Praktikum im SS 2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Vincent Bode, MSc., Minh Chung, MSc., Dennis-Florian Herr, MSc., Bengisu Elis, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Assignment 03 – Due: May 19, 2022, 14:00

Memory Hierarchy Properties

In this assignment, we look at various properties of the memory hierarchy, we try to understand its performance effects and compare the different systems in BEAST.

1. **Cache Line Utilization**

   For the first part of this assignment we change the triad benchmark from Assignment 1 by introducing a stride between memory accesses of triad computations, with parameter `N` of the `triad` function now being the number of triad computations done over the arrays. That is, each array now needs to have `N * STRIDE` elements. This allows us to compare the runtime for a fixed workload (depending on `N`), but using different data layout in memory. As this results in reads/writes of only every stride-th element in the vector, we emulate a memory access pattern similar to what is known as an array of structures (AoS). AoS appears in many object-oriented applications, where in particular data structures of a certain size (e.g., a `struct` of `stride` consecutive `doubles`) are packed into an array and the application uses only one element of the data structure (e.g., the first `double` in each `struct`) when iterating through the AoS (see Fig. 1).

   (a) Use the parallel vector triad implementation from Assignment 1. The triad loop `j` now becomes `for (long j=0; j<N*STRIDE; j+=STRIDE)`. Add parsing of a 3rd command line argument `STRIDE`, add `STRIDE` as additional parameter to function `triad`, and also adjust allocation/initialization.

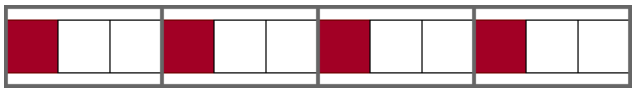

Figure 1: Illustration of AoS: in this illustration three elements are packed in a `struct` and we have an array of four `structs`. The application iterates through this array and only accesses the first element, i.e. highlighted in red.

```
// basic elements of the linked list
struct entry { double v; int next; };

void init(int N, int k, struct entry* A)
{
  int mask = N-1; // N is power-of-2
  for( int i=0; i<N; ++i ) {
    A[i].v    = (double) i;
    A[i].next = (k*(i+1)) & mask;
  }
}
```

Figure 2: Initialization of list (task 2).

(b) Measure the performance in MFlops/sec of the triad microbenchmark with one thread pinned to a core and values of N (as parameter of `triad`) going from 32 to 2^{25} (use values up to $N = 2^{23}$ on the A64FX), but this time add different curves for various values of `STRIDE` in the following set: $\{1, 2, 4, 8, 16, 32, 64\}$. Perform these experiments on all systems in BEAST. Analyze the results of your experiments, and try to explain why you observe different curves for various stride values. Furthermore, compare the results from the different systems and try to explain the observed behaviours.

(c) Now repeat the same experiments by setting the number of threads to a value which gave you the best performance in Assignment 1. Do not forget to pin the threads and document your setting of OpenMP thread pinning. Explain the results.

2. **Linked List Traversal**

Measurements in this task only use one thread - no parallelization is requested.

List data structures are essential for a lot of algorithms. To allow for fast insertion and removal of elements of the list, often, so-called linked lists are used. After lots of insertions and removals, a traversal of a linked list results in almost random accesses to memory. We emulate this access pattern by traversing the elements of an array with N elements in a pseudo-random order. We emulate the pseudo-random accesses by accessing the $(k \cdot i \bmod N)$-th element of A for the i-th access, with the ratio of k and N being close to the golden ratio ($\frac{1+\sqrt{5}}{2}$). We ensure that all elements of A are visited by asserting that k and N are coprime integers. Also we choose a power-of-2 for N as this allows a fast version of the modulo operator by masking out the lower $log_2(N)$ bits of N through an AND operation.

We compare two code variants of exactly the same traversal, with the same memory accesses being done, and same computations done. The only difference is that the first version calculates the indexes during traversal from i, while the second one gets the index of the next element to access from the previously accessed element. Elements store a next-index value and another double value to be summed up. As we want exactly the same accesses in both traversals, we need to use the accessed index in both cases by some dummy calculation. To avoid that the compiler optimizes out this calculation, the final result is stored to a location outside of the traversal function. The initialization is shown in in Fig. 2.

The source snippet in Fig. 3 provides the first version of our emulated traversal, with the indexes to be accessed being calculated. The source snippet in Fig. 4 shows the second variant with the index of the element to be accessed next being loaded from the previously accessed element.

For the k used to approximate the golden ratio, make sure that k and N actually are co-prime.

```
double sum_indexcalc(int N, int k, int REP, struct entry* A, double *psum,
    int* pdummy)
{
  auto t0 = std::chrono::high_resolution_clock::now();
  int mask = N-1; // N is power-of-2
  int dummy = 0;
  double sum = 0.0;
  for( int r=0; r<REP; ++r ) {
    int next = 0;
    for( int i=0; i<N; ++i ) {
      sum   += A[next].v;
      dummy |= A[next].next;
      next   = (k*(i+1)) & mask;
  }
  auto t1 = std::chrono::high_resolution_clock::now();
  *psum = mysum; *pdummy = dummy;

  // return average time per element access in nanoseconds
}
```

Figure 3: Code snippet with index calculation during traversal.

You might want to implement an assertion scheme for verifying that all elements are accessed in both of your functions. (e.g., by first setting $v = 0$ for all entries, then set $v = 1$ in the traversal, and check that v of all entries are as expected).

```
double sum_indexload(int N, int k, int REP, struct entry* A, double *psum,
    int* pdummy)
{
  auto t0 = std::chrono::high_resolution_clock::now();
  int mask = N-1; // N is power-of-2
  int dummy = 0;
  double sum = 0.0;
  for( int r=0; r<REP; ++r ) {
    int next = 0;
    for( int i=0; i<N; ++i ) {
      sum   += A[next].v;
      dummy |= (k*(i+1)) & mask;
      next   = A[next].next;
  }
  auto t1 = std::chrono::high_resolution_clock::now();
  *psum = mysum; *pdummy = dummy;

 // return average time per element access in nanoseconds
}
```

Figure 4: Code snippet using indexes pre-calculated before traversal.

(a) Implement the full code to run traversals of `sum_indexcalc()` and `sum_indexload()`. Measure the performance of both variants by plotting the average time required for traversing one element of the list, with array size N set to $\{2^{10}, 2^{11}, ...2^{30}\}$ and k on the one hand set to 1 and on the other hand set to the value such that k/N is near the golden ratio. Make sure to bind the thread during the runs to a fixed core. Do the measurement on all systems in BEAST.

(b) Try to explain the results of your experiments. Especially, what is the influence of the

size N, setting of k, and the difference of the two code variants `sum_indexcalc()` and `sum_indexload()`?

(c) For random access traversal with a large enough N, the first access into a list element should be a cache miss. With this assumption, what is the utilized memory bandwidth? How large is it for the various systems, for each of the variants?

(d) The memory access latency is the elapsed time between a core triggering a load of a value from main memory and being able to use the loaded value. Assume that in contrast to main memory access latency, the second access into the same element always hits L1 and thus is neglectable. With this assumption, can you derive the memory access latency to local memory (i.e. within a NUMA domain) of the various systems from the results of the previous sub-task? Which value of N, k, and which variant can be used to derive the memory access latency?
Hint: the chosen setting must ensure that there is no prefetching happening and no memory parallelism possible (i.e. only one outstanding memory access at each point in time). Why is this important?

(e) Can you derive the access latency to other levels in the memory hierarchy (data caches) from the measurements of sub-task (a)? For example, as Table 1 shows.
Hint: you can use the "lscpu" command to obtain relevant information about the caches. Think about what is a reasonable value of N on each case.

	L1	L2	L3	Memory
rome	10ns / $N = 2^a$	20ns / $N = 2^b$	30ns / $N = 2^c$	40ns / $N = 2^d$
ice	...	...		
cs	...			
thx				

Table 1: Example table for access latencies of the memory hierarchies.

(f) Compare the measured latencies on the A64FX system to the vendor-provided values. You can find them in the A64FX microarchitectural manual [1] (load-to-use).
Hint: the A64FX cores run at 1.8GHz.

(g) Look up the manual page of "numactl". Use this tool to display the NUMA node distances on each system. What is the semantic of the reported values for the NUMA node distances? Which of the values corresponds to the measurement from sub-task (d)?
Hint: read this section [2] about the System Locality Information Table (SLIT).

(h) The tool `numactl` allows you to explicitly incorporate the memory allocation in a separate NUMA domain to enforce sequential traversals in a NUMA domain different from the one the code runs on. For each distinct NUMA distance, perform one measurement of the resulting memory access latency using `numactl` and an appropriate value of N. Document how you performed the measurements and provide a table for the latency data similar to Table 1. Compare the NUMA node distance reported earlier by `numactl` to the values derived from your measurements.

[1] https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.6.pdf pages 64,65,68

[2] https://uefi.org/specs/ACPI/6.4/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html#system-locality-information-table-slit