

# nVidia CUDA

A guide to nVidia CUDA programming and GPU usage

## CUDA Programming on Anselm

The default programming model for GPU accelerators on Anselm is Nvidia CUDA. To set up the environment for CUDA use

```
$ module load cuda
```

If the user code is hybrid and uses both CUDA and MPI, the MPI environment has to be set up as well. One way to do this is to use the PrgEnv-gnu module, which sets up correct combination of GNU compiler and MPI library.

```
$ module load PrgEnv-gnu
```

CUDA code can be compiled directly on login1 or login2 nodes. User does not have to use compute nodes with GPU accelerator for compilation. To compile a CUDA source code, use nvcc compiler.

```
$ nvcc --version
```

CUDA Toolkit comes with large number of examples, that can be helpful to start with. To compile and test these examples user should copy them to its home directory

```
$ cd ~
```

```
$ mkdir cuda-samples
```

```
$ cp -R /apps/nvidia/cuda/6.5.14/samples/* ~/cuda-samples/
```

To compile an examples, change directory to the particular example (here the example used is deviceQuery) and run “make” to start the compilation

```
$ cd ~/cuda-samples/1_Uutilities/deviceQuery
```

```
$ make
```

To run the code user can use PBS interactive session to get access to a node from qnvidia queue (note: use your project name with parameter -A in the qsub command) and execute the binary file

```
$ qsub -I -q qnvidia -A OPEN-0-0
```

```
$ module load cuda
```

```
$ ~/cuda-samples/1_Uutilities/deviceQuery/deviceQuery
```

Expected output of the deviceQuery example executed on a node with Tesla K20m is

```
CUDA Device Query (Runtime API) version (CUDART static linking)
```

```
Detected 1 CUDA Capable device(s)
```

```

Device 0: "Tesla K20m"
CUDA Driver Version / Runtime Version 5.0 / 5.0
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 4800 MBytes (5032706048 bytes)
(13) Multiprocessors x (192) CUDA Cores/MP: 2496 CUDA Cores
GPU Clock rate: 706 MHz (0.71 GHz)
Memory Clock rate: 2600 Mhz
Memory Bus Width: 320-bit
L2 Cache Size: 1310720 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 2147483647 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Enabled
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 2 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.0, CUDA Runtime Version = 5.0, NumDevs

```

## Code example

In this section we provide a basic CUDA based vector addition code example. You can directly copy and paste the code to test it.

```
$ vim test.cu
```

```

#define N (2048*2048)
#define THREADS_PER_BLOCK 512

#include <stdio.h>

```

```

#include <stdlib.h>

// GPU kernel function to add two vectors
__global__ void add_gpu( int *a, int *b, int *c, int n){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}

// CPU function to add two vectors
void add_cpu( int *a, int *b, int *c, int n) {
    for (int i=0; i < n; i++)
        c[i] = a[i] + b[i];
}

// CPU function to generate a vector of random integers
void random_ints( int *a, int n) {
    for (int i = 0; i < n; i++)
        a[i] = rand() % 10000; // random number between 0 and 9999
}

// CPU function to compare two vectors
int compare_ints( int *a, int *b, int n ){
    int pass = 0;
    for (int i = 0; i < N; i++){
        if (a[i] != b[i]) {
            printf("Value mismatch at location %d, values %d and %dn",i, a[i], b[i]);
            pass = 1;
        }
    }
    if (pass == 0) printf ("Test passedn"); else printf ("Test Failedn");
    return pass;
}

int main( void ) {

    int *a, *b, *c; // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for N integers

    // Allocate GPU/device copies of dev_a, dev_b, dev_c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    // Allocate CPU/host copies of a, b, c

```

```

a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );

// Fill input vectors with random integer numbers
random_ints( a, N );
random_ints( b, N );

// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add_gpu() kernel with blocks and threads
add_gpu<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c, N );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

//Check the results with CPU implementation
int *c_h; c_h = (int*)malloc( size );
add_cpu( a, b, c_h, N );
compare_ints(c, c_h, N);

// Clean CPU memory allocations
free( a ); free( b ); free( c ); free (c_h);

// Clean GPU memory allocations
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}

```

This code can be compiled using following command

```
$ nvcc test.cu -o test_cuda
```

To run the code use interactive PBS session to get access to one of the GPU accelerated nodes

```

$ qsub -I -q qnvidia -A OPEN-0-0
$ module load cuda
$ ./test_cuda

```

## CUDA Libraries

### CuBLAS

The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the complete standard BLAS library with 152 standard BLAS routines. Basic description of the library together with basic performance comparison with MKL can be found [here](#).

CuBLAS example: SAXPY\*\*

SAXPY function multiplies the vector  $x$  by the scalar  $\alpha$  and adds it to the vector  $y$  overwriting the latest vector with the result. The description of the cuBLAS function can be found in NVIDIA CUDA documentation. Code can be pasted in the file and compiled without any modification.

```
/* Includes, system */
#include <stdio.h>
#include <stdlib.h>

/* Includes, cuda */
#include <cuda_runtime.h>
#include <cublas_v2.h>

/* Vector size */
#define N (32)

/* Host implementation of a simple version of saxpy */
void saxpy(int n, float alpha, const float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = alpha*x[i] + y[i];
}

/* Main */
int main(int argc, char **argv)
{
    float *h_X, *h_Y, *h_Y_ref;
    float *d_X = 0;
    float *d_Y = 0;

    const float alpha = 1.0f;
    int i;

    cublasHandle_t handle;

    /* Initialize CUBLAS */
```

```

printf("simpleCUBLAS test running..n");
cublasCreate(&handle);

/* Allocate host memory for the matrices */
h_X = (float *)malloc(N * sizeof(h_X[0]));
h_Y = (float *)malloc(N * sizeof(h_Y[0]));
h_Y_ref = (float *)malloc(N * sizeof(h_Y_ref[0]));

/* Fill the matrices with test data */
for (i = 0; i < N; i++)
{
    h_X[i] = rand() / (float)RAND_MAX;
    h_Y[i] = rand() / (float)RAND_MAX;
    h_Y_ref[i] = h_Y[i];
}

/* Allocate device memory for the matrices */
cudaMalloc((void **)&d_X, N * sizeof(d_X[0]));
cudaMalloc((void **)&d_Y, N * sizeof(d_Y[0]));

/* Initialize the device matrices with the host matrices */
cublasSetVector(N, sizeof(h_X[0]), h_X, 1, d_X, 1);
cublasSetVector(N, sizeof(h_Y[0]), h_Y, 1, d_Y, 1);

/* Performs operation using plain C code */
saxpy(N, alpha, h_X, h_Y_ref);

/* Performs operation using cublas */
cublasSaxpy(handle, N, &alpha, d_X, 1, d_Y, 1);

/* Read the result back */
cublasGetVector(N, sizeof(h_Y[0]), d_Y, 1, h_Y, 1);

/* Check result against reference */
for (i = 0; i < N; ++i)
    printf("CPU res = %f t GPU res = %f t diff = %f n", h_Y_ref[i], h_Y[i], h_Y_ref[i] - h_Y[i]);

/* Memory clean up */
free(h_X); free(h_Y); free(h_Y_ref);
cudaFree(d_X); cudaFree(d_Y);

/* Shutdown */
cublasDestroy(handle);
}

```

Please note: cuBLAS has its own function for data transfers between CPU and

GPU memory: - cublasSetVector - transfers data from CPU to GPU memory  
- cublasGetVector - transfers data from GPU to CPU memory

To compile the code using NVCC compiler a “-lcublas” compiler flag has to be specified:

```
$ module load cuda
$ nvcc -lcublas test_cublas.cu -o test_cublas_nvcc
```

To compile the same code with GCC:

```
$ module load cuda
$ gcc -std=c99 test_cublas.c -o test_cublas_icc -lcublas -lcudart
```

To compile the same code with Intel compiler:

```
$ module load cuda intel
$ icc -std=c99 test_cublas.c -o test_cublas_icc -lcublas -lcudart
```