# Exercises for Architectures of Supercomputers

10th Exercise, 29./30.1.2020

Johannes Hofmann

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

# Exercise Agenda



- Introduction to CUDA (Nvidia's GPU programming interface)
- Exercise: CUDA STREAM benchmark
- Hints: Working with GPUs in the cluster

# Introduction to CUDA



- Compute Unified Device Architecture (CUDA)
- Regular C/C++ programs
  - File name extension `.cu` instead of `.c` or `.cpp`
  - Code is compiled using the **NV**idia **C**/C++ **C**ompiler (`nvcc`)
- Functions are either run on the host (CPU) or the device (GPU)
  - Separation by qualifiers
    - `__host__` / `__global__` functions are run on the host (CPU)
    - `__device__` functions are run on the GPU (also called **kernels**)
- As with regular C/C++ code, execution begins in the `main()` function on the host (CPU)
- GPU kernels are executed on the device (GPU) when they are called from the host code
  - Special syntax to call GPU kernels:
    `KernelName<<<blocksPerGrid,threadsPerBlock>>>(arg0, arg1, ...);`
  - `blocksPerGrid` and `threadsPerBlock` special variables used to map kernel instances to GPU

# Introduction to CUDA



- Parallelization (multicore and SIMD) is taken care of by CUDA
  - Kernel code is scalar
  - Multiple kernel instances are created by the CUDA runtime system
    - How many is determined by the programmer using the `threadsPerBlock` and `blocksPerGrid` variables (see blackboard!)
    - Each kernel instance gets unique ids that enable the instance to identify its position in the grid (`blockIdx` and `threadIdx`)
- Kernel example

```
#include <stdio.h>
__device__ void HelloKernel() {
    printf("This is Thread (%d, %d), (%d, %d)\n",
        blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y);
}
int main(int argc, char *argv[]) {
    dim3 threadsPerBlock(3, 3, 1);
    dim3 blocksPerGrid(2, 2, 1);
    HelloKernel<<<blocksPerGrid, threadsPerBlock >>>();
    return 0;
}
```

# Introduction to CUDA



```
$ nvcc cuda_hello_world.cu -o cuda_hello_world
$ ./cuda_hello_world
This is Thread (0, 0), (0, 0)
This is Thread (0, 0), (1, 0)
This is Thread (0, 0), (2, 0)
This is Thread (0, 0), (0, 1)
This is Thread (0, 0), (1, 1)
This is Thread (0, 0), (2, 1)
This is Thread (0, 0), (0, 2)
This is Thread (0, 0), (1, 2)
This is Thread (0, 0), (2, 2)
This is Thread (1, 0), (0, 1)
This is Thread (1, 0), (1, 1)
This is Thread (1, 0), (2, 1)
[...]
This is Thread (1, 1), (0, 2)
This is Thread (1, 1), (1, 2)
This is Thread (1, 1), (2, 2)
```

- Thread
  - Kernel instance, which is executed on the GPU
    (on a single “CUDA core”)

- Block
  - A set of threads that are executed on a single GPU multiprocessor
  - Number of threads per block chosen by programmer (`threadsPerBlock`)

- Grid
  - Set of all blocks
  - Number of blocks per grid chosen by programmer (`blocksPerGrid`)

# Introduction to CUDA (kernel calls)



- Kernel calls use the following syntax

```
KernelName<<<blocksPerGrid, threadsPerBlock>>>(parameters, ...);
```

- Identifying a threads unique grid position
  - `threadIdx.{x,y,z}` position inside block
  - `blockIdx.{x,y,z}` position of block in grid
  - `blockDim.{x,y,z}` extends of a block
  - Example

```
int grid_pos_x = blockIdx.x * blockDim.x + threadIdx.x;
int grid_pos_y = blockIdx.y * blockDim.y + threadIdx.y;
int grid_pos_z = blockIdx.z * blockDim.z + threadIdx.z;
```

- Kernels are executed asynchronously
  - CPU initiates kernel execution on the GPU, then continues working
  - The CPU can be instructed to wait until execution on the GPU has finished using `cudaThreadSynchronize()`

# Introduction to CUDA (memory)



- Kernels cannot directly access host memory (i.e., the CPU's memory)
  - If you want to access data in host memory, you must first copy the data from host to device memory
  - Then pass a pointer to device memory to the kernel and access the data in the device memory
- Allocate memory on the device
  - `cudaMalloc`, similar to `malloc` on the CPU
    - `cudaError_t cudaMalloc(void ** devPtr, size_t size);`
    - Example:

```
int *array;
cudaMalloc((void **)&array, 100 * sizeof(int));
```

