- Introduction to Scientific Programming for GPGPU with CUDA

SCAI
SuperComputing Applications and Innovation

# Rights & Credits

These slides are CINECA 2014 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

http://creativecommons.org/licenses/by-nc-nd/3.0/

Slides and examples were authored by:

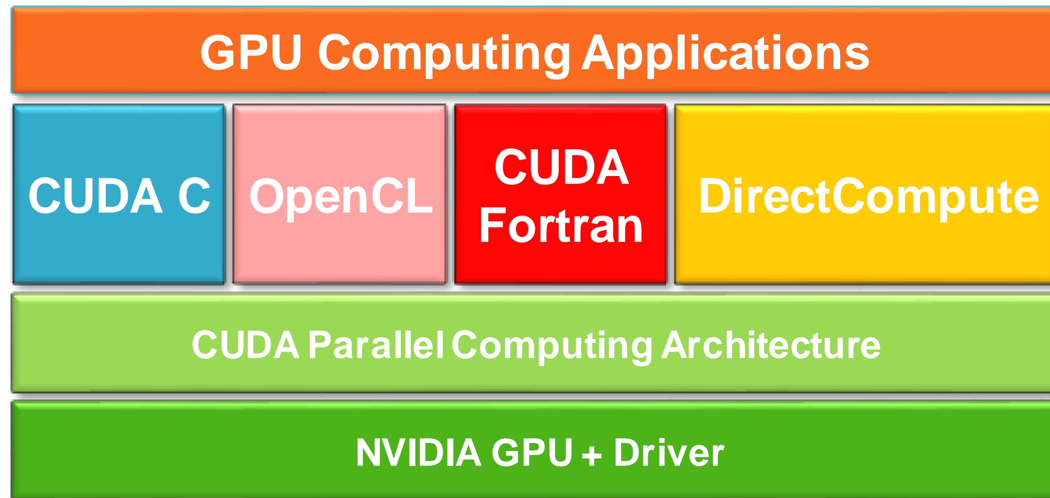
Isabella Baccarelli, Luca Ferraro, Sergio Orlandini

CINECA SCAI SuperComputing Applications and Innovation

# A General-Purpose Parallel Computing Architecture

Compute Unified Device Architecture (NVIDIA 2007)

- 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 extensions to higher level programming language to express thread parallelism into the new PTX instruction set
  - a developer toolkit to compile, debug, profile programs and run them easily in a heterogeneous systems



# A simple CUDA port

```
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);
  }

  ...
}
```

CINECA SCAI SuperComputing Applications and Innovation

# CUDA Kernels

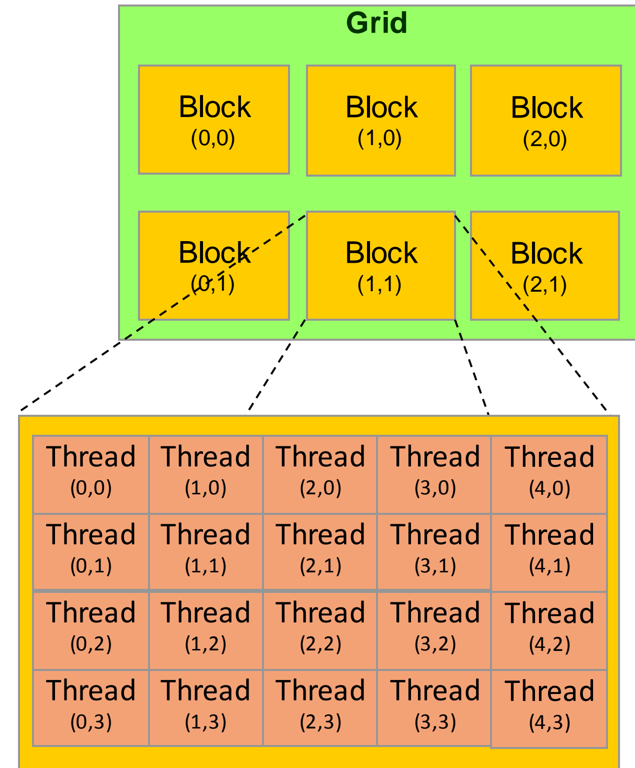
- A CUDA kernel function is defined using the **__global__** declaration
- when a CUDA kernel is call, it will be exectued N times in parallel by N different CUDA threads on the device
- the number of CUDA threads that execute that kernel is specified using a new syntax, called **execution configuration**

```
cudaKernelFunction <<<...>>> (function arguments)
```

- each thread has a unique thread ID
  - the thread ID is accessible within the CUDA kernel scope through the built-in **threadIdx** variable
  - the built-in variables **threadIdx** are a 3-component vector
    - use .x, .y, .z to access its components

# CUDA Threads, Blocks, Grid

- threads are organized into block of threads
  - blocks can be 1D, 2D, 3D sized in threads
- blocks are organized into a grid of blocks
  - each block of thread will be executed independently
    - no assumtion is made on which block is executed first
- 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




`threadIdx:`
thread coordinates inside a block

`blockDim:`
block dimensions in thread units

`blockIdx:`
block coordinates inside the grid

`gridDim:`
grid dimensions in block units

# Simple 1D CUDA vector add

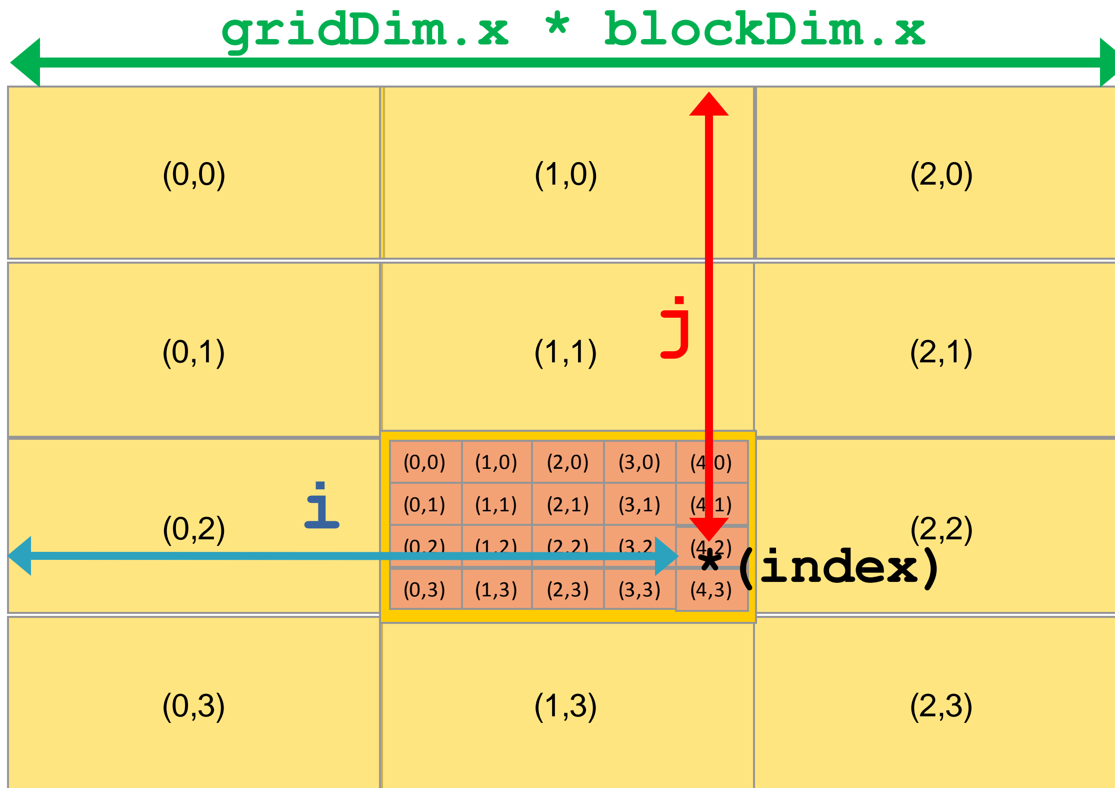
```
__global__
void gpuVectAdd( int N, const double *u, const double *v, double *z)
{
    // use GPU thread id as index
    index = blockIdx.x * blockDim.x + threadIdx.x;

    // 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);
   ...
}
```

# Simple 2D CUDA matrix add

- As an example of a 2D data processing mapping onto CUDA grid of threads, the following code adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*

```
__global__ void matrixAdd(int N, const float *A,  const float *B, float *C) {
  int i = blockIdx * blockDim.x + threadIdx.x;
  int j = blockIdx * 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[]) {
  ...
  dim3 threadsPerBlock(16, 16);
  dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);

  matrixAdd <<< numBlocks, theadsPerBlock >>> (N, A, B, C);
  ...
}
```

