Michigan State University

CMSE/CSE 822 – Parallel Computing **Fall 2019**

Homework 2

Alexander Harnisch

---

1) Performance Modeling

a)

There is some missing information here and assumptions have to be made. However, it's not important since it only effects the numbers to plug in and get out, not the logic behind it.

The kernel performs 6 floating point operations (FLOP) in each iteration. Assuming no caching we have seven floating point number reading and one writing operation for each loop iterations. However, we can assume that the same variables are kept in the registers in each loop iterations, or at least in fast memory. Which means the entire y and z arrays both only have to be loaded once. Assuming single precision with 4 byte per float that translates to 12 byte of memory access and an arithmetic intensity I of

$$I = \frac{6\,\text{FLOP}}{12\,\text{byte}} = \frac{1}{2}\,\frac{\text{FLOP}}{\text{byte}}\,. \tag{1}$$

b)

In a simple roofline model for some I the critical peak performance π_{crit} is given by βI, where β is the peak memory bandwidth. So for $I = 0.5\,\frac{\text{FLOP}}{\text{byte}}$ from a) we get:

$$\pi_{\text{crit}} = 30\,\frac{\text{GB}}{\text{s}} \cdot \frac{1}{2}\,\frac{\text{FLOP}}{\text{byte}} = 15\,\frac{\text{GFLOP}}{\text{s}}\,. \tag{2}$$

So in case the processor's peak performance is greater than 15 GFLOP/s the kernel is compute bound, otherwise memory bound.

c)

A simple roofline model plot is given by Figure 1. The performance for an arithmetic intensity of $I = 0.5\,\mathrm{FLOP/byte}$ is $15\,\mathrm{GFLOP/s}$.

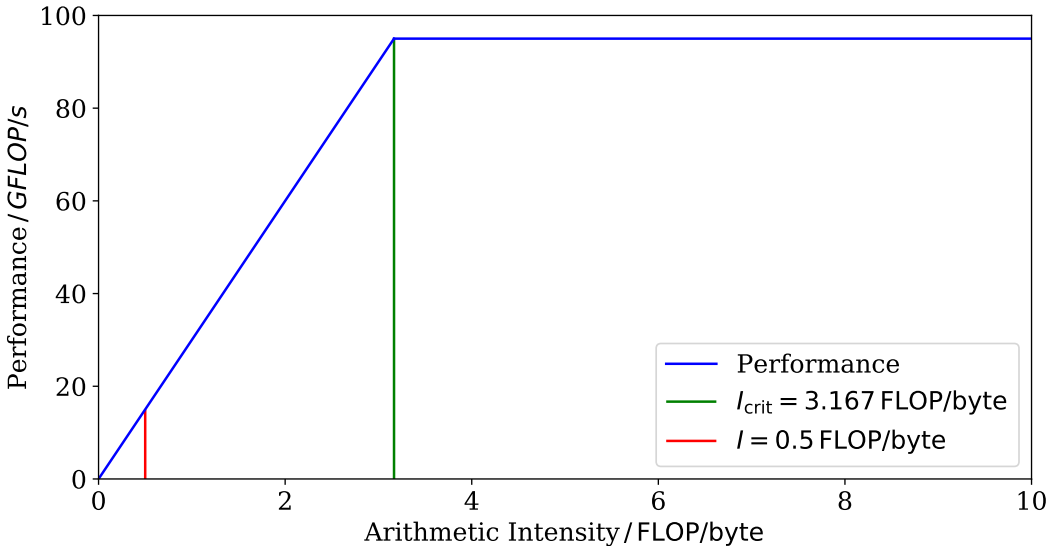

Figure 1: Simple roofline model.

