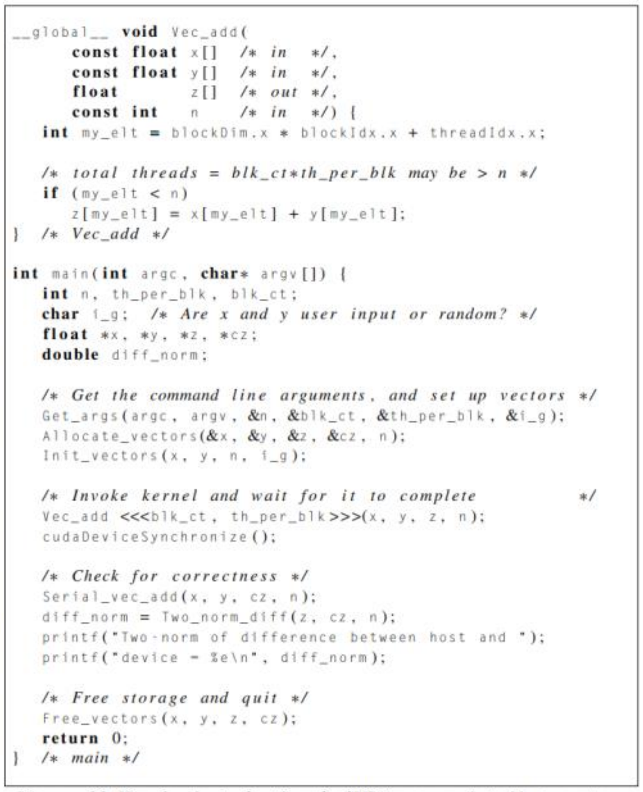
Vector addition

GPUs and CUDA are designed to be especially effective when they run data-parallel programs.

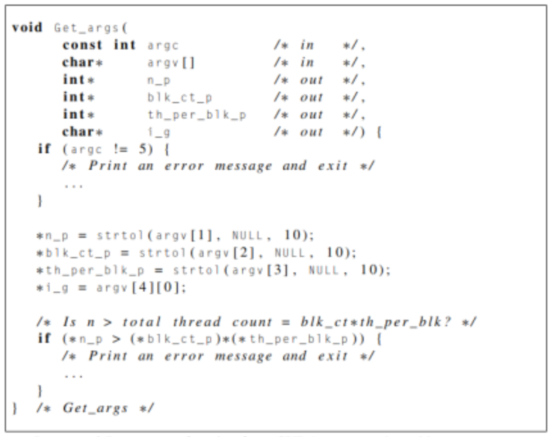
So let’s write a very simple, data-parallel CUDA program that’s embarrassingly parallel: a program that adds two vectors or arrays.

- We will initialize three arrays: x, y, z (will be the array with the computation of adding x[i] + y[i]).
    - x and y will be initialized on the host
    - Since GPU's tend to have more 32 bit than 64 bit floating points we will use float arrays instead of doubles
    - `float *x, *y, *z;`
- The kernel will then start n threads and each thread will take care of one iteration of appending to z.
- After allocating and initializing the arrays, we’ll call the kernel, and after the kernel completes execution, the program checks the result, frees memory, and quits.
- The program (complete with all of the helper functions except for Two_norm_diff function, serial_vec_add and Init_vectors):



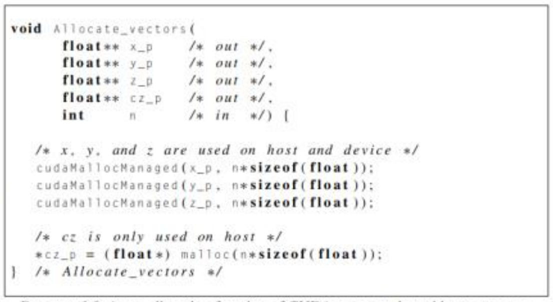
```
 1  __global__ void Vec_add(
 2        const float x[]   /* in   */,
 3        const float y[]   /* in   */,
 4        float       z[]   /* out  */,
 5        const int   n     /* in   */) {
 6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
 7
 8     /* total threads = blk_ct*th_per_blk may be > n */
 9     if (my_elt < n)
10        z[my_elt] = x[my_elt] + y[my_elt];
11  }  /* Vec_add */
12
13  int main(int argc, char* argv[]) {
14     int n, th_per_blk, blk_ct;
15     char i_g;  /* Are x and y user input or random? */
16     float *x, *y, *z, *cz;
17     double diff_norm;
18
19     /* Get the command line arguments, and set up vectors */
20     Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
21     Allocate_vectors(&x, &y, &z, &cz, n);
22     Init_vectors(x, y, n, i_g);
23
24     /* Invoke kernel and wait for it to complete          */
25     Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
26     cudaDeviceSynchronize();
27
28     /* Check for correctness */
29     Serial_vec_add(x, y, cz, n);
30     diff_norm = Two_norm_diff(z, cz, n);
31     printf("Two-norm of difference between host and ");
32     printf("device = %e\n", diff_norm);
33
34     /* Free storage and quit */
35     Free_vectors(x, y, z, cz);
36     return 0;
37  }  /* main */
```

Program 6.3: Kernel and main function of a CUDA program that adds two vectors.



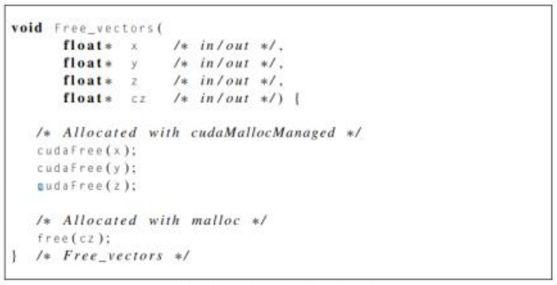
```
void Get_args(
      const int  argc           /* in   */,
      char*      argv[]         /* in   */,
      int*       n_p            /* out  */,
      int*       blk_ct_p       /* out  */,
      int*       th_per_blk_p   /* out  */,
      char*      i_g            /* out  */) {
   if (argc != 5) {
      /* Print an error message and exit */
      ...
   }

   *n_p = strtol(argv[1], NULL, 10);
   *blk_ct_p = strtol(argv[2], NULL, 10);
   *th_per_blk_p = strtol(argv[3], NULL, 10);
   *i_g = argv[4][0];

   /* Is n > total thread count = blk_ct*th_per_blk? */
   if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {
      /* Print an error message and exit */
      ...
   }
}  /* Get_args */
```

Program 6.5: Get_args function from CUDA program that adds two vectors.



```
void Allocate_vectors(
      float** x_p    /* out */,
      float** y_p    /* out */,
      float** z_p    /* out */,
      float** cz_p   /* out */,
      int     n      /* in  */) {

   /* x, y, and z are used on host and device */
   cudaMallocManaged(x_p, n*sizeof(float));
   cudaMallocManaged(y_p, n*sizeof(float));
   cudaMallocManaged(z_p, n*sizeof(float));

   /* cz is only used on host */
   *cz_p = (float*) malloc(n*sizeof(float));
}  /* Allocate_vectors */
```

Program 6.6: Array allocation function of CUDA program that adds two vectors.



```
void Free_vectors(
      float*  x    /* in/out */,
      float*  y    /* in/out */,
      float*  z    /* in/out */,
      float*  cz   /* in/out */) {

   /* Allocated with cudaMallocManaged */
   cudaFree(x);
   cudaFree(y);
   cudaFree(z);

   /* Allocated with malloc */
   free(cz);
}  /* Free_vectors */
```

Program 6.8: CUDA function that frees four arrays.

The Kernel

- In lines 1-11 of the kernel (function that is running the parallel code on the GPU) we first determine which element of z (the array we want to append to) the thread should take care of
    - Essentially which iteration the thread should take care of
        - As a side not, we can do gridDim.x * blockDim.x which will equal the total number of threads we have.

        - In this program we expect the have more threads than iterations
    - We want to assign a unique "global" (meaning a rank number which takes into account all of the other threads) to each thread by using the formula
        - `rank = blockDim.x * blockIdx.x + threadIdx.x`   My_rank can also be my_elt
            - BlockDim.x is the amount of total blocks we have
            - blockInd.x is the current "rank" of the block we are in
            - threadIdx.x is the current thread we are in, respective of the block we are in too
        - For example,

Table 6.4 Global thread ranks or indexes in a grid with 4 blocks and 5 threads per block.

	threadIdx.x				
blockIdx.x	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

- Before running the code on line 10 which computes and appends to the z array we want to run an if statement that checks to see if the current global rank for the thread we are in doesn't exceed the z array size (n)
    - We could have more threads than size of array and in that case we would simply not do anything in those threads
    - `z[my_elt] = x[my_elt] + y[my_elt];`

The Get_args() function

After declaring the variables, the `main` function calls a `Get_args` function, which returns *n*, the number of elements in the arrays, `blk_ct`, the number of thread blocks, and `th_per_blk`, the number of threads in each block. It gets these from the command line. It also returns a **char** `i_g`. This tells the program whether the user will *i*nput `x` and `y` or whether it should *g*enerate them using a random number generator. If the user doesn't enter the correct number of command line arguments, the function prints a usage summary and terminates execution. Also if *n* is greater than the total number of threads, it prints a message and terminates. (See Program 6.5.) Note that `Get_args` is written in standard C, and it runs completely on the host.

- The get_args function handles asking the user through the command line for things needed by the program
- The i_g variable is obtained by asking the user whether they want a randomely generated x and y or if they want to input it
- We also handle the error here where there are not enough threads to handle the size of the array

The Allocate_vectors and managed memory

- In the Allocate_vectors function we allocate memory to the arrays x, y, z, cz (of type float)
    - x, y, and z are all used on both the host and device
        - For this reason, we allocate them memory using a CUDA function

```
__host__ cudaError_t cudaMallocManaged (
                 void**      devPtr    /* out */,
                 size_t      size      /* in  */,
                 unsigned    flags     /* in  */);
```

            - The host qualifier (__host__) is a CUDA addition to C and it indicates that the function should be called and ran on the host
                - The __host__ qualifier is by default in every CUDA function
            - FYI, most CUDA functions have a return type of cudaError_t to tell you if there was an error, but for not we will ignore these return statements
            - The first argument is a pointer to a pointer: it refers to the pointer that's being allocated. The second argument specifies the number of bytes that should be allocated. The `flags` argument controls which kernels can access the allocated memory. It defaults to `cudaMemAttachGlobal` and can be omitted.
            - The function cudaMallocManaged is one of several CUDA memory allocation functions. It allocates memory

that will be automatically managed by the "unified memory system."
- This is a relatively recent addition to CUDA and it allows a programmer to write CUDA programs as if the host and device shared a single memory: pointers referring to memory allocated with cudaMallocManaged can be used on both the device and the host, even when the host and the device have separate physical memories.
  - It simplifies programming but it can have some drawbacks:

    1. Unified memory requires a device with compute capability ≥ 3.0, and a 64-bit host operating system.
    2. On devices with compute capability < 6.0 memory allocated with `cudaMallocManaged` cannot be simultaneously accessed by both the device and the host. When a kernel is executing, it has exclusive access to memory allocated with `cudaMallocManaged`.
    3. Kernels that use unified memory can be slower than kernels that treat device memory as separate from host memory.

    - For number 3, it has to do with data transfer b/w cpu and gpu and when we use this function it becomes up to the system to decide when to transfer data which can be slow.
    - A programmer could choose not to use this function and explicitly write their own which could make the program more efficient if they know how write code to reduce the cost of transferring data.

- The cz array is only used on the host and its purpose it to compute the sum of z using only one core from the CPU
  - In this case we simply use the C function malloc

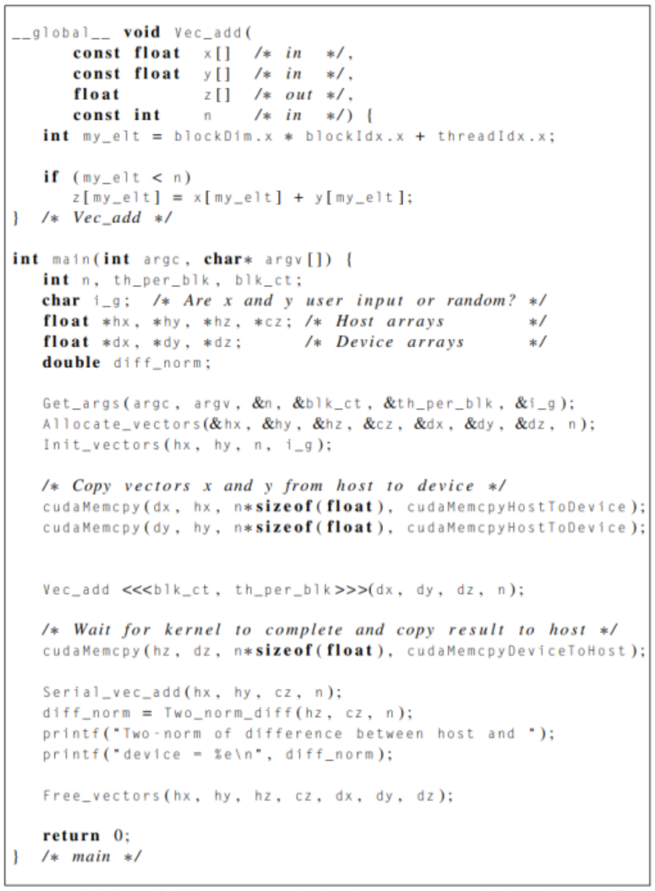
Other functions called from main
- Except for the Free_vectors function, the other host functions that we call from main are just standard C.
- Other functions that are standard C functions:
  - The function Init_vectors either reads x and y from stdin using scanf or generates them using the C library function random. It uses the last command line argument i_g to decide which it should do.
  - The Serial_vec_add function (Program 6.4) just adds x and y on the host using a for loop. It stores the result in the host array cz.
  - The `Two_norm_diff` function computes the "distance" between the vector `z` computed by the kernel and the vector `cz` computed by `Serial_vec_add`. So it takes the difference between corresponding components of `z` and `cz`, squares them, adds the squares, and takes the square root:

$$\sqrt{(z[0]-cz[0])^2+(z[1]-cz[1])^2+\cdots+(z[n-1]-cz[n-1])^2}.$$

- Free_vectors function (CUDA involved function)
  - This function just frees the arrays allocated by the Allocate_vectors function
  - The array cz is simply freed using the C library free() since it is a regularly initialized array using malloc
  - The other arrays x, y, z that were allocated using cudaMallocManaged must be freed using cudaFree function:
    - `__host__ __device__ cudaError_t cudaFree ( void* ptr )`
      - The __device__ qualifier is a CUDA addition to C, and it indicates that the function can be called from the device
      - So cudaFree can be called by both the host and device since its using both qualifiers
      - One thing to note: if a pointer is allocated on the device, it cannot be freed on the host, and vice versa.
        - This means if you allocated things on the device you must deallocate it there and if you allocated things in the host you must deallocate there
        - It's important to note that unless memory allocated on the device is explicitly freed by the program, it won't be freed until the program terminates. So if a CUDA program calls two (or more) kernels, and the memory used by the first kernel isn't explicitly freed before the second is called, it will remain allocated, regardless of whether the second kernel actually uses it.

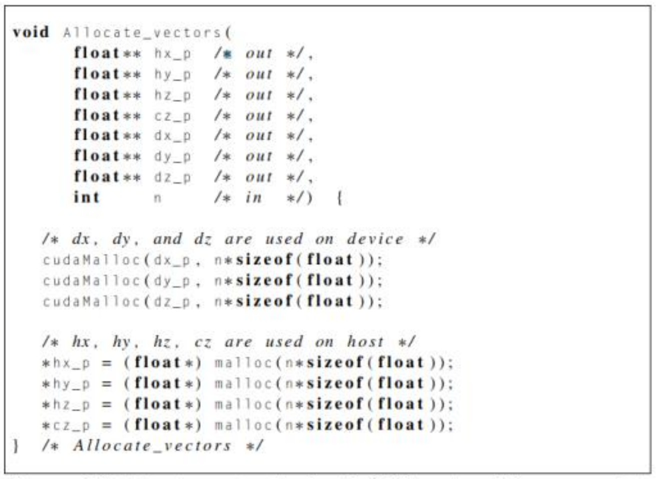
Explicit memory transfers
- In our program for adding vectors we used the CUDA function cudaMallocManaged() to initialize the arrays that were going to be used by both the host and device.
  - Remember, we used this function because it takes care of data transfer b/w CPU and GPU for us and makes for less

    coding to write
    - It has its drawbacks
- Here we will rewrite our program without using cudaMallocManaged and explicitly define the transfer of data
- Here is the new program:



```
__global__ void Vec_add(
      const float  x[]  /* in   */,
      const float  y[]  /* in   */,
      float        z[]  /* out  */,
      const int    n    /* in   */) {
   int my_elt = blockDim.x * blockIdx.x + threadIdx.x;

   if (my_elt < n)
      z[my_elt] = x[my_elt] + y[my_elt];
}  /* Vec_add */

int main(int argc, char* argv[]) {
   int n, th_per_blk, blk_ct;
   char i_g;  /* Are x and y user input or random? */
   float *hx, *hy, *hz, *cz; /* Host arrays       */
   float *dx, *dy, *dz;      /* Device arrays     */
   double diff_norm;

   Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
   Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz, n);
   Init_vectors(hx, hy, n, i_g);

   /* Copy vectors x and y from host to device */
   cudaMemcpy(dx, hx, n*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(dy, hy, n*sizeof(float), cudaMemcpyHostToDevice);


   Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);

   /* Wait for kernel to complete and copy result to host */
   cudaMemcpy(hz, dz, n*sizeof(float), cudaMemcpyDeviceToHost);

   Serial_vec_add(hx, hy, cz, n);
   diff_norm = Two_norm_diff(hz, cz, n);
   printf("Two-norm of difference between host and ");
   printf("device = %e\n", diff_norm);

   Free_vectors(hx, hy, hz, cz, dx, dy, dz);

   return 0;
}  /* main */
```

Program 6.9: Part of CUDA program that implements vector addition without unified memory.



```
void Allocate_vectors(
      float** hx_p  /* out */,
      float** hy_p  /* out */,
      float** hz_p  /* out */,
      float** cz_p  /* out */,
      float** dx_p  /* out */,
      float** dy_p  /* out */,
      float** dz_p  /* out */,
      int     n     /* in  */)  {

   /* dx, dy, and dz are used on device */
   cudaMalloc(dx_p, n*sizeof(float));
   cudaMalloc(dy_p, n*sizeof(float));
   cudaMalloc(dz_p, n*sizeof(float));

   /* hx, hy, hz, cz are used on host */
   *hx_p = (float*) malloc(n*sizeof(float));
   *hy_p = (float*) malloc(n*sizeof(float));
   *hz_p = (float*) malloc(n*sizeof(float));
   *cz_p = (float*) malloc(n*sizeof(float));
}  /* Allocate_vectors */
```

Program 6.10: Allocate_vectors function for CUDA vector addition program that doesn't use unified memory.

- This is a program that does not provide unified memory (which came from cudaMallocManaged)
- The first things to note is that much of the code is similar
    - The kernel arguments are unchanged: x,y,z and n. And it still does the same thing.
- The main function is pretty similar but we do have some differences due to no unified memory
    - Since unified memory isnt assumed, the pointers we had on the host arent valid on the device and so we must handle that by making arrays for both host and device
        - an address on the host may be illegal on the device, or, even worse, it might refer to memory that the device is using for some other purpose.
        - Similar problems occur if we try to use a device address on the host.
        - We must combat this by declaring three arrays that are valid on the host and three arrays that are valid on the device: hx,hy,hz for the host and dx,dy,dz for the device
            - The arrays will be used as the arrays that both sides will use when talking to each other
    - We declared these arrays but now we allocate space for these arrays in the Allocate_vectors() function passing in all of the arrays and other args
        - Since we arent using unified memory by way of cudaMallocManaged we will use ordinary C malloc calls for the host arrays and the CUDA function cudaMalloc() (which is different of course from cudaMallocManaged) for the device arrays
            -
```
__host__  __device__  cudaError_t cudaMalloc (
                      void**  dev_p   /* out */,
                      size_t  size    /* in  */);
```
                - The first argument is a reference to a pointer that will be used on the device.
                - The second argument specifies the number of bytes to allocate on the device.

- Now that all the arrays have been allocated, we then initialize the host arrays with data using the helper function Init_vectors() to give the host arrays their data
- We will not copy over the data that exists in the host arrays to the device arrays using a CUDA function

  -
    ```
    __host__ cudaError_t cudaMemcpy (
                  void*            dest     /* out */,
                  const void*      source   /* in  */,
                  size_t           count    /* in  */,
                  cudaMemcpyKind   kind     /* in  */);
    ```
    - This copies count bytes from the memory referred to by source into the memory referred to by dest. The type of the kind argument, cudaMemcpyKind, is an enumerated type defined by CUDA that specifies where the source and dest pointers are located. For our purposes the two values of interest are cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost. The first indicates that we're copying from the host to the device, and the second indicates that we're copying from the device to the host.
- The call to the kernel in Line 28 uses the pointers dx, dy, and dz, because these are addresses that are valid on the device.
- After the call to the kernel, we copy the result of the vector addition from the device to the host in Line 31 using cudaMemcpy again.
  - The z variable is now filled with data from the kernel and we tranfer it back into the variable hz which the device can work with
- Note:
  - A call to cudaMemcpy is synchronous, so it waits for the kernel to finish executing before carrying out the transfer. So in this version of vector addition we do not need to use cudaDeviceSynchronize to ensure that the kernel has completed before proceeding.
- After copying the result from the device back to the host, the program checks the result, frees the memory allocated on the host and the device, and terminates. So for this part of the program, the only difference from the original program is that we're freeing seven pointers instead of four. As before, the `Free_vectors` function frees the storage allocated on the host with the C library function `free`. It uses `cudaFree` to free the storage allocated on the device.