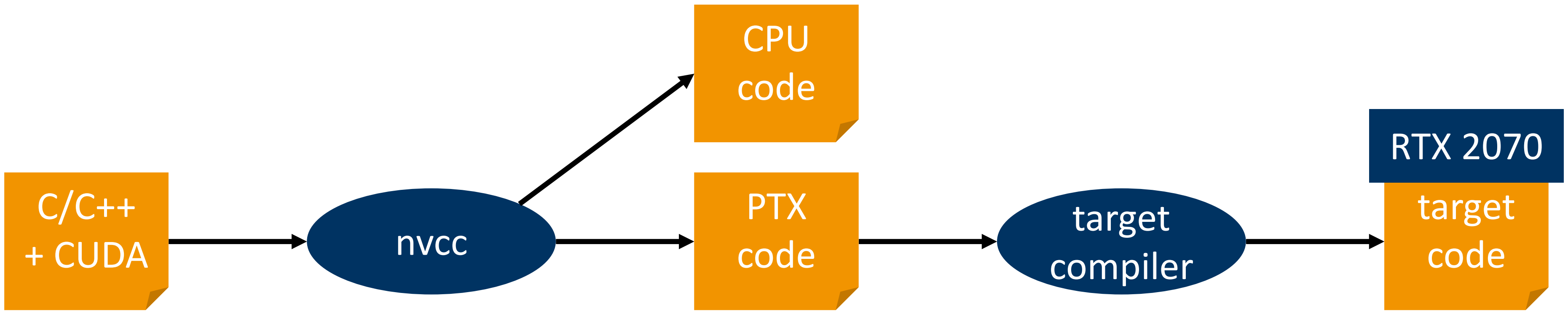
Best Practice: How to Write Correct CUDA Programs

Philipp Gschwandtner

It’s not all About Computational Speed!

- GPUs provide high performance for suitable applications
  - 7 clusters out of top 10 of Top500 use accelerators (8 out of top 10 of Green500)

- But software and hardware stack are very different compared to CPUs



- Getting the wrong result very fast isn’t very useful!

What can go Wrong?

- Functional bugs
  (in ascending order of difficulty)
  - Failure to launch
  - Crash
  - Hang
  - Incorrect result

- Non-functional bugs
  - Slow execution
    (➔ performance debugging)

➔ Imagine everything that can go wrong in a sequential program, and add to that two separately acting hardware devices, one with massive parallelism.

Kernel Execution is Asynchronous

- Launch operation of a kernel does not block host code
  - Proper synchronization requires `cudaDeviceSynchronize()`
- Synchronization is not for free
  - Performance penalty
  - Only synchronize when necessary

```
// ...
kernel<<<gridDim,blockDim>>>(...);
// kernel might not have
// run or finished yet
cudaDeviceSynchronize();
// kernel definitely has
// finished execution
```

Kernel Execution is In-Order

- Multiple Kernels submitted to the same stream execute in order
  - Stream represents a queue
  - Guaranteed without explicit synchronization

```
// ...
kernelA<<<gridDim,blockDim>>>(...);
kernelB<<<gridDim,blockDim>>>(...);
// kernel A/B might not have
// run or finished yet, but B will
// not start before A has finished
cudaDeviceSynchronize();
// both kernels definitely
// have finished execution
```

`cudaDeviceSynchronize()`

- Blocks until GPU has finished all tasks launched so far, e.g.
  - Kernels
  - Asynchronous memcpy operations
  - `printf()` output inside GPU code
- Will return an error if any of the preceding tasks has failed
- Must be issued individually per GPU in multi-GPU setups
- Also available: `cudaStreamSynchronize()` when using multiple streams

Thread Synchronization

- Mainly used in conjunction with shared memory
  - Not discussed in detail, to be covered by Lukas later in the course
- Several levels of synchronization, among which block-level synchronization
  - By calling `__syncthreads()` in GPU code
  - Acts like a barrier for all threads in the same block
  - Must be encountered by all threads of this block
  - Has no effect on threads of other blocks of the same grid

Thread Synchronization: Undefined Behavior

- `__syncthreads()` inside conditional
  - No problem
  - But: conditional must evaluate to the same value (true/false) for all threads of the same block

- Otherwise: undefined behavior

```
__global__ void kernel(float* data) {
  if(data[threadIdx.x] > 10) {
    // all threads of this block
    // must execute this call
    __syncthreads();
  }
}
```

Practical Exercise

- Goal: Evaluate correct use of `__syncthreads()`
- Read the source code of `day_2/ho1/synccheck.cu`
- Compile and run
- Interpret the result!
  - What is the problem?
  - How can we fix it?

