



# Project Report

for

# Project 1

as part of VU on

High Performance Computing

SS 2018

Andreas Klebinger, David Fischak

e1426266@student.tuwien.ac.at , e1128194@student.tuwien.ac.at

01426266, 01128194

14 May 2018

# 1 Obtaining a Roofline Model

## 1.1 Machine 1

### 1.1.1 Theoretical Peak Performance

- `earth`:
  - Intel Core i5 750 @ 2.67 GHz
  - 4 Cores

  This processor also uses the SSE2 extensio but has only one unit for floating-point addition and multiplication. Therefore, at most 4 floating-point calculations are executed with each cycle.
  - Theoretical peak performance:
    $1\,\text{instruction} \cdot 4\,\text{operations} \cdot 2.67\,\text{GHz} \cdot 4\,\text{cores} = \mathbf{42.72\,GFLOPS/s}$
  - Peak memory bandwidth: $\mathbf{12.2\,GB/s}$

### 1.1.2 NUMA-STREAM

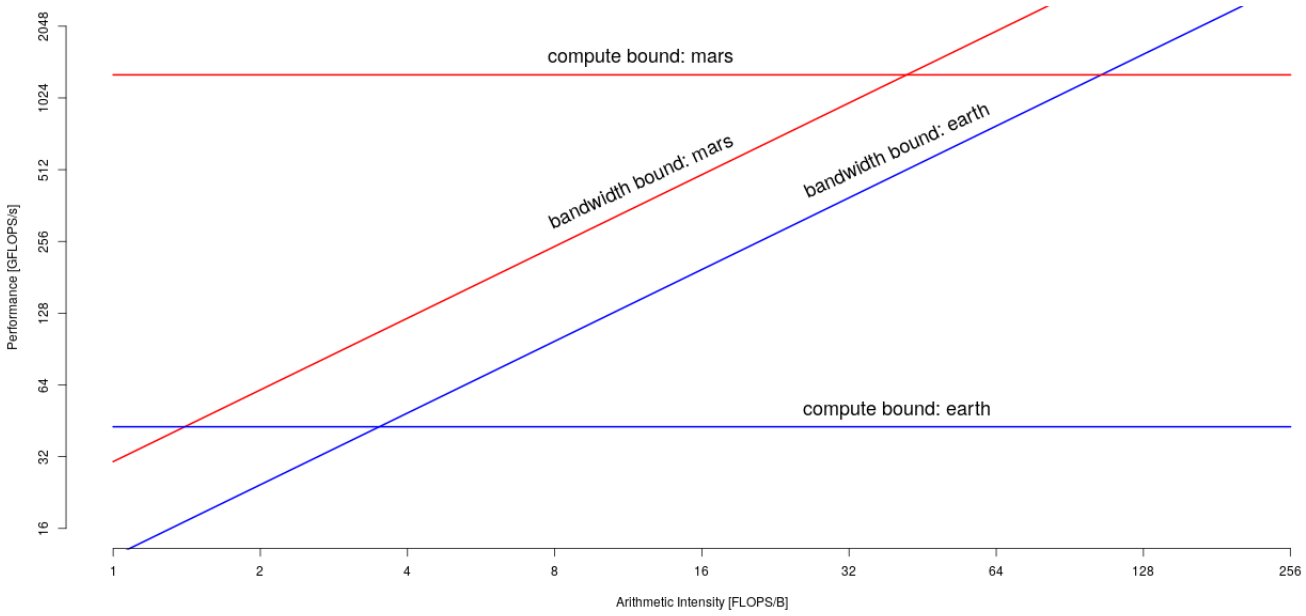
- Parameters used to compile `numa-stream`:
```
gcc -O3 -std=c99 stream.c -lnuma -fopenmp \
-DN=80000000 -DNTIMES=100 -o stream-gcc
```

- Output of `stream-gcc` on `earth`:

```
-------------------------------
STREAM version $Revision: 5.9
-------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------
Array size = 80000000
Total memory required = 1831.1 MB.
Each test is run 100 times, but only
the *best* time for each is used.
-------------------------------
Number of Threads requested = 2
Number of available nodes = 1
-------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 81670 microseconds.
(= 81670 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------
WARNING - The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------
```

```
Function Rate (MB/s) Avg time Min time Max time

Copy:  11315.5286 0.1137 0.1131 0.1237

Scale:  11134.7738 0.1156 0.1150 0.1184

Add:  12139.7541 0.1589 0.1582 0.1625

Triad:  12197.3056 0.1581 0.1574 0.1597

-------------------------------

Solution Validates

-------------------------------
```

### 1.1.3 Roofline Plot




## 1.2 Machine 2 / Mars

The mars system has:

- 8 Sockets with
- an Intel Xeon E7-8850 @ 2.00 GHz
- with 10 Cores each.

