COMPUTER GRAPHICS

California State University, Fresno

What is the GPU Good at?

- The GPU is good at data-parallel processing
  - The same computation executed on many data elements in parallel – low control flow overhead with **high SP floating point arithmetic intensity**
  - Many calculations per memory access
  - Currently also need high floating point to integer ratio
- High floating-point arithmetic intensity and many data elements mean that memory access latency can be hidden with calculations instead of big data caches

General Purpose GPU

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

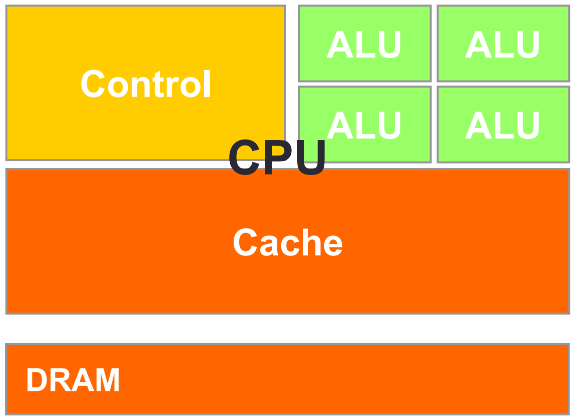
GPU Computing Development

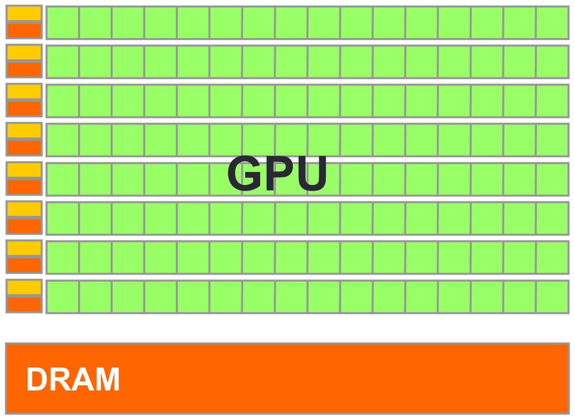
- Laptops, desktops, workstations, servers, clusters – (cell phones? iPods?)

- NVIDIA Volta VG100 GPU 2017
  - Projected peak performance of 250 PFLOPS

GPU Architecture

- The GPU is specialized for compute-intensive, massively data parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than **data caching** and **flow control**



