ASL Assignment 4

Fabian Wüthrich

April 10, 2020

1. Cache Mechanics (40 pts)

   (a) **Cache Misses `calculate1`**

   The vector v has a size of 8448 bytes and doesn't fit in cache completely. This implies that for every row of A we have to load v into cache again.

   The elements `v[j]` and `A[i * m + j]` are 8448 bytes apart in memory so they are assigned to different sets in cache. Therefore, the block of `v` and the block of `A` will not interfere with each other. Every 16th iteration we have a miss for the first `v[j]` and `A[i * m + j]`. The second access to `v[j]` is not a miss, as the value was loaded before. In total, there are $\frac{2mn}{16} = \frac{mn}{8}$ misses and we access `v` and `A` $3nm$ times. The miss rate is then $\frac{1}{24} \approx 0.042$.

   Cache Misses `calculate2`

   In the inner loop `v[j]` get loaded once and is then reused n times. There is a potential conflict with `A[(i + n - 1) * m + j]` as this value is mapped to the same set. As we have a 2-way set associative cache with LRU replacement, this conflict is resolved by leaving `v[i]` in the first block (as it was used in the previous loop) and replace `A[(i + n - 1) * m + (j - 1)]` (placed in the second block in the previous iteration). This gives us a compulsory miss in every 16th iteration so we have $\frac{m}{16}$ misses for v.

   The values of A get mapped to different sets (each value is placed 4 sets apart from the set of the previous value) so there will be no conflicts between values of A. In every 16th iteration we have n compulsory misses followed by a series of cache hits. Thus, there are $\frac{nm}{16}$ misses for A which gives $\frac{m+mn}{16}$ total misses.

   The number of accesses is the same as above so we get a cache miss rate of

   $$\frac{m+nm}{16} \cdot \frac{1}{3mn} = \frac{n+1}{48n} = \frac{3}{128} \approx 0.023$$

   which is better than `calculate1`.

   (b) **Cache Misses `calculate1`**

   As n doesn't influence the cache access pattern of `calculate1`, the miss rate is still $\frac{1}{24}$.

   Cache Misses `calculate2`

   Due to the LRU policy v stays in cache as before so the miss rate is still $\frac{m}{16}$.

   For A we get a new conflict as `A[(i + 8) * m + j]` is mapped to the same set as `A[i * m + j]`. This conflict is again resolve by putting `A[(i + 8) * m + j]` in the other block of the set. In the next iteration we a hit for both `A[i * m + (j + 1)]` and `A[(i + 8)  * m + (j + 1)]` as they were loaded with the previous line.

   A problem occurs when we get to the set that contains v. In this case one block is already occupied by `v[j]` and we have to kick out the value of A stored in the other block. This causes two misses in every inner loop iteration plus the additional $n-2$ misses in every 16th outer loop iteration. Thus, we have $2m + (n-2)\frac{m}{16}$ misses for A.

   Adding everything together gives us $\frac{mn+31m}{16}$ cache misses and a cache miss rate of

   $$\frac{mn+31m}{16} \cdot \frac{1}{3mn} = \frac{n+31}{48n} = \frac{47}{768} \approx 0.061$$

(c) When $n = 16$ we observed a conflict miss in the set where v is stored. If we increase n further, we can see that we get more conflict misses (e.g. `A[0 * m + j]` and `A[16 * m + j]` get mapped to the same set which causes a conflict miss in the next outer loop iteration) and the miss rate increases.

Now if we set $n = 23$ we produce conflict misses for `A[0 * m + j]` ... `A[7 * m + j]` which get removed from cache. We have to load these values again in the next outer loop iteration and because of the LRU policy we overwrite `A[8 * m + j]` ... `A[15 * m + j]`.

The pattern continues like that so in the end we have only misses for A. Increasing n is not necessary as `A[7 * m + j]` falls into the same set as v, so was already replaced in a previous iteration of the inner loop. A smaller n gives a lower cache rate because there would be still values of A left in the cache, which gives a hit in the next outer loop iteration. Therefore, $n = 23$ is the minimum value that produces the highest miss rate.