2) Cache optimization: Matrix Vector Multiplication

General Comment: I am very interested in fully understanding what is going on here. However, I simply don't have the time to properly think about all the plots I've generated. I keep exact track of how I spend my productive time, partly to point out to professors when their homework takes up more time than it is supposed to. I spent 28 hours in total on this homework (and almost all of it on the second part). This is simply too much. Yes I did a few things that are definitely beyond of what was asked for, but without it it seemed very hard to get any reliable data. This was also partly due to the first published version of the homework being very confusing and it being updated after I already waisted 10 hours of my time on it. The available time for the homework for this class in a graduate student's weekly schedule (assuming a 40 hour work week) is 3 hours and 40 minutes on average over the entire duration of the class. In other words: This week's assignment was about a factor of 7.7 longer than what it is supposed to. MSU claims to care about student's mental health, so please act like it and keep that in mind for future assignments.

a) Implementation

See code in repository.

b) and c) Performance Analysis and Cache Performance Measurement

For all of the testing in this subsection, I wrote scripts for fully automated job submission and data collection in BASH and PYTHON. Using this system, I tested 4432 different sets of parameters (N, M and B). For each run the relative speedup as well as cache misses for L1, L2 and L3 were collected. All of the results are summarized in the following plots. The raw data can be found in the file *data.txt* in my repository, outside of the submission directory.

Figure 2 shows the speedup for all tested matrices in dependence of the blocking factor B. Each data point is the average speedup for all matrices with the corresponding B value. This was the first data I looked at, also splitting it down by matrix sizes, to get a general feel for the data and choose a reasonable blocking factor for tests with fixed blocking factors and varied matrix shape. $B = 200$ was chosen for that, also because it lays within the window of highest speedup in Figure 3.

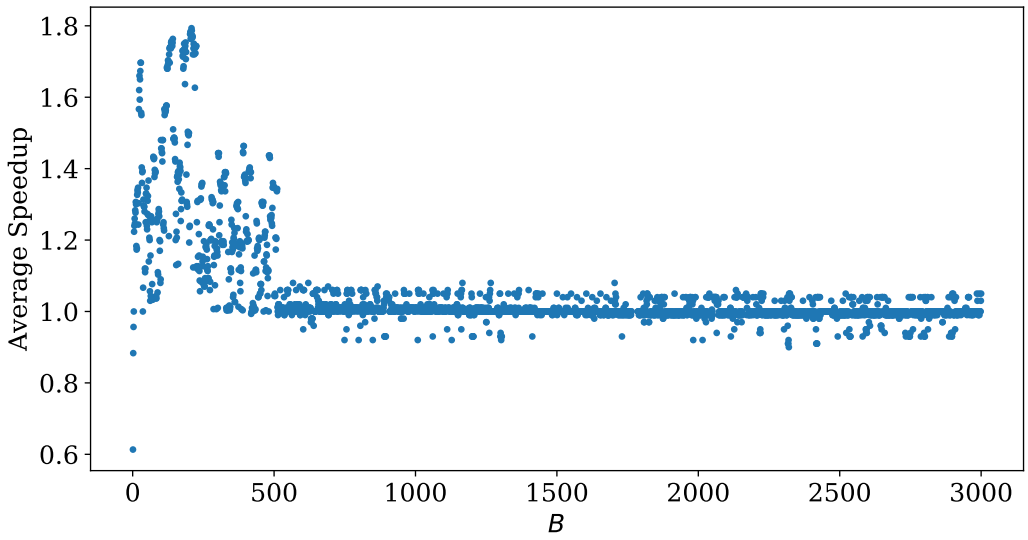

Figure 2: Average speedup for all tested matrices in dependence of the blocking factor B.

Figure 3 is probably the most insightful one. It shows the achieved speedup and ratio of cache misses for all blocking factors between 0 and 512 for a constant matrix shape of $N = M = 10000$. This plot has many interesting features, many of which I do not understand (for example: there is a periodicity in L1 misses for $B > 285$ that also periodically effects L3 misses and speedup).

We can clearly see a correlation between L1 cache miss rate and speedup. For $B > 228$ the L1 cache seems to be too small for the blocking and it starts to become less and less effective for larger blocking factors. We can see how the speedup slowly declines back to one as the L1 cache miss rate increases and the ratio approaches one, in which case there is practically no difference between the naive and optimized implementation any more as all of the cached values have been overwritten once they are needed again.

However, for some reason the L2 miss rate still seems to be significantly lower for the optimized method without any significant speedup, which I do not fully understand. My best guess is that the L2 latency is just too high for the achieved changes in L2 miss rate to have any significant effect on speedup in this scenario. We can find more evidence

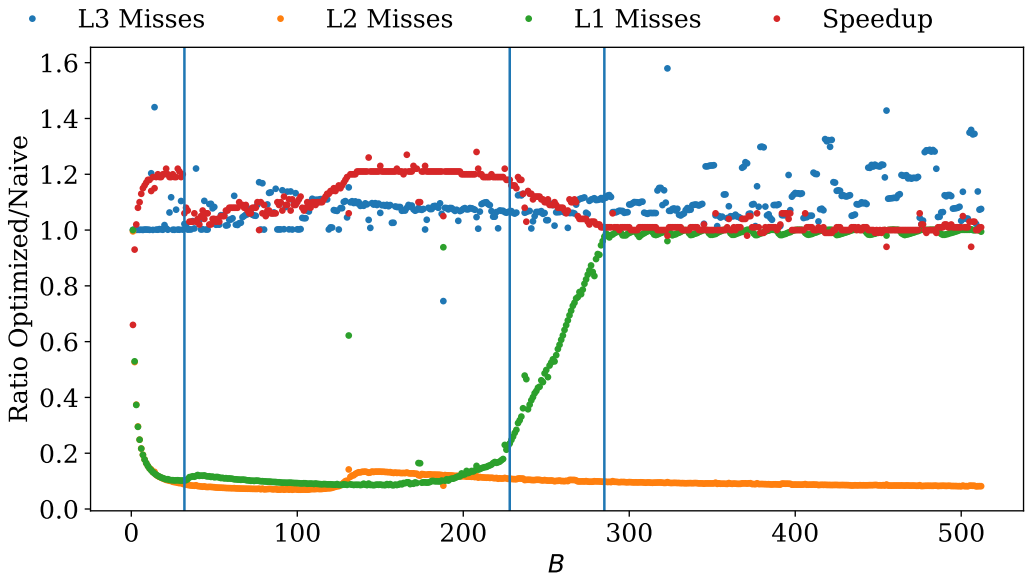

Figure 3: Results for a fixed matrix size of $N = M = 10000$ for different blocking factors B. For reference, the plot includes blue vertical lines at $B = 32$, $B = 228$ and $B = 285$.

for this hypothesis in my other plots, which also do not show a significant correlation between L2 miss rate and speedup (see Figure 7 and Figure 9).

It is also unclear why there is no observable speedup for $32 < B < 125$ despite the L1 and L2 miss rate being consistently low. Maybe it has to do with cache latency and overhead due to the added loops. The initial rise in speedup is cut off by a drop at exactly $B = 32$, which is probably related to a lower cache latency with a cache block size of 32. But then there should be a similar effect for $B = 64$, which there is not.

To start seeing L2 cache misses we have to test larger blocking factors. For that purpose, Figure 4 is an extension of Figure 3 with blocking factors up to $B = 3000$. We can clearly make out the rising edge of L2 cache misses. However, as mentioned before L2 cache misses have no effect on performance in this setting. The middle of the rising edge can be found at $B = 2083$.

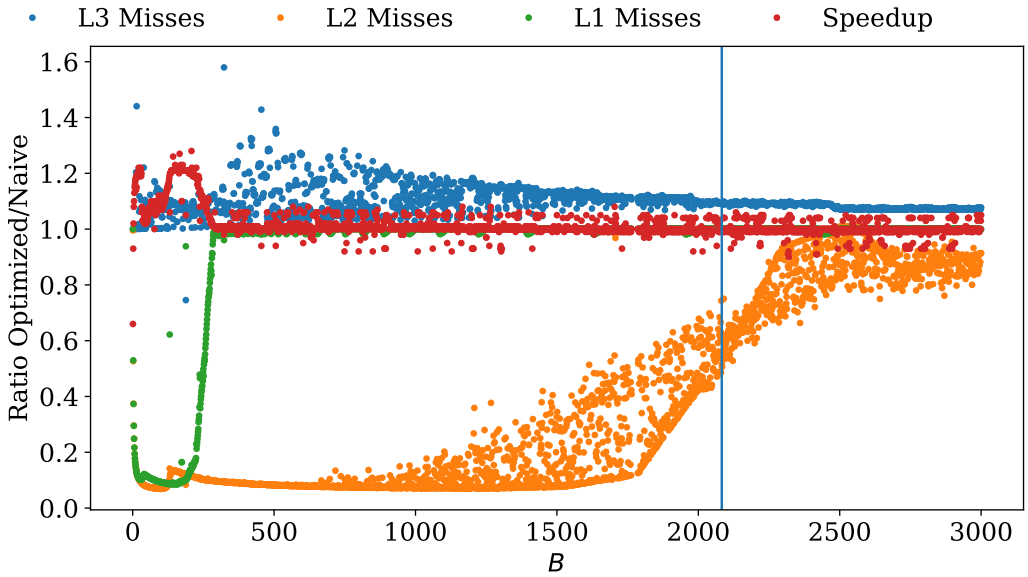

Figure 4: Results for a fixed matrix size of $N = M = 10000$ for different blocking factors B. For reference, the plot includes a blue vertical line at $B = 2083$.

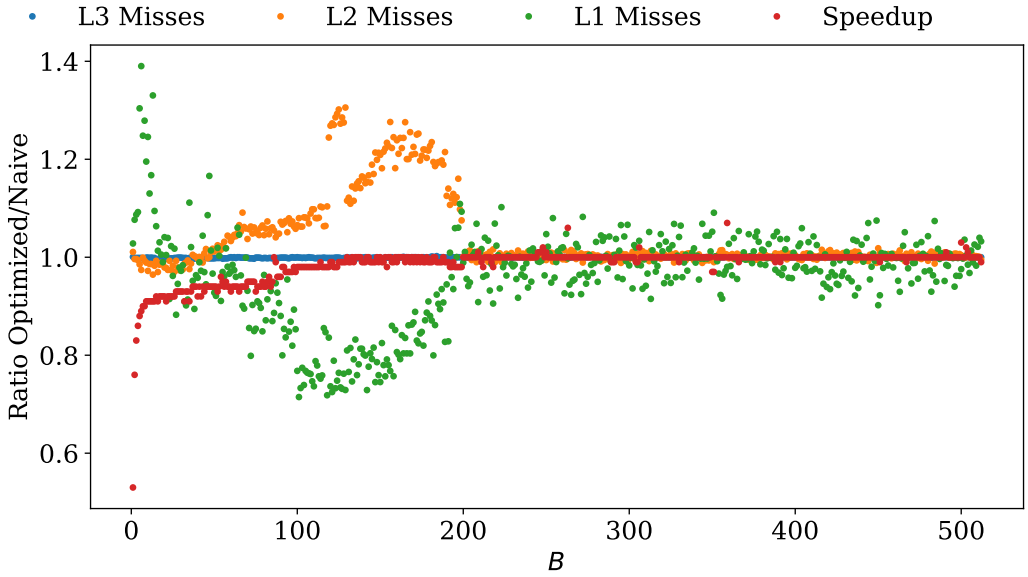

Figure 5: Results for a narrow matrix with $N = 100000$ and $M = 200$ for different blocking factors B.