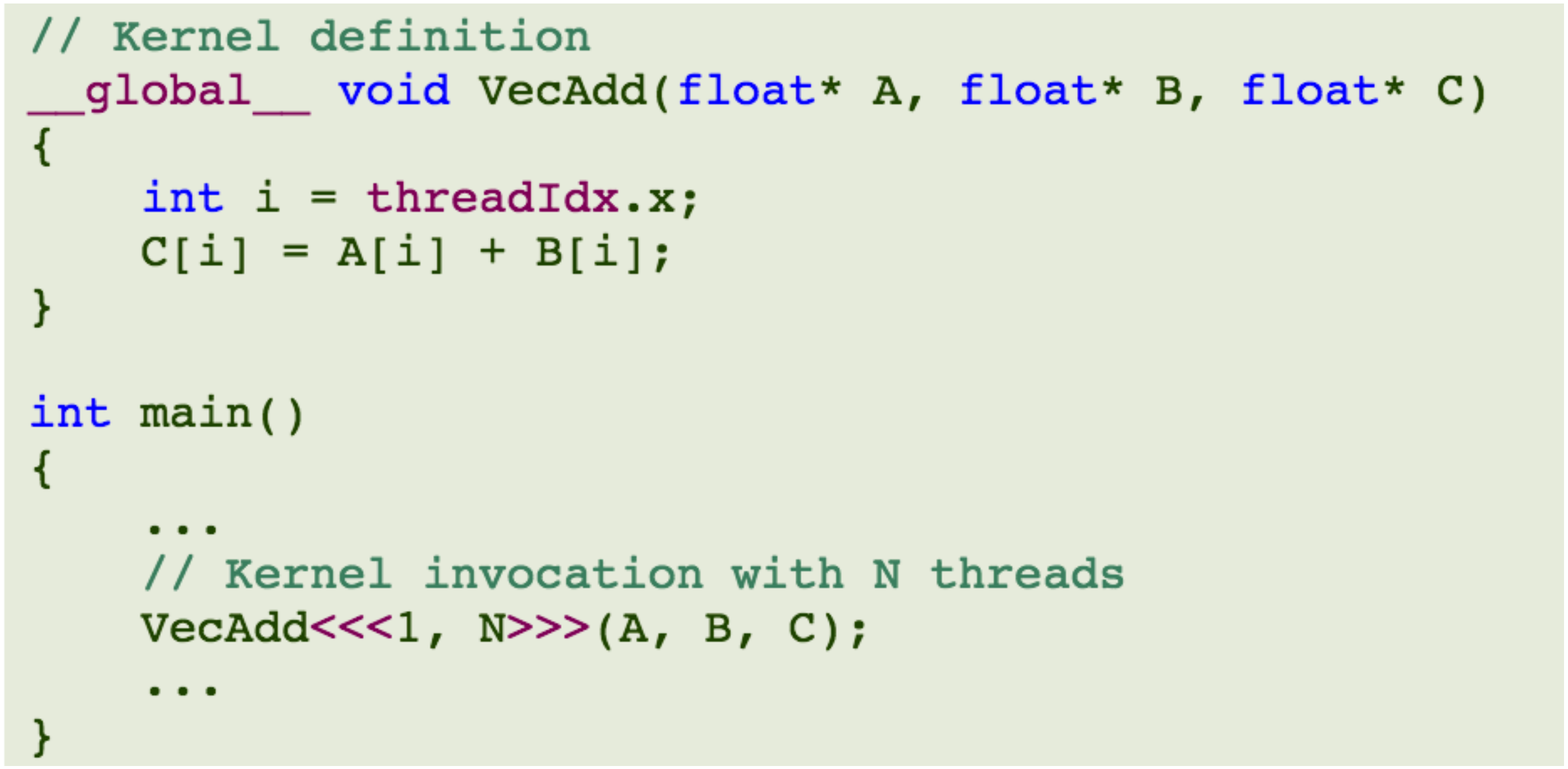


CUDA

- “Compute Unified Device Architecture”
- CUDA C extends C by allowing the programmer to define C functions, called *kernels*
- A kernel is defined using the __global__
- The number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> *execution configuration*

Computational Example

- GPU on simple vector addition



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

- GPU on simple matrix addition

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- threadIdx is a 3-component vector identified using a
  - one-dimensional,
  - two-dimensional, or
  - three-dimensional *thread index*
- invoke computation across the elements in a domain such as a vector, matrix, or volume.

Threads and Blocks

- The independent threads are organized into blocks
- The blocks are completely independent
- Each block is given a small area of shared memory that exists on the multiprocessor.
- Each thread can share data with threads in the same block
- threads in different blocks may be assigned to different multiprocessors concurrently
- Any thread in the block is delayed at this synchronization point until all the other threads in the block complete its task.

thread ID and block ID

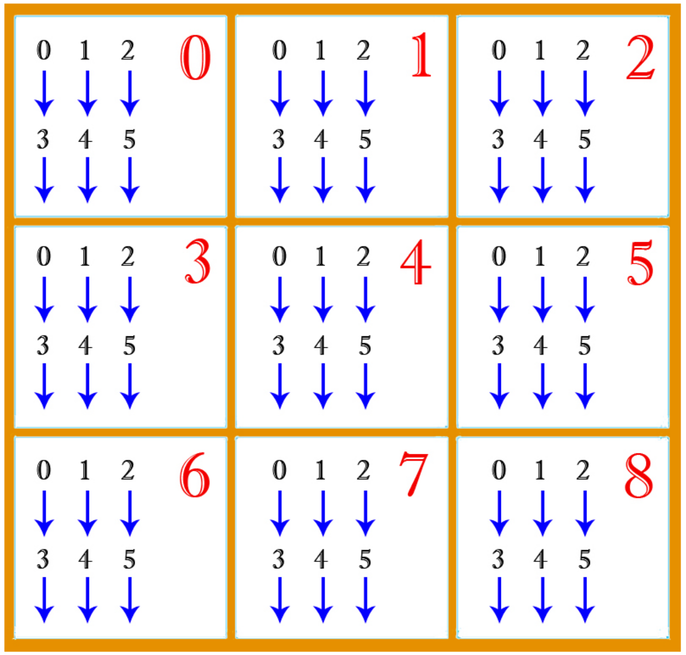
- Each thread is identified by its *thread ID* and each block identified by its block ID.
- thread IDs are designed as thread index within the block
- two-dimensional block of size (Dx;Dy)
  - thread ID of a thread of index (x; y) is *(x + yDx)*
- three-dimensional block of size (Dx;Dy;Dz)
  - the thread ID of a thread of index (x; y; z) is (x + yDx + zDxDy)

