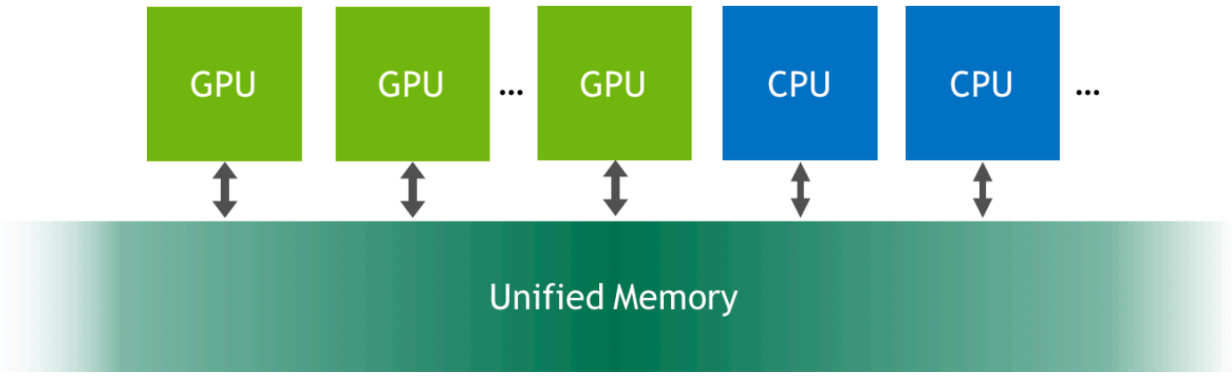
CUDA Unified Memory

Introduction

Unified Memory is a component of the CUDA programming model, first introduced in CUDA 6.0, that defines a managed memory space in which all processors see a single coherent memory image with a common address space.

Unified Memory is a single memory address space accessible from any processor in a system (see figure below). This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs.



Allocating Unified Memory is as simple as replacing calls to `malloc()` or `new` with calls to `cudaMallocManaged()`, an allocation function that returns a pointer accessible from any processor (`ptr` in the following).

```
cudaError_t cudaMallocManaged(void** ptr, size_t size);
```

Programming example

The following code example is a simple program written without Unified Memory. It combines two numbers together on the GPU with a per-thread ID and returns the values in an array. Without managed memory, both host- and device-side storage for the return values is required (`host_ret` and `ret` in the example), as is an explicit copy between the two using `cudaMemcpy()`.

```
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);
    for(int i = 0; i < 1000; i++)
```

```
        printf("%d: A+B = %d\n", i, host_ret[i]);
    free(host_ret);
    cudaFree(ret);
    return 0;
}
```

The following code is the same program with Unified Memory, which allows direct access of GPU data from the host. Notice the `cudaMallocManaged()` routine, which returns a pointer valid from both host and device code. This allows ret to be used without a separate `host_ret` copy, greatly simplifying and reducing the size of the program.

```
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));
    AplusB<<< 1, 1000 >>>(ret, 10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    cudaFree(ret);
    return 0;
}
```

Finally, language integration allows direct reference of a GPU-declared `__managed__` variable and simplifies a program further when global variables are used.

```
__device__ __managed__ int ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();
    for(int i = 0; i < 1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
    return 0;
}
```

Note the absence of explicit `cudaMemcpy()` commands and the fact that the return array ret is visible on both CPU and GPU.

It is worth a comment on the synchronization between host and device. Notice how in the non-managed example, the synchronous `cudaMemcpy()` routine is used both to synchronize the kernel (that is, to wait for it to finish running), and to transfer the data to the host. The Unified Memory examples do not call `cudaMemcpy()` and so require an explicit

`cudaDeviceSynchronize()` before the host program can safely use the output from the GPU.

References

Unified Memory for CUDA Beginners, https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Unified Memory Programming, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd