Programming the GPU

- The challenge for the GPU programmer is not simply getting good performance on the GPU, but also in coordinating the scheduling of computation on the system processor and the GPU and the transfer of data between system memory and GPU memory.

- NVIDIA decided to develop a C-like language and programming environment that would improve the productivity of GPU programmers. . The name of their system is CUDA, for Compute Unified Device Architecture. CUDA produces C/C++ for the system processor (host) and a C and C++ dialect for the GPU (device, hence the D in CUDA). A similar programming language is OpenCL, which several companies are developing to offer a vendor-independent language for multiple platforms.

- NVIDIA decided that the unifying theme of all these forms of parallelism is the CUDA Thread. the compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. Hence, NVIDIA classifies the CUDA programming model as Single Instruction, Multiple Thread (SIMT).
- These threads are blocked together and executed in groups of 32 threads, called a Thread Block. We call the hardware that executes a whole block of threads a multithreaded SIMD Processor.

- We need just a few details before we can give an example of a CUDA program:
  - To distinguish between functions for the GPU (device) and functions for the system processor (host), CUDA uses _device_or_global_ for the former and -host-for the latter.
  - CUDA variables declared as in the device or global functions are allocated to the GPU Memory (see below), which is accessible by all multi threaded SIMD processors.

■ The extended function call syntax for the function name that runs on the GPU is name-dimGrid, dimBlock>>>(... parameter list ...)

where *dimGrid* and *dimBlock* specify the dimensions of the code (in blocks) and the dimensions of a block (in threads).

■ In addition to the identifier for blocks (*blockIdx*) and the identifier for threads per block (*threadIdx*), CUDA provides a keyword for the number of threads per block (*blockDim*), which comes from the *dimBlock* parameter in the bullet above.

- Before seeing the CUDA code, let's start with conventional C code for the DAXPY loop from Section 4.2:

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
        for(int i=0; i< n;++i)
        y[i]= a*x[i]+ y[i]
}
```

- Below is the CUDA version. We launch n threads, one per vector element, with 256 CUDA Threads per thread block in a multithreaded SIMD Processor. The GPU function starts by calculating the corresponding element index i based on the block ID, the number of threads per block, and the thread ID. As long as this index is within the array (i< n), it performs the multiply and add.

```
// Invoke DAXPY with 256 threads per Thread Block _-host-
  int nblocks =(n+255)/256;
 daxpy<<<nblocks,256>>>(n,2.0, x, y);
 // DAXPY in CUDA
 device_
void daxpy(int n, double a, double*x, double*y)
{
  int i= blockIdx.x*blockDim.x+ threadIdx.x;
  if(i< n) y[i]= a*x[i]+ y[i];
}
```

- Comparing the C and CUDA codes, we see a common pattern to parallelizing data-parallel CUDA code. The C version has a loop where each iteration is independent of the others, allowing the loop to be transformed straightforwardly into a parallel code where each loop iteration becomes an independent thread.
- The programmer determines the parallelism in CUDA explicitly by specifying the grid dimensions and the number of threads per SIMD Processor. By assigning a single thread to each element, there is no need to synchronize among threads when writing results to memory.\
- The GPU hardware handles parallel execution and thread management; it is not done by applications or by the operating system. To simplify scheduling by the hardware, CUDA requires that thread blocks be able to execute independently and in any order.