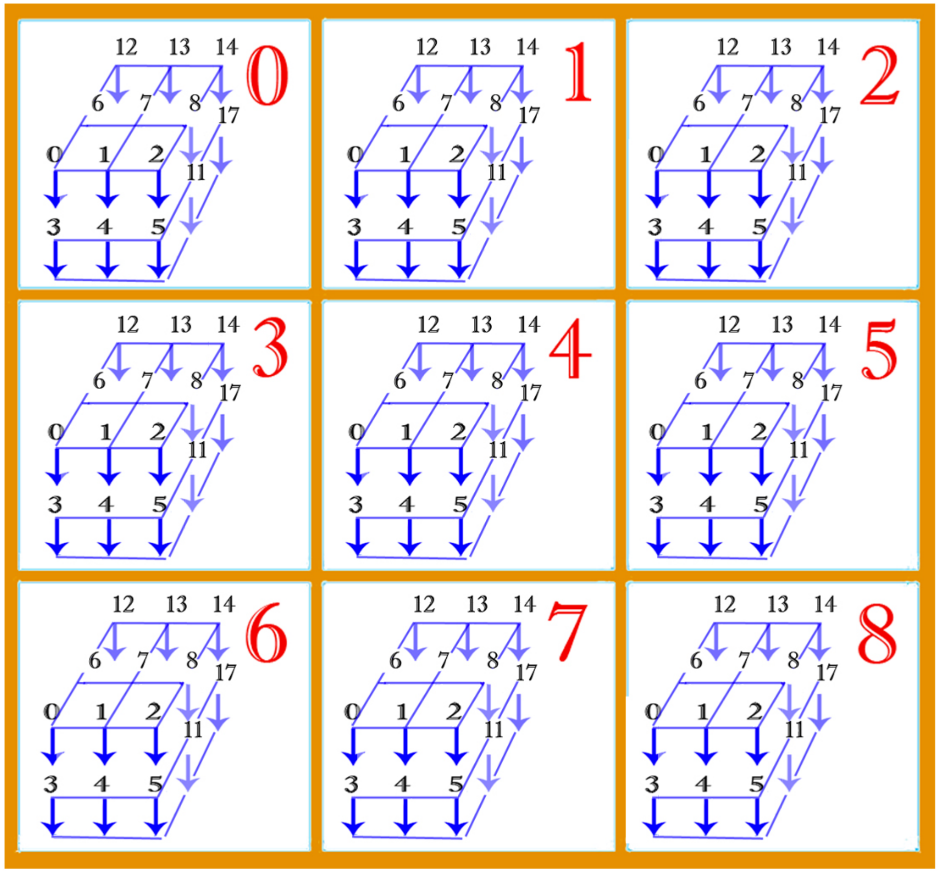
Indexing

↓ Threads

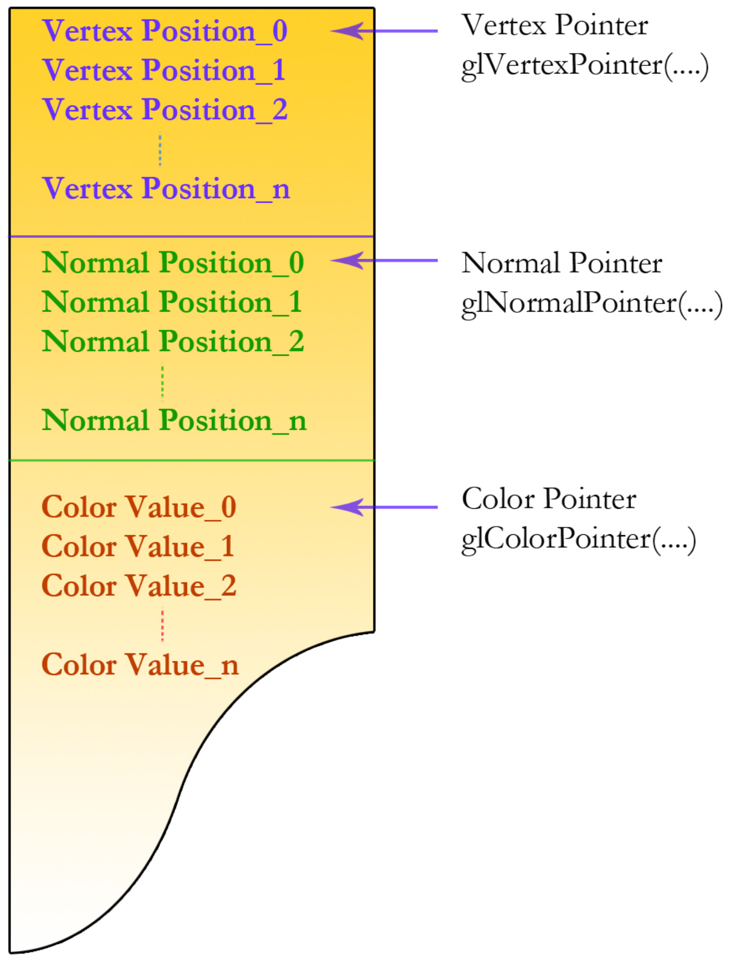
# Thread ID

# Block ID





OpenGL Serialized VBO data layout



Data Mapping with CUDA

- Data mapping begins CUDA assigned each element with an index number (idx).
- *idx = block ID * Number of threads per block + thread ID*
- This assignment presents one to one alignment between idx value and the serialized index while mapping the data into each thread

Accessing VBO Data

- For a given vertex DATA[idx],
- corresponding color value can be found in location at *DATA[idx+number of vertices]*
- the Normal coordinates will be located at *DATA[idx+number of vertices+Number of color elements]*.