GPU Origins and Evolution

- GPUs and CPUs do not share a common architectural ancestor. While CPUs evolved from general-purpose processors, GPUs descended from specialized graphics accelerators.
- The primary purpose of GPUs remains graphics excellence, but they have evolved to support a wider range of computational applications.

GPU Computing Revolution

- GPU computing
  - The explosion of interest in GPU computing occurred when this hardware potential was combined with programming languages that made GPUs easier to program.
- Programming model
  - CUDA (Compute Unified Device Architecture) is NVIDIA's C-like language and programming environment designed to improve GPU programmer productivity by addressing both heterogeneous computing challenges and multifaceted parallelism

CUDA Programming Challenges

Heterogeneous Computing

Coordinating computation between the system processor (host) and the GPU (device)

Data Transfer Management

Efficiently moving data between system memory and GPU memory

Multiple Forms of Parallelism

Handling multithreading, MIMD, SIMD, and instruction-level parallelism simultaneously

CUDA produces C/C++ for the system processor (host) and a C/C++ dialect for the GPU (device). A similar vendor-independent language is OpenCL, which supports multiple platforms.

CUDA Thread: The Unifying Concept

- NVIDIA chose the CUDA Thread as the unifying theme for all forms of parallelism. Using this lowest level of parallelism as the programming primitive, the compiler and hardware can organize thousands of CUDA Threads to utilize various styles of parallelism within a GPU.
  - Multithreading
  - MIMD (Multiple Instruction, Multiple Data)
  - SIMD (Single Instruction, Multiple Data)
  - Instruction-level parallelism
- NVIDIA classifies this programming model as SIMT (Single Instruction, Multiple Thread)

CUDA Programming Basics

Key CUDA Elements

- Functions marked with `__device__` or `__global__` run on the GPU
- Functions marked with `__host__` run on the system processor
- Variables in `__device__` or `__global__` functions are allocated to GPU Memory
- Extended function call syntax:

```
name<<<dimGrid, dimBlock>>>(... parameter list ...)
```

Thread Organization

- `blockIdx`: identifier for blocks
- `threadIdx`: identifier for threads per block
- `dimGrid`: number of blocks being launched
- `dimBlock`: number of threads per block
- Threads are organized in blocks of 32 threads called Thread Blocks

DAXPY Example: C vs. CUDA

DAXPY in C

```
// 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];
}
```

DAXPY in CUDA

```
// 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];
}
```