TÉCNICO LISBOA

Assignment 2

- Tutorial -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

Overview

This assignment is intended to help you develop an understanding of the two primary forms of parallel execution present in a modern multi-core CPU:

1. Parallel execution using multiple cores (You'll see effects of Intel Hyper-threading as well.)
2. SIMD execution within a single processing core
3. Roofline: CPU architecture and application profiling and analysis

You will also gain experience measuring and reasoning about the performance of parallel programs (a challenging, but important, skill you will use throughout this class). This assignment involves only a small amount of programming, but a lot of analysis!

Deadlines and Submission

NO SUBMISSIONS! This work serves as a tutorial and preparation – **NO EVALUATION**. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

To get started:

1. The assignment starter code is available on the course webpage (section: Labs). `wget` it, `unzip` it and start having fun with it.

2. Intel SPMD Program Compiler (ISPC) is needed to compile many of the programs used in this assignment. It is already installed in our LSD2 machines, but you need to "enable it" by running the following command:

   ```
   source /opt/intel/oneapi/ispc/latest/env/vars.sh
   ```

   NOTE: If you want to use your own machine, you will first need to install the ISPC. You can find all the info here: http://ispc.github.io/, with the binaries available on this downloads page[1].

1. The CARM Tool (for the roofline analysis part), should be downloaded to your local machine. You can also download it on the lab machines to conduct your own analysis **(not mandatory).**

   ```
   git clone https://github.com/champ-hub/carm-roofline.git
   ```

[1] https://ispc.github.io/downloads.html

Program 1: Parallel Fractal Generation Using ISPC

Like the assignment from Lab1A, Program 1 in Lab2 computes a mandelbrot fractal image, but it achieves even greater speedups by utilizing both the CPU's four cores and the SIMD execution units within each core.

In Lab1A, you parallelized image generation by creating one thread for each processing core in the system. Then, you assigned parts of the computation to each of these concurrently executing threads. (Since threads were one-to-one with processing cores in Lab1A, you effectively assigned work explicitly to cores.) Instead of specifying a specific mapping of computations to concurrently executing threads, Program 1 in Lab2 uses ISPC language constructs to describe *independent computations*. These computations may be executed in parallel without violating program correctness (and indeed they will!). In the case of the Mandelbrot image, computing the value of each pixel is an independent computation. With this information, the ISPC compiler and runtime system take on the responsibility of generating a program that utilizes the CPU's collection of parallel execution resources as efficiently as possible.

You will make a simple fix to Program 1 which is written in a combination of C++ and ISPC (the error causes a performance problem, not a correctness one). With the correct fix, you should observe performance that is over 100 times greater than that of the original sequential Mandelbrot implementation from `mandelbrotSerial()`.

Program 1, Part 1. ISPC Basics

Before proceeding, you are encouraged to familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at https://ispc.github.io/example.html. The example program in the walkthrough is almost exactly the same as this program's implementation of `mandelbrot_ispc()` in `mandelbrot.ispc`. In the assignment code, we have changed the bounds of the `foreach` loop to yield a more straightforward implementation.

What you need to do:

1. Compile and run the program `mandelbrot ispc`. **The ISPC compiler is currently configured to emit 16-wide AVX512 vector instructions.** What is the maximum speedup you expect given what you know about the CPU you are running the code on? Why might the number you observe be less than this ideal? (Hint: Consider the characteristics of the computation you are performing. Try to pinpoint the parts of the image that present challenges for SIMD execution. Comparing the performance of rendering views 1 and 2 of the Mandelbrot set may help confirm your hypothesis.)

We remind you that for the code described in this subsection, the ISPC compiler maps gangs of program instances to SIMD instructions executed on a single core. This parallelization scheme differs from that of Lab1A, where speedup was achieved by running threads on multiple cores.

Program 1, Part 2: ISPC Tasks

ISPCs SPMD execution model and mechanisms like `foreach` facilitate the creation of programs that utilize SIMD processing. The language also provides an additional mechanism utilizing multiple cores in an ISPC computation. This mechanism is launching *ISPC tasks*.

See the `launch[2]` command in the function `mandelbrot_ispc_withtasks`. This command launches two tasks. Each task defines a computation that will be executed by a gang of ISPC program instances. As given by the function `mandelbrot_ispc_task`, each task computes a region of the final image. Similar to how the `foreach` construct defines loop iterations that can be carried out in any order (and in parallel) by ISPC program instances, the tasks created by this launch operation can be processed in any order (and in parallel on different CPU cores).

What you need to do:

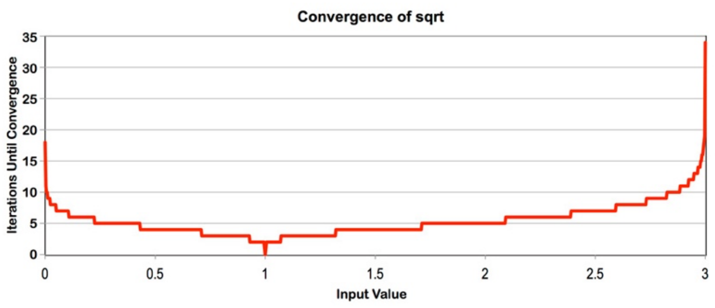
1. Run `mandelbrot_ispc` with the parameter `--tasks`. What speedup do you observe on view 1? What is the speedup over the version of mandelbrot_ispc that does not partition that computation into tasks?
2. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by changing the number of tasks the code creates. By only changing code in the function `mandelbrot_ispc_withtasks()`, you should be able to achieve performance that exceeds the sequential version of the code by over 100 times! How did you determine how many tasks to create? Why does the number you chose work best?

Program 2: Iterative sqrt

Program 2 is an ISPC program that computes the square root of 20 million random numbers between 0 and 3. It uses a fast, iterative implementation of square root that uses Newton's method to solve the equation $1/x^2 - s = 0$. This gives a value $x \approx \sqrt{1/s}$. Multiplying x by s gives an approximation to $\sqrt{s}$.

The value 1.0 is used as the initial guess in this implementation. The graph below shows the number of iterations required for sqrt to converge to an accurate solution for values in the (0-3) range. (The implementation does not converge for inputs outside this range). Notice that the speed of convergence depends on the accuracy of the initial guess.

Note: This problem is a review to double-check your understanding, as it covers similar concepts as program 1.



What you need to do:

1. Build and run `sqrt`. Report the ISPC implementation speedup for single CPU core (no tasks) and when using all cores (with tasks). What is the speedup due to SIMD parallelization? What is the speedup due to multi-core parallelization?

2. Modify the contents of the array values to improve the relative speedup of the ISPC implementations. Describe a very-good-case input that **maximizes speedup over the sequential version of the code** and report the resulting speedup achieved (for both the with- and without-tasks ISPC implementations). Does your modification improve SIMD speedup? Does it improve multi-core speedup (i.e., the benefit of moving from ISPC without-tasks to ISPC with tasks)? Please explain why.
3. Construct a very-bad-case input for sqrt that **minimizes speedup for ISPC with no tasks**. Describe this input, describe why you chose it, and report the resulting relative performance of the ISPC implementations. What is the reason for the loss in efficiency? **(keep in mind we are using the `--target=avx2` option for ISPC, which generates 8-wide SIMD instructions)**.

Program 3: BLAS saxpy

Program 3 is an implementation of the `saxpy` routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems. `saxpy` computes the simple operation $\mathrm{result} = \mathrm{scale} * \mathrm{X} + \mathrm{Y}$, where X, Y, and result are vectors of N elements (in Program 3, N = 20 million) and scale is a scalar. Note that `saxpy` performs two math operations (one multiply, one add) for every three elements used. `saxpy` is a *trivially parallelizable computation* and features predictable, regular data access and predictable execution cost.

What you need to do:

1. Compile and run `saxpy`. The program will report the performance of ISPC (without tasks) and ISPC (with tasks) implementations of `saxpy`. What speedup from using ISPC with tasks do you observe? Explain the performance of this program. Do you think it can be substantially improved? (For example, could you rewrite the code to achieve near linear speedup? Yes or No? Please justify your answer.)

Notes: Don't think too hard! We expect a simple answer, but the results from running this problem might trigger more questions in your head.

Cache-Aware Roofline Model :: The CARM Tool

Roofline Part 1: CPU Architecture Analysis

Each lab machine contains 11th Gen Intel i7-11700, which uses the RocketLake architecture. This CPU runs at 4.8 GHz turbo frequency (nominal 2.5GHz), has eight cores and supports up to two hardware threads per core (for a total of 16 "hyperthreads"). Each core contains one AVX512 and two AVX2/SSE/Scalar FP units as presented in the theoretical class, alongside four memory ports (although only three are capable of using AVX512).

We will start by confirming these architectural limits using the Cache-aware Roofline Model (CARM). For this, we will download the CARM Tool presented in class by cloning its repository from GitHub[2], you should also clone it to your local machine to use the GUI to visualize results, using the command:

[2] https://github.com/champ-hub/carm-roofline/tree/main

```
git clone https://github.com/champ-hub/carm-roofline.git
```

Then you can execute CARM benchmarks to test the different vector extensions and their corresponding architectural limits. However, to save time we also provide the results from these benchmarks in the `Results` folder inside the lab kit. To visualize the results in the GUI on your machine, simply copy the `Results` folder into your local installation of the CARM Tool. You can do this via `scp` on your machine using the command:

```
scp -r csph_<group_num>@193.136.147.<machine_number>:/path/to/results /path/to/tool
```

where the "*/path/to/results*" is the path to the results directory in the lab machine inside the CARM tool folder, and the "*/path/to/tool*" is the path on your local machine where the CARM tool is locally installed. You can also just place the `Results` folder from the `lab2_kit` in the CARM tool directory (instead of sending the `Results` folder from the lab machines to your local machine). It's up to you ☺

Note: You can still run the benchmarks yourself to check out some of the results, but do keep in mind they can take some time to finish (30+ minutes). Instructions on how to do it are provided below.

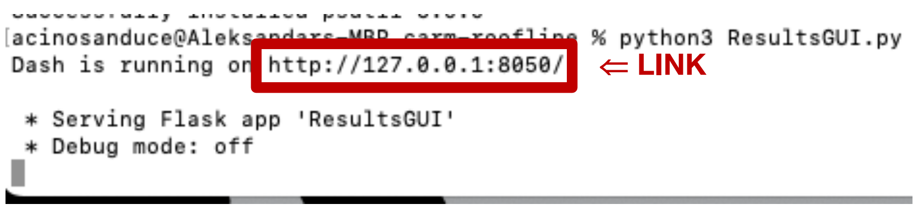
You can now visualize the results in your browser using the GUI of the tool. For this, run the command in your local machine (**not** in the LSD2 machine!):

```
python3 ResultsGUI.py
```

You might be missing some of the modules required by the GUI to function, in which case simply install them using pip3, which should cover all the dependencies under this command:

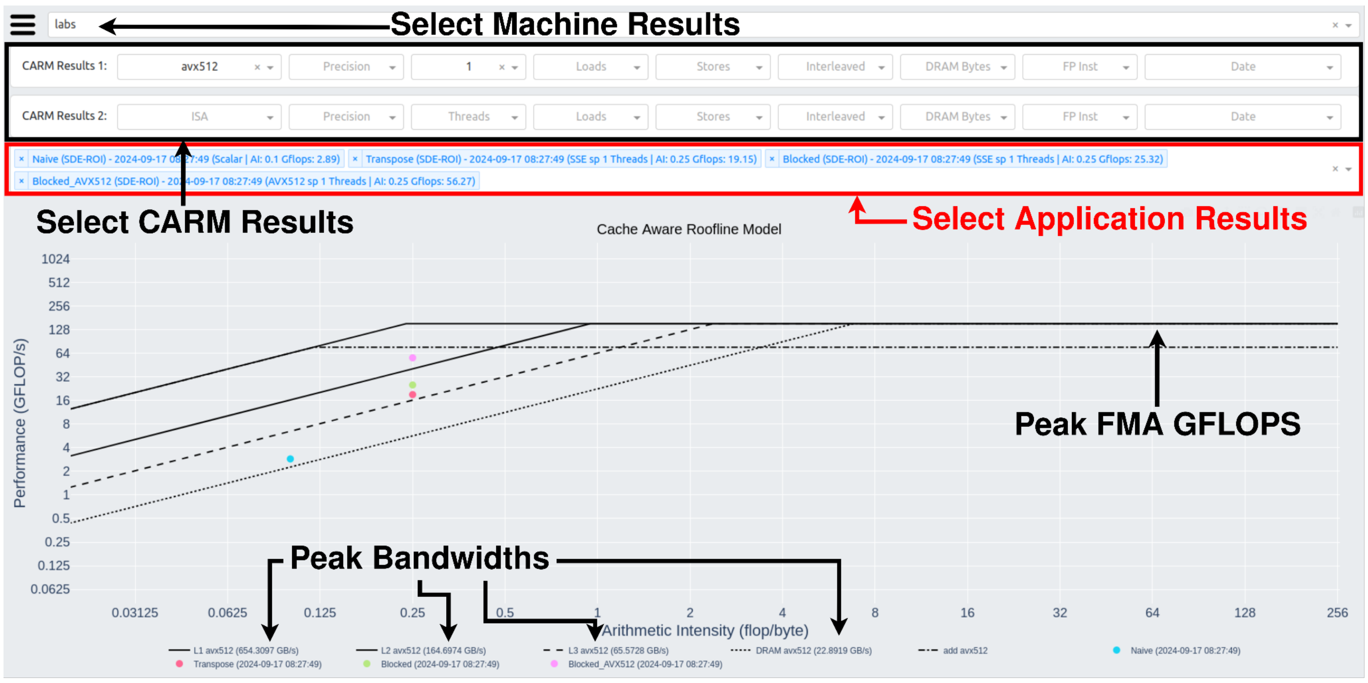
```
pip3 install plotly multiprocess dash dash_bootstrap_components diskcache "dash[diskache]" numpy pandas
```

Now, you should be able to start the GUI (`run python3 ResultsGUI.py`), which will give you a link to open in your browser (see figure below). Copy/paste this link in your browser and DO NOT close the terminal window!



```
acinosanduce@Aleksandars-MBP carm-roofline % python3 ResultsGUI.py
Dash is running on http://127.0.0.1:8050/
 * Serving Flask app 'ResultsGUI'
 * Debug mode: off
```

From here, simply select `lab2_machine` from the machine list and then you will be able to consult the various results presented via the dropdown menus as you see in the figure below. To start with, in the *CARM Results 1* part, select `avx512` in the *ISA* dropdown, and `1` in the *Threads* dropdown.


labs

CARM Results 1:	avx512	Precision	1	Loads	Stores	Interleaved	DRAM Bytes	FP Inst	Date
CARM Results 2:	ISA	Precision	Threads	Loads	Stores	Interleaved	DRAM Bytes	FP Inst	Date

Naive (SDE-ROI) - 2024-09-17 08:27:49 (Scalar | AI: 0.1 Gflops: 2.89)
Transpose (SDE-ROI) - 2024-09-17 08:27:49 (SSE sp 1 Threads | AI: 0.25 Gflops: 19.15)
Blocked (SDE-ROI) - 2024-09-17 08:27:49 (SSE sp 1 Threads | AI: 0.25 Gflops: 25.32)
Blocked_AVX512 (SDE-ROI) - 2024-09-17 08:27:49 (AVX512 sp 1 Threads | AI: 0.25 Gflops: 56.27)

L1 avx512 (654.3097 GB/s)
L2 avx512 (164.6974 GB/s)
L3 avx512 (65.5728 GB/s)
DRAM avx512 (22.8919 GB/s)
add avx512
Naive (2024-09-17 08:27:49)
Transpose (2024-09-17 08:27:49)
Blocked (2024-09-17 08:27:49)
Blocked_AVX512 (2024-09-17 08:27:49)

In the main panel, you can see the roofline plot for the selected parameters (AVX-512 and 1 thread). As you already know from the theoretical classes, the horizontal (flat) roof depicts the peak compute performance in GFLOPS measured (for the selected parameters). Here we show two horizontal roofs: the topmost one corresponding to the Fused Multiply-Add (FMA) operations, and the dashed MUL/ADD roof below it. The diagonal slopes represent the maximum performance achievable for different levels of the memory hierarchy, where the topmost slope corresponds to the L1 roof, followed by the L2 and L3 roofs, and finally the DRAM roof (see dotted lines). In the CARM tool, these roofs are constructed by running specific micro-benchmarks created to fully explore the capabilities of architecture's compute and memory resources, i.e., the CARM micro-benchmarks as referred above.

The characteristic values obtained from this testing are reported in the legend below the roofline plot, e.g., you can notice in the figure above that the peak measured L1 bandwidth is ~654.3Gb/s, while if you hover over the horizontal roof you can read that the peak measured AVX512 FP performance is ~153.7 GFLOPS for FMAs. In the theoretical classes, we were deriving the single-core maximum theoretical performance of this architecture, as well as the maximum theoretical L1 bandwidth. How do these values compare to the ones from the slides? Fill in the table below (consider a frequency of 4.8GHz).

Single-core Maximums (AVX512)	**FMA Performance [GFLOPS]**	**MUL/ADD Performance [GFLOPS]**	**L1 Bandwidth [GB/s]**
Theoretical			
Measured			

One would expect the measured GFLOPS to ideally match the throughput of the available FP units (per core on the CPU). To confirm this, we can convert the GFLOPS measurement to Instructions per Cycle (IPC) by dividing the GFLOPS by the number of FP operations in the corresponding vector extension used and then dividing again by the frequency of the CPU like so:

$$IPC_{compute} = \frac{GFLOPS \div FP\ Operations}{Frequency}.$$

So, if the measured GFLOPS corresponds to the maximum peak, the calculated IPC should match the number of available FP units in the CPU. We can also apply a similar procedure for memory, to obtain the corresponding memory IPC, by consulting the peak bandwidth measured in GB/s, dividing it by the amount of bytes moved by each instruction, and scaling with the frequency, such that:

$$IPC_{memory} = \frac{GB/s \div Bytes}{Frequency}.$$

In the table below, you can find the theoretical number of double-precision (DP, 64-bit) FP operations that can be delivered in a single clock cycle per FP unit, as well as the total amount of available FP units per data type per single core. Keep in mind that if you are referring to FMA GFLOPS, you need to multiply the FP operations by two. Also, if you are considering single-precision (SP, 32-bit) FP then you should again multiply the FP operations by two (except for Scalar), since you can have a double amount of elements in the same SIMD width. We also provide the number of memory bytes for each vector extension that can be delivered in a single clock cycle per each memory port, namely: scalar – 64 bits (8 bytes), SSE – 128 bits, AVX2 – 256 bits, and AVX512 – 512 bits (64 bytes). We also list the total amount of Load and Store ports available per singe core. For memory instructions, only the Scalar instructions move fewer bytes when single-precision is used, since it is 32-bits. As you expect, you should use these values when calculating the compute and memory IPCs.

FP Vector ISAs	**Scalar**	**SSE**	**AVX2**	**AVX512**
FP Operations / Unit	**1**	**2**	**4**	**8**
DP/SP FP Units	**2**	**2**	**2**	**1**
L1 Bytes per port	**SP: 4 | DP: 8**	**16**	**32**	**64**
Load and Store Ports	**4**	**4**	**4**	**3**

Fill in the table below with compute/memory IPCs for this architecture, both theoretical (using the values from the table above) and measured (using the values from the CARM tool). For the measured values, you will need to consult different roofline plots in the CARM tool, which correspond to different configurations in terms of the ISA used (Scalar, SSE, AVX2, AVX512), and the number of threads (cores: 1 or 8). In all cases, you should consider a frequency of 4.8GHz, and for the theoretical values do not forget to multiply the theoretical maximums per FP unit/port with the total amount of FP units/ports available.

FP Vector ISAs	Scalar	SSE	AVX2	AVX512
1 thread L1 IPC	Measured | Theoretical	|	|	|
1 thread FMA IPC	|	|	|	|
8 threads L1 IPC	|	|	|	|
8 threads FMA IPC	|	|	|	|

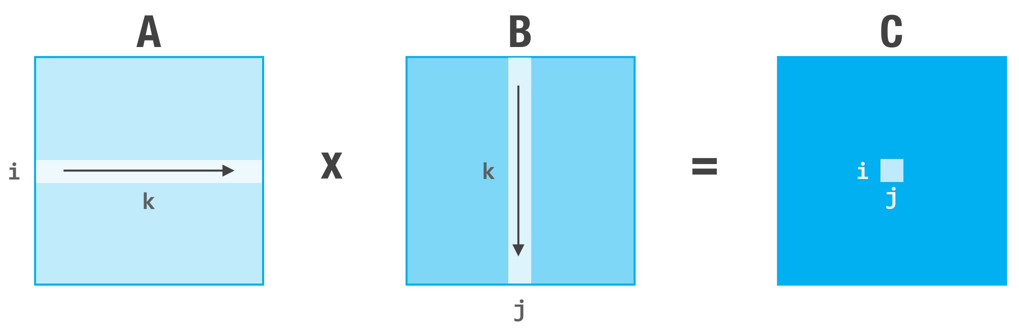
Do you observe any deviations? Can you explain them?

Roofline Part 2: Optimization Example SGEMM (tutorial)

For part 2, we will analyze a Single Precision General Matrix Multiply (SGEMM) implementation, i.e., AxB=C, in the CARM Tool to guide us through some optimization steps to improve its performance in the lab machines. We will use the CARM tool for running the application, which will be also automatically profiled with the Intel SDE for Dynamic Binary Instrumentation (see at the end how to run and profile apps in the CARM tool). However, to save time, we include the results of these analyses in the `Results` folder in the lab kit so you can quickly visualize them in the CARM Tool GUI on your machine. Feel free to conduct the analysis yourself **only if you have time**.

Optimization 1 (naïve): Open the `naive_SGEMM.cpp` file, which provides a straightforward implementation of the matrix multiplication (see *matrix_multiply* function that uses 3 "for" loops, also provided below). You can ignore the remaining part of the code, since it mainly serves to initialize the matrix and benchmarking.

```
for (i = 0; i < msize; i++) {
   for (j = 0; j < msize; j++) {
      for (k = 0; k < msize; k++) {
         c[i][j] += a[i][k] * b[k][j];
      }
   }
}
```



$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

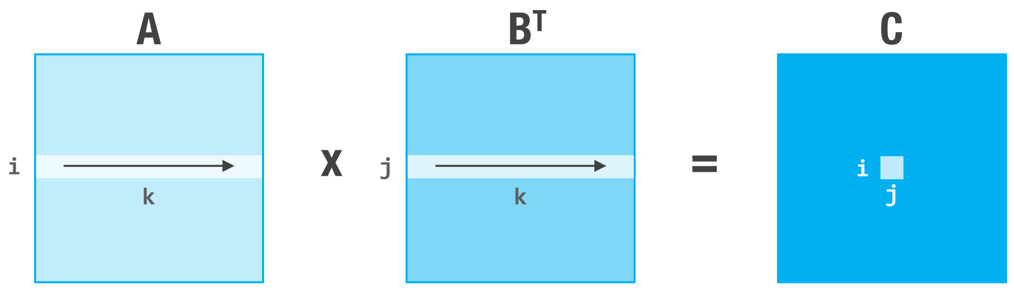
Complexity

- N^3 multiplications
- N^3 additions

In the CARM tool GUI, select the `naive_SGEMM` results from the "Select Application Results" panel, which should result in a single (blue) dot appearing in the roofline plot. Since our application is currently using Scalar[3] and single-precision elements on one thread, we should also select the Scalar single-precision single-thread CARM results. From this, we can observe our application is plotted between the L3 and DRAM lines and is clearly memory-bound according to the CARM. For this reason, we will try to improve the memory access pattern of the application in the next optimization step.

[3] This can be verified using DynamoRIO instead of SDE which produces an opcode breakdown based on ISAs used, however for the sake of time we did not include DynamoRIO analysis in this lab

Optimization 2 (transpose): In this step, we will try to improve the data access and its reuse by pre-transposing the matrix B (see code below).



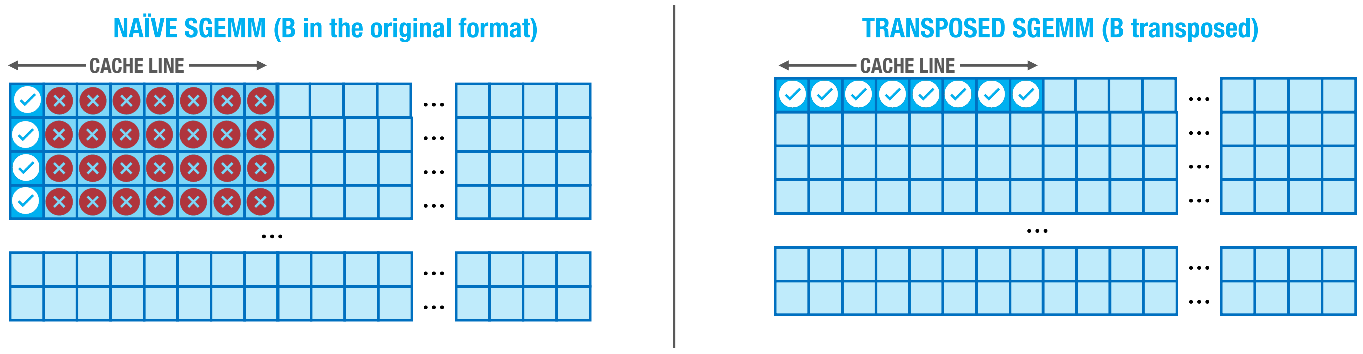
$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{j,k}^{T}$$

Complexity

- N^3 multiplications
- N^3 additions

The transposition allows us to access columns of matrix B row-wise, which greatly improves our access pattern and data reuse in the caches. In detail, to produce a single element of C, we need to multiply all elements of a row of A with all elements of a column of B. However, in the naïve implementation, when accessing an element in a column of B, there is little to no reuse of other elements fetched in the same cache line (in this architecture, the cache line size is 64 bytes, and we effectively use only 8 bytes per cache line, i.e., a single element of a column of B). By transposing the B matrix, we store the column elements row-wise, thus making the full (re-)use of all fetched B elements within a single cache line (see below). As such, it should lead to performance gains in a memory-bound application. This transposition is implemented in the `transpose_SGEMM.cpp` file, thus you should add the "*Transpose_GEMM*" point to the roofline plot in the CARM Tool GUI. As you can notice, the performance of this point now "breaks the roofline", which should be impossible!

```
for (i = 0; i < msize; i++) {
   for (j = 0; j < msize; j++) {
      t[i][j] = b[j][i];
   }
}
for (i = 0; i < msize; i++) {
   for (j = 0; j < msize; j++) {
      for (k = 0; k < msize; k++) {
         c[i][j] += a[i][k] * t[j][k];
      }
   }
}
```

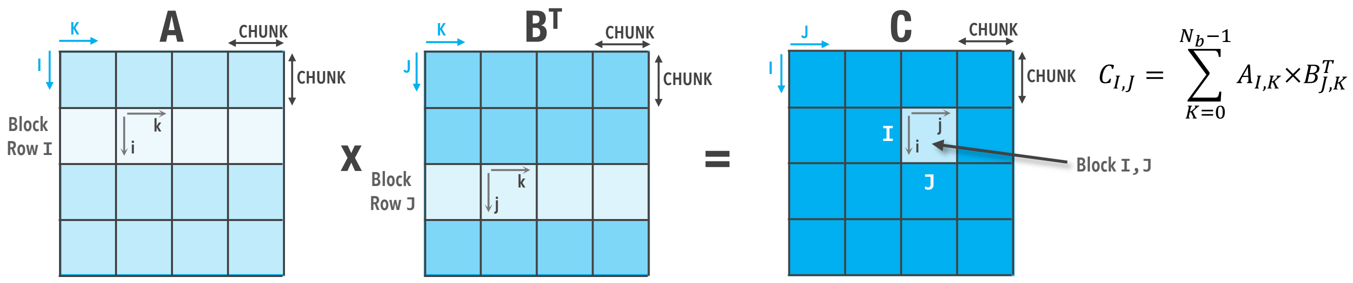

NAÏVE SGEMM (B in the original format)
CACHE LINE
TRANSPOSED SGEMM (B transposed)
CACHE LINE

Given the predictable access pattern, this new version of the code can also benefit from *automatic vectorization* by the compiler even with no new additional flags. The SSE[3] instructions are now being utilized, thus, for this reason, we should plot the results of this analysis against the SSE CARM results to get the correct analysis. You should select SSE, single-precision (sp), and single-threaded roofs.

We can observe that we are able to go above the L3 line with much higher performance (total GFLOPS) when compared to our previous implementation. This showcases the great performance increases that can be obtained by the vectorization of our code. Furthermore, since we also improved our access patterns thanks to the transposition, the application is now able to go above the L3 cache line.

However, the application is still under the L2 line which indicates that the application is not able to take full advantage of the L2 cache level. Since we are still in the memory-bound area, we should again pursue a memory-related optimization.

Optimization 3 (blocking): Our next optimization step considers partitioning the data into blocks (tiles), as presented in the code on the next page.



$$C_{I,J} = \sum_{K=0}^{N_b-1} A_{I,K} \times B^T_{J,K}$$

This strategy should allow the data to better fit into the L2 cache and provide better reuse, thus improving our performance further. This optimization is implemented in the blocked_SGEMM.cpp file, thus you should select the "*Blocked_SGEMM (SDE-ROI)*" in the CARM Tool GUI.

```
for (i = 0; i < msize; i++) {
   for (j = 0; j < msize; j++) {
      t[i][j] = b[j][i];
   }
}
for (ichunk = 0; ichunk < msize; ichunk += CHUNK_SIZE) {
   for (jchunk = 0; jchunk < msize; jchunk += CHUNK_SIZE) {
      for (i = 0; i < CHUNK_SIZE; i++) {
         ci = ichunk + i;
         for (j = 0; j < CHUNK_SIZE; j++) {
            cj = jchunk + j;
            for (k = 0; k < msize; k++) {
               c[ci][cj] += a[ci][k] * t[cj][k];
            }
         }
      }
   }
}
```

In this case, we also utilize the compiler flag "*-O3*" to further squeeze all possible performance gains from the compiler side. As we can see, the new results reach around 30% more GFLOPS when compared with the transpose implementation, and we are now able to surpass the L2 bandwidth line. Given that we are now limited by the compute (ADD/MUL) roof, it is worth pursuing optimizations that deal with the computations.

Optimization 4 (avx512): We can still further optimize by taking advantage of the AVX512 vector ISA extension, which is present in the Rocket Lake CPUs of the lab machines. For this, we can recompile the same example but include the flag "*-mavx512f*" which will allow the compiler to try to vectorize our code using AVX512 instructions instead of SSE. To observe the performance of this optimization step, you should select "*Blocked_AVX512_GEMM*", and use the AVX512 single-precision single-thread roofline in the CARM Tool GUI.

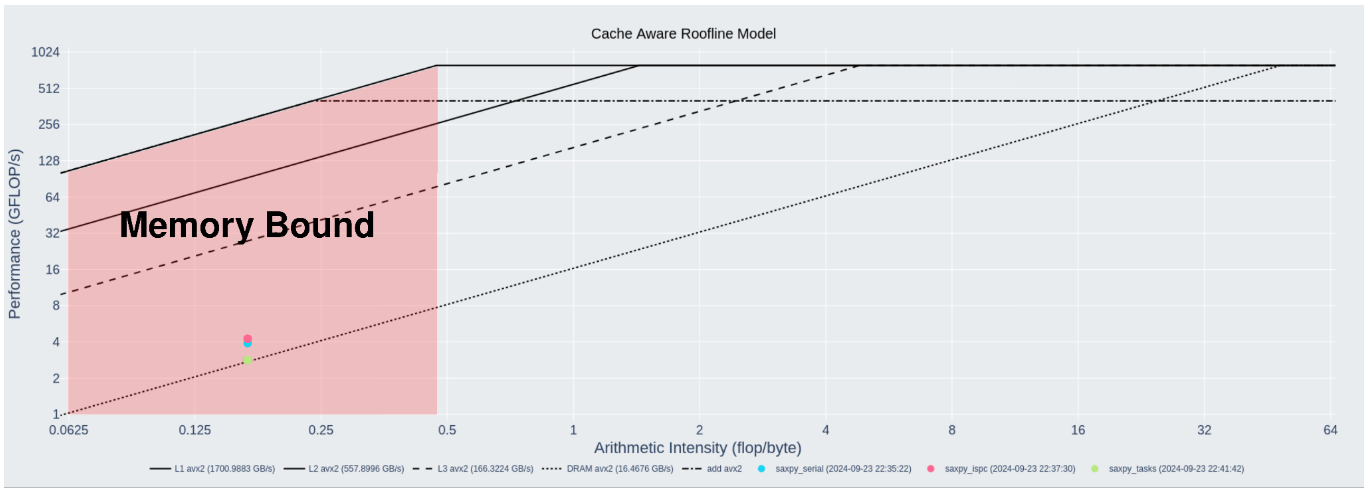
This new optimization is now able to achieve over twice the amount of GFLOPS when compared with the SSE implementation, surpassing the SSE CARM roofs (55.27 GFLOPS for AVX512 versus 35.35 GFLOPS for SSE). You can see those values by hovering with the mouse over the application point.

How much of the speedup have we achieved when compared to the naïve implementation?

Roofline Part 3: BLAS Saxpy (again!)

For part 3, we can also analyze Saxpy from Program 3 in the roofline model using the CARM Tool. For this, you can select the different Saxpy implementations that were profiled "*saxpy_serial*", "*saxpy_ispc*", and "*saxpy_tasks*" using the AVX2 8-threaded roof.

What differences do you notice between them? Do you think that this profiling can be helpful to prove your hypothesis from the problem 3 solution?



Roofline :: Optional part

Running your own benchmarks (totally optional, run only if you have a lot of free time)

You can still run the benchmarks yourself to check out some of the results, but do keep in mind they can take some time to finish (30+ minutes), so make sure you only do this **only if you have extra time (WARNING: it will take 30+ minutes to run this benchmark on lab machines!)**. The benchmarks can be started using the command:

```
python3 run.py -v 3 -p sp
```

The "*-v 3*" flag provides us with extra information about the results of each benchmark that is then used to construct the CARM. This will run the benchmarks for all available vector extensions (aka SIMD widths) on 1 thread using single-precision. You can also run the double-precision tests by changing the "*-p sp*" flag to "*-p dp*" or by deleting the "*-p*" flag entirely.

Compile and profile applications (totally optional, run only if you have a lot of free time)

Compilation: The `icpx` compiler is used to compile these codes, which you need to enable by running the command:

```
. /opt/intel/oneapi/setvars.sh
```

This implementation source codes is available in the `<file_name>.cpp` files (e.g., `naive_SGEMM.cpp`, `transpose_SGEMM.cpp` etc.*)*, and to compile it with the icpx compiler you should type:

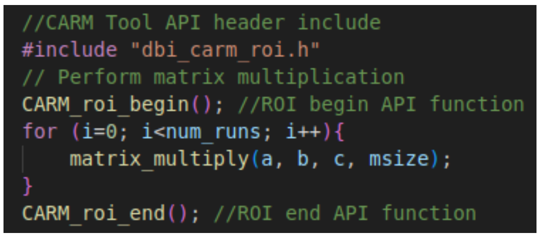
```
icpx <file_name>.cpp -o <exe_name>
```

For example, to compile *naïve_SGEMM*, you need to type `icpx naive_SGEMM.cpp -o naive_SGEMM`.

Analysis (Profiling): To execute the analysis using SDE you can use the command:

```
python3 DBI_AI_Calculator.py --sde --roi /opt/sde/sde-external-9.33.0-2024-01-07-lin
                    /path/to/executable executable_arguments
```

You can profile complete applications, as well as only the certain part of it, whoch is designated as Region of Interest (ROI). For this, we need to use the "*--roi*" flag and you must specify a ROI in your code using the CARM Tool API. You will also need to include the `dbi_carm_roi.h` header file in the same folder as the executable and in the code as shown in the figure on the right side.



```
//CARM Tool API header include
#include "dbi_carm_roi.h"
// Perform matrix multiplication
CARM_roi_begin(); //ROI begin API function
for (i=0; i<num_runs; i++){
    matrix_multiply(a, b, c, msize);
}
CARM_roi_end(); //ROI end API function
```

For all the optimization steps of matrix multiplication, the provided codes have the ROI will already be defined, the header file is included and in the same folder as the source codes, so no further action is required to conduct the analysis.