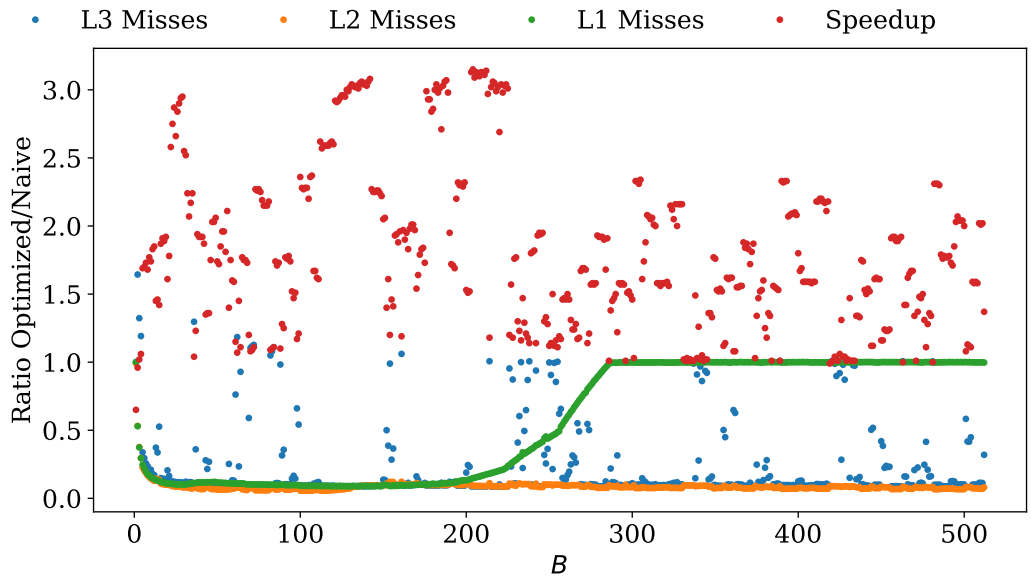

Figure 6: Results for a wide matrix with $M = 100000$ and $N = 200$ for different blocking factors B.

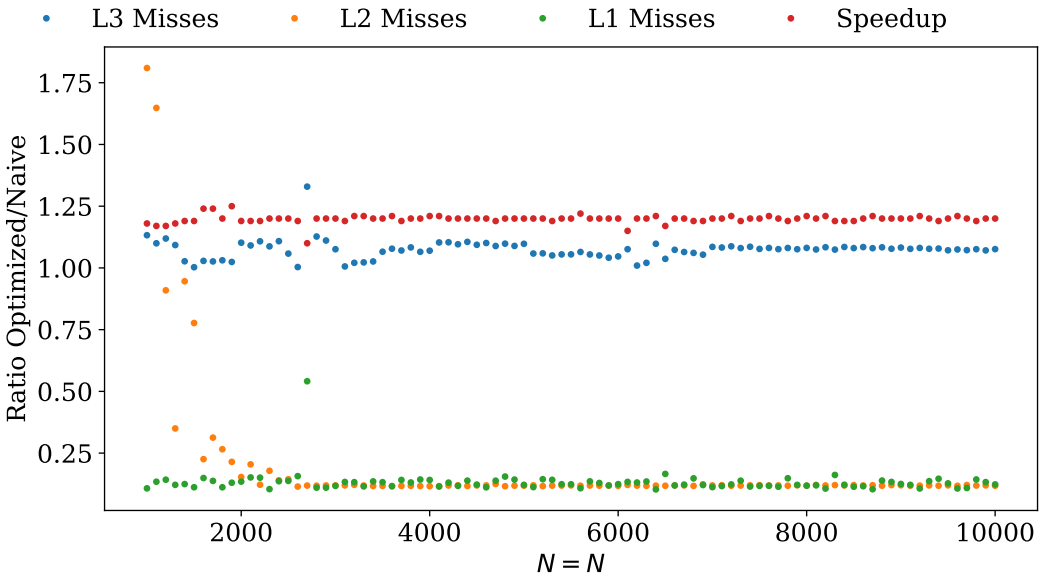

Figure 7: Results for a fixed blocking factor of $B = 200$ for different square matrices ($N = m$).

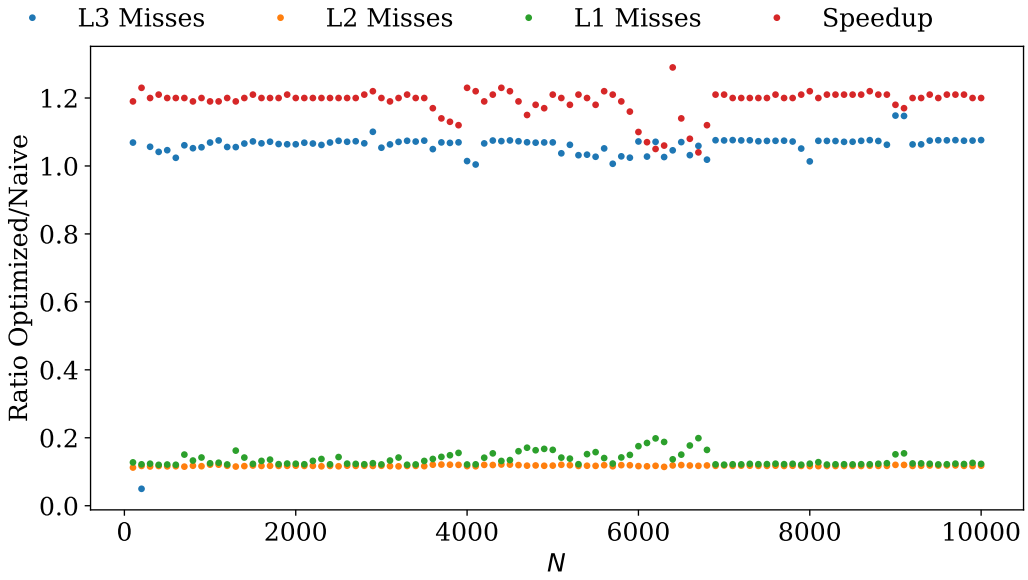

Figure 8: Results for a fixed blocking factor of $B = 200$ and fixed $M = 10000$ for different N.

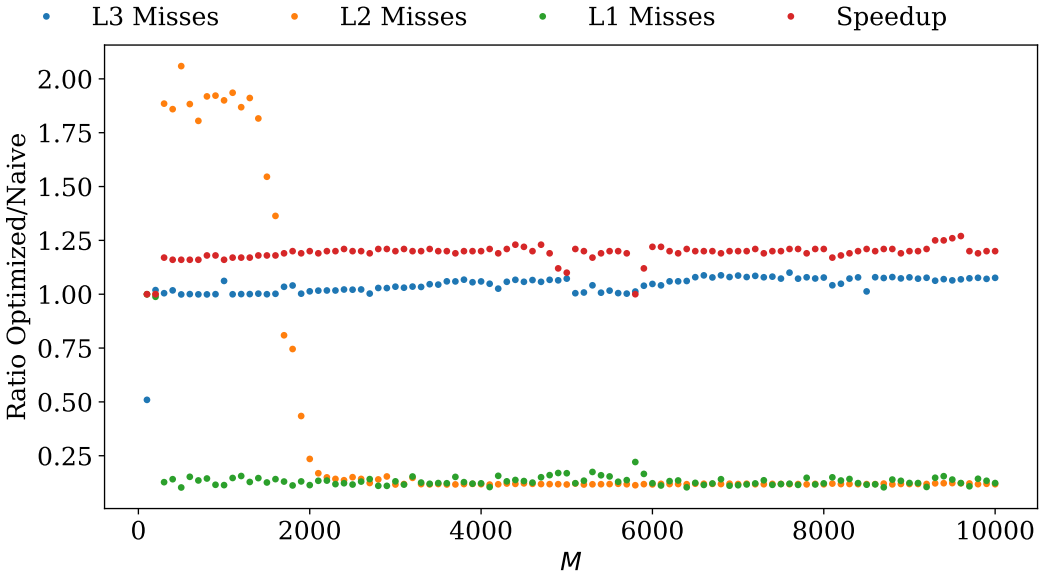

Figure 9: Results for a fixed blocking factor of $B = 200$ and fixed $N = 10000$ for different M.

The data visualized in Figure 7, Figure 8 and Figure 9 leads to conclude that for sufficiently large matrices and a reasonable blocking factor, there is no correlation between matrix shape or matrix size and performance. The performance of the optimized implementation is consistently and significantly better than the naive implementation.

