

Programming paradigms for GPU devices



Day 1

Sergio Orlandini
s.orlandini@ Cineca.it

Luca Ferraro
l.ferraro@ Cineca.it

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

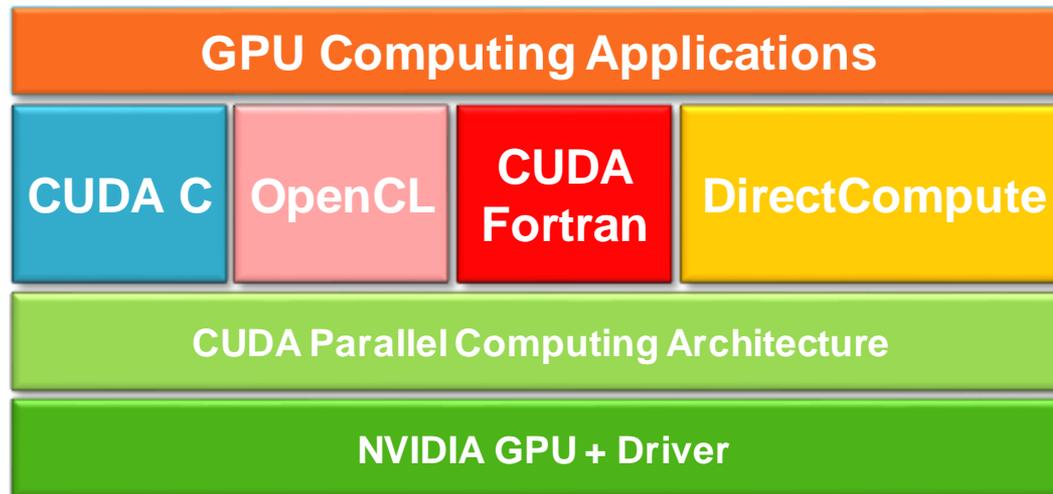
- CUDA programming model
 - Heterogeneous execution
 - Writing a CUDA Kernels
 - Thread Hierarchy
- Getting started with CUDA programming:
 - the Vector-Vector Add
 - handling data transfers from CPU to GPU memory and back
 - write and launch the CUDA program



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.



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 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];
}
```

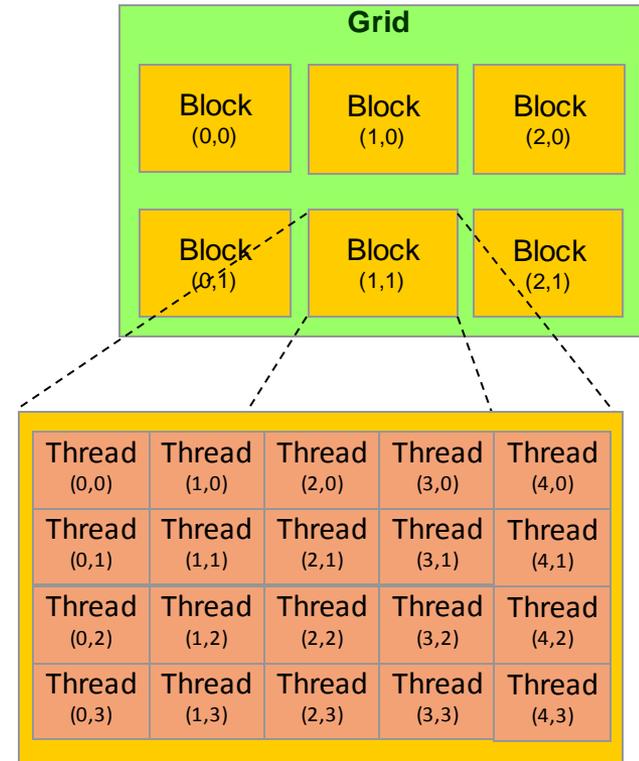
```
int main(int argc, char *argv[]) {
    ...

    // 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;

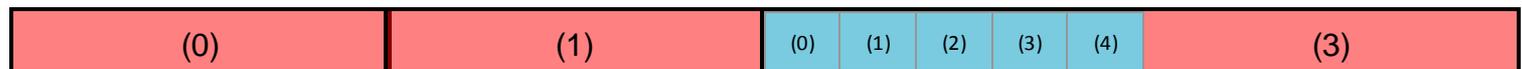
    // check out of border access
    if ( index < N ) {
        z[index] = u[index] + v[index];
    }
}

