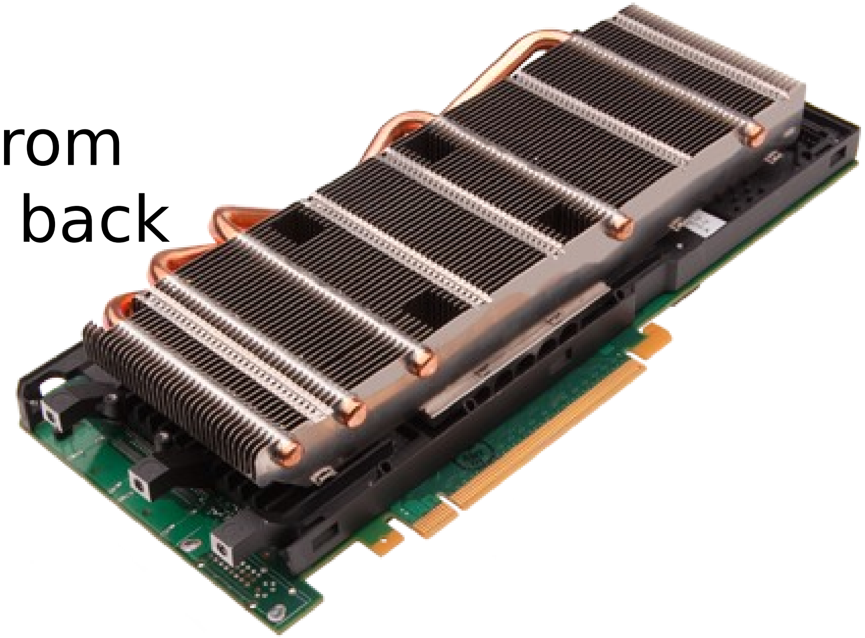
Programming paradigms for GPU devices

28th Summer School on Parallel Computing

1-12 July 2019

Sergio Orlandini
s.orlandini@cineca.it

- GPU programming model
  - Design a porting to GPU architecture
  - Thread hierarchy and indexing
  - Writing a kernel
  - Handling data transfers from CPU to GPU memory and back
  - Vector-Vector Add
  - Write and launch a GPU program



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 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

Identify data-parallel intensive portions

```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
```

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 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

Manage kernel calls

Insert calls to CUDA kernels using the execution configuration syntax:

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

specifying the thread/block hierarchy you want to apply:

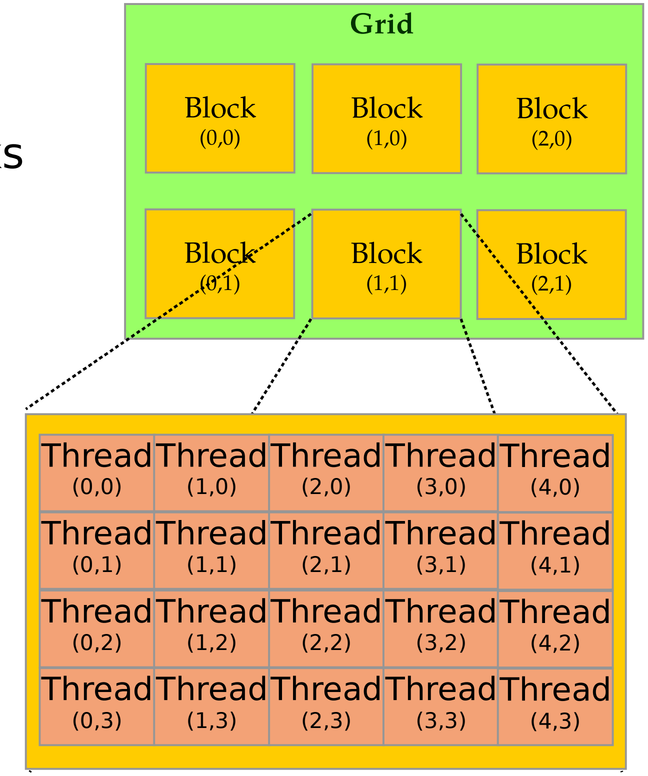
- **numBlocks:** specify grid size in terms of thread blocks along each dimension
- **numThreads:** specify the block size in terms of threads along each dimension

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

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

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

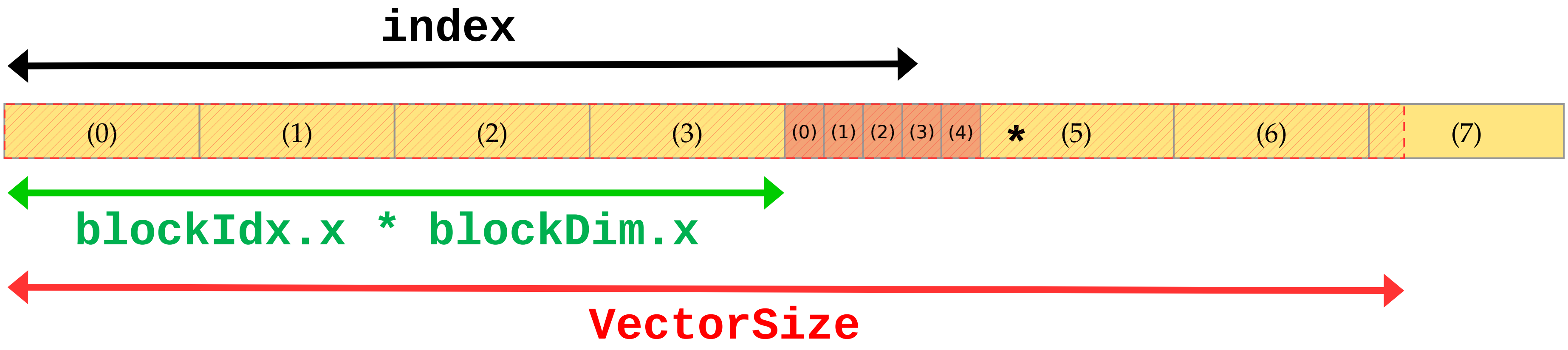
`threadIdx:`
thread coordinates inside a block

`blockIdx:`
block coordinates inside the grid

`blockDim:`
block dimensions in thread units

`gridDim:`
grid dimensions in block units

Composing 1D CUDA Thread Indexing




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

`threadIdx`: thread coordinates inside a block

`blockIdx`: block coordinates inside the grid

`blockDim`: block dimensions in thread units

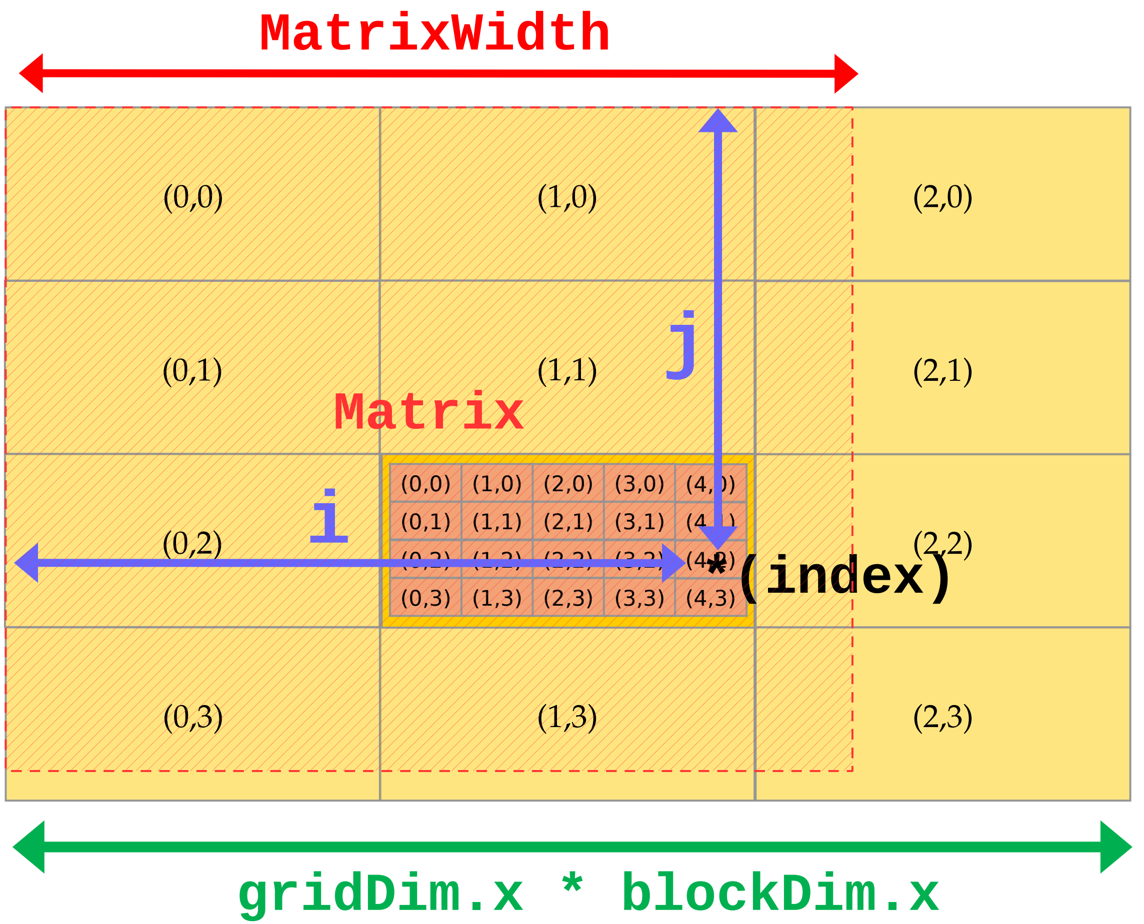
`gridDim`: grid dimensions in block units