However, we can observe significant differences for the two extreme cases tested which are visualized in Figure 5 and Figure 6. For the extreme case of a very narrow matrix shown in Figure 5 the optimized implementation underperforms (probably due to added overhead by the additional loops) until the block size approaches the width of the matrix, in which case both implementations are equivalent (also for block sizes larger than the width). We observe interesting behaviour in L1 and L2 miss rates for blocking factors smaller than the matrix width: The L2 cache miss rate increases roughly by the same factor the L1 miss rate decreases. This might be due to suboptimal branch prediction for this scenario.

The special case of an extremely wide matrix demonstrated by Figure 6 shows extreme performance of the optimized implementation with speedups of over a factor of 3 for some cases. The L1 and L2 miss rate behavior is the same observed for sufficiently large square matrices. Additionally, here we also observe significantly lower L3 miss rates which seem to be the cause for the large speedup factors as as the L3 miss rate shows a negative correlation with speedup. However, I can not make out a clear pattern, the behavior is very irregular. This is most likely due to low statistics. The matrix is comparably small and randomly initialized. This test should probably be repeated several times to draw conclusions from the averages instead of single runs. The general trend is easy to explain though: The optimized implementation works its way through the entire input matrix and input vector block in blocks form left to right, never requiring the first values again while the naive method works at one row of the input matrix at a time. In the case of very few and long rows every row reaches the L3 cache capacity which explains the observed behavior.

Therefore, in the special case of extremely rectangular matrices, it might make sense to always transpose very narrow matrices first to always work with extremely wide matrices instead to achieve the performance advantage of the optimized implementation.

The assignment also asks to make an estimation on the relative L1 and L2 latencies based on the performance and cache misses data. I do not observe a correlation between L2 misses and performance in the data I have collected, but I do find a strong correlation between L1 cache misses and performance. Therefore I can not give a precise estimate on the relative latencies, I can only say that the L1 latency is negligible when compared to the L2 latency, which means the L2 latency is probably at least one order of magnitude larger than the L1 latency.

d) Inference about the Memory Hierarchy

The CPUs used here are Xeon E5-2680 v4 by Intel. It has 14 cores. Each core has 448 KiB of L1 cache divided into 14 blocks of 32 KiB. The L2 cache consists of 14 256 KiB blocks adding up to a total of 3.5 MiB. The L3 cache is ten times that large, 35 MiB consisting of 14 2.5 MiB blocks.

To make more sense of the data and use it to learn about the memory hierarchy, we need to look at the optimized implementation and think about how many double precision numbers are repeatedly used (and therefore potentially kept in fast memory).

The three most inner loops with loop variables k, j and i loop over the respective en-

tries within a block for blocking factor B. The two outer loops move the block. Considering a fixed block, so only the three most inner loops, the algorithm repeatedly accesses 16 numbers in the output vector block, B^2 numbers in the input matrix and $16B$ numbers in the input vector block. That means ideally we would like all

$$N = B^2 + 16(B + 1) \tag{3}$$

numbers to be available in the fastest memory. Therefore, we should see a significant drop in speedup when the critical value for B is reached, at which the L1 cache can not hold all of these numbers anymore.

This is exactly what we observe. As discussed in the previous section, Figure 3 shows a significant drop in speedup for $B = 228$, which is where we start to see a significant increase in L1 cache misses. That means for $B = 228$ we reach the L1 cache capacity. Plugging that into (3) leads to an inferred cache capacity of $N = 55{,}648$ double precision floating point numbers. The double precision used here is 8 byte, therefore the inferred L1 cache size is roughly $55{,}648 \cdot 8\,\text{byte} = 445.184\,\text{KiB}$. This is in perfect agreement with the hardware specifications. For $B = 229$ we already exceed the L1 capacity with $N = 56{,}121$ which would require a cache size of 448.968 KiB.

To infer the L2 cache size we take the observed middle of the rising edge for L2 cache misses from Figure 4 at $B = 2083$. Plugging that into (3) results in $N = 4{,}372{,}233$ which means we can estimate the L2 cache size to be about 35 MiB, which again is in perfect agreement with the given hardware specifications.