Harness the Power of GPUs: An Introduction to GPGPU Programming
Lecture 1: Introduction

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

June 16, 2014
Organization of this class

• 13 sites, each site should have an instructor/TA/helper

• Please put your microphones on mute, if you are not talking/asking questions

• Course set up in lectures and labs
  – Q&A time at end of each lecture
• Chat for offline discussions

• All you need available at: http://go.iu.edu/cn5
Course outline

- Course runs June 16-20, from 11 until 4 (5 hours per day)
- Two blocks per day (60 minute lecture, 75 minute lab each), 30 minute lunch break in between
- All times are EDT

- Monday:
  - 11:00-12:30: Lecture: Introduction to parallelism and GPU computing, concept of compute offloading, Data transfers, basic CUDA functions
  - 12:30-13:15: Lab: Access BlueWaters, Compile CUDA sample and run it
  - 13:15-13:45: Lunch break
  - 13:45-14:45: Lecture: Kernels
  - 14:45-16:00: Lab: Data Movement, First Kernel (VecAdd)
- Tuesday:
  - 11:00-12:00: Lecture: GPU Architecture
  - 12:00-13:15: Lab: Error Handling, 2D Data Structures
  - 13:15-13:45: Lunch break
  - 13:45-14:45: Lecture: Memory Hierarchies and Management
  - 14:45-16:00: Lab: Data Movement, First Kernel (VecAdd)
- Wednesday:
  - 11:00-12:00: Lecture: Using Shared Memory, Atomic Operations
  - 12:00-13:15: Lab: Advanced Matrix-Matrix-Multiplication, Dot-Product
  - 13:15-13:45: Lunch break
  - 13:45-14:45: Lecture: Streams, Dynamic Parallelism
  - 14:45-16:00: Lab: Quicksort
- Thursday:
  - 11:00-12:00: Lecture: Introduction to OpenACC
  - 12:00-13:15: Lab: First OpenACC examples
  - 13:15-13:45: Lunch break
  - 13:45-14:45: Lecture: Advanced OpenACC
  - 14:45-16:00: Lab: Using data regions, present data
- Friday:
  - 11:00-12:00: Lecture: Parallelization techniques and optimizations
  - 12:00-13:15: Lab: N-Body simulation
  - 13:15-13:45: Lunch break
  - 13:45-14:45: OpenCL as a platform independent alternative
  - 14:45-16:00: Lab: Catchup, Questions
Outline

• Parallelism as an every day concept
• Types of parallelism
• Offloading tasks to a GPU
• Managing data transfers
• Error handling
• The five basics in CUDA
Parallelism is everywhere

Parallelism occurs whenever multiple agents work together in solving a larger problem. You use it when one agent cannot solve the problem in time.
Parallel computing

- Take a large computational problem
- Break it into smaller parts
- Solve the parts concurrently

- If necessary:
  - Communicate partial results
  - Repeat until solution is reached
Types of parallelism

- Bit-wise parallelism
- Instruction level parallelism
- Data parallelism
- Task parallelism
Task parallelism

- Every agent works on a different task (Pipelining)
  - Example: Video processing
  - Agent 1: Load frames
  - Agent 2: Remove blur
  - Agent 3: Adjust colors,…
  - …

- Agent = thread

- Works well on multi-core CPUs
- Only good for coarse-grain work on GPUs
- Requires multiple tasks in the overall problem to be solved
Data parallelism

- Every agent performs the same task on different pieces of data
  - Example: Video processing
  - Agent 1: works on top left corner
  - Agent 2: works on top right corner
  - Agent 3: works on bottom left corner
  - ...

- Agent = thread
- Works well on CPUs and GPUs
- Requires dividable data structures
GPU system setup

- CPU
- Main Memory
- Accelerator
- Local Memory
- Application
- Accelerator Library
- Subprogram

Connections:
- System Bus
- DMA Transfers
- calls
- invokes
Defining terms: HOST and DEVICE

HOST

DEVICE
Allocating and freeing memory on the device

```c
int main( int argc, char *argv[] ) {

    float *devicepointer; // Pointer to float element

    // The following line allocates memory for one float on the GPU and
    // sets devicepointer to the beginning of that memory area
    cudaMalloc( (void**)&devicepointer, sizeof(float) );
    // The following line releases the allocated memory for devicepointer
    // on the GPU so that it may be used again by another allocation
    cudaFree( devicepointer );

    return 0;
}
```
Data transfers use cudaMemcpy

