CUDA Parallel Programming Tutorial

Richard Membarth
richard.membarth@cs.fau.de

Hardware-Software-Co-Design
University of Erlangen-Nuremberg

19.03.2009
Outline

- Tasks for CUDA
- CUDA programming model
- Getting started
- Example codes
Tasks for CUDA

- Provide ability to run code on GPU
- Manage resources
- Partition data to fit on cores
- Schedule blocks to cores
Data Partitioning

- Partition data in smaller blocks that can be processed by one core
- Up to 512 threads in one block
- All blocks define the grid
- All blocks execute same program (kernel)
- Independent blocks
- Only ONE kernel at a time
Memory Hierarchy

Memory types (fastest memory first):

- Registers
- Shared memory
- Device memory (texture, constant, local, global)
Tesla Architecture

- 30 cores, 240 ALUs (1 mul-add)
- \((1 \text{ mul-add } + 1 \text{ mul}): 240 \times (2+1) \times 1.3 \text{ GHz} = 936 \text{ GFLOPS}\)
- 4.0 GB GDDR3, 102 GB/s Mem BW, 4GB/s PCIe BW to CPU
CUDA: Extended C

- Function qualifiers
- Variable qualifiers
- Built-in keywords
- Intrinsics
- Function calls
Function Qualifiers

Functions: __device__, __global__, __host__

__global__ void filter(int *in, int *out) {
    ...
}

- Default: __host__
- No function pointers
- No recursion
- No static variables
- No variable number of arguments
- No return value
Variable Qualifiers

▸ Variables: __device__, __constant__, __shared__

  __constant__ float matrix[10] = {1.0f, ...};
  __shared__ int [32][2];

▸ Default: Variables reside in registers
Built-In Variables

- Available inside of kernel code
- Thread index within current block:
  threadIdx.x, threadIdx.y, threadIdx.z
- Block index within grid:
  blockIdx.x, blockIdx.y
- Dimension of grid, block:
  gridDim.x, gridDim.y
  blockDim.x, blockDim.y, blockDim.z
- Warp size: warpSize
Intrinsics

- **void** `__syncthreads();`
- Synchronizes in all thread of current block
- Use in conditional code may lead to deadlocks
- Intrinsics for most mathematical functions exists, e.g. `__sinf(x), __cosf(x), __expf(x), ...`
- Texture functions
Function Calls

Launch parameters:
  ▶ Grid dimension (up to 2D)
  ▶ Block dimension (up to 3D)
  ▶ Optional: stream ID
  ▶ Optional: shared memory size
  ▶ kernel<<<grid, block, stream, shared_mem>>>()

```c
__global__ void filter(int *in, int *out);
...
dim3 grid(16, 16);
dim3 block(16, 16);
filter <<< grid, block, 0, 0 >>> (in, out);
filter <<< grid, block >>> (in, out);
```
Getting Started

- Compiler path
- Sample Makefile
- Debugging
- Memory management
- Time measurement
Compiler Path

- gcc/g++ compiler for host code
- nvcc compiler for device code
- gcc/g++ for linking
- icc/icpc works as well
Simple Project Makefile

- Use different files for host and device code
- Compile device/host code with nvcc
- Compile additional code with gcc
- Adjust Makefile from SDK:

```makefile
# Add source files here
EXECUTABLE := vector_add

# CUDA source files (compiled with cudacc)
CUFILES := vector_add_host.cu

# CUDA dependency files
CU_DEPS := \vector_add_device.cu \defines.h

# C/C++ source files (compiled with gcc / c++)
CCFILES := \vector_add_cpu.cpp

#set directory for common.mk
CUDA_SDK_PATH ?= /opt/cuda/sdk
ROOTDIR := $(CUDA_SDK_PATH)/projects
ROOTBINDIR := bin
ROOTOBJDIR := obj

include $(CUDA_SDK_PATH)/common/common.mk
```
Building the Program

Makefile offers different options:

▶ Production mode: make
▶ Debug mode: make dbg=1
▶ Emulation mode: make emu=1
▶ Debug+Emulation mode: make dbg=1 emu=1
Debugging

SDK offers wrappers for function calls:

- For CUDA function calls: `cutilSafeCall(function);`
- For kernel launches (calls internally `cudaThreadSynchronize()`):
  `cutilCheckMsg(function);`
- For SDK functions: `cutilCheckError(function);`

Additional tools (recommended):

- CudaVisualProfiler
- `valgrind` – in emulation mode only, there is no MMU on the GPU!
- `gdb` – in emulation mode: `#ifdef __DEVICE_EMULATION__`
- real (!) `gdb` support, for GNU Linux – unfortunately 32bit only :(
Memory Management

- Host manages GPU memory
  - `cudaMalloc(void **pointer, size_t size);`
  - `cudaMemset(void *pointer, int value, size_t count);`
  - `cudaFree(void *pointer);`

- Memcopy for GPU:
  - `cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind direction)`

- `cudaMemcpyKind`:
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
Time Measurement

- Initialization biases execution time
- Don’t measure first kernel launch!
- SDK provides timer:

```c
int timer=0;
cutCreateTimer(&timer);
cutStartTimer(timer);
...
cutStopTimer(timer);
cutGetTimerValue(timer);
cutDeleteTimer(timer);
```

- Use events for asynchronous functions:

```c
cudaEvent_t start_event, stop_event;
cutilSafeCall (cudaEventCreate (&start_event));
cutilSafeCall (cudaEventCreate (&stop_event));
cudaEventRecord (start_event, 0);  // record in stream-0, to ensure that all
   previous CUDA calls have completed
...
cudaEventRecord (stop_event, 0);
cudaEventSynchronize (stop_event); // block until the event is actually
   recorded
cudaEventElapsedTime (&time_memcpy, start_event, stop_event);
```
Example

Vector addition:

- CPU Implementation
- Host code
- Device code
Vector Addition - CPU Implementation

```c
void vector_add(float *iA, float *iB, float* oC, int width) {
    int i;
    for (i=0; i<width; i++) {
        oC[i] = iA[i] + iB[i];
    }
}
```
#include <cutil_inline.h>
// include CUDA and SDK headers - CUDA 2.0
#include <cuda.h>
#include <cutil.h>

#include "vector_add_kernel.cu"

int main(int argc, char** argv) {
    int dev;

    // CUDA 2.1
    dev = cutGetMaxGflopsDeviceId();
cudaSetDevice(dev);

    // CUDA 2.0
    CUT_DEVICE_INIT(argc, argv);
}
// allocate device memory
int *device_idata_A, *device_idata_B, *device_odata_C;
cudaMalloc((void**) &device_idata_A, mem_size);
cudaMalloc((void**) &device_idata_B, mem_size);
cudaMalloc((void**) &device_odata_C, mem_size);

// copy host memory to device
cudaMemcpy(device_idata_A, host_idata_A, mem_size,
    cudaMemcpyHostToDevice);
cudaMemcpy(device_idata_B, host_idata_B, mem_size,
    cudaMemcpyHostToDevice);
...

// copy result from device to host
cudaMemcpy(host_odata_C, device_odata_C, mem_size,
    cudaMemcpyDeviceToHost);

// free memory
cudaFree(device_idata_A);
cudaFree(device_idata_B);
cudaFree(device_odata_C);
Vector Addition - Launch Kernel

// setup execution parameters
\texttt{dim3 grid(1, 1);}  
\texttt{dim3 threads(num\_elements, 1);}  

// execute the kernel
\texttt{vec\_add\lll grid, threads \rrr(device\_idata\_A, device\_idata\_B, device\_odata\_C);}  
\texttt{cuda\_Thread\_Synchronize();}
Vector Addition - Kernel Function

```c
__global__ void vector_add(float *iA, float *iB, float* oC) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;

    oC[idx] = iA[idx] + iB[idx];
}
```
Questions?

Krakow, Pontifical Residency
Courtesy of Robert Grimm