ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2023
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: Tim Holt, Malik Lechekhab, Dimosthenis Pasadakis, Xiaohe Niu, Niall Alexander Siegenheim, Schmidt Julien, Jitin Jami

Project 1 — Euler Cluster Warm-Up, AVX Vectorization, and Optimizing Matrix Multiplication
Due date: 06 March 2023, 11:59pm

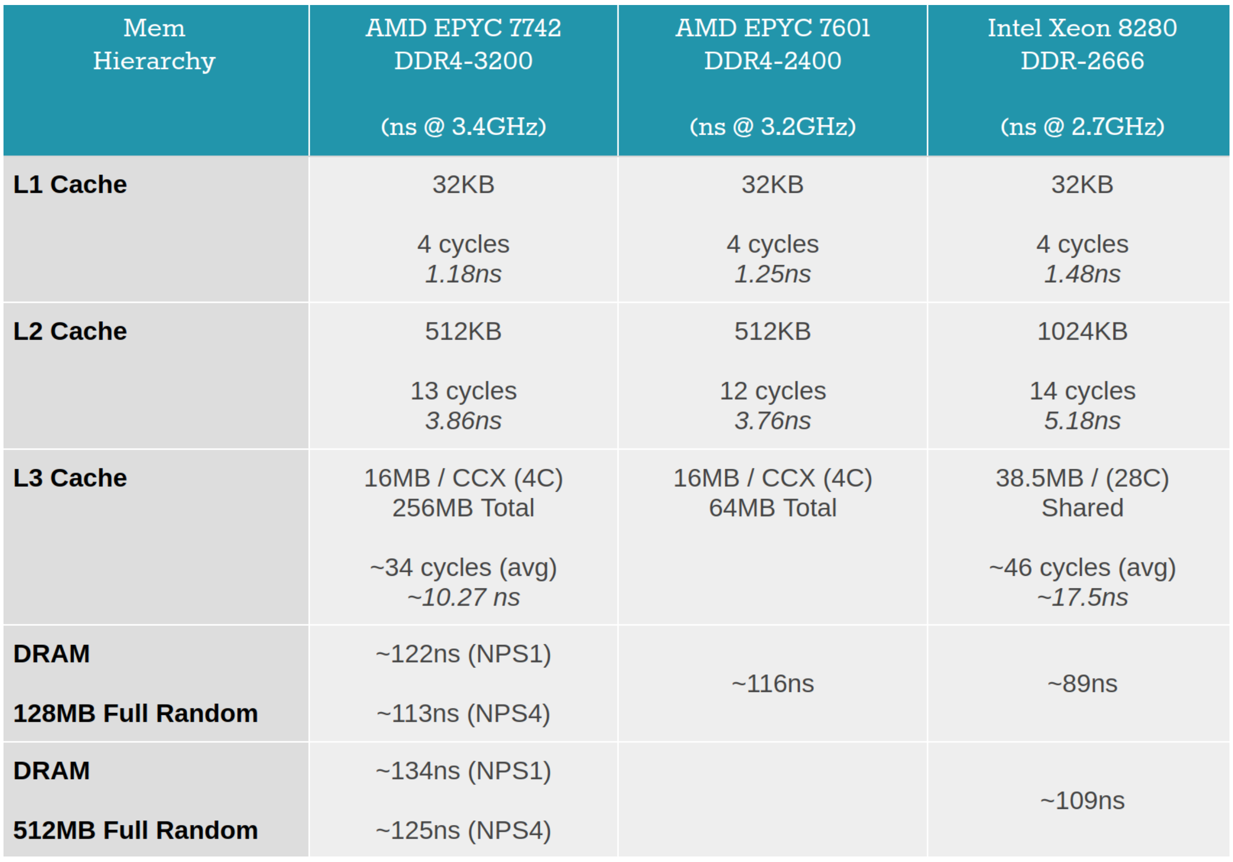
1. Euler warm-up [10 points]

Consult the documentation of the Linux-Cluster and get familiar with the hardware of the Euler cluster [1].

- What is the module system and how do you use it?
- What are the following terms: LSF, SLURM, and LoadLeveler? What do they have in common?
- What software is installed on the Euler Cluster to manage workload and resources of multiple users?
- How can you execute programs on a cluster's compute node? Describe the interactive and batch modes.
- Using the batch mode, write a simple "Hello World" program which prints the host-name of the current machine.
- Compile the program and write a batch script [2] which runs your program on two different nodes of the Euler cluster. (In parallel, do not just execute the batch script twice.)
- Check the logs to ensure that two different host-names were printed.
  Include the source code, batch script, and log in your submission.

2. Explaining the impact of memory hierarchies [10 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.
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. Additionally, data items that are located next to each other in memory are likely to be accessed by a code successively, so they can be loaded together to the cache as well to improve memory access; this is called spatial locality. We will exploit locality of reference to improve



Mem Hierarchy	AMD EPYC 7742 DDR4-3200 (ns @ 3.4GHz)	AMD EPYC 7601 DDR4-2400 (ns @ 3.2GHz)	Intel Xeon 8280 DDR-2666 (ns @ 2.7GHz)
L1 Cache	32KB 4 cycles *1.18ns*	32KB 4 cycles *1.25ns*	32KB 4 cycles *1.48ns*
L2 Cache	512KB 13 cycles *3.86ns*	512KB 12 cycles *3.76ns*	1024KB 14 cycles *5.18ns*
L3 Cache	16MB / CCX (4C) 256MB Total ~34 cycles (avg) *~10.27 ns*	16MB / CCX (4C) 64MB Total	38.5MB / (28C) Shared ~46 cycles (avg) *~17.5ns*
DRAM **128MB Full Random**	~122ns (NPS1) ~113ns (NPS4)	~116ns	~89ns
DRAM **512MB Full Random**	~134ns (NPS1) ~125ns (NPS4)		~109ns

Figure 1: Example of Memory Subsystems on various AMD and Intel Architectures.

performance of the code in section 4 and 5 of this 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 Euler cluster:

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

You might find the following useful:

```
$ lscpu
$ cat /proc/cpuinfo
$ cat /proc/meminfo
```

2. The directory `membench` on the Moodle course webpage contains

[1] *Motivation for Improving Matrix Multiplication* or in the book *Introduction to High Performance Computing for Scientists and Engineers* [5], in particular Chapter 1 ”Modern processors”, Chapter 2 on ”Basic optimization techniques for serial code”, and Chapter 3 on ”Data access optimization.”

- `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;

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
```

- on the Euler cluster :
  - compile on login node

```
$ cd membench
$ module load new
$ module load gcc/6.3.0
$ make
```

  - start batch job from login node

```
$ sbatch run_membench.sh
```

  (In the case of Euler, 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 Euler 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 Euler 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.

3. Auto-vectorisation [10 points]

Please read Chapter ”Automatic Vectorization” of the Intel C++ Compiler Developer Guide and Reference [3] and answer the following questions:

- Why is it important for data structures to be aligned?
- What are some obstacles that can prevent vectorisation?
- Is there a way to assist the compiler through language extensions? If yes, please give some examples. If not, explain why not.
- Which loop optimisations are performed by the compiler in order to vectorise and pipeline loops?

4. The perfect DGEMM micro-kernel [30 points]

DGEMM is an abbreviation for GEneral Matrix Multiplication in Double precision. Given matrices $A \in R^{MxK}$, $B \in R^{KxN}$, and $C \in R^{MxN}$, this operation shall implement

$$C_{m,n} = \sum_{k=O}^{K-1} A_{m,k} B_{k,n}, \quad m = 0, \dots, M-1, \quad j = 0, \dots, N-1. \tag{1}$$

A reference implementation is given in `dgemm.cpp`. Your task is to optimise DGEMM for the AMD EPYC 7742 microarchitecture. Describe your optimization strategy in your report. Try to reach 50% of the theoretical peak performance.

Rules:

- You may only use a single core.
- You must set $K = 128$. The parameters $M, N \in \mathbb{N}$ may be chosen freely, but should be small (i.e. < 32).
- The results must match the reference code up to machine precision.
- You may use any combination of the following techniques: Auto-vectorisation, #pragma-directives, intrinsics, inline assembly.
  - Hints: Read the literature, especially [6]. (You will need to look at it anyway for the next assignment.)
- Use the Intel C++ compiler.
- Have a look at the requirements and think about the following: Where is your data located in the memory hierarchy?
- Dive into the microarchitecture and find out how many vector registers and vector processing units there are? How many cycles does it take to perform one multiplication and addition? How many cycles does it take to accomplish a load vector instruction from memory?
- Compute the theoretical peak performance of your target processor.
- Imagine that the vector register file is the fastest cache, i.e. try to reuse data in vector registers as much as you can.
- It is a good practice to compile your application with ”-S -fsource-asm”. This will generate an assembly listing which allows you to inspect the code generated by the compiler. It is especially useful when you rely on auto-vectorization.

5. Optimize General Square Matrix-Matrix Multiplication [40 points]

Problem statement

Your next task in this project[2] is to use the results from section 4 and integrate it into an optimized matrix multiplication function on the Euler 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 Euler processors.

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

Part One [30 points]

Write an optimized single-threaded matrix multiply kernel. This will run on only one core.

Part Two [10 points]

Write an optimized multi-threaded matrix multiply kernel. This will run on one processor, using all the available cores.

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 `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:

- Part One: Directory matmul_part1
- Part Two: Directory matmul_part2

The first directory contains starter code for part one, the serial matrix multiply; the second directory contains starter code for part two, the multi-threaded matrix multiply using OpenMP. Both versions 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).
- `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. It also 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 Euler cluster and download the first part of the assignment. This will look something like the following:

```
[user@eu-login]$ cd matmul_part1
[user@eu-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 new
[user@eu-login]$ module load gcc/6.3.0 mkl/2018.1
[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@eu-login]$ sbatch run_matrixmult.sh
```

or p cores by setting the slurm option `--cpus-per-task=p`, and the two environment variables `OMP_NUM_THREADS=p` and `MKL_NUM_THREADS=p` in the job script `run_matrixmult.sh`.
Our jobs are now submitted to the Euler cluster 's job queue. We can now check on the status of our submitted jobs using a few different commands.

```
[user@eu-login]$ $ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           9986188 normal.4h matrixmu  karoger  R       0:30      1 eu-g5-024-3
```

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. You can copy the `timing.ps` file to your laptop and open it with your favorite PDF file viewer.

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 Euler cluster 's Skylake 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 Euler cluster

One core has a clock rate of 3.20 GHz, so it can issue 3.2 billion instructions per second. Skylake processors also have a 256-bit vector width, meaning each instruction can operate on 8 32-bit data elements at a time. Furthermore, the Skylake 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 Euler cluster 's Skylake (Intel Xeon Processor E3-1585L v5) node is

- 3 GHz* 8-element vector * 2 ops in an FMA = 48 GFlops/s

Multicore on the Euler cluster

The calculation for multicore is very similar; we simply multiply our computation by 4 , the number of cores (for Intel Xeon Processor E3-1585L v5).

- 4 Cores * 3 GHz * 8-element vector * 2 ops in an FMA = 192 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.

5.1. Optimizing Part One [30 points]

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. If you attended the course Design of Parallel and High Performance Computing by Prof. Torsten Hoefler and Prof. Markus Püschel in Fall 2022 you might write a register-blocked kernel, either by writing an inner-level fixed-size matrix multiply and hoping (and maybe checking) that the compiler inlines it, writing AVX intrinsics, or even writing inline assembly instructions.
- Compare performance of your implementation to the Intel MKL by compiling and running the driver program and visualizing the performance results.

We recommend you look through the reference material to guide your optimization process, or use all the lecture notes from the previous ETH courses on HPC e.g. Design of Parallel and High Performance Computing - HS 2022 or High Performance Computing for Science and Engineering (HPCSE I) - HS 2022

5.2. Optimizing Part Two [10 points]

For part two, implement a multithreaded code using OpenMP. OpenMP allows you to add compiler pragmas which signify to the compiler that a loop can be executed in parallel. For more information on optimizing your code using OpenMP, we recommend reviewing the ETH lecture notes on shared memory programming as well as looking through the OpenMP-related resources in the references below [7].

Additional notes and submission details

Submit **all 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 a detailed Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to Moodle .

- Your submission should be a zip or tar archive, formatted like project_number_lastname_firstname.zip / .tgz. It must contain:
  - dgemm-blocked.c, a C-language source file containing your implementation of the routine:
    void square_dgemm(int, double*, double*, double*);
  - Makefile. 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 (report) 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 archive file through Moodle .
- 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 Euler cluster cluster and on your local machine; and
  - how the performance changed when running your optimized code on a different machine. For this, you may run your implementation on your laptop.

Your grade will mostly depend on two factors:

- performance sustained by your codes on the Euler 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 (`project1-info.pdf`). Please always use the project template which is available on the Moodle webpage for your submission.

References

[1] Getting started with the Euler cluster https://scicomp.ethz.ch/wiki/Euler

[2] Using the batch system on the Euler cluster `https://scicomp.ethz.ch/wiki/Using_the_batch_system`

[3] Intel C++ Compiler Classic Developer Guide and Reference https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-automatic-vectorization

[4] Intel Optimized Math Library for Numerical Computing – https://www.intel.com/oneapi/

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

[6] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw., 34(3):12:1-12:25, May 2008. https://dl.acm.org/doi/10.1145/1356052.1356053

[7] OpenMP Tutorial - LLNL Computation: https://hpc-tutorials.llnl.gov/openmp/