DD2356 Assignment 1
Performance Modeling & Benchmarking

Yannis Paetzelt - paetzelt@kth.se

April 23, 2021

Exercise 1

1. The Haswell nodes on Beskow advertise a clock speed of 2.3GHz. For one such core we have *clock-rate∗DP/s* FLOPs where *DP/s* is known to be 16 for the Haswell architecture. The test program is using double-precision floating point values, thus we use the processor's double-precision operation per second value (*DP/s*). This yields $2.3 * 10^9 * 16 = 36.8$ GFLOPs. Taking the reciprocal we obtain the number of seconds per operation, $c = 2.71 * 10^{-11} s$.

   The total execution time is modeled by $Time = nnz * 2c$ where $nnz = 5 * nrows$ for $nrows \in \{10^2, 10^4, 10^6, 10^8\}$. Theoretical execution times for different matrix sizes are then:

$$Time_0 = 5 * 10^2 * 2 * 2.71 * 10^{-11} = 2.71 * 10^{-8}$$
$$Time_1 = 5 * 10^4 * 2 * 2.71 * 10^{-11} = 2.71 * 10^{-6}$$
$$Time_2 = 5 * 10^6 * 2 * 2.71 * 10^{-11} = 2.71 * 10^{-4}$$
$$Time_3 = 5 * 10^8 * 2 * 2.71 * 10^{-11} = 2.71 * 10^{-2}$$

2. Running the program on Beskow we obtain the following execution times for various matrix sizes (modified slightly to produce more significant digits in the output):

```
Time for Sparse Ax, nrows=100, nnz=460, T = 0.0000009537
Time for Sparse Ax, nrows=10000, nnz=49600, T = 0.0000910759
Time for Sparse Ax, nrows=1000000, nnz=4996000, T = 0.0118219852
Time for Sparse Ax, nrows=100000000, nnz=499960000, T = 1.0672960281
```

   Comparing the measured times with the theoretical times we see that in reality the computations sometimes take several orders of magnitude longer than estimated. We can compute the number of FLOPs performed for each run and compare to the theoretical value of 36.8

GFLOPs. The number of FLOPs is computed as follows, $(nnz * 2)/Time$.

$$FLOPs_0 = (460 * 2)/9.537 * 10^{-7} = 9.65 * 10^8$$
$$FLOPs_1 = (49600 * 2)/9.107 * 10^{-5} = 1.089 * 10^9$$
$$FLOPs_2 = (4996000 * 2)/1.182 * 10^{-2} = 8.453 * 10^8$$
$$FLOPs_3 = (499960000 * 2)/1.067 = 9.37 * 10^8$$

3. Presumably there is a bottleneck in the system preventing full utilization of the processor. It is likely that this would be the memory architecture causing additional delay related to reading and writing to main-memory.

4. Using $nnz * (sizeof(double) + sizeof(int)) + nrows * (sizeof(double) + sizeof(int))$, and assuming $sizeof(double) = 8B$ and $sizeof(int) = 4B$, we compute the read bandwidth for each run:

$$BW_0 = (460 * 12B + 100 * 12B)/9.537 * 10^{-7} = 7.046 * 10^9 B/s$$
$$BW_1 = (49600 * 12B + 10^4 * 12B)/9.107 * 10^{-5} = 7.853 * 10^9 B/s$$
$$BW_2 = (4996000 * 12B + 10^6 * 12B)/1.182 * 10^{-2} = 6.087 * 10^9 B/s$$
$$BW_3 = (499960000 * 12B + 10^8 * 12B)/1.067 = 6.747 * 10^9 B/s$$

5. Running STREAM on a Beskow Haswell node yields the following (extraneous output omitted):

```
//clipped
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           18270.9     0.008861     0.008757     0.009001
Scale:          11688.2     0.013922     0.013689     0.014121
Add:            13105.3     0.018624     0.018313     0.018932
Triad:          13083.4     0.018624     0.018344     0.018912
-------------------------------------------------------------
//clipped
```

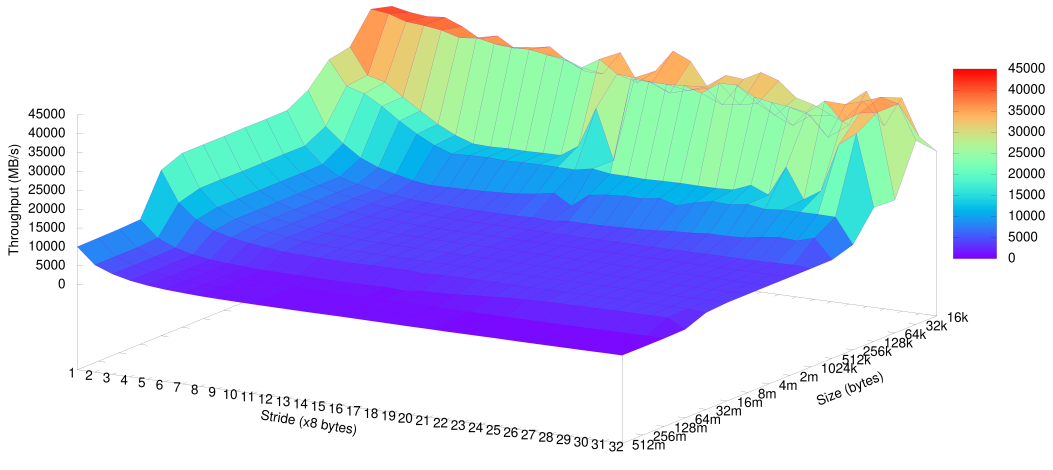
The benchmark has a much higher bandwidth than SpMV. This is because STREAM is running on all the cores in the node when SpMV is only running on one core.

