GPU Computing Workshop
CSU 2013

Getting Started

Garland Durham
Quantos Analytics

nvidia-smi

At command line, run command "nvidia-smi" to get/set GPU properties.

```
nvidia-smi
    Options:
       -q      query
       -L      list attached devices

See "man nvidia-smi" for full documentation.
```

—

Get device information

See http://docs.nvidia.com/cuda/cuda-runtime-api/index.html for details on these and other CUDA API functions.

```
cudaError_t cudaGetDeviceCount ( int* count )

    Returns the number of compute-capable devices.


cudaError_t cudaSetDevice ( int  device )

    Set device to be used for GPU executions.


cudaError_t cudaGetDevice ( int* device )

    Returns which device is currently being used.


cudaError_t cudaGetDeviceProperties ( cudaDeviceProp* prop, int  device )

    Returns information about the compute-device.
```

—

Error checking

- Most functions in the CUDA API return an object of type `cudaError_t`.
- Also, if an error occurs in any CUDA API call, a CUDA error flag is set. The most recent error (if any) can be checked.
- It is good practice to check these (can prevent much frustration...)

```
cudaError_t err = cudaGetLastError();

    Get last error.


cudaError_t err = cudaDeviceSynchronize()

    Some operations run asynchronously (control returns to program
    before operation completes).  This command blocks until all CUDA
    operations complete and then checks for errors.
```

—

Error checking — continued

It is convenient to define a couple of macros.

(See code/util/mycuda.cu for more sample macros and utilities.)

```
#define CUDA_CHECK_ERROR()    check_error( __FILE__, __LINE__ )
void check_error( const char* file, const int line ) {
/***
 * Example:
 *     cudaMalloc( &ptr, n*sizeof(float) )
 *     CUDA_CHECK_ERROR()
 */
    cudaError_t err = cudaGetLastError();
    if (err) {
        fprintf( stdout, "%s(%i): check_error  ::  %s\nAborting...\n",
                 file, line, cudaGetErrorString( err ) );
        exit(-1);
    }
}
```

```
#define CUDA_SAFE_CALL(err)    safe_call(err, __FILE__, __LINE__)
void safe_call( cudaError_t err, const char* file, const int line ) {
/***
 * Example:
 *     CUDA_SAFE_CALL( cudaMalloc( &ptr, n*sizeof(float) ));
 */
     if (err) {
        fprintf( stdout, "%s(%i): mycuda::safe_call  ::  %s\nAborting...\n",
                 file, line, cudaGetErrorString( err ) );
        exit(-1);
    }
}
```

—

Example — devQuery

See `code/devQuery`.

- Try compiling and running.
- Note error handling.
- Look at makefile.

—

Allocating memory on GPU and copying to/from host

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )

    Allocates memory on device.

    Notes:
        * Memory allocated with this function must be freed with cudaFree()
        * Device memory is only accessible from the device.



cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
                         cudaMemcpyKind kind )

   Copies <count> bytes from the memory area pointed to by src to the memory area
   pointed to by dst, where kind is one of:
       * cudaMemcpyHostToHost
       * cudaMemcpyHostToDevice
       * cudaMemcpyDeviceToHost
       * cudaMemcpyDeviceToDevice.
```

—

Allocating and copying — sample code

Note that it is common to distinguish between device and host copies of variables as, e.g., `x_h` and `x_d`.

```
const int n=1000;

float * x_d;
cudaMalloc( &x_d, n*sizeof(float) );

float * x_h;
x_h = (float*)malloc( n*sizeof(float) );
for (int i=0; i<n; i++) x_h[i]=i;

cudaMemcpy( x_d, x_h, n*sizeof(float), cudaMemcpyHostToDevice );

 < do stuff >

cudaMemcpy( x_h, x_d, n*sizeof(float), cudaMemcpyDeviceToHost );

 < look at results >

cudaFree( x_d );
free( x_h );
```

—

Allocating and copying — utilities

A couple of utilities to simplify this can be found at `code/util/mycuda.cu`. (Note that these throw exceptions in case of failure.)