(d) The problem in the previous subtask was that `calculate2` does not use the full cache block before it goes to the next row. Thus, the block is not in cache anymore if we want to access it in the next outer loop iteration.

We can fix this with blocking as seen in the lecture, so we traverse A in $n \times 16$ blocks. The following listing shows the improvement.

```c
void  calculate2(float *A, float* v, int m, int n) {
  for (int j = 0; j < m; j += 16) {
    for (int i = 0; i < n; ++i) {
      for (int k = 0; k < 16; ++k) {
        v[j + k] = max(v[j + k], A[i * m + j + k]);
      }
    }
  }
}
```

With this improvement we use the complete cache block in the inner most loop and then we do not use these values again, so we will not get any conflict misses as in the previous subtask. For v we get again the $\frac{m}{16}$ compulsory misses. For A we get a in every outer loop iteration one cache miss per row i.e. $\frac{m}{16} \cdot n$. Thus, we have $\frac{m+mn}{16}$ misses in total and a cache miss rate of

$$\frac{m + mn}{16} \cdot \frac{1}{3mn} = \frac{n+1}{48n} = \frac{1}{46} \approx 0.022$$

As n doesn't influence the cache miss rate of `calculate1`, the function has still a miss rate of 0.042 which is inferior to the improvement of `calculate2`.

2. Stride Access (20 pts)

(a) The function `calculate` accesses v twice with $stride = 4$. In the first iteration of the outer loop we get a hit in every second iteration of the inner loop (one cache block can hold 8 doubles). We cannot improve the number of hits in the first round by adjusting n because of the compulsory misses. Now we have to pick n such that every element of v is still in the cache if we access v in the second iteration and we have only hits. The whole cache can hold 512 double values so if we choose n to access more values we get conflicts and more misses in the second iteration of the outer loop. Thus, the maximum value is $n = 128$ which loads the value at index 508 as last element into the cache.

(b) As in the previous answer we cannot influence the misses in the first outer loop iteration by adjusting n. In order to generate more misses in the second iteration we fill the cache with 768 doubles which overrides the first 256 doubles of v in the cache. In the second iteration we have to load the first 256 doubles again and this will overwrite the next 256 doubles that where in cache (LRU policy). Then we load the second portion and this removes the last chunk of 256 doubles from the cache. Now we get a miss in every second iteration in the second round which is the lowest hit rate we can achieve. Therefore, the minimum value to get the lowest hit rate is $n = 192$.

(c) Now we have $stride = 32$ which forces a compulsory miss in every iteration in the first round. In order to create only hits in the second round (highest hit rate) we have to set $n = 16$ which access the value at index 480 and doesn't generate any conflicts with the previous sets.

For the lowest hit rate we set $n = 24$ which is the minimum value to load 768 doubles so we remove the values in a equally bad way as in the previous sub-task. Now we have only misses for both outer loop iterations which is the lowest hit rate we can achieve.

(d) With *stride* $= 16$ every two consecutive values we access are 128 bytes apart. This is too large to fit into a block of 64 bytes and we do not have spatial locality. Because of that we have just compulsory misses in the first round and no hit at all.

As $n = 40$ we load 640 doubles which is with 5120 bytes too large to fit in cache. The cache can hold 512 doubles so the first 128 doubles (`v[0]...v[127]`) get overwritten by the last 128 doubles (`v[512]...v[639]`) of v in the first round.

In the second iteration we get conflicts at the 16 sets in the beginning because the required values were overwritten in the first iteration. When we consider the LRU policy one can see that we have only misses for the first 16 sets because the values overwrite each other.

The last 16 sets of the cache generate now conflicts so we can reuse the 256 doubles which we placed there in the first iteration. With a stride of 16 we achieve 16 hits on these 256 doubles. We access v 80 times so we get a hit rate of $\frac{16}{80} \approx 0.2$.

For completeness I summarize the hit/miss pattern for the whole calculation. In the first outer loop iteration we have 40 compulsory misses. Then when we traverse v a second time we have 8 misses, 8 hits, 8 misses, 8 hits and again 8 misses in the end.

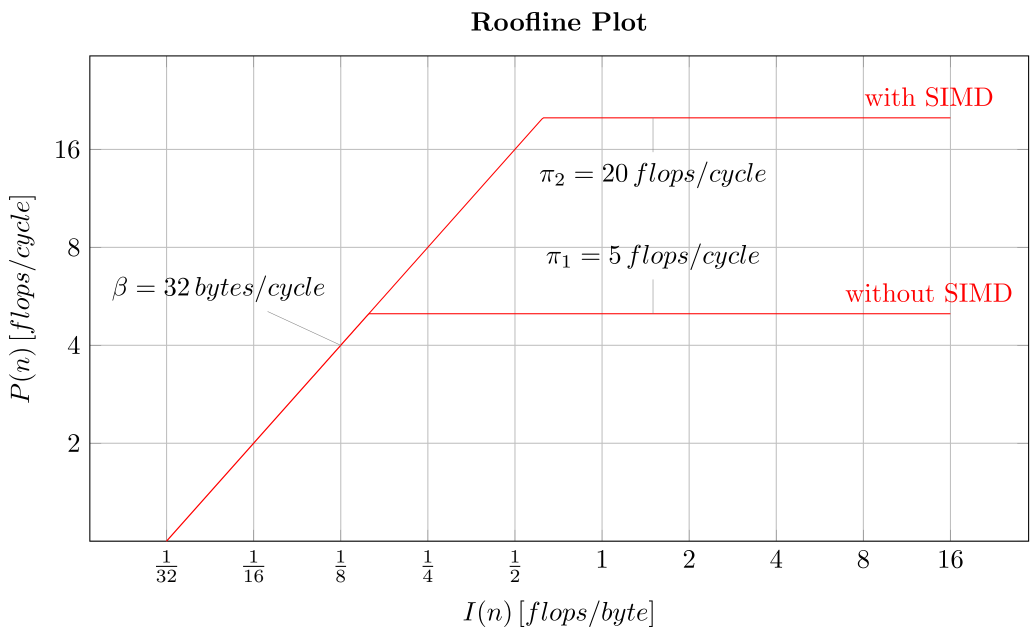
3. Rooflines (40 pts)

(a) Without SIMD instructions we can perform one FMA instruction on P0, one FMA instruction on P1 and one addition on P2. As an FMA instruction executes two floating point instructions per cycle we get a peak performance of $\pi_1 = 2 + 2 + 1 = 5\, flops/cycle$.

With SIMD instructions enabled we can use the previous instructions on vector of four doubles which gives a peak performance of $\pi_2 = 8 + 8 + 4 = 20\, flops/cycle$.

The given computer has a maximum read bandwidth of $\beta = 32\, bytes/cycle$ which is the same for the model with SIMD and the one without.

Using the peak performance and the maximum read bandwidth we can plot the following roofline models.



(b) For all three functions we consider only floating point operations and assume that index calculations do not influence the floating point units.

In `computation1` we just use floating point additions and have three ports to distribute them. As each port has a throughput of $1\, operation/cycle$ and a latency of $1\, cycle$ we can do one addition per cycle on each port. This gives us an upper performance bound of

$$P_1 \leq 3\, flops/cycle$$

In `computation1` we execute five flops per loop iteration so we have

$$W_1(n) = 5n\, flops$$

For the data movement we consider only reads and in each loop iteration we access three doubles (we ignore the constants as they can be hold in registers). As eight doubles fit into one cache block we have only two compulsory misses per loop iteration because `y[i+4]` is already in the cache. Therefore, the data movement is

$$Q_1(n) \geq 8 \cdot 2n = 16n\, bytes$$

Now we can calculate the operational intensity

$$I_1 \leq \frac{5n}{16n} = \frac{5}{16}\, flops/byte$$

In `computation2` we do exclusively multiplications but we have only two ports where we can schedule them. This gives a performance bound of

$$P_2 \leq 2\, flops/cycle$$

As we perform the same work and move the same amount of data as in `computation1`, the operational intensity does not change. Therefore, we get

