Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

10 CUDA DGEMM

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

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

January 30, 2024

Floating-Point Performance

Scope of exercise

- In this exercise we will compare the floating-point performance of a CPU and a GPU
  - 2x Intel Xeon E5-2630 v4 CPUs (meggie node)
  - 1x Nvidia RTX3080 GPU (tinyGPU node)
- We calculate the theoretical performance in advance
- We measure the actual performance with state of the art library implementations

Theoretical Floating-Point Performance

The theoretical floating-point performance limit for a chip can be calculated with:

$$P_{chip} \text{ [Flop/s]} = n_{cores} \cdot f_{core} \text{ [Hz]} \cdot n_{ILP} \text{ [instr/cy]} \cdot n_{SIMD} \text{ [Flop/instr]} \cdot n_{FMA}$$

- n_{cores}: Multi-core parallelism
- f_{cores}: Clock speed of the cores
- n_{ILP}: Instruction-level parallelism / Superscalar
- n_{SIMD}: Single Instruction Multiple Data Vectorization
- n_{SIMD}: Fused Multiply-Add instruction

Experimental Floating-Point Performance

The actual floating-point performance limit for a chip can be measured with a benchmark. In this exercise we will run the LINPACK benchmark:

- LINPACK measures the time to factorize a system of linear equations filled with `double` values
- Standard benchmark for determining the achievable floating-point performance of a chip
- Used to rank the TOP500 Supercomputers
- For big matrices the DGEMM (Double-Precision General Matrix-Matrix Multiplication) library is used

Task 10.1

Theoretical Flop/s

- Determine the theoretical floating-point performance of a meggie node
- Determine the theoretical floating-point performance of an RTX 3080
- How huge is the difference?

HINT:

- Meggie nodes consist of two CPUs, each having 10 cores with 2.2 GHz base frequency. Each cycle, every core is able to perform one `AVX ADD` and one `AVX MULT`.
- RTX 3080 GPUs consist of 68 Streaming Multiprocessors. Each processor runs with a frequency of 1.44 GHz and consists of two double precision units which also support fused multiply add instructions.

Task 10.2

Flop/s with optimized libraries

- Use 80% of the available memory
  - CPU: 58 GiB $\rightarrow$ 58 GiB * 80% / 3 = 15 GiB per Matrix
  - GPU: 10 GiB $\rightarrow$ 58 GiB * 80% / 3 = 2.6 GiB per Matrix
- Calculate the floating-point performance with $2 \cdot N^3$ divided by the runtime
- For CPU use the DGEMM routine from the Intel Math Kernel Library (MKL)
  - Intel MKL is part of the `icc`
  - Add `-mkl` to the linker flags in your makefile
  - Make sure to use `double` precision
- For GPU use the DGEMM routine from the CUDA Basic Linear Algebra Subprograms (CuBLAS) library
  - A CuBLAS example can be found here
  - Make sure to use `double` precision
  - Compile your code with

    ```
    nvcc -arch sm_86 file.cu -lcublas -lcurand -o binary
    ```
- Benchmark the GPU and CPU to determine the floating-point performance with these optimized libraries
- Compare the performance with the results from task 10.1

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 (`icc`)
  - 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
  - GPU: Consider memory transfer overhead
- 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