Return Codes of CUDA API

- Always check return code of CUDA calls
  - Will tell you if your function call succeeded or failed
  - Ask `cudaGetErrorString()` for a readable message
  - Failing function calls might affect subsequent function calls

- Consider what to do in case of failure
  - At least tell the user the program failed
  - Cleanup resources allocated so far
  - …

Common CUDA Idiom

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

inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true) {
  if(code != cudaSuccess) {
    fprintf(stderr,"assert: %s %s %d\n", cudaGetErrorString(code), file, line);
    if(abort) {
      exit(code);
    }
  }
}
// call like this
gpuErrorCheck(cudaMalloc(...)); // if fails, print message and continue
gpuErrorCheck(cudaMalloc(...), true); // if fails, print message and abort
```

Reasons for Incorrect Results

- Specification errors – computation correct but result does not match science
  - Validation – go fix your math!
- Implementation errors – computation does not match specification
  - Verification – go fix your code!
- Numerical accuracy issues
  - Numerical precision (half vs. single vs. double)
  - (Non-)Associativity of operations
  - IEEE 754 & 80-bit compliance

Precision

- GPUs offer choice of floating-point bit width
  - Trade-off between speed and precision
  - Make sure to compare against same-precision results

- Math library implementations
  - CUDA provides own implementation for math functions such as `sinf()`, `cosf()`, …
  - These differ from e.g. glibc implementations for x86
  - Results for same input might differ!
  - Fast versions available `__sinf()`, `__cosf()`, …

Practical Exercise

- Goal: Test difference in precision between trigonometric implementations
- Read the source code of `day_2/ho1/cos.cu`
- Compile and run with 5992555 as input (see `cos.txt`)
- Examine the output

Associativity

- Floating-point math is not associative
  - almost every operation involves rounding errors of some sort
  - (A+B)+C != A+(B+C)
- Not restricted to CUDA
  - but inherent part of any parallel computation with floating point math

Sequential Equivalence

- strong sequential equivalence
  - bitwise identical results to sequential implementation
  - potentially big impact on performance (e.g. choice of parallelization strategy)
  - requires preserving the order of computations compared to sequential implementation

- weak sequential equivalence
  - mathematically equivalent but not bitwise identical
  - does not require preserving the order of computations

- Always check your requirements!
  - If your algorithm doesn't require a specific order, why should its implementation?

Coding Guidelines

- write clean code that prevents bugs or facilitates their detection, e.g.
  - use meaningful identifiers
  - minimize vertical distance of variable declaration, definition & use
  - follow the <u>D</u>on't <u>R</u>epeat <u>Y</u>ourself (DRY) principle (single component per feature)

- Use the toolchain, Luke!
  - read & heed compiler warnings
  - write and regularly run unit and/or integration tests, especially aimed at (varying degrees of) parallelism
  - use code coverage tests
  - use continuous integration
  - use source version control

Unit Testing

- Structure kernel code in multiple `__device__` functions instead of a single `__global__`
  - Allows them to be tested individually
  - Improves readability
- Declare functions both `__device__` and `__host__`
  - Causes `nvcc` to emit both CPU and GPU code for these functions
  - Enables testing on CPU and GPU
  - Also may reduce code duplication for CPU+GPU execution paths

Conclusion

- Always check return codes of CUDA API calls
  - Make it a habit to use a macro definition as discussed

- Do not put more severe constraints on implementation than on algorithm
  - If the algorithm doesn’t require double precision numerical accuracy, why use it?
  - When porting code from CPU to GPU, consider precision
  - Small differences in the result are not necessarily an implementation error

- Watch out for unspecified behavior
  - e.g. __syncthreads() in an index-dependent conditional statement

- Adhere to coding guidelines
  - Will save you a lot of time and effort down the road

Practical Exercise

- Goal: First porting of a CUDA program from scratch

- Examine `day_2/ho1/heat_stencil_omp.c`, compile and run (Makefile is provided)
  - Naïve 2D heat stencil implementation (mathematically inaccurate)

- Port to CUDA using the knowledge you gained so far

- Output of both programs should be the same

Image Sources

- Yoda: https://www.deviantart.com/biggiepoppa/art/Master-Yoda-Star-Wars-395511111