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

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

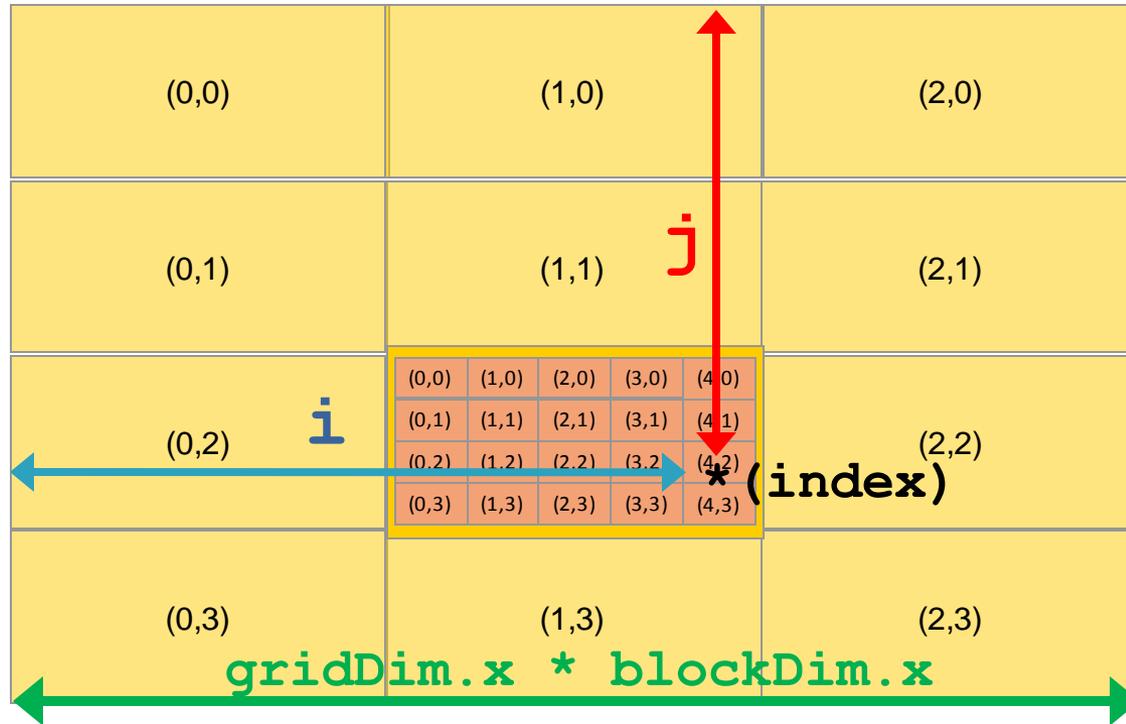
    // use 1D grid blocks
    dim3 gridSize = (N + blockSize-1) / blockSize.x;

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



^ (index)

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 <<< gridDim, blockDim >>> (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.

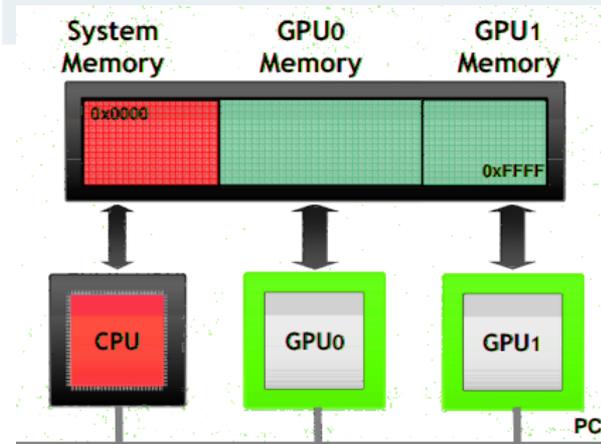
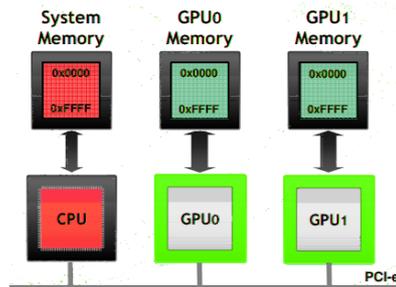
Memory copy between CPU and GPU

- `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

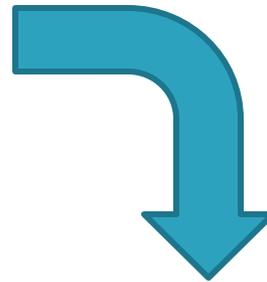
call printVector (z, N)

end program
```

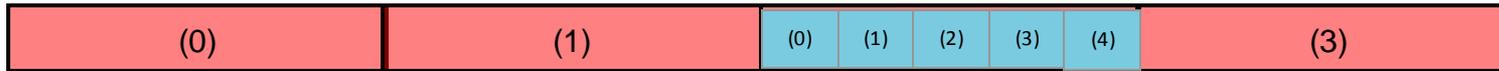
- each *thread* execute the same kernel, but act on different data:
 - turn the loop into a CUDA kernel function
 - map each CUDA *thread* onto a unique index to access data
 - let each *thread* retrieve, compute and store its own data using the unique address
 - prevent out of border access to data 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];
}
```



^ (index)

```
__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 kernels are not defined within a module, then an explicit interface must be provided for each kernel.

- **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
```