# Composing 2D CUDA Thread Indexing




`threadIdx`:
thread coordinates inside a block

`blockIdx`:
block coordinates inside the grid

`blockDim`:
block dimensions in thread units

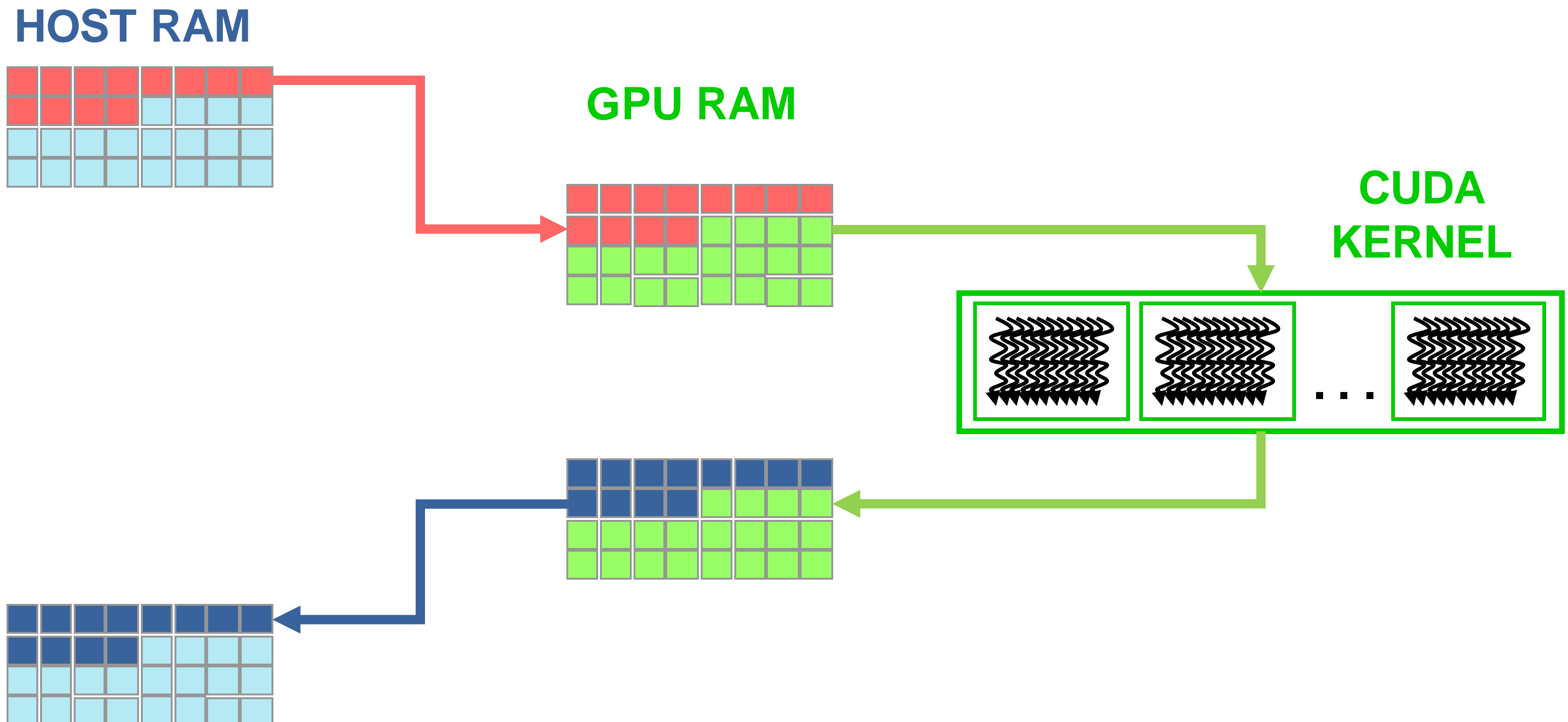
`gridDim`:
grid dimensions in block units

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

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

CINECA SCAI SuperComputing Applications and Innovation

# Data movement

- data must be moved from HOST to DEVICE memory in order to be processed by a CUDA kernel
- when data is processed, and no more needed on the GPU, it is transferred back to HOST



# 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 should 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)

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

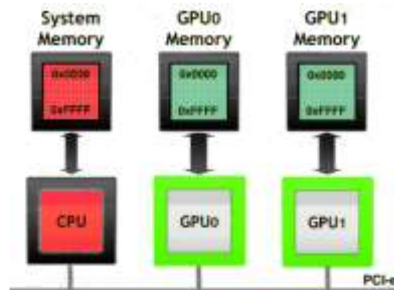
# Memory copy between CPU and GPU

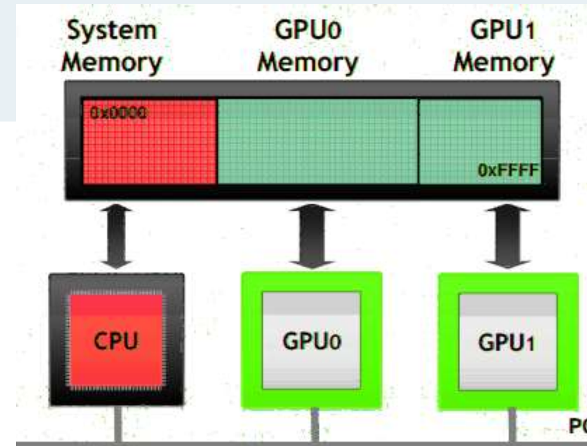
- cudaMemcpy(void *dst, void *src, size_t size, direction)
  - dst: destination buffer pointer
  - src: source buffer pointer
  - size: number of bytes to copy
  - direction: macro name which defines the direction of data copy
    - from CPU to GPU: cudaMemcpyHostToDevice (H2D)
    - from GPU to CPU: 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 transfered
  - 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 |
|---|---|
| una  definizione per ogni permitazione di sorgente/destinazione | una sola definizione direzionale |
| `cudaMemcpyHostToHost`<br>`cudaMemcpyHostToDevice`<br>`cudaMemcpyDeviceToHost`<br>`cudaMemcpyDeviceToDevice` | `cudaMemcpyDefault` |



# CUDA 6.x - Unified Memory

- CUDA 6.0 introduces a mechanism to fully control which memory segment should be accessed by a kernel and how long data should persist in
- Unified Memory creates a pool of managed memory that is shared between the CPU and GPU
- Managed memory is accessible to both the CPU and GPU using a single pointer
- the system *automatically* migrates data allocated in Unified Memory between host and device
  - 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
- yet, it can result in degraded performances with respect direct and explicit control of data movements

# 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);
}
```

CINECA SCAI SuperComputing Applications and Innovation

# Three steps for a CUDA port

1. identify data-parallel computational intensive parts
   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 thead 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 parts

```c
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;
}
```

```fortran
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
```

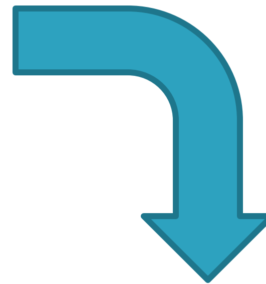
CINECA SCAI
SuperComputing Applications and Innovation

# Vector Sum

2. translate identified parallel parts into CUDA kernels

- each *thread* executes the same kernel, but acts 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];
```

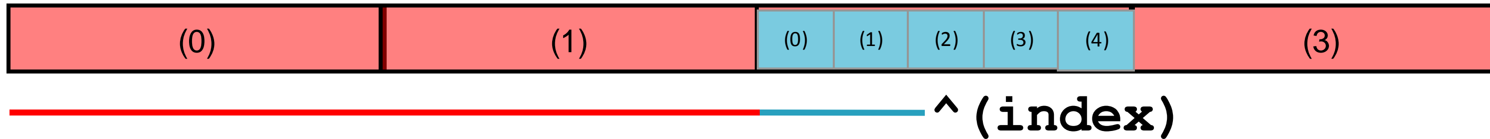


```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
  // index is a unique identifier for each GPU thread
  int index =  blockIdx * blockDim.x + threadIdx.x ;
  if (index < N)
     z[index] = u[index] + v[index];
}
```

CINECA SCAI SuperComputing Applications and Innovation

# Vector Sum

2. translate identified parallel parts 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];
}
```

**__global__**

qualifier which declare a CUDA kernel

CUDA kernels are special 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 immediatelly to the host code
- an explicit synchronization should be used to check if a CUDA kernel has completed

# Vector Sum

2. translate identified parallel parts into CUDA kernels

```
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
```

CINECA SCAI SuperComputing Applications and Innovation

# Vector Sum

2. translate identified parallel parts into CUDA kernels

```
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 kernel are not defined inside modules, you must provide an interface for each kernel

# Vector Sum

## 3. manage memory transfers and kernel calls

- API CUDA C: `cudaMalloc(void **p, size_t size)`
  - allocate size bytes of GPU global memory
  - return an address of the *device* memory space

```
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));
```

- CUDA Fortran programmers can allocate arrays on GPU global memory declaring them with the DEVICE attribute
- DEVICE arrays can be allocated with a standard ALLOCATE

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

allocate( u_dev(N), v_dev(N), z_dev(N) )
```

CINECA SCAI SuperComputing Applications and Innovation

# Vector Sum

3. manage memory transfers and kernel calls

```
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 copy
    - we need to pass the address of u_dev to let its value be modified after the function call
    - 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 ....

- API CUDA C:

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

  - copy size bytes starting from dst pointer to src pointer memory

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

- CUDA Fortran programmers can transfer data from a HOST array to a DEVICE array, and vice versa, simply using the assignment operator

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

# Vector Sum

## 3. manage memory transfers and kernel calls

Insert calls to CUDA kernels using the execution configuration syntax:

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

specifing 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 )
```

CINECA SCAI SuperComputing Applications and Innovation

# 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
```