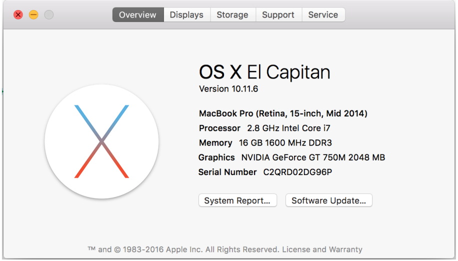
Getting Started with CUDA

September 30, 2016

Outline

The Device

- Before programing in CUDA, and even before installing a driver, need to check if the OS sees an NVIDIA device.
- On linux this can be as easy as **`ls -l /dev/nv*`** or use the **`lspci`** command which provides detailed information about all PCI buses and devices in the system
- On OS X goto “About This Mac” (beware graphics-switching)



GPU Device Drivers

- CUDA installer comes with a driver but do not blindly rely on this driver version to match your specific GPU.
- Go to the NVIDIA drivers download page to lookup current driver version for your device
- On Windows and Linux systems, the NVIDIA display driver installs the **`nvidia-smi`** command line utility. The NVIDIA System Management Interface is intended to aid in the management and monitoring of NVIDIA GPU devices.
- The **`nvidia-smi`** command line utility is the go-to tool for querying installed drivers, device information, device state, device performance (e.g. power, temp, clock speed), etc

NVIDIA CUDA Toolkit

General requirements to use CUDA on your system:

- a CUDA-capable GPU device connected via PCI
- gcc or Clang compiler and toolchain
- the NVIDIA CUDA Toolkit

Links to detailed installation guides for each OS below:

- Windows
- Linux
- OSX

CUDA Programing: The Very Basics

```
#include<stdio.h>

// GPU Kernel definition
__global__ void sayHello(void){
    printf("Hello World from the GPU!\n");
}

int main(void){

    //launch kernel
    sayHello<<<1,5>>>();

    //wait for the kernel to finish
    cudaDeviceSynchronize();

    //that's all
    return 0;
}
```

Listing 1: Hello world in CUDA (runtime API)

CUDA API Level

- For managing the device and organizing threads, CUDA consists of a higher-level *runtime* API and a low-level *driver* API.
- Each function of the runtime API calls are broken down into more basic driver API operations.
- While the driver API does offer more explicit control over how the GPU device is used but does not provide any performance gains over the runtime API and is more difficult to program.
- Most (all?) modern CUDA applications and libraries (e.g. cuBLAS, cuFFT) are built on the runtime API.
- Note driver API calls start with "cu" while runtime API calls start with "cuda" (e.g. `cudaDeviceSynchronize()`)

The CUDA Compiler: **`nvcc`**

- The CUDA **`nvcc`** compiler is based on the widely used LLVM open source compiler infrastructure.
- CUDA programs consist of a mixture of *host* code for the CPU and *device* code for the GPU.
- The **`nvcc`** compiler separates the device code from the host code during the compilation process.
- The host code is standard C and is compiled with C compilers while the CUDA C device code is compiled by **`nvcc`**.
- During the link stage, CUDA runtime libraries (libcudart.so.4) are added for kernel procedure calls and explicit GPU device manipulation.
- Many additional details available via the **`nvcc`** documentation

`nvcc` and PTX

- PTX which stands for “Parallel Thread eXecution” is the *intermediate representation* of the compiled GPU code that can be further compiled into native GPU microcode.
- This is the mechanism that enables CUDA applications to be “future-proof” against instruction set innovations by NVIDIA.
- The PTX code is typically compiled into hardware specific microcode in an on-deman fashion (**JIT**ted) by the CUDA driver. This online compilation process happens automatically when running CUDART applications compiled with the `--fatbin` option (default).
- The PTX code can be manually compiled into microcode using the PTX assembler **`ptxas`**. The resulting CUDA binary microcode is called a "cubin" (pronounced like “Cuban”).
- Cubin files can be disassembled with `cuobjdump` using the option `--dump-sass`. See the docs for more information on CUDA binary utilities.

`nvcc` and PTX continued ...

- Both `.cubin` microcode and `PTX` representations of each kernel are included in the **`nvcc`** "fatbin" executable.
- If the executable is run on hardware which does not support any of the `.cubin` representations, the driver compiles the `PTX` version.
- Since `PTX` compilation can be time consuming, the driver caches theses compiled kernels on disk for repeated invocation.
- Note that `PTX` code can be generated at runtime and compiled explicitly by the driver by calling `cuModuleLoadEx()`.

Compiling CUDA Code

- As a compiler driver, **`nvcc`** does nothing more than set up a build environment and spawn a combination of native tools (e.g. the C compiler installed on the system) and CUDA specific command-line tools (e.g. **`ptxas`**)
- To compile CUDA program simply invoke **`nvcc`** `myprog.cu`
- Use the `--verbose` option to view the build process or `--dryrun` option to generate the build commands without actually executing them.
- There are many options for guiding the code generation. In particular the `--gpu-architecture` option for specifying which `PTX` version to emit and the `--gpu-code` option for specifying which version of Streaming Multiprocessor (SM) microcode (`.cubin`) to produce (`sm_1[0123]`, `sm_2[01]`, `sm_3[05]`).