- Free memory (compare `free`)

```
cudaFree(array);
```

- Copy data from host memory to device memory or other way:
  - `cudaMemcpy(dst, src, nbytes, cudaMemcpyHostToDevice);`
  - `cudaMemcpy(dst, src, nbytes, cudaMemcpyDeviceToHost);`

# Introduction to CUDA (debugging continued)



- `printf()` can be used inside kernels
  - When creating thousands of kernel instances, it can be useful to restrict output to a single thread:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
if (x == 0)
    printf("hello from thread %d\n", x);
```

- By default there **NO** notification if a kernel fails, e.g. because you access unallocated memory which causes a segmentation fault
  - `cuda-memcheck` Memory debugging tool

```
$ cuda-memcheck ./my_cuda_binary
```

  - Check the return values of CUDA functions for errors
    - See next slide

# Introduction to CUDA (debugging)



- Check return value of CUDA functions

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file,
                      int line, bool abort=true)
{
    if (code != cudaSuccess) {
        fprintf(stderr,"GPUassert: %s %s %d\n",
                cudaGetErrorString(code), file, line);
    if (abort)
            exit(code);
    }
}
```

- Application in your code

```
gpuErrchk(cudaMemcpy(d_B, h_B, N * sizeof(double),
          cudaMemcpyHostToDevice));
```

# Exercise Agenda



- Introduction to CUDA (Nvidia's GPU programming interface)
- **Exercise: CUDA STREAM benchmark**
- Hints: Working with GPUs in the cluster

# STREAM benchmark: Part 1



- You can find a reference implementation of the STREAM benchmark for CPUs at http://www.cs.virginia.edu/stream/

- Download the code and modify the `Makefile`
  - Make sure to use the Intel C Compiler (`icc`)
  - Use the following compiler options:
    - `-O3`
    - `-opt-streaming-stores always`

- Compile the STREAM benchmark for the CPU

- Measure the memory bandwidth of the emmy node's CPUs
  - Use all physical cores of both process, no SMT
  - Use `likwid-pin` to pin threads appropriately
  - The reported bandwidth for the STREAM triad will serve as reference

# STREAM benchmark: Part 2



- Implement the STREAM triad in CUDA
  - STREAM Triad in C:

```
for (i=0; i<N; ++i)
      A[i] = B[i] * c + C[i]
```

- Use double-precision numbers as data type
- Each array should contain 1 GB of data
- The input arrays `B` and `C` are initialized with a value of 1.0 in host memory
- Afterwards, the data should be copied to device memory
  - Use `cudaMalloc` and `cudaMemcpy`
- Execute the STREAM triad kernel
- Verify your kernel is correct by comparing the result calculated by the kernel to a reference solution calculated by the CPU
  - To do so, you must copy the result from the device memory to host memory

# STREAM benchmark: Part 2



- Measure the kernel's execution time using the `get_time` function
  - Make sure to wait until the kernel finishes execution
- Calculate the sustained memory bandwidth
  - Divide data volume by runtime
- Now include the overhead of copying the data from host to device memory before you execute the kernel. Also include the overhead of copying the result from the device to the host memory. Now calculate the sustained memory bandwidth using the runtime including the overhead
- Compare both sustained bandwidths to the reference bandwidth of the CPU
  - Can you give a recommendation as to when it is sensible to use an accelerator vs. when it is not?

# Exercise Agenda



- Introduction to CUDA (Nvidia's GPU programming interface)
- Exercise: CUDA STREAM benchmark
- **Hints: Working with GPUs in the cluster**

# Working with GPUs in the cluster



- Not all Emmy nodes are equipped with GPUs
  - To request a GPU node use the `anygpu`, `k20` or `v100` properties when requesting a node
    ```
    $ qsub -lnodes=1:ppn=40:anygpu,walltime=1:30:00 -I
    ```

- Before you can use the CUDA compiler (`nvcc`), you have to make the CUDA environment available using the module system
  (Note: this environment is only available on GPU nodes)
  ```
  $ module load cuda
  ```

- To use the recent features of CUDA, you have to specify an appropriate ”device capability” when compiling.
  (Use at least `sm_35` if you want to use `printf`)
  ```
  $ nvcc -arch sm_35 StreamTriad.cu
  ```