Heidelberg University
Institute of Computer Engineering
Computer Engineering Group

Holger Fröning
Bálint Soproni

GPU Computing - Architecture and Programming
Winter Term 2023/2024

Exercise 4

- **Hand in via Moodle until Monday, Nov 13, 2023, 09:00**
- **Include all names on the top sheet. Hand in a <u>single</u> PDF.**
- **Please compress your results into a single archive (`.zip` or `.tar.gz`).**
- **Please employ the following naming convention `hpc<XX>_ex<N>.{zip,tar.gz}`, where `XX` is your group number, and `N` is the number of the current exercise.**
- **A maximum of four students is allowed to collaborate on the exercises.**
- **In case an exercise requires programming:**
  - **include clean and documented code**
  - **include a Makefile for compiling**

General notes

- Please use Pascal Thread Scheduling to avoid unwanted behavior by setting -arch=compute_60 -code=sm_70 compiler flags [1].
- Furthermore please use the cuda/11.4 and devtoolset/8 module.
- Use proper error checking in your code [2].
- Run cuda-memcheck on your code, and resolve all errors.

4.1 Reading

Read the following two papers and provide reviews as explained in the first lecture (see slides):

- Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News 38, 3 (June 2010), 451-460.
- Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro 28, 2 (March 2008), 39-55.

(10 points)

Hint for the following exercises: The compiler might notice that you are loading data into registers without ever using it. To prevent such optimizations, it is helpful to conditionally assign a value of a register to an output variable of the kernel. Then the compiler cannot track back the reference and won't optimize out the code we are trying to analyze here. Such an example code could be:

```
if ( tid == 0 ) *out = t_{reg};
```

A good way to check if your code is optimized away is by using godbolt[3]

[1] `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`
[2] `https://leimao.github.io/blog/Proper-CUDA-Error-Checking/`
[3] `https://godbolt.org/z/hqrThWaMW`

Holger Fröning
Bálint Soproni

4.2 Shared Memory Analysis - Basis

Goal of this exercise is to assess the performance of the shared memory that is part of each Streaming Multiprocessor (SM) of the GPU. Allocate shared memory in a kernel. Perform tests with at least 10 measurement points and report graphically. Report the bandwidth [GB/s] on the y-axis and the size on the x-axis. If another parameter is required (threads per block, etc), use different lines and a legend.
To assess the basic performance of shared memory, ensure that each thread operates on an exclusive location in shared memory; in other terms avoid bank conflicts.
First, validate previous results for global memory by performing these experiments:

- Read data of varying size (1kB to 48kB) from global memory into shared memory. Use only one thread block; determine the optimal number of threads per block. Report graphically and interpret.
- Write back data of varying size (1kB to 48kB) from shared memory into global memory. Use only one thread block; determine the optimal number of threads per block. Report graphically and interpret.
- Now choose an arbitrary (but reasonable) data size. For both directions, vary the number of thread blocks to maximize performance and report it.
  Note: If you use multiple thread blocks, then each one has its own shared memory so the total throughput is the aggregated throughput of each thread block. Also consider that the amount of data moved depends on the block count.

Now measure the shared memory performance:

- Read data of varying size (1kB to 48kB) from shared memory to registers. Determine the optimal number of threads per block. Report graphically and interpret.
- Write back data of varying size (1kB to 48kB) from registers into shared memory. Determine the optimal number of threads per block. Report graphically and interpret.

(14 points)

4.3 Shared Memory Analysis - Conflicts

The shared memory is organized in banks, and certain access patterns may result in huge performance penalties. In order to assess these penalties, perform experiments in which the access pattern (how threads operate on shared memory) is varied.

- Implement a kernel that loads values from shared memory into a thread-local register. Use a parameter for the number of iterations; choose an appropriate number to ensure stable results. Let each thread operate on a 4B value (e.g. float).
- Measure the time spent for these repeated shared memory loads using the clock64() function. Note that too long executions can results in overflows.
- Use a stride when accessing shared memory. Vary the stride in between 1 and 64 (preferable using steps of 1). Make sure that only valid addresses are accessed, but try not to use modulo to avoid problematic branch divergence issues.
- Choose an appropriate number of threads per block (tpb).
- Report graphically the effect of different strides (stride on x axis, clock count (or time) on y axis).
- Interpret and explain the results, in particular comment on how many shared memory banks there are and why.

(12 points)

4.4 Matrix Multiply – CPU sequential version

Implement an unoptimized, sequential CPU version of a matrix multiply operation C = A*B. Matrices can be assumed to be square. Your program should accept the matrix size (elements per dimension) as parameter and report run time for the matrix multiply operation. Initialize according to: A[i,j] = i+j, B[i,j] = i*j. Don't include initialization time in the measurement.
Note: Please make sure that you run it on the compute node and not on the headnode. To do so, run your code with srun or sbatch

- Report C for an input of 5x5 in the table below. In order to verify correctness.
- Report the run time graphically by varying the problem size. Interpret your results.
- Choose a problem size for which the CPU caches are no longer effective. You can look up the size of the L3 cache of the cluster's processor on its manufacturer product page. Report sustained GFLOP/s for this size. Interpret your results.

C[i,j]	C[,0]	C[,1]	C[,2]	C[,3]	C[,4]
C[0,]					
C[1,]					
C[2,]					
C[3,]					
C[4,]					

Note: we will use this code in the next exercise when we start implementing and optimizing a matrix-multiply operation on a GPU. We therefore recommend that you use a clean and modular design.
(8 points)

4.5 Willingness to present

Please declare whether you are willing to present any of the previous exercises.
The declaration can be made on a per-exercise basis. Each declaration corresponds to 50% of the exercise points. You can only declare your willingness to present exercises for which you also hand in a solution. If no willingness to present is declared you may still be required to present an exercise for which your group has handed in a solution. This may happen if as example nobody else has declared their willingness to present.

- Reading (Task: 4.1)
- Shared Memory Analysis - Basis (Task: 4.2)
- Shared Memory Analysis - Conflicts (Task: 4.3)
- Matrix Multiply – CPU sequential version (Task: 4.4)

(22 points)

Total: 66 points