HOST

DEVICE

HostToDevice
# define NELEMENTS 16

int main( int argc, char *argv[] ) {

    float hostvariable[NELEMENTS]; // float-array on the HOST
    float *devicepointer;          // allocated pointer to the DEVICE

    cudaMemcpy( devicepointer,          // Pointer to DEVICE memory
               hostvariable,            // Pointer to host memory
               sizeof(float)*NELEMENTS, // number of bytes to transfer
               cudaMemcpyHostToDevice ); // direction of transfer

    return 0;
}

Data transfers use cudaMemcpy (2)
Copying data from the DEVICE to the HOST

#define NELEMENTS 16
int main( int argc, char *argv[] ) {

    float hostvariable[NELEMENTS]; //float-array on the HOST
    float *devicepointer; //allocated pointer to the DEVICE

    cudaMemcpy( hostvariable, //Pointer to HOST memory
                 devicepointer, //Pointer to DEVICE memory
                 sizeof(float) )*NELEMENTS, //number of bytes to transfer
                 cudaMemcpyDeviceToHost ); //direction of transfer

    return 0;
}
There is more than one copy operation available

- cudaMemcpy
- cudaMemcpy2D
- cudaMemcpy2DArrayToArray
- cudaMemcpy2DAsync
- cudaMemcpy2DFromArray
- cudaMemcpy2DFromArrayAsync
- cudaMemcpy2DToArray
- cudaMemcpy2DToArrayAsync
- cudaMemcpy3D
- cudaMemcpy3DAsync
- cudaMemcpy3DPeer
- cudaMemcpy3DPeerAsync
- cudaMemcpyFromArray
- cudaMemcpyFromArrayAsync
- cudaMemcpyFromSymbol
- cudaMemcpyFromSymbolAsync
- cudaMemcpyToArray
- cudaMemcpyToArrayAsync
- cudaMemcpyToSymbol
- cudaMemcpyToSymbolAsync
Possible transfer directions

- `cudaMemcpyHostToDevice` → Transfers data from host to device
- `cudaMemcpyDeviceToDevice` → Transfers data on the device
- `cudaMemcpyDeviceToHost` → Transfers data from device to host
- `cudaMemcpyHostToHost` → Transfers data on the host
Catching errors with cudaError_t

CUDA functions return a value of type cudaError_t:

```c
typedef enum cudaError cudaError_t

It can have the following values:

- `cudaSuccess`: The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see `cudaEventQuery()` and `cudaStreamQuery()`).
- `cudaErrorMissingConfiguration`: The device function being invoked (usually via `cudaLaunch()` was not previously configured via the `cudaConfigureCall()` function.
- `cudaErrorMemoryAllocation`: The API call failed because it was unable to allocate enough memory to perform the requested operation.
- `cudaErrorInitializationError`: The API call failed because the CUDA driver and runtime could not be initialized.
- `cudaErrorLaunchFailure`: An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until `cudaThreadExit()` is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.
```
Automatic interpretation of the error

In order to interpret cudaError_t one can use the following function

```c
const char* cudaGetErrorString( cudaError_t error)
```

Example:
```c
cudaError_t error = cudaGetLastError();
printf( "CUDA error: %s\n", cudaGetErrorString( error ) );
```
Routine to catch and handle errors

static void HandleError( cudaError_t err,
                        const char *file, int line ) {

    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n",
                cudaGetErrorString( err ),  file, line );
        exit( EXIT_FAILURE );
    }
}

#define HANDLE_ERROR( err )
    (HandleError( err, __FILE__, __LINE__ ))
Handling errors by default

- Prepend all CUDA calls with HANDLE_ERROR

Example:
HANDLE_ERROR( cudaMalloc( (void**)&devicepointer, sizeof(float) ) );
Compiling and running a CUDA program

• On remote systems make sure that the CUDA environment is available (usually requires a module load cudatoolkit or similar)
• Name your CUDA files with the suffix .cu
• Compile your program using nvcc (e.g. nvcc myprogram.cu)
• Execute your program by running ./a.out
• On remote systems the node you compile on might not feature a GPU, you will have to use a batch system to access the “right” node
The five basic CUDA functions

- `cudaMalloc` to allocate memory on the device
- `cudaMemcpy` to transfer data to and from the device
- Kernel invocations (we will cover this later)
- Handling errors
- `cudaFree` to release allocated memory on the device
The five results of CUDA programming

• Compiler error
• Program crashes
• Program produces wrong results
• Program runs very slow
• Program runs fast and correct
Five tricks to make it easier

- Use comments (it’s all logical now, but how about a week from now?)
- Use error handling (it will tell you what went wrong)
- Test for correctness
- Mark your data locations (e.g. put a “_d” and a “_h” as a suffix)
- Use building blocks where possible
I need help!

- Check the CUDA programming guide
- Recommended books:
QUESTIONS?