High-Performance Computing, Winter 2021

Exercise 1: Processor and Memory

Task 1. Pipelining
Pipelining is an effective technique to increase throughput on instruction level. Assume a given architecture with a latency of 3 cycles for the execution of a single *add* operation (not counting other pipeline stages, such as load and store, ...).

Task 1.1. Pipelining Issues
We focus on the reduction of an array as described by the following listing:

```
const int N=20;
double s = 0.0;
double A[N]; // init A (not shown here)
for (i=0; i<N; i++)
  s = s + A[i];
```

Are there (real) dependencies in the code with respect to the execution of the *add* operation? Or can the execution of the *add* operation be perfectly pipelined? Justify your answer.
With respect to your result, can you imagine which optimization is important for this reduction example?
Hint: It might be required to introduce additional variables with your chosen optimization.

Task 1.2. Theoretical Speedup
Now, assume perfect pipeline utilization of the execution of the add operation. What can be the maximum speedup with the given N? What is the theoretical speedup for an infinitely big N?

Task 2. Caches
Modern processors use cache memory, or short caches, to bridge the gap between processor performance and memory access speed. In this task different attributes of caches are discussed. Therefore, we analyze a simplified processor architecture.

Task 2.1. Associativity
Caches implement a certain type of associativity. Name and describe the different types of associativity as presented in the lecture.

Task 2.2. Locality
In relation to data locality, *spatial locality* and *temporal locality* are two important terms. Give a definition for both terms and explain their differences.

Task 2.3. Speedup
Be there a cache of access time $T_c = 20\,\text{ms}$, a local memory with access time $T_m = 180\,\text{ms}$ and a cache-hit-rate $\beta = 0.5$. Calculate the average access time for an arbitrary memory location. Additionally calculate how much speedup is gained due to use of the cache in this example.

Task 2.4. Capacity
Next, we assume a direct mapped cache has a size of $C = 1\,\text{MiB}$ ($=2^{20}$ byte)[1] and one cache line has a length of $C_L = 64\ \text{B} = 2^6\ \text{B}$. How many bits of a memory address are used as an index to identify the corresponding cache line when this cache line is fetched from main memory?

Task 2.5. Coherency
Explain in short what the term *cache coherency* means.

Task 3. STREAM2 benchmark
The STREAM benchmark (`http://www.cs.virginia.edu/stream/`) is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels.

Compile the STREAM2 benchmark, which is an adaptation of STREAM measuring with different vector sizes and using different vector kernels:

```
$ make
```

For reliable results, you have to bind the running program to one processor core. Otherwise, the operating system may migrate it to any other core during the execution. That will cause overhead for context switches on the one hand side. On the other hand, it might generate remote memory accesses on NUMA nodes (will be covered later in the lecture). As consequence, varying runtimes are measured if threads are not pinned.
If using a serial program, `taskset` is sufficient. This command cuts down the execution of a process to a given set of CPUs / cores on the system (defined by `-c`). That means that the process cannot run on any other cores but the ones given in the set. However, the operating system may still migrate the process between cores the given set. If you pass a singleton set to `taskset` (as we will do here), then you can force the execution to a specific core.

Run the benchmark in a batch job. For that, use the provided batch script `batchStream2.sh`. Submit the batch job by

```
$ sbatch batchStream2.sh
```

You can test it on a frontend node as well by running

```
$ taskset -c <ID> ./stream2.exe | tee stream2.txt
```

As `ID`, you can use any core number given by `lstopo`. At best not core 0. On this core, the operating system usually has load so that performance might not be the best.
Moreover, the output is piped to an arbitrary file (here `stream2.txt`), so that you can further use the gathered information.

[1]Note: We always use the International System of Units (SI), which means that unit prefixes for kilo (k), mega (M), etc. represent mulitplication by powers of ten. In contrast, binary prefixes like kibi (Ki), mebi (Mi) represent multiplication by powers of two.

Have a short look at the output. It delivers the throughput in MB/s for each operation (FILL, COPY, DAXPY, DOT). Look at the source code to see a description of these operations.

Represent the results graphically using gnuplot (a command reference can be found at `http://www.gnuplot.info/docs_4.6/gnuplot.pdf`). A prepared gnuplot file (`plotData.plt`) can be found in the current directory:

```
set output "stream2.png"
set terminal png
set logscale x
set xlabel '#elements in vector'
set ylabel 'MB/s'
plot FILE using 1:3 title "FILL" with linespoints, FILE using 1:4 title
"COPY" with linespoints, FILE using 1:5 title "DAXPY" with linespoints,
FILE using 1:6 title "DOT" with linespoints
```

For usage, pass as argument (`-e`) the file to analyze to gnuplot:

```
$ gnuplot -e "FILE='stream2.txt'" plotData.plt
```

Open the plotted png file

```
$ display stream2.png
```

What can you see? Explain the stairs in the figure.

Discussion of this exercise on November 5, 2021.