Compiling and Executing CUDA Programs in Emulation Mode

High Performance Scientific Computing II

ICSI-541

Spring 2010

Topic Overview

- Overview of compiling and executing CUDA programs in emulation mode
- Downloading and installing the CUDA software
- Verifying the installation of the CUDA Toolkit and SDK
- Compiling and executing a CUDA program in emulation mode

Compiling and Executing CUDA Programs in Emulation Mode

- CUDA-compliant code can be compiled and executed on a system that does not have a CUDA-capable GPU
- When running a CUDA application in emulation mode, threads are run on the host CPU and not on any GPU
  - For each thread in a thread block, the runtime environment creates a thread on for execution on the host CPU
- For execution in emulation mode, the programmer must ensure that:
  - The host is able to run up to the maximum number of threads per thread block – plus one for the master thread
  - Enough memory is available to run all threads, as each thread gets 256 KB of stack space

Compiling and Executing CUDA Programs in Emulation Mode

- When executed in emulation mode, the performance will be far less than that on a CUDA-capable GPU
  - Emulation mode should NOT be used for performance tuning!
  - Emulation-mode only emulates the GPU device, it does not simulate it
- ASIDE: Emulation vs Simulation
  - From a computing standpoint, the definition depends on who you ask
  - The best clarification I found was:

“If you want to convince people that watching television gives you stomach-aches, you can simulate this by holding your chest/ abdomen and moan. You can emulate it by eating a kilo of unripe apples.”

--Peter Hans van den Muijzenberg

Compiling and Executing CUDA Programs in Emulation Mode

- Certain errors are difficult to detect
  - Race conditions, since the number of threads executing simultaneously is much smaller than on a GPU
  - When dereferencing a pointer to global memory on the host or a pointer to host memory on the device, device execution almost certainly fails in some undefined way, whereas device emulation can produce correct results
  - Most of the time the same floating-point computation will not produce exactly the same result when performed on the device as when performed on the host in device emulation mode
  - The warp size is equal to 1 in device emulation mode. Therefore, the warp vote functions produce different results than in dev
- A CUDA program file compiled in emulation mode can be augmented with code which cannot run on a GPU, e.g., I/O operations to files or the screen

Required Software from NVIDIA

- Software required for building CUDA-compliant programs
  - **CUDA Toolkit -** contains the tools and files needed to compile and build a CUDA application
  - **CUDA SDK -** sample projects that provide source code and other resources for constructing CUDA programs
  - **The NVIDIA driver is not required - unless you have a CUDA-capable GPU**

Evaluation Environment

- This presentation only covers compiling and executing CUDA-compliant code on a Linux platform
  - The effort necessary on a MS-Windows platform may be less than on Linux
- Platform tested on and NVIDIA software used
  - Tests for this presentation were run on an HP Compaq nc8430 laptop with Intel dual-core T2600 processor
  - Operating system: Ubuntu Linux 9.04
  - Graphics card: ATI Radeon Mobility X1600
  - NVIDA CUDA Development Tools 2.3 (both the toolkit and SDK)
  - GCC version 4.3
  - **NO NVIDIA DRIVERS WERE INSTALLED!**

Downloading and Installing the CUDA Software

- For version 2.3 of the CUDA Toolkit and SDK, go to:

  http://www.nvidia.com/object/cuda_get.html

- Installation of CUDA Toolkit
  - Do installation as root user
  - Execute the installation script downloaded (e.g. `cudatoolkit_2.3_linux_32_ubuntu9.04.run`)
  - Default installation directory is `/usr/local/cuda`
  - After installation, place `/usr/local/cuda/bin` in your `PATH` environment variable, and `/usr/local/cuda/lib` in your `LD_LIBRARY_PATH` environment variable
  - In your `~/.bashrc` file, add:

```
PATH=$PATH:/usr/local/cuda/bin;  export PATH
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib
export LD_LIBRARY_PATH
```

Downloading and Installing the CUDA Software

- Installation of CUDA SDK
  - Do installation as a regular user
  - Execute the installation script downloaded (`cudasdk_2.3_linux.run`)
  - Default installation directory is `~/NVIDIA_GPU_Computing_SDK`
  - Sample CUDA programs are in `~/NVIDIA_GPU_Computing_SDK/C`
- If the symbolic link `/usr/lib/libglut.so` does not exist, but `/usr/lib/libglut.so.3` does, do:

  ```
  ln -s /usr/lib/libglut.so.3  /usr/lib/libglut.so
  ```

  This is necessary to get the CUDA programs to compile.
- The CUDA compiler is nvcc (`/opt/cuda/bin/nvcc`)

Verifying the Installation of the NVIDIA CUDA Software

- As a regular user, cd to `~/NVIDIA_GPU_Computing_SDK/C`
- Compile all sample programs in emulation mode by running:

```
make emu=1
```

- Resulting binary executables are in:

```
~/NVIDIA_GPU_Computing_SDK/C/bin/linux/emurelease
```

- All compiled sample programs use libraries pointed to by `LD_LIBRARY_PATH` environment variable, so make sure `/usr/local/cuda/bin` is in the path
  - `printenv LD_LIBRARY_PATH`

