Università della Svizzera italiana

Institute of Computing CI

HPC Software Atelier, Fall Semester 2022
Team: Prof. O. Schenk, Dr. J. Kardoš
T. Holt, M. Lechekhab, Z. Panthakkalakath

Project 1
Memory Hierarchies and Matrix-Matrix Multiplications
Due date: 12 October 2022, 11:59pm (midnight)

1. Explaining the impact of memory hierarchies [25 points]

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. The main memory is much slower, but also much larger than cache. This is typically 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 the main memory.

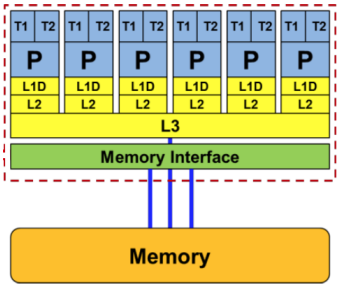

Figure 1: Memory hierarchy of a multicore architecture.

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 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 part 2 of this mini-project. In this part we will benchmark the memory subsystem to see the effect of the memory hierarchy. A detailed explanation of memory hierarchy can be found in [1].

Problem statement

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

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

[1] *Motivation for Improving Matrix Multiplication* or in the book *Introduction to High Performance Computing for Scientists and Engineers* [2], in particular Chapter 2 on "Basic optimization techniques for serial code" and Chapter 3 on "Data access optimization."

You might find the following useful:

```
$ module load likwid
$ likwid-topology

$ cat /proc/meminfo
```

2. The directory `membench` on GitHub and the iCorsi course webpage 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
     ```

     Note: If you have a laptop with CPU based on arm64 architecture (e.g. Mac M1 chip) you will probably run into errors while compiling the benchmark script, e.g. *error: invalid output constraint '=a' in asm*. In order to compile the code, you have to run the commands in a Rosetta[2] enabled terminal. Rosetta is Apple's translation layer which emulatex x86_64 architecture on Apple Silicon. For instructions, please follow one of the tutorials available, e.g. "Set Up Your Apple M1 for Development".
   - on the ICS cluster :
     - compile on login node

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

     - start batch job from login node

       ```
       $ sbatch run_membench.sh
       ```

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

3. Using the resulting `generic.ps` files (view them with your favorite 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. Analyze the resulting `generic.ps` file produced by `membench` on the ICS cluster (open `generic.ps` file with your favorite PDF viewer):
   - Which array sizes and which stride values demonstrate good temporal locality? Please explain.

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

[2] `https://support.apple.com/en-us/HT211861`

2. Optimize Square Matrix-Matrix Multiplication [60 points]

Problem statement

Your second task in this project[3] is to write an optimized matrix multiplication function on the ICS computer. We will give you a generic matrix multiplication code (also called matmul or dgemm), and it will be your job to tune our code to run efficiently on the ICS processors.
Write an optimized single-threaded matrix multiply kernel. This will run on only one core.

Matrix multiplication

Matrix multiplication is the basic building block in many scientific computations; 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 the 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
```

Starter Code

Download the starter code: Directory matmul
The directory contains starter code for the matrix multiply. It contain the following source files:

- `dgemm-blocked.c` – A simple blocked implementation of matrix multiply. It is your job to optimize the square_dgemm() function in this file.
- `dgemm-blas.c` – A wrapper which calls the vendor's optimized BLAS implementation of matrix multiply (here, MKL).

[3] This document is originally based on a project from Professor Katherine A. Yelick from the Computer Science Department at the University of Berkeleyhttp://www.cs.berkeley.edu/ yelick/

- `dgemm-naive.c` – For illustrative purposes, a naive implementation of matrix multiply using three nested loops.
- `benchmark.c` – A driver program that runs your code. You will not modify this file, except perhaps to change the MAX_SPEED constant if you wish to test on another computer (more about this below).
- `Makefile` – A simple makefile to build the executables.
- `run_matrixmult.sh` – Script that executes all three executables and produces log files (`*.data`) that contain the performance logs.
- `plot_matrixmult.sh` – Script that plots the data in the performance logs and produces a figure showing the results.

Running our Code

The starter code should work out of the box. To get started, we recommend you to log into the ICS cluster and download the first part of the assignment. This will look something like the following:

```
[user@login]$ git pull
[user@login]$ cd matmul
[user@login]$ ls
Makefile benchmark.c dgemm-blas.c dgemm-blocked.c
dgemm-naive.c run_matrixmult.sh
```

Next let's build the code.

```
[user@eu-login]$ module load gcc intel-mkl
[user@eu-login]$ make
```

We now have three binaries: benchmark-blas, benchmark-blocked, and benchmark-naive. The easiest way to run the code is to submit a batch job. We have already provided batch files which will launch jobs for each matrix multiply version using one core:

```
[user@login]$ sbatch run_matrixmult.sh
```

Our jobs are now submitted to the ICS cluster 's job queue. We can now check on the status of our submitted jobs using a few different commands.

```
[user@login]$ squeue
   JOBID     USER PARTITION     NAME ST   START_TIME     END_TIME TIME_LEFT  NODES
 1173224  user         slim matrixmu  R     14:47:17     15:17:17     29:50      1
 NODELIST(REASON)  PRIORITY
 icsnode17       707
```

When our job is finished, we'll find new files in our directory containing the output of our program. For example, we will find the files matrixmult-xxx.out and matrixmult-xxx.err . The first file contains the standard output of our program, and the second file contains the standard error. Additionally, the performance data are stored in *.data files. The script also generates `timing.ps` file, a visual representation of the results. You can copy the `timing.ps` file to your laptop and open it with your favorite PDF file viewer.

Interactive Session

You may find it useful to launch an interactive session when developing your code. This lets you compile and run code interactively on a compute node that you have reserved. In addition, running interactively lets you use the special interactive queue, which means you'll receive you allocation quicker. The `benchmark.c` file generates matrices of a number of different sizes and benchmarks the performance. It outputs the performance in FLOPS and in a percentage of theoretical peak attained. Your job is to get your matrix multiply's performance as close to the theoretical peak as possible.

Theoretical Peak

Our benchmark reports numbers as a percentage of theoretical peak. Here, we show you how we calculate the theoretical peak of the ICS cluster 's Haswell processors. If you'd like to run the assignment on your own processor, you should follow this process to arrive at the theoretical peak of your own machine, and then replace the MAX_SPEED constant in benchmark.c with the theoretical peak of your machine.

One Core on the ICS cluster

One core has a clock rate of 2.30 GHz, so it can issue 2.3 billion instructions per second. Haswell processors also have a 256-bit vector width, meaning each instruction can operate on 8 32-bit data elements at a time. Furthermore, the Haswell microarchitecture includes a fused multiply-add (FMA) instruction, which means 2 floating point operations can be performed in a single instruction. So, the theoretical peak of the ICS cluster 's Haswell node is

- 2.3 GHz* 8-element vector * 2 ops in an FMA = 36.8 GFlops/s

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 a 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 rowwise 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 interpret it in rowwise storage, we obtain the desired result AB. Have a look at `dgemm-blas.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 `dgemm-naive.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 `benchmark-blocked` binary compiled from the `square_dgemm` code you wrote, the result correctness will be automatically checked for different matrix sizes.

2.1. Optimizing the matrix multiplication

Now, it's time to optimize!

- The `dgemm-blocked.c` contains the naive implementation of the square matrix multiply. Modify the code so that it performs blocking. Test your code and tune block sizes to obtain the best performance.
- Compare performance of your implementation to the Intel MKL by compiling and running the driver program and visualizing the performance results.

3. Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

Additional notes and submission details

Submit the source code files (together with your 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. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi .

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:

  - dgemm-blocked.c, a C-language source file containing your implementation of the routine: void square_dgemm(int, double*, double*, double*);
  - described in pseudocode above. We provide an example dgemm-blocked.c, below.
  - Makefile, only if you modified it. If you modified it, make sure it still correctly builds the provided benchmark.c, which we will use to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - these formats and naming convention, please. Not following these instructions leads to more busy work for the TA's, which makes the TA's sad...

- Submit your .tgz through iCorsi .
- Your Latex write-up should contain the following
  - names of all the students that you discussed your submission with. You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently;
  - the optimizations used or attempted;
  - the results of those optimizations;
  - the reason for any odd behavior (e.g., dips) in performance;
  - comments on your `dgemm-blocked.c` implementation and performance visualizations produced on the cluser and on your local machine; and
  - how the performance changed when running your optimized code on a different machine.
- For the last requirement, you may run your implementation on your laptop.

Your grade will mostly depend on two factors:

- performance sustained by your codes on the ICS machines;
- explanations of the performance features you observed (including what didn't work)

Additional resources:

You may find useful the "Motivation for Improving Matrix Multiplication" document provided on the HPC course web page (`project2-info.pdf`). Please always use the project template which is available on the iCorsi webpage for your submission.

References

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

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

[3] OpenMP Tutorial - LLNL Computation: https://computing.llnl.gov/tutorials/openMP/