Università della Svizzera italiana | **Institute of Computational Science ICS**

High Performance Computing **2019**

Instructor: Prof. Olaf Schenk TA: Juraj Kardos, Radim Janalik, Aryan Eftekhari

Mini-project

Due date: 15 October 2019, 13:30

Memory Hierarchies, Single-Core Optmization (100 points)

In this mini-project you will practice data access optimization and performance-oriented programming for cluster environments.

Data can be stored in a computer system in many different ways. CPUs have a set of registers, which can be accessed without delay. In addition there are one or more small but very fast caches holding copies of recently used data items. Main memory is much slower, but also much larger than cache. Finally, data can be stored on disk and copied to main memory as needed. This a is a complex hierarchy, and it is vital to understand how data transfer works between the different levels in order to identify performance bottlenecks.

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die. The need for caches can be easily understood by realizing that data transfer rates to main memory are painfully slow compared to the CPU's arithmetic performance. Caches can alleviate the effects of the DRAM gap in many cases. Usually there are several levels of cache (see Figure 1), called L1D (D stands for data, L1 is usually shared with instruction cache), L2 and L3 respectively.

When the arithmetic unit (AU) executes an instruction (e.g. add, mult) it assumes that the operands are located in the registers. If they are not, the CPU first needs to issue load instructions to fetch the data from some location in the memory hierarchy. Whenever the CPU issues a load request for transferring a data item to a register, first-level cache logic checks whether this item already resides in cache. If it does, this is called a cache hit and the request can be satisfied immediately, with low latency. In case of a cache miss, however, data must be fetched from outer cache levels or, in the worst case, from main memory.

Caches can only have a positive effect on performance if the data access pattern of an application shows some locality of reference. More specifically, data items that have been loaded into a cache are

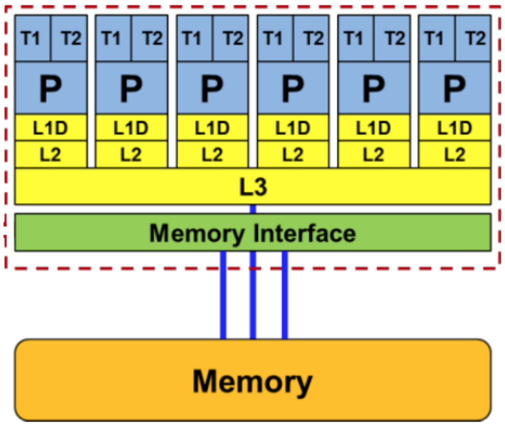

Figure 1. Memory hierarchy of multi-core architecture

to be used again "soon enough" to not have been evicted in the meantime. This is also called temporal locality. We will exploit temporal locality to improve performance of the code in the part 2 of this mini-project. In this part we will benchmark the memory subsystem to see the effect of the memory hierarchy. Detailed explanation of memory hierarchy can be found in the additional material on the course webpage, titled *Motivation for Improving Matrix Multiplication* or in the book *Introduction to High Performance Computing for Scientists and Engineers* [3].

1. Explaning impact of memory hierarchies [30 points]

Solve the following problems:

1. Identify parameters of the memory hierarchy on the **compute node** of the icsmaster cluster:

Main memory	. . . GB
L3 cache	. . . MB
L2 cache	. . . kB
L1 cache	. . . kB

   You might find useful:

```
$ module load likwid
$ likwid-topology

$ cat /proc/meminfo
```

2. The directory `membench` on GitHub contains:

   - `membench.c` – a program in C to measure the performance (benchmark) of different memory access patterns;
   - `Makefile` – a Makefile to compile and run the code;

- `gnuplot` – a GnuPlot script for displaying performance results.
- `run_membench.sh` – a bash script for collecting performance results.
- `plot_membench.sh` – a bash script for displaying performance results.

Compile `membench.c` into `membench` binary and run it using the provided `Makefile`:

- on your local machine, e.g. laptop (you may need to install `gnuplot`):

```
$ cd membench
$ make
$ ./run_membench.sh
$ ./plot_membench.sh
```

- on the icsmaster cluster:
  - compile on login node

```
$ cd membench
$ module load gcc/6.1.0 gnuplot
$ make
```

  - run on compute node

```
$ ./run_membench.sh
```

  - or start batch job from login node

```
$ sbatch run_membench.sh
```

    (In case of icsmaster, the resulting `generic.ps` will be available in few minutes (check the job status with `squeue`).)

  - plot graph on login node

```
$ ./plot_membench.sh
```

3. Using the resulting `generic.ps` files (view them with your favourite PDF viewer) and `membench.c` program source, characterize the memory access pattern used in the following cases:
   - $csize = 128$ and $stride = 1$;
   - $csize = 2^{20}$ and $stride = csize/2$.
4. Analyse the resulting `generic.ps` file produced by `membench` on the icsmaster cluster (open `generic.ps` file with your favourite PDF viewer):
   - Which array sizes and which stride values demonstrate good temporal locality? Explain.

Please include the answers in your Latex report. It should also contain the `generic.ps` files produced by `membench` on icsmaster cluster and on your local machine (please specify the type and operating system of the local machine you used) and explanation or the resulting graph in more detail.

2. Optimize Square Matrix-Matrix Multiplication [70 points]

Matrix multiplication is the basic building block in many scientific computations; and since it is an $\mathcal{O}(n^3)$ algorithm, these codes often spend a lot of their time in matrix multiplication. However, the arithmetic complexity is not the limiting factor on modern architectures. The actual performance of the algorithm is also influenced by the memory transfers. We will illustrate the effect with a common technique for improving cache performance, called blocking. Please refer to the additional material on the course webpage, titled *Motivation for Improving Matrix Multiplication* or in the book.

Since we want to write fast programs, we must take the architecture into account. The most naive code to multiply matrices is short, simple, and very slow:

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end
  end
end
```

Instead, we want to implement the algorithm that is aware of the memory hierarchy and tries to minimize number of references to the slow memory (please refer to the "Motivation for Improving Matrix Multiplication" document provided on the HPC course web page `mini-project-info.pdf` for more detailed explanation):

```
for i=1 to n/s
   for j=1 to n/s
      Load C_{i,j} into fast memory
      for k=1 to n/s
         Load A_{i,k} into fast memory
         Load B_{k,j} into fast memory
         NaiveMM (A_{i,k}, B_{k,j},  C_{i,j}) using only fast memory
      end for
      Store C_{i,j} into slow memory
   end for
end for
```

The directory `matrixmult` on GitHub contains:

- `basic_dgemm.c` – a trivial unoptimized matrix multiply code in C;
- `blas_dgemm.c` – a wrapper-function calling external optimized matrix multiply function provided by Basic Linear Algebra Subroutines (BLAS) library. The BLAS is a standard interface for building blocks like matrix multiplication. Many vendors provide optimized versions of the BLAS for their processor architectures;
- `blocked_dgemm.c` – a placeholder for blocked square matrix muliply you will implement;
- `matmul.c` – the "driver" program, which provides sample data and time measurements for user-defined and BLAS versions of matrix multiplication above;

- `Makefile` – a Makefile to compile and run the code;
- `timing.gnuplot` – a GnuPlot script for displaying performance results.
- `run_matrixmult.sh` – a bash script for collecting performance results.
- `plot_matrixmult.sh` – a bash script for displaying performance results.

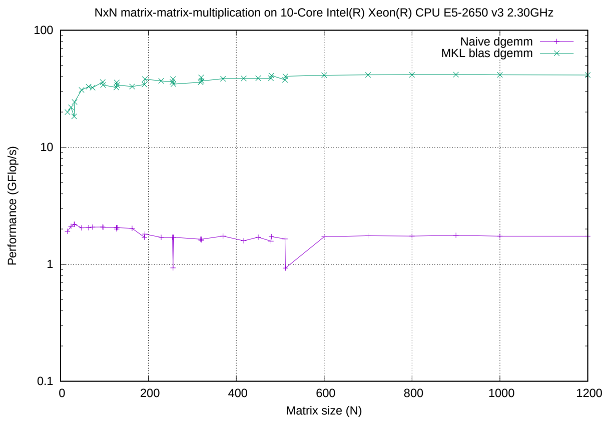

Figure 2. Performance of matrix-matrix multiplication on the 10-Core Intel(R) Xeon(R) CPU E5-2650 v3.

The performance of the naïve basic “three loops” version `basic_dgemm.c` and the optimized blocked code from the Intel MKL library [1] `blas_dgemm.c` is shown in Fig. 2.

Solve the following problems:

You will develop an optimized function for multiplying square matrices.

1. Replace the empty body of `square_dgemm` function in `blocked_dgemm.c` with your own implementation of blocked square matrix multiply in C:

```c
void square_dgemm (const unsigned M,
                   const double *A, const double *B, double *C);
```

2. Compare your implementation performance to the OpenBLAS or Intel MKL by compiling & running the driver program and visualizing the performance results:

   - on your local machine, e.g. laptop (you may need to install `gnuplot`):

```
$ cd matrixmult
$ make
$ ./run_matrixmult.sh
$ ./plot_matrixmult.sh
```

   - on the icsmaster cluster:
     - compile on login node

```
$ cd matrixmult
$ module load gcc/6.1.0 mkl/11.3 gnuplot
$ make
```

- run on compute node

```
$ ./run_matrixmult.sh
```

- or start batch job from login node

```
$ sbatch run_matrixmult.sh
```

  (In case of icsmaster, the resulting `timing.ps` will be available in few minutes (check the job status with `squeue`).)

- plot graph on login node

```
$ ./plot_matrixmult.sh
```

3. Explain the inhomogeneous behavior of the performance curves in the resulting `timing.ps` files;
4. Explain the differences in performance between icsmaster cluster and your local machine;
5. Make your conclusion about the relative difficulty of using someone else's library to writing and optimizing your own routine.

Note that the matrices are stored in C style row-major order. However, the BLAS library expects matrices stored in column-major order. When we provide matrix stored in row-wise ordering to the BLAS, the library will interpret it as its transpose. Knowing this, we can use an identity $B^T A^T = (AB)^T$ and provide matrices A and B to BLAS in row-wise storage, swap the order when calling `dgemm` and expect the transpose of the result, $(AB)^T$. But the result is returned again in column-wise storage, so if we interpet it in row-wise storage, we obtain the desired result AB. Have a look at `blas_dgemm.c` to see how the A and B are passed to `dgemm`. Also, your program will actually be doing a multiply and add operation $C := C + A \cdot B$. Look at the code in `basic_dgemm.c` or study the `dgemm` signature if you find this confusing.

The driver program supports result validation (enabled by default). So during the run of `blocked_dgemm` binary compiled from the `square_dgemm` code you wrote, the results correctness will be automatically checked for different matrix sizes.

The Latex report should include comments on your `blocked_dgemm.c` implementation and performance visualizations produced on icsmaster cluster and on your local machine.

Additional Notes

Submission:

Submit the source code files (together with used `Makefile`) in an archive file (tar, zip, etc) and summarize your results and the observations for all exercises by writing an extended Latex report. 4 Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi `https://www2.icorsi.ch/course/view.php?id=7797`.

Additional resources:

You may find useful the "Motivation for Improving Matrix Multiplication" document provided on the HPC course web page (`mini-project-info.pdf`).

References

[1] Intel MKL – https://software.intel.com/en-us/intel-mkl

[2] OpenBLAS – http://www.openblas.net

[3] Hager, G. and Wellein, G. Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Inc. 2010. ISBN 9781439811924.