Returning results from CUDA kernels

Some quick things about CUDA kernels

- All CUDA kernels have a return type of void, so they cannot return a value
- They also cant return anything to the host via standard C pass by reference
  - pass by reference: when you pass a function the address of a variable and it can modify that value in the function
  - The reason for this is that addresses on the host are, in most systems, invalid on the device, and vice versa.
  - Example:

```
__global__ void Add(int x, int y, int* sum_p) {
   *sum_p = x + y;
}  /*  Add */

int main(void) {
   int sum = -5;
   Add <<<1, 1>>> (2, 3, &sum);
   cudaDeviceSynchronize();
   printf("The sum is %d\n", sum);

   return 0;
}
```

    - This will either print -5 or the device will hang
    - The reason is that the address &sum is probably invalid on the device, thus *sum_p = x +y is t trying to assign a value to an invalid memory location

There are approaches to returning a "result" to the host from a kernel

- One is to declare pointer variables and allocate a single memory location.
  - On a system that supports unified memory, the computed value will be automatically copied back to host memory:

```
__global__ void Add(int x, int y, int* sum_p) {
   *sum_p = x + y;
}  /*  Add */
```

Unified memory in this program can be noted since we are using cudaMallocManaged

6.9 Returning results from CUDA

```
int main(void) {
   int* sum_p;
   cudaMallocManaged(&sum_p, sizeof(int));
   *sum_p = -5;
   Add <<<1, 1>>> (2, 3, sum_p);
   cudaDeviceSynchronize();
   printf("The sum is %d\n", *sum_p);
   cudaFree(sum_p);

   return 0;
}
```

  - If your system doesn't support unified memory, the same idea will work, but the result will have to be explicitly copied from the device to the host:

```
__global__ void Add(int x, int y, int *sum_p) {
   *sum_p = x + y;
}  /*  Add */

int main(void) {
   int *hsum_p, *dsum_p;
   hsum_p = (int*) malloc(sizeof(int));
   cudaMalloc(&dsum_p, sizeof(int));
```

Note:

- I think this program is missing a second cudaMemcpy to copy the hsum_p value of -5 into the device variable dsum_p
  - This would probably go right before the call the kernel

```
__global__ void Add(int x, int y, int *sum_p) {
   *sum_p = x + y;
}  /*  Add */

int main(void) {
   int *hsum_p, *dsum_p;
   hsum_p = (int*) malloc(sizeof(int));
   cudaMalloc(&dsum_p, sizeof(int));
   *hsum_p = -5;
   Add <<<1, 1>>> (2, 3, dsum_p);
   cudaMemcpy(hsum_p, dsum_p, sizeof(int),
         cudaMemcpyDeviceToHost);
   printf("The sum is %d\n", *hsum_p);
   free(hsum_p);
   cudaFree(dsum_p);

   return 0;
}
```

Note:
- I think this program is missing a second cudaMemcpy to copy the hsum_p value of -5 into the device variable dsum_p
  - This would probably go right before the call the kernel

- Note that in both the unified and non-unified memory settings, we're returning a single value from the device to the host.
- If unified memory is available, another option is to use a global managed variable for the sum:

```
__managed__ int sum;

__global__ void Add(int x, int y) {
   sum = x + y;
}  /*  Add */

int main(void) {
   sum = -5;
   Add <<<1, 1>>> (2, 3);
```

Note:
- The qualifier __managed__ declares sum to be a managed int that is accessible to all the functions, regardless of whether they run on the host or the device
- Since it's managed, the same restrictions apply to it that apply to managed variables allocated with cudaMallocManaged.
  So this option is unavailable on systems with compute capability < 3.0, and on systems with compute capability < 6.0, sum can't be accessed on the host while the kernel is running.
  - So after the call to Add has started, the host can't access sum until after the call to cudaDeviceSynchronize has completed.
- Since this last approach uses a global variable, it has the usual problem of reduced modularity associated with global variables


```
   cudaDeviceSynchronize();
   printf("After kernel:  The sum is %d\n", sum);

   return 0;
}
```