Multi-GPU

Learning CUDA to Solve Scientific Problems.

Miguel Cárdenas Montes

Centro de Investigaciones Energéticas Medioambientales y Tecnológicas,
Madrid, Spain
miguel.cardenas@ciemat.es

2010

Table of Contents

Objectives

- To use multiple GPUs within the same application in order to improve the performance.

Technical Issues

- Zero-copy.
- Multigpu.

Zero-copy

Zero-copy I

- In the previous lecture, the concept of pinned-memory was explained. This memory is activated using the instruction *cudaHostAlloc()* and passing the flag *cudaHostAllocDefault*.
- The flag *cudaHostAllocMapped* allows allocating pinned memory but in this case the host memory can be accessed directly from within the CUDA kernels.
- This memory does not require copies to and from the GPU, and therfore it is termed *zero-copy* memory.

EUFORIA

Zero-copy I

- First at all, it is necessary to check if the device supports mapping host memory.

```
int main( void ) {
  cudaDeviceProp prop;
  int whichDevice;
  HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
  HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
  if (prop.canMapHostMemory !=1) {
    printf("Device cannot map memory \n");
    return 0;
  }
```

EUFORIA

Zero-copy II. How to proceed?

- Next the runtime must be placed into a state enabling to allocate zero-copy buffers.
- For this, the instruction*cudaSetDeviceFlags()* with the flag *cudaDeviceMapHost()* have to be indicated.

```
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
```

EUFORIA

Zero-copy III. How to proceed?

- Then the original allocations are not longer necessary (removed).
- New instruction *cudaHostAlloc* with the flags *cudaHostAllocWriteCombined* and *cudaHostAllocMapped* is activated.
- The copy instruction are not necessary. On the other hand, valid pointers on the GPU to the data allocated on the CPU are necessary.

```
//a_h = (float *)malloc(size);        // Allocate array on host
cudaHostAlloc( (void**)&a_h, size, cudaHostAllocWriteCombined|cudaHostAllocMapped);
//cudaMalloc((void **) &a_d, size);   // Allocate array on device
//b_h = (float *)malloc(size);        // Allocate array on host
cudaHostAlloc( (void**)&b_h, size, cudaHostAllocWriteCombined|cudaHostAllocMapped);
//cudaMalloc((void **) &b_d, size);   // Allocate array on device
//c_h = (float *)malloc(size);        // Allocate array on host
cudaHostAlloc( (void**)&c_h, size, cudaHostAllocWriteCombined|cudaHostAllocMapped);
//cudaMalloc((void **) &c_d, size);   // Allocate array on device


//cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
//cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);

cudaHostGetDevicePointer( &a_d, a_h, 0);
cudaHostGetDevicePointer( &b_d, b_h, 0);
cudaHostGetDevicePointer( &c_d, c_h, 0);
```

EUFORIA

Zero-copy IV

- It is necessary to force the synchronization of the previous instructions commanded —kernel are asynchronous!.
- Kernel invocation is not modified.

```
square_array <<< n_blocks, block_size >>> (a_d, b_d, c_d, N);
cudaThreadSynchronize(); // compulsory synchronization
```


Zero-copy performance

Zero-copy Performance

- In integrated GPUs, the use of zero-copy technique produces a performance win because the memory is physically shared with the host.
- In the case of discrete GPUs, the performance is improved if the inputs and outputs are used exactly once.
- In situations where the memory gets read multiple times, zero-copy should be avoided. A large penalty could be avoided by simply copying the data to the GPU first.
- *cudaGetDeviceProperties()* has a field named *integrated* which indicates the character of the card.

Using Multiple GPUs

Using Multiple GPUs I

- More and more systems contain multiple GPUs, meaning that they also have multiple CUDA capable processors.
- NVIDIA also sells products, such as the GeForce GTX 295, that contain more than one GPU.
- A GeForce GTX 295, while physically occupying a single expansion slot, will appear to your CUDA applications as two separate GPUs.

Using Multiple GPUs II

- For multiple GPU's, the CPU first launches multiple p-threads.
- The CUDA runtime requires that each GPU must be associated to a separate p-thread running on the CPU.
- The CPU then forms a task pool, each task being processing of one part of the computational tasks when one of the GPU's becomes available, its corresponding p-thread pick up a task, transfer data to GPU and begin the process.