CUDA Vector add – 1D thread grid

```
__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 - 1) / blockSize.x + 1;

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

CINECA SCAI SuperComputing Applications and Innovation

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

CUDA Matrix add - 2D thread grid

```
__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 = j * N + i;

  if ( i < N && j < N )
    C[index] = A[index] + B[index];
}

int main(int argc, char *argv[]) {
  ...
  // use 2D block threads
  dim3 blockSize(32,32);
  // use 2D grid blocks
  dim3 gridSize( (N-1)/block.x + 1, (N-1)/block.y + 1 );
  // add NxN matrices on GPU
  matrixAdd <<< gridSize, blockSize >>> (N, A, B, C);
  ...
}
```

CINECA SCAI
SuperComputing Applications and Innovation

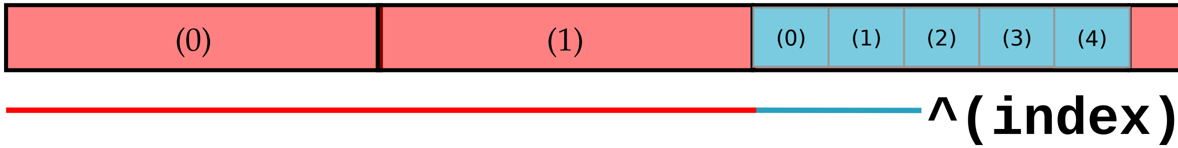
Translate parallel portions into kernels

- 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 * blockDim.x + threadIdx.x ;
  if (index < N)
     z[index] = u[index] + v[index];
}
```

Translate parallel portions into 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

Translate parallel portions into 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
```

Translate parallel portions into 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 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.

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

CINECA SCAI SuperComputing Applications and Innovation

Memory allocation on GPU device

```
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…

Memory allocation on GPU device

- 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) )
```

Memory Initialization on GPU device

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

It fills the first count <u>bytes</u> 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.

CINECA SCAI SuperComputing Applications and Innovation

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

Manage memory transfers

- 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

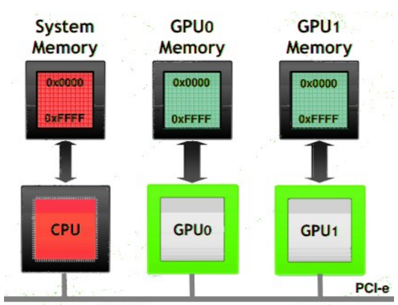
```
u_dev = u ; v_dev = v
```

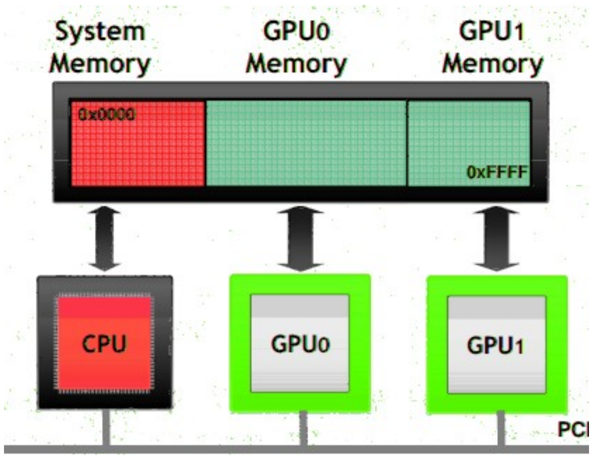
CUDA 4.x - Unified Virtual Addressing

- 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.

`cudaMemcpyHostToHost`
`cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost`
`cudaMemcpyDeviceToDevice`
	`cudaMemcpyDefault`





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);
}
```

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
```

CINECA SCAI SuperComputing Applications and Innovation

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini