Introduction Programming in CUDA (continued)

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

Some materials/slides are adapted from:
Andreas Moshovos' Course at the University of Toronto
UIUC course by Wen-Mei Hwu and David Kirk

Predefined Vector Data Types

- Max Vector Size: 4
- Can be used both in host and in device code.
- Structures accessed with .x, .y, .z, .w fields

```c
int3 i3; uint3 vect;
ulong4 list; float3 f3;
short2 s2; char4 cs;
```

- default constructors, "make_TYPE (...)"
  ```c
  float4 f4 = make_float4 (1f, 10f, 1.2f, 0.5f);
  ```

- dim3
  - type built on uint3
  - Used to specify dimensions
  - Default value is (1, 1, 1)

Adapted From: A. Moshovos
Execution Configuration

- Must specify when calling a __global__ function:
  
  ```
  <<< Dg, Db [, Ns [, S]] >>>
  ```

- `dim3 Dg`: grid dimensions in blocks
- `dim3 Db`: block dimensions in threads
- `size_t Ns`: per block additional number of shared memory bytes to allocate
  - optional, defaults to 0
- `cudaStream_t S`: request stream (queue)
  - optional, default to 0.
  - Used to have multiple kernels active simultaneously
  - Compute capability >= 1.1

Adapted From: A. Moshovos

Built-in Variables

- `dim3 gridSize`
  - Number of blocks per grid, in 2D (.z always 1)

- `uint3 blockIdx`
  - Block ID, in 2D (blockIdx.z = 1 always)

- `dim3 blockDim`
  - Number of threads per block, in 3D

- `uint3 threadIdx`
  - Thread ID in block, in 3D

Adapted From: A. Moshovos
Execution Configuration Examples

- 1D grid / 1D blocks
  
  \[
  \text{dim3 } gd(1024) \\
  \text{dim3 } bd(64) \\
  \text{akernel}<<<gd, bd>>>(...) \\
  \]
  
  \text{gridDim.x = 1024, gridDim.y = 1, blockDim.x = 64, blockDim.y = 1, blockDim.z = 1}

- 2D grid / 3D blocks
  
  \[
  \text{dim3 } gd(4, 128) \\
  \text{dim3 } bd(64, 16, 4) \\
  \text{akernel}<<<gd, bd>>>(...) \\
  \]
  
  \text{gridDim.x = 4, gridDim.y = 128, blockDim.x = 64, blockDim.y = 16, blockDim.z = 4}

Adapted From: A. Moshovos

Error Handling

- Most cuda...() functions return a \texttt{cudaError_t}
  
  - If \texttt{cudaSuccess}: Request completed without a problem

- \texttt{cudaGetLastError}():
  
  - returns the last error to the CPU
  
  - Use with \texttt{cudaThreadSynchronize}():
    
    \begin{verbatim}
    cudaError_t code; 
    cudaThreadSynchronize (); 
    code = cudaGetLastError (); 
    \end{verbatim}

- \texttt{char \*cudaGetErrorString(cudaError_t code);}:
  
  - returns a human-readable description of the error code

Adapted From: A. Moshovos
Error Handling Utility Function

```c
void cudaDie (const char *msg)
{
    cudaError_t err;
    cudaThreadSynchronize ();
    err = cudaGetLastError();

    if (err == cudaSuccess) return;
    fprintf (stderr, "CUDA error: %s: %s.\n",
        msg,
        cudaGetErrorString (err));
    exit(EXIT_FAILURE);
}
```

Adapted from: http://www.ddj.com/hpc-high-performance-computing/207603131

CUDA_SAFE_CALL

- A preprocessor macro
  
  **CUDA_SAFE_CALL (cuda call )**

- Example:
  
  ```c
  CUDA_SAFE_CALL (cudaMemcpy (a_h, a_d, arr_size,
    cudaMemcpyDeviceToHost) );
  ```

- Must define `#define _DEBUG`
  - No code emitted when undefined: Performance

- Use `make dbg=1` under NVIDIA_CUDA_SDK

Adapted From: A. Moshovos
Measuring Time

- On Host
  - gettimeofday
  - CUDA Timers

- On Device
  - GPU Clock

---

UNIX/Linux `gettimeofday`

```c
#include <sys/time.h>
#include <time.h>

struct timeval start, end;
unsigned long cpu_time;

gmtimeofday (&start, NULL);

THE CODE WE'RE INTERESTED IN

gmtimeofday (&end, NULL);

cpu_time = (end.tv_sec - start.tv_sec) * 1000000 +
           (end.tv_usec - start.tv_usec);
```
CUDA Timer Utility

```c
#include <cuda.h>
#include <cutil.h>

unsigned int htimer;
float start, end;

cutCreateTimer (&htimer);

start = cutGetTimerValue(htimer);
cutStartTimer(htimer);

WHAT WE ARE INTERESTED IN

What if we don’t put this here!

cutStopTimer(htimer);
end = cutGetTimerValue(htimer);

printf ("execution time in millisecond: %f\n", end-start);
```

Using CUDA clock ()

- Can be used in device code
- Returns per multiprocessor counter which is incremented every clock cycle
- Note: threads are time-sliced
- Using clock():
  - Every thread measures start and end
  - Then must find min start and max end
  - Accurate
- Take a look at the clock example in the CUDA SDK

```
uint start, end;

start = clock();
kernel computation
end = clock();

if (end > start)
time = end - start;
else
time = end + (0xffffffff - start)
```
Measurement Methodology

- You will not get exactly the same time measurements every time
  - Other processes running / external events (e.g., network activity)
  - Cannot control
  - “Non-determinism”

- Must take sufficient samples
  - say 10 or more
  - There is theory on what the number of samples must be

- Measure average

Adapted From: A. Moshovos

Compiling a CUDA Program

C/C++ CUDA Application → NVCC → CPU Code → PTX Code → PTX to Target Compiler → Physical → G80 → ... → GPU

Parallel Thread eXecution (PTX)
- Virtual Machine and ISA
- Programming model
- Execution resources and state

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign
Compiling and Linking CUDA Programs

- NVCC
  - Compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
  - Output
    - C code (host CPU Code)
      - Must then be compiled with the rest of the application using another tool
    - PTX
      - Object code directly
      - Or, PTX source, interpreted at runtime

- Linking
  - Any executable with CUDA code requires two dynamic libraries:
    - The CUDA runtime library (cudart)
    - The CUDA core library (cuda)

Debugging CUDA Programs

- An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of __syncthreads
Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.

- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.

NVIDIA Parallel Nsight

- New Visual Studio Based GPU Integrated Development Environment


- Available in Beta (as of Oct 2009)

- Try it!