Using Multiple GPUs III

- Back to dot product example.
- To avoid learning a new example, lets convert our dot product to use multiple GPUs.
- To make our lives easier, we will summarize all the data necessary to compute a dot product in a single structure.

```
struct DataStruct {
    int     deviceID;
    int     size;
    float   *a;
    float   *b;
    float   returnValue;
} ;
```


Using Multiple GPUs IV

- To use N GPUs, it is necessary to know exactly how many are installed or accessible.
- The instruction *cudaGetDeviceCount()* can be used to determine how many cuda-capable processors are installed.

```
int main( void ) {
  int deviceCount;
  HANDLE_ERROR( cudaGetDeviceCount( &deviceCount ) );
  if (deviceCount < 2) {
    printf("Less than 2 GPU. Only fund %d\n", deviceCount);
    return 0;
  }
```

Using Multiple GPUs V

- To keep things as simple as possible, we will allocate standard host memory for our inputs and fill them with data exactly how we have done in the past.

```
float   *a = (float*) malloc( sizeof(float) * N ) ;
HANDLE_NULL( a ) ;
float   *b = (float*) malloc( sizeof(float) * N ) ;
HANDLE_NULL( b ) ;
// fill in the host memory with data
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
```


Using Multiple GPUs VI

- The trick to using multiple GPUs with the CUDA runtime API is realizing that each GPU needs to be controlled by a different CPU thread.
- Since we have used only a single GPU before, we have not needed to worry about this.
- We fill a structure with data necessary to perform the computations. Although the system could have any number of GPUs greater than one, we will use only two of them for clarity:

```
DataStruct  data[2] ;

data[0].deviceID = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;

data[1].deviceID = 1;
data[1].size = N/2;
data[1].a = a + N/2;
data[1].b = b + N/2;
```


Using Multiple GPUs VII

- To proceed, we pass one of the DataStruct variables to a utility function termed start_thread(). We also pass start_thread() a pointer to a function to be called by the newly created thread; this example is thread function is called routine( ).
- The function start_thread() will create a new thread that then calls the specifed function, passing the DataStruct to this function. The other call to routine( ) gets made from the default application thread (so we have created only one additional thread).
- Before following, we have the main application thread wait for the other thread to finish by calling end_thread( ).
- Since both threads have completed at this point in main(), it is safe to clean up and display the result (next slice!).

```
CUTThread   thread = start_thread( routine, &( data[0] ) ) ;
routine( &(data[1] ) ) ;

end_thread( thread ) ;
```


Using Multiple GPUs VIII

- Since both threads have completed at this point in main(), it is safe to clean up and display the result.
- Notice that we sum the results computed by each thread. This is the last step in our dot product reduction.

```
    free( a ) ;
    free( b ) ;
    printf( "Value calculated:  %f\n" ,
            data[0].returnValue + data[1].returnValue ) ;
    return 0;
}
```

Using Multiple GPUs IX

- We declare routine() as taking and returning a void* so that you can reuse the start_thread() code with arbitrary implementations of a thread function.
- This is a standard procedure for callback functions in C.

```
void* routine( void *pvoidData ) {
    DataStruct  *data = ( DataStruct*) pvoidData;
    HANDLE_ERROR( cudaSetDevice( data->deviceID ) ) ;
```


Using Multiple GPUs X

- Each thread calls cudaSetDevice( ), and each passes a different ID to this function.
- As a result, we know each thread will be manipulating a different GPU.
- In our case the GPUs have identical performance, GTX 295
- These details are important to load balance your application across the systems GPUs

```
int     size = data->size;
float   *a, *b, c, *partial_c;
float   *dev_a, *dev_b, *dev_partial_c;
// allocate memory on the CPU side
a = data->a;
b = data->b;
partial_c = (float*) malloc( blocksPerGrid*sizeof(float) ) ;
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( ( void**) &dev_a, size*sizeof(float) ) ) ;
HANDLE_ERROR( cudaMalloc( ( void**) &dev_b, size*sizeof(float) ) ) ;
HANDLE_ERROR( cudaMalloc( ( void**) &dev_partial_c, blocksPerGrid*sizeof( float) ) ) ;
// copy the arrays 'a' and ' b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float) , cudaMemcpyHostToDevice ) ) ;
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float) , cudaMemcpyHostToDevice ) ) ;
```


Using Multiple GPUs XI

- Then we launch our dot product kernel, copy the results back, and finish the computation on the GPU

```
    dot<<<blocksPerGrid, threadsPerBlock>>>( size, dev_a, dev_b, dev_partial_c ) ;
    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,  blocksPerGrid*sizeof(float) ,
                              cudaMemcpyDeviceToHost ) ) ;
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i] ;
    }
```


Using Multiple GPUs XII

- As usual, we clean up our GPU buffers and return the dot product we have computed in the returnValue field of our DataStruct.

```
    HANDLE_ERROR( cudaFree( dev_a ) ) ;
    HANDLE_ERROR( cudaFree( dev_b ) ) ;
    HANDLE_ERROR( cudaFree( dev_partial_c ) ) ;
    // free memory on the CPU side
    free( partial_c ) ;
    data->returnValue = c;
    return 0;
}
```

Using Multiple GPUs XIII

- For the sake of completeness, the dot product kernel is presented.

```
__global__ void dot( int size, float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < size) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```


Thanks

Thanks

Questions?

More questions?