Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

9 CUDA STREAM Benchmark

Kenan Gündogan[1] **Philipp Gündisch**[1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Chair of Computer Science 3 (Computer Architecture)

January 19, 2025

CUDA

Thread Hierarchy

- Thread
  - Kernel instance running on a single CUDA core
- Block
  - Group of **threads** executed on a streaming multiprocessor (SM)
- Grid
  - Group of **blocks** executed on a GPU
- A CUDA kernel is executed as a grid of blocks of threads
- Further documentation
  - CUDA
  - CUDA Programming Model

CUDA

Kernel Call

- Kernel call syntax
  - `kernelName<<<blocks, threadsPerBlock>>>(Parameter);`
- Kernel definition syntax
  - `__global__ void kernelName(parameter) ...`
- Thread position determination
  - `threadIdx.{x,y,z}` block position
  - `blockIdx.{x,y,z}` grid position
  - `blockDim.{x,y,z}` block dimension
  - example: `int x = blockIdx.x * blockDim.x + threadIdx.x;`
- Asynchronous execution of kernels
  - CPU just initiates the execution on the GPU
  - The program running on the CPU can wait for the GPU kernels to finish by calling the `cudaDeviceSynchronize()` function

CUDA

Memory

- Kernel cannot access the host memory (i.e. CPU memory)
  - Data must be copied from host (CPU) memory to device (GPU) memory and vice versa
  - Parameters are passed copy-by-value when the kernel is called
- Memory allocation on device memory
  - `cudaError_t cudaMalloc(void** devPtr, size_t size);`
- Memory freeing on device memory
  - `cudaError_t cudaFree(void* devPtr);`
- Copy data from host to device memory and vice versa
  - `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind);`

Debugging

- Memory debugging tool
  - `cuda-memcheck ./my-cuda-binary`
- CUDA Debugger
  - `cuda-gdb`

CUDA

Cluster

- To run cuda on GPUs we switch to the Alex cluster:
  - Head node: `alex.nhr.fau.de`
  - Measurement device is NVIDIA A100 GPU
  - sbatch argument: `-gres=gpu:a100:1`
  - See Alex cluster for more details
- CUDA compiler `nvcc` must be loaded explicitly via:
  - `module load cuda`

Task 9.1

Measure CPU Memory Bandwidth

- Measure the memory bandwidth of the CPU via the STREAM Triad benchmark
- Download the source code `stream.c` from the STREAM reference implementation
- Compile `stream.c`
  - Use the Intel compiler
  - Enforce non-temporal stores with flag `-qopt-streaming-stores always`
  - Enable highest optimization level with flag `-O3`
- Measure the memory bandwidth of the host system
  - All physical cores (both sockets)
  - Use `likwid-pin` tool to pin the threads
  - Use the STREAM Triad value as reference value for the comparison with your following GPU implementation

Task 9.2

Measure Internal GPU Memory Bandwidth

- Implement the STREAM Triad benchmark in CUDA as a kernel function for the GPU
  - Triad benchmark in C:
    ```
    for(i = 0; i < n; ++i) {
    A[i] = B[i] * c + C[i];
    }
    ```
  - 2 load operations, 1 store operation, 1 multiplication, 1 addition, c is an arbitrary constant unequal to {0.0, 1.0}
- Total data size should be 3 GiB *(i.e. 1 GiB per array)*
- Allocate and initialize the arrays `B` and `C` in the CPU memory and then copy them into the GPU memory
- Call your kernel function to run the STREAM Triad on the GPU
- Kernel function writes to array `A`
- Finally, copy array `A` from the device memory back to the host memory to verify the correctness of your kernel implementation
- Measure solely the kernel execution time and calculate the bandwidth

Task 9.3

Measure External GPU Memory Bandwidth

- Measure the kernel execution time plus the data transfer times of the copy operations and calculate the bandwidth
- Compare the GPU’s bandwidths to the CPU’s bandwidth
  - What is your recommendation on when to use the GPU instead of CPU?

Appendix: Checklist

Performance Optimization (1/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Compiling
  - Choice of the compiler (`icx`)
  - Compiler flag to optimize aggressively (e.g. `-O3`)
  - Compiler flag to adapt for specific hardware (e.g. `-xHost`)
- Programming Techniques (if applicable)
  - Use `#define` and `const` instead of variables
  - Data type aware programming
  - Use aligned memory (e.g. `_mm_malloc()` or `posix_memalign()`)
  - Consecutive address iteration
  - Variable declarations outside of loops
  - Reduce function calls
  - Use intrinsics (to utilize SIMD)
  - Cache aware programming (Spatial Blocking)
  - Prefetcher aware programming (L1 Cache Blocking)

Appendix: Checklist

Performance Optimization (2/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum
- Optimization Process
  - Check assembler code while optimizing
  - Check performance gains while optimizing
  - Use profiling tools
  - Ensure correctness of code
  - Optimize iteratively
  - Optimize single core performance first
  - **Parallelize your code on the CPU first**