Verifying the Installation of the NVIDIA CUDA Software

- Once the CUDA Toolkit and SDK are installed, verify the sample programs in the SDK can be compiled and executed
  - cd to `~/NVIDIA_GPU_Computing_SDK/C/bin/linux/emurelease`
  - Try running `./deviceQuery`. Sample output (next slide)
- Other sample CUDA programs to try running:

bandwidthTest	convolutionFFT2D	eigenvalues
matrixMul	MonteCarlo	simpleMultiGPU

```
There is no device supporting CUDA.

Device 0: "Device Emulation (CPU)"
  CUDA Driver Version:                              0.0
  CUDA Runtime Version:                             2.30
  CUDA Capability Major revision number:            9999
  CUDA Capability Minor revision number:            9999
  Total amount of global memory:                    4294967295 bytes
  Number of multiprocessors:                        16
  Number of cores:                                  128
  Total amount of constant memory:                  65536 bytes
  Total amount of shared memory per block:          16384 bytes
  Total number of registers available per block:    8192
  Warp size:                                        1
  Maximum number of threads per block:              512
  Maximum sizes of each dimension of a block:       512 x 512 x 64
  Maximum sizes of each dimension of a grid:        65535 x 65535 x 1
  Maximum memory pitch:                             262144 bytes
  Texture alignment:                                256 bytes
  Clock rate:                                       1.35 GHz
  Concurrent copy and execution:                    No
  Run time limit on kernels:                        No
  Integrated:                                       Yes
  Support host page-locked memory mapping:          Yes
  Compute mode:                                     Default (multiple host threads can use this
     device simultaneously)

Test PASSED
```

Compiling your own CUDA program for execution in emulation mode

- Once you have verified the installation of the CUDA Toolkit and SDK, you are ready to build your own CUDA program
- CUDA code files normally have the `.cu` extension in their filename
- Use the NVIDIA CUDA compiler, nvcc, to build the executable
  - The `–device-emulation` flag generates code for the GPU emulation library
  - All code for an application (including libraries used) must be compiled consistently either for device (GPU) emulation or for device execution. Otherwise, a runtime error will occur
- As an example, consider the matrix-matrix multiplication CUDA program, `MatMul.cu` (see slide)
- To compile and link `MatMul.cu` for execution in emulation mode:

```
nvcc --link --device-emulation -o MatMul MatMul.cu
```

- The resulting binary executable, `MatMul`, can now be executed on the host in emulation mode, producing the same results as if it were compiled and executed on an NVIDIA GPU

```
//////////////////////////////////////////////////////////////////////////////////
// PROGRAM:       MatMul.cu
// COMPILE AS:    nvcc --link --device-emulation -o MatMul MatMul.cu
// EXECUTE AS:    ./MatMul
// PURPOSE:       Multiplies two 4x4 matrices (A and B) together, and prints out
//                the resulting 4x4 matrix ©
//
// The bulk of this code is taken from Section 3.2.2 of the document “NVIDIA
// CUDA Programming Guide, Version 2.3.1”
//////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
   int width;
   int height;
   float* elements;
} Matrix;

// Thread block size
// #define BLOCK_SIZE 16
#define BLOCK_SIZE 4
```

```
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
int i;
   // Load A and B to device memory
   Matrix d_A;
   d_A.width = A.width;
   d_A.height = A.height;
   size_t size = A.width * A.height * sizeof(float);
   cudaMalloc((void**)&d_A.elements, size);
   cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

   Matrix d_B;
   d_B.width = B.width;
   d_B.height = B.height;
   size = B.width * B.height * sizeof(float);
   cudaMalloc((void**)&d_B.elements, size);
   cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

   // Allocate C in device memory
   Matrix d_C;
   d_C.width = C.width;
   d_C.height = C.height;
   size = C.width * C.height * sizeof(float);
```

```
    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    for (int e = 0; e < A.width; ++e)
       Cvalue += A.elements[row * A.width + e] * B.elements[e * B.width + col];

    C.elements[row * C.width + col] = Cvalue;
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
    Matrix AA, BB, CC;
    int i;

    AA.width = AA.height = 4;
    BB.width = BB.height = 4;
    CC.width = CC.height = 4;

    AA.elements = (float *)malloc(AA.width * AA.height * sizeof(float));
    BB.elements = (float *)malloc(BB.width * BB.height * sizeof(float));
    CC.elements = (float *)malloc(CC.width * CC.height * sizeof(float));

    for (i = 0; i < AA.width * AA.height; i++)
       AA.elements[i] = (float)i;

    for (i = 0; i < BB.width * BB.height; i++)
       BB.elements[i] = 10.0 * (float)i;
```

```
    for (i = 0; i < AA.width * AA.height; i++)
       printf("AA.elements[%d] = %f\n", i, AA.elements[i]);

    for (i = 0; i < BB.width * BB.height; i++)
       printf("BB.elements[%d] = %f\n", i, BB.elements[i]);

    MatMul(AA,BB,CC);

    for (i = 0; i < CC.width * CC.height; i++)
       printf("CC.elements[%d] = %f\n", i, CC.elements[i]);

    free(AA.elements);   free(BB.elements);      free(CC.elements);
}
```