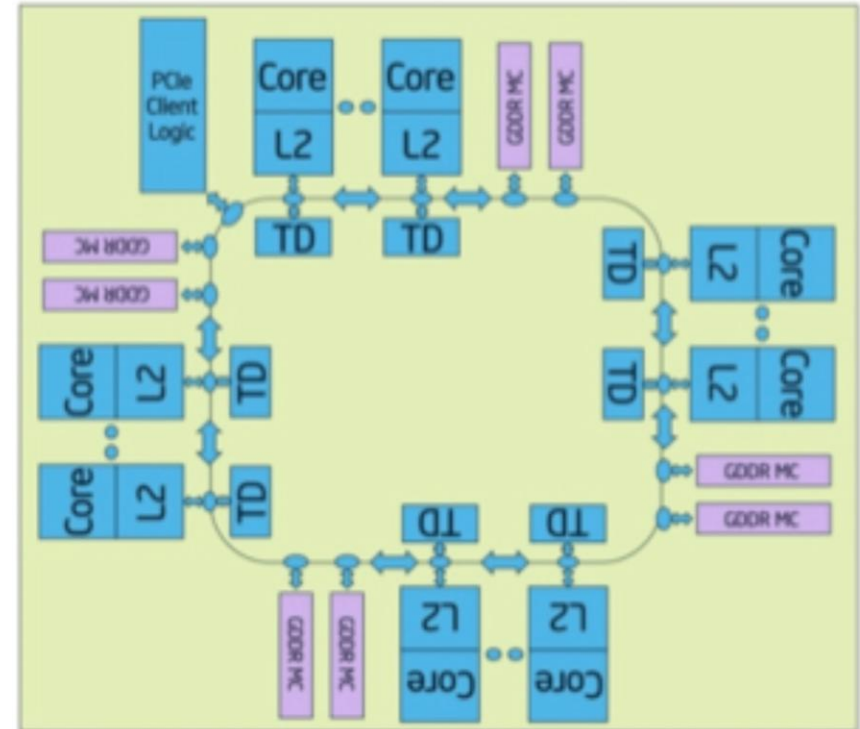
SpeedUp vs 6 CPU cores

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:

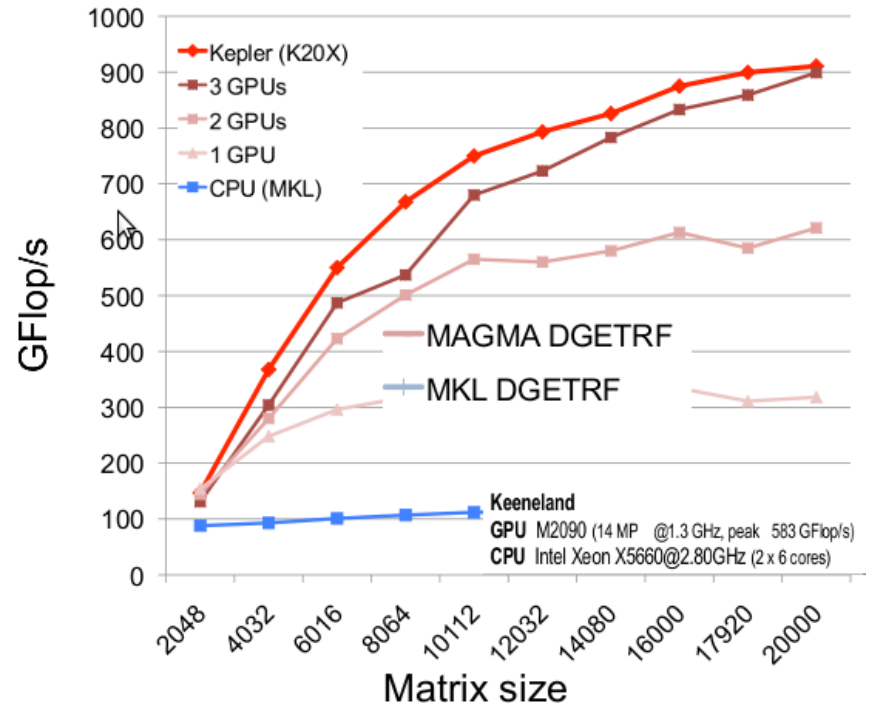
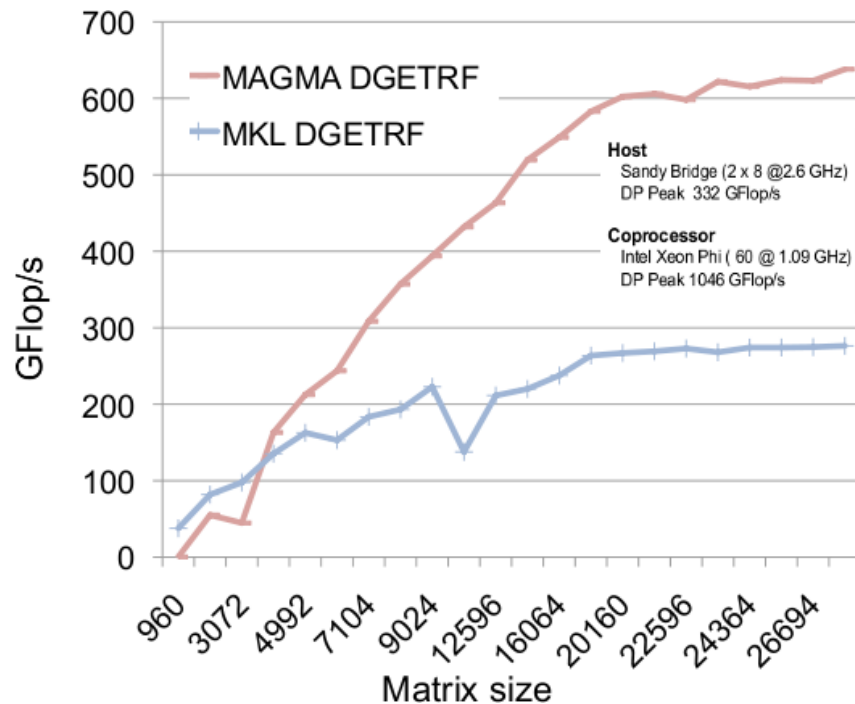
- using pragmas to augment existing codes so they offload work from the host processor to the Intel Xeon Phi coprocessors(s)
- Accessing the coprocessor as an accelerator through optimized libraries such as the Intel MKL (Math Kernel Library)

■ 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 developping 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>