$$I_2 \leq \frac{5}{16}\, flops/byte$$

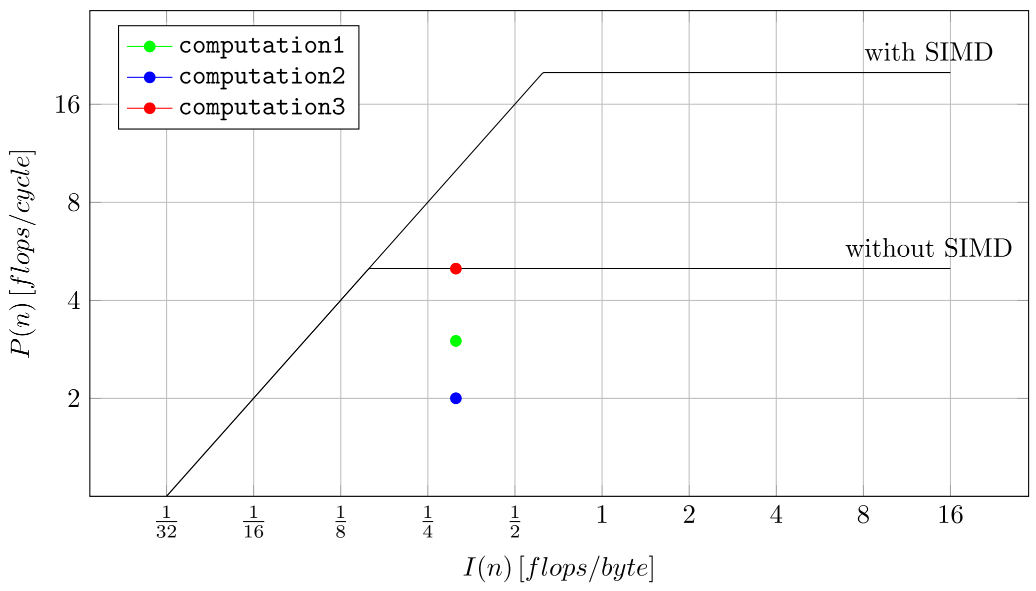
In `computation3` we use FMAs and one addition to calculate `x[i]`. In order to get a upper bound on the performance we assume that they get distributed optimally on the ports. Then we can execute two FMAs (P0 and P1) and one addition (P3) per cycle. Thus, we get an performance bound of

$$P_3 \leq 5\, flops/cycle$$

Again we perform the same work and move the same amount of data so the operational intensity stays the same

$$I_3 \leq \frac{5}{16}\, flops/byte$$

Roofline Plot `computation1` / `computation2` / `computation3`



(c) When we use vector intrinsics we perform the same number of operations and move the same amount of data from main memory into cache. Thus, the operational intensity does not change when using SIMD instructions. With vector instructions we could potentially get a 4x speedup but the computation gets eventually memory bound.

If we use SIMD for `computation1` the performance should reach $4 \cdot 3 = 12\, flops/cycle$ but we reach only $10\, flops/cycle$ because it is memory bound by the roofline model (one can get the bound from the roofline plot). Therefore, `computation1` is **3.33x** faster with SIMD.

For `computation2` we can reach a performance of $4 \cdot 2 = 8\, flops/cycle$ which is not memory bound. Hence, we get a full **4x** speedup for `computation2`.

`computation3` is also memory bound so we can reach only $10\, flops/cycle$ which gives a speedup of **2x**.

(d) The modification does not change the number of operations per loop iteration so we perform the same amount of work which is still $W(n) = 5n\, flops$ for each function.