```
// Example:
//     float *x_h = host_malloc<float>(1024);
//     float *x_d = device_malloc<float>( 1024 );
//     copy_host_to_device( x_d, x_h, n );
//     < do stuff...>
//     copy_device_to_host( x_h, x_d, n );
//     device_free( x_d );
//     host_free( x_h );
```

```
    template <typename T>
T * device_malloc( int n ) throw (exception) {
      T * ptr;
      if ( cudaMalloc( &ptr, n*sizeof(T) ))
              throw exception( "mycuda::device_malloc" );
      return ptr;
}
```

```
    template <typename T>
void device_free( T* ptr  ) throw( exception ) {
    if ( cudaFree( ptr ) )
        throw exception( "mycuda::device_free" );
}
```

```
    template <typename T>
void copy_device_to_host( T * dest, T * src, int n ) throw( exception ) {
    cudaError_t err = cudaMemcpy( dest, src, n*sizeof(T), cudaMemcpyDeviceToHost );
    if (err) throw( exception( "mycuda::copy_from_device"));
}
```

```
    template <typename T>
void copy_host_to_device( T * dest, T * src, int n ) throw( exception ) {
    cudaError_t err = cudaMemcpy( dest, src, n*sizeof(T), cudaMemcpyHostToDevice );
    if (err) throw( exception( "mycuda::copy_to_device" ));
}
```

Basic arithmetic

- Functions executed on the device are referred to as "kernels".
- Kernels are identified with the `__global__` keyword. The return type must be void. Arguments must be pointers to device variables or passed by value.
- Threads are arranged in blocks. Each thread has access to several built-in constants identifying its position in the grid.

  In the simplest case:

```
gridDim.x     : number of blocks in grid
blockIdx.x    : index of the block to which this thread belongs
blockDim.x    : number of threads in each block
threadIdx.x   : index of this thread within the block
```

—

Example — vecAdd

```
__global__ void
vecAdd( float * z, float a, float * x, float b, float * y, int n ) {
/***
 * Example: vecAdd
 *
 *    Computes z = a*x + b*y
 *    a, b are scalar
 *    x, y, z are device pointers.
 *
 *    Notes:
 *        * If there are fewer threads than data elements, some (or all)
 *          threads handle more than one data element.
 */
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = a*x[i] + b*y[i];
    }
}
```

—

Calling a kernel

The grid layout must be specified when the kernel is called.

```
// Example: Calling vecAdd

const int gridsize  = 1000;
const int blocksize = 1024;

float *x_d = device_malloc<float>( 1024 );
float *y_d = device_malloc<float>( 1024 );
float *z_d = device_malloc<float>( 1024 );

 < copy data to x_d and y_d >

vecAdd <<< gridsize, blocksize >>> ( z_d, 4.0f, x_d, 5.0f, y_d );

 < copy data from z_d back to host >
```

—

Example 02 — vecAdd

See `code/vecAdd`.

- Try compiling and running.
- Try setting `n` to some very large value and see if you can throw an exception.

—

Remarks

- Threads are executed in "warps" of 32 threads at a time. So, typically, blocksize should be a multiple of 32.

- The grid size and block size can both be multidimensional. That is suppose that it convenient to specify a 5x3x20 grid comprised of block, each of which has 4x8x16 threads.

  Nvidia has a special data structure, dim3, with fields x, y and z, for this purpose. Thus, one can specify, e.g.,

  ```
  dim3 gridsize(5,3,20);
  dim3 blocksize(4,8,16);
  ```

  In this case, the x, y and z dimensions and indices are accessed in the kernel as:

  ```
  gridDim.x      gridDim.y     griddim.z
  blockDim.x     blockDim.y    blockdim.z
  blockIdx.x     blockIdx.y    blockIdx.z
  threadIdx.x    threadIdx.y   threadIdx.z
  ```

  The total number of blocks, number of thread in each block and total number of threads are thus given by

  ```
  int nblocks              = gridDim.x*griddim.y*gridDim.z;
  int nthreads_per_block   = blockdim.x*blockdim.y*blockdim.z;
  int nthreads             = nblocks*nthreads_per_block;
  ```

—

A few useful utilities

See code/util/mycuda.cu.