Exercise 2

Benchmarking was performed on the Haswell nodes on Beskow, these are Xeon E5-2698v3's. Cache sizes for this processor area as follows, L1: 512KB, L2: 4MB, and L3: 40MB.

Small array sizes near 32k in combination with any size stride consistently get the most performance at a throughput of around 35GB/s.

Consistently low performance is seen with arrays from 1024k up to 512m in combination with high strides from 4 and up.



At a stride of 1, spacial locality is high and the L1 cache is read often, albeit more slowly than high strides, allowing the cpu time to prefetch the next chunck of data into the higher level caches. The overall effect of this is the L1 cache can fetch from the next cache level where the data is guaranteed to be, rather than resorting to accessing the main-memory, reducing overall access times for reads.

Temporal locality refers to reusing the same data multiple times. Spacial locality refers to (re)using nearby data, either in reference to a contiguous block of memory or in the cache.

Changing the array size affects how much of the array fits in the cache at a time, in turn affecting how much of the array can be reused without slow memory-access in computation at a given time. Increasing array size may decrease spacial locality.

Changing the read stride affects how much of the array is read from the cache at a time. Small strides have the effect of using nearby data in the cache more often, therefore increasing spatial locality.

Exercise 3

Switching the processor environment can be done by running `module swap PrgEnv-cray PrgEnv-gnu`.

The average running time of the benchmark when compiled with `-O2` seems to be near zero. Increasing N has no perceivable effect on the measured runtime.

Looking at the assembly code it is obvious why the runtime is as the way it is. Enabling the optimization, the compiler notices that the result of the computation is never used and so does not emit any code for doing the computation in the first place. If we remove the optimisation flag, we do obtain a measurable runtime for the benchmark: `Execution time:  0.00002813 s`.

Running the clock granularity test on Beskow yields the following output:
`# min dist = 9.54e-07, max dist = 1.19e-06, total time = 2.15e-06`
This indicates a clock granularity of approximately 1 μs.

Using RDTSC to compute time based on the number of CPU cycles yields:
`# min dist = 1.13e-08, max dist = 1.13e-08, total time = 0.00e+00`
Indicating a granularity closer to on the order of 10 ns.

Implementing the modifications to the benchmark code and running it multiple times gives a good idea of the true performance. Below is an example of the output for one run.

```
Execution time:  0.00000265 s
Min time:  0.00000191 s
```

Exercise 4

On Beskow, the `perf` command should be placed after `srun` as otherwise we would be measuring aspects of the job scheduling system rather than the executable we are interested in.

EVENT	MSIZE = 64 Naive	MSIZE = 64 Optimized	MSIZE = 1000 Naive	MSIZE = 1000 Optimized
Elapsed time (seconds)	0.009426274	0.008338597	18.152895708	5.816990363
IPC	1.3436	2.2429	1.0516	4.1413
L1 miss ratio (%)	7.61	2.49	60.28	6.27
L1 miss rate PTI	24.8917	7.09988	200.71302	17.91405
LLC miss ratio	N/A	N/A	1.9055e-5	2.435e-4
LLC miss rate PTI	N/A	N/A	3.9986e-4	9.9653e-5

Table 1: `perf` statistics for `matrix_multiply`

The loop-reordering optimization seems most beneficial for large matrix sizes, due to more efficient use of the cache.

Exercise 5

EVENT	N = 64	N = 128	N = 2048	N = 2049
Elapsed time (seconds)	0.007108168	0.010620070	5.026974547	4.329650309
Bandwidth/Rate (MB/s)	9.68e+03	4.16e+03	7.12e+02	8.94e+02
IPC	1.2	1.02	0.19	0.22
L1 miss ratio (%)	8.68	55.67	112.59	112.65
L1 miss rate PTI	20.0711	124.2291	223.5149	223.4811
LLC miss ratio (%)	N/A	N/A	76.11	15.33
LLC miss rate PTI	N/A	N/A	104.8409	16.8474

Table 2: `perf` statistics for `transpose`

Small matrices fit into the cache, so there is a smaller performance penalty as main-memory is accessed less frequently. With large matrices, only parts of each matrix can fit into a cache line, and unpadded power-of-two stride sizes in the case of the 2048 transpose cause frequent cache line evictions, causing performance degradation and a lower overall bandwidth when compared to the padded 2049 transpose.

To improve cache re-use one can modify the transpose function to use blocking with an appropriate block size such that required array accesses are local to one cache line. Similarly the loops can be reordered to improve temporal locality.

Exercise 6

For GCC, supplying the `-O3` flag applies vectorization by default. GCC will provide a vectorization report with `-fopt-info-vec-all`. Other command line options that affect vectorization on various systems include: `-maltivec` for powerpc* platforms, `-msse\-msse2` on x86_64, and `-ffast-math` or `-fassociative-math` for math related optimizations (also enabled by `-Ofast`).

Compiling the optimized matrix_multiply code with `-O3`, the vectorization report indicates that some loops could be vectorised, but that most could not due to alignment issues with the data types. Some types also do not have a vector-type equivalent that the compiler can use. Additionally, the compiler cannot optimize function calls, unless they are inlined.