If we analyze the strided access pattern we can see that more data is moved from memory to the cache. Every iteration we get a compulsory miss (sometimes we get none or two misses but the average each other out). Each miss moves eight doubles from main memory into cache in every iteration so we get

$$Q(n) \geq 8 \cdot 8 \cdot n = 64n\, bytes$$

In all three functions the same access pattern is used and only the operations are changed. Thus, we get a new operational intensity for all three functions of

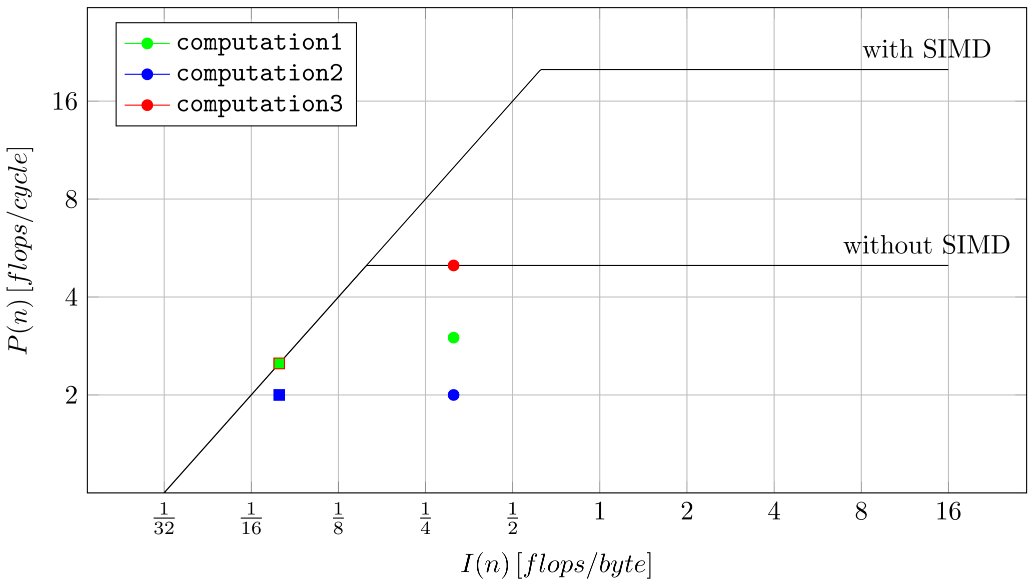
$$I(n) \leq \frac{5n}{64n} = \frac{5}{64}\, flops/byte$$

If we add the new values to the roofline plot we can see that the new functions move to the memory bound area of the roofline model. The performance bound of `computation2` is still the same but we get lower bounds for `computation1` and `computation3` because they cannot bring in all required data for best performance. Therefore, the new bounds are

$$P_1' \leq P_3' \leq 32 \cdot \frac{5}{64} = 2.5\, flops/cycle$$

We get the following roofline plot where the squares show the new bounds.

Roofline Plot `computation1` / `computation2` / `computation3`



4. Skinny Matrix-Matrix Multiplication (15 pts)

(a) As everything fits in cache and we have no conflict misses, we must transfer each value once from main memory to cache. For A we read m^2 values, for B we read mn values and for C we access $2 \cdot mn$ values (considering reads and writes). Under the assumption that we have a write-back/write-allocate cache and C is only written back at the end of the computation, we can bound the I/O cost to

$$Q_a(n, m) = 8m^2 + 8mn + 8 \cdot 2mn = 8m^2 + 24mn\, bytes$$

(b) When simulating the skinny MMM algorithm one can notice that each element of A is read once and C is read and written once to main memory (see *Hint*). Thus, the same number of values as before is moved between cache and main memory.

As we can fit only one row of B into cache, we get additional conflict misses during the computation. In the function `suboperation_Ci` we have a miss for every new row of B (as only one row fits in cache) so we load mn values in each function call. The function `suboperation_Ci` gets called $\alpha = \frac{m}{m_c}$ times. We define M as the number of elements that fit in cache so $M = n + m_c n + 1$. Therefore, we get a lower bound of

$$\begin{aligned}
Q(n, m, M) &= 8m^2 + 16mn + 8\alpha mn \\
&= 8m^2 + 16mn + \frac{8m^2 n}{m_c} \\
&= 9m^2 + 16mn + \frac{8m^2 n^2}{M - n - 1} && (m_c = \frac{M - n - 1}{n})
\end{aligned}$$