```
    template <typename T1, typename T2>
__global__
void fill( T1 * z, T2 a, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = a;
    }
}
```

```
    template <typename T1, typename T2>
__global__
void seq( T1 * z, T2 first, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = first +i;
    }
}
```

```
    template <typename T1, typename T2>
__global__
void seq( T1 * z, T2 first, T2 inc, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = first +i*inc;
    }
}
```

—

Utilities — continued

```
    template <typename T1, typename T2>
__global__
void rep( T1 * z, int n, T2 * x, int nx, int ncopies ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = x[i/ncopies];
    }
}
```

```
    template <typename T1, typename T2>
__global__
void tile( T1 * z, int n, T2 * x, int nx, int ncopies ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = x[i % nx];
    }
}
```

—

Utilities — continued

It may also be useful to have a few simple math operations in a "utilities" package, e.g.,

```
__global__
void aX_plus_bY( float * z, float a, float * x, float b, float * y, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = a*x[i] + b*y[i];
    }
}
```

```
__global__
void max_X_Y( float * z, float * x, float * y, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        if (isnan(x[i])) {
            z[i] = x[i];
        } else {
            z[i] = (x[i]>y[i] ? x[i] : y[i] );
        }
    }
}
```

```
__global__
void exp_X( float * z, float * x, int n ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int nthreads = blockDim.x*gridDim.x;
    for (int i=tid; i<n; i+=nthreads) {
        z[i] = exp(x[i]);
    }
}
```

—

Example 03 — simpleMath

See `code/simpleMath` for some sample code using these utilities and implementing simple kernels.

- Try modifying, compiling, and running.

—

Using mapped memory

- Mapped memory can be accessed from either host or device.
- In the past, host and device pointers were different, and the device pointer needed to be obtained with a call to cudaHostGetDevicePointer().
- However, newer versions of CUDA use "unified addressing", which means that the same pointer is valid for both host and device.

```
NOTES

   * Call cudaSetDeviceFlags( cudaDeviceMapHost ) before calling
     any other functions from CUDA API (not needed if device supports
     unified virtual address space, as most recent devices do).

   * Allocate using cudaHostAlloc()

   * Remember to synchronize (using cudaDeviceSynchronize())
     before reading data written on device.

   * Free using cudaFreeHost()

EXAMPLE

    cudaSetDeviceFlags( cudaDeviceMapHost )
    float *ptr;
    cudaHostAlloc( &ptr, n*sizeof(float), cudaHostAllocMapped );
    < Do stuff...>
    cudaDeviceSynchronize();
    < Read on host >
    cudaFreeHost( ptr )
```

—

Mapped memory utilities

See code/util/mycuda.cu.

```
    template <typename T>
T * mapped_malloc( int n ) throw (exception) {
      T * ptr;
      if ( cudaHostAlloc( &ptr, n*sizeof(T), cudaHostAllocMapped ))
              throw exception( "mycuda::mapped_malloc" );
      return ptr;
}


    template <typename T>
void mapped_free( T* ptr  ) throw( exception ) {
    if ( cudaFreeHost( ptr ) )
        throw exception( "mycuda::mapped_free" );
}


void device_synchronize() throw( exception ) {
    cudaError_t err = cudaDeviceSynchronize();
    if (err) throw( exception( "mycuda::device_synchronize" ) );
}
```

—

Example 04 — mapped memory

See `code/mappedMemory`.

- Try commenting out the `cudaDeviceSynchronize()` statements.
- See if you can make it fail by omitting cudaDeviceSynchronize().

—

Timing CUDA code using cudaEvent

To time CUDA code, need to use a timer based on `CudaEvent` (since CUDA operatons run asynchronously).

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0);

 < do something on GPU >

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Elapsed time: %5.3f\n", elapsedTime );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

—

Profiler

There is also a profiler distributed with CUDA. This is often a more convenient way to assess performance.

—

Exercises

—