A Note on CUDA Header Files

- There are three key header files when programing in CUDA:
  - `cuda.h` defining types and host functions for the CUDA *driver* API.
  - `cuda_runtime_api.h` which defines types and host functions and types for the CUDA *runtime* API.
  - `cuda_runtime.h` contains a superset of definitions including everything from `cuda_runtime_api.h`, as well as built-in type definitions, function overlays for the CUDA language extensions, and device intrinsic functions.
- Notice that when compiling with **`nvcc`** the appropriate CUDA headers are included automatically.

Profiling with **nvprof** and **nvvp**

- As of CUDA 5.0, **nvprof** is available to help collect timeline information from the application CPU and GPU activity (e.g. kernel execution, memory transfers, and API calls).
- The Visual Profiler, **nvvp**, displays a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement.
- In addition, **nvvp** will analyze the application to detect potential performance bottlenecks and provide recomendations to eliminate or reduce those bottlenecks.
- Both **nvprof** and **nvvp** are powerful tools to help understand where time is being spent in an application. In many GPU based workloads it is important to understand the compute to communication ratio. That is, number of instructions per byte accessed. Most HPC workloads are bound by memory bandwidth.

Debugging CUDA Applications

- CUDA-GDB is the NVIDIA tool for debugging CUDA kernels running on Linux and Mac operating systems.
- CUDA-GDB is an extension to the x86-64 port of GDB, the GNU Project debugger.
- The CUDA debugger is designed to allow simultaneous debugging of both GPU and CPU code within the same application as well as supports debugging of both the CUDA driver API and/or the CUDA runtime API.
- In order to use debugging tools, compile with **`nvcc`** `-g -G` which embeds debugging information for both host and device code as well as disable optimizations so that program state can be inspected during execution.

Debugging CUDA Applications

- CUDA-MEMCHECK is a functional correctness checking suite included in the CUDA toolkit.
- In short, `memcheck` is a memory access error and leak detection tool.
- The `memcheck` tool can precisely detect and report out of bounds and misaligned memory accesses to global, local, shared and global atomic instructions in CUDA applications
- To use the `memcheck` tool on Linux, compile with
  `nvcc -Xcompiler -rdynamic -lineinfo`
- Similarly, on Windows compile with
  `nvcc -Xcompiler /Zi -lineinfo`
- When compiling with these additional options, **`nvcc`** generates executables that will contain sufficient metadata for `memcheck` to display helpful messages but maintain performance characteristics of the original application.

Querying Device Information

- The runtime API provides a variety of functions for managing GPU devices and querying their associated information.
- To figure out how many devices are visible to the host use `cudaGetDeviceCount()`
- To switch between GPU devices use the `cudaSetDevice()`
- Each GPU device properties can be queried using `cudaGetDeviceProperties()` which returns a struct of type `cudaDeviceProp` containing various information about the device.

Device Properties Example: runtime API

```c
#include<stdio.h>
#include<cuda_runtime.h>

int main(void){
     int deviceCount;
     cudaDeviceProp deviceProp;
     cudaGetDeviceCount(&deviceCount);
     cudaGetDeviceProperties(&deviceProp,0);
     printf("There are %d gpu devices\n",deviceCount);
     printf("Device %s has %f GB of global memory\n",
          deviceProp.name,
          deviceProp.totalGlobalMem/pow(1024.0,3)
     );
}
```

Listing 2: Getting information about devices in CUDA (runtime API)

Driver API for Device Attributes

- Similarly the driver API has a variety of low-level functions for device attributes.
- Before a device can be queried for attributes, the device must be initialized with `cuInit(int devId)`
- Most driver API calls require handle to the particular device which is provided by `cuDeviceGet()`
- There are explicit functions for device count, device name, and total global memory.
- All other device attributes are queried via the `cuDeviceGetAttribute()` function and a set of attribute macros.

Device Properties Example: driver API

```
#include<stdio.h>
#include<cuda.h>

int main(void){
    int deviceCount;
    char deviceName[256];
    CUdevice device;
    size_t szMem; int szProc;
    cuInit(0);
    cuDeviceGetCount(&deviceCount);
    cuDeviceGet(&device,0);
    cuDeviceGetName(deviceName,255,device);
    cuDeviceTotalMem(&szMem,device);
    cuDeviceGetAttribute(&szProc,
        CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT,device);
    printf("There are %d devices detected\n",deviceCount);
    printf("Device %s has %f GB of global memory\n",
        deviceName,szMem/pow(1024.0,3));
    printf("Device multiprocessor count: %d\n",szProc);
}
```

Listing 3: Getting information about devices in CUDA (driver API)

CUDA Code Samples

- The CUDA Toolkit is prepackaged with a variety of CUDA sample codes which cover both runtime and driver APIs.
- These samples cover everything from basic bandwidth tests to using features such as Zero-Copy Memory, Asynchronous Data Transfers, Unified Virtual Addressing, Peer-to-Peer Communication, Concurrent Kernels, sharing data between CUDA and Direct3D/OpenGL graphics APIs, using CUDA with MPI and OpenMP, Image Processing, Video encode/decode, CFD, FDTD, and more.
- Typically CUDA code samples are selected for installation when installing the CUDA Toolkit.
- The code samples can be installed after toolkit installation using the `cuda-install-samples-X.X.sh` script located in the root CUDA directory.