GPU Computing with CUDA

Hands-on: Matrix/Vector Operations

Dan Melanz & Andrew Seidl
Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Portland, Oregon
August 4, 2013

Cluster Information

- SSH: euler.wacc.wisc.edu
  - Username/password: should have been emailed
- OS: Scientific Linux 6.2
- Batch system: Torque 3.0
  - Submission scripts are regular shell scripts, with a few Torque/PBS-specific comment lines
  - Full documentation at: http://go.wisc.edu/1eq923

Example Job

exampleJob.sh

```
#!/bin/bash
#PBS -l nodes=1:gpus=1
# This requests one node and one GPU
cd $PBS_O_WORKDIR # go to job submission directory
./exampleJob
```

To submit:

```
qsub exampleJob.sh
```

Stdout and stderr will be placed in:

```
jobname.{0,e}jobid
```

CUDA Programming

- Simple CUDA programs have a basic flow:
  1)The host initializes an array with data
  2)The array is copied from the host to the memory on the CUDA device
  3)The CUDA device operates on the data in the device array
  4)The content of the device array is copied back to the host

One quick example...

Example 1: Vector Scaling

- Vector scaling can be easily performed serially:
  - Given a vector, a, with size N and a scalar, alpha:

```
for( int i=0; i<N; i++)
        a[i] = alpha*a[i];
```

Example 1: Vector Scaling

- Can be performed in parallel:
  - Stage 1: The host initializes an array with data:

```
float *a_h, *a_d;  // Pointer to host & device arrays
float alpha = 2;
const int N = 10;  // Number of elements in arrays
size_t size = N * sizeof(float);
a_h = (float *)malloc(size);        // Allocate array on host
cudaMalloc((void **) &a_d, size);   // Allocate array on device

// Initialize host array and copy it to CUDA device
for (int i=0; i<N; i++) a_h[i] = (float)i;
```

Example 1: Vector Scaling

- Can be performed in parallel:
  - Stage 2: The array is copied from the host to the memory on the CUDA device:

```
//copy it to CUDA device
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
```

Example 1: Vector Scaling

- Can be performed in parallel:
  - Stage 3: The CUDA device operates on the data in the array:

```
// Do calculation on device:
 int block_size = 4;
 int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
 axpy_GPU <<< n_blocks, block_size >>> (a_d, alpha, N);
```

Example 1: Vector Scaling

- Can be performed in parallel:
  - Stage 4: The array is copied back to the host, do some housecleaning…

```
// Retrieve result from device and store it in host array
 cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

 // Print results
 for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);

 // Cleanup
 free(a_h);
 cudaFree(a_d);
```

End Programming Job #1

Example 2: Vector Addition

- Can be performed serially:
  - Given vectors A and B, each of size N, store the result in C:

```
for( int i=0; i<N; ++i) {
      C[i] = A[i] + B[i];
}
```

Your turn now…

- Remember the basic flow of CUDA programs:
  1)The host initializes an array with data.
  2)The array is copied from the host to the memory on the CUDA device.
  3)The CUDA device operates on the data in the array.
  4)The array is copied back to the host.

End Programming Job #2

Example 3: Vector Dot Product

- Dot product:
  - Given vectors a and b, each with size N, store the result in the scalar c:

$$c = \mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \ldots + a_N b_N$$

There are several ways to do this

- Here is one alternative:
  1) Use 3 vectors: *A*, *B*, and *C*
  2) Populate the values of *A* and *B* randomly
  3) For each value in *C* less than *N*, store *A*[i]***B*[i] in *C*[i]
  4) Synchronize the threads!
  5) Pick a single thread to add up all of the values in *C* and store in a single value to pass back to the host (**NOTE**: This step only works if you are using a single block, i.e., less than 1526 threads!)

End Programming Job #3