As such it is an 80 core NUMA shared memory system.

### 1.2.1 Theoretical Peak Performance

The Xeon E7-8850 is a Westermere-EX architecture based on Nehalem.[3] As such is has two discrete ALU's for addition and multiplication. Each ALU can process up to 4 SP Floats per cycle when using the appropriate SIMD instructions.[1]

This results in a peak performance for the whole mars system of:

$$\text{Throughput} = 4\text{SPF} * 2(\text{ALUs}) * 2Ghz * 80 = 1280\,GFLOPs$$

### 1.2.2 STREAM or NUMA-STREAM

We measured the maximal bandwith using the NUMA-STREAM benchmark. The maximum Bandwith measured was 30.5GB/s

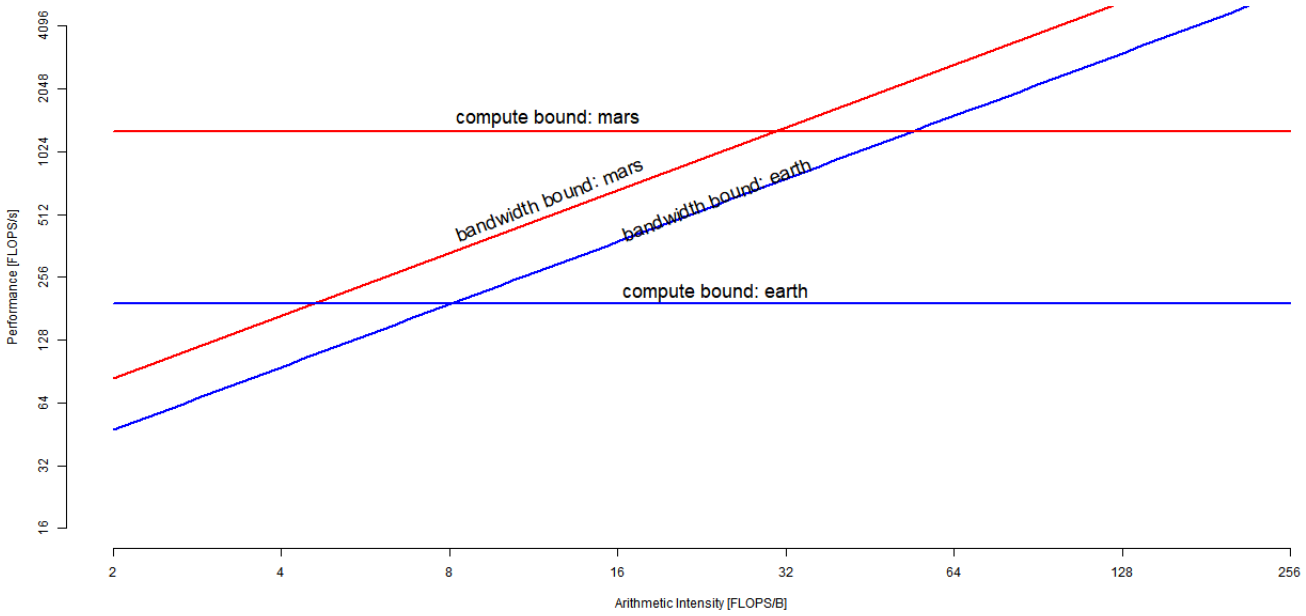



Figure 1: Calculated roofline model for mars

| Function | MB/s |
|---|---|
| Copy | 30344 |
| Scale | **30521** |
| Add | 30126 |
| Triad | 30260 |

### 1.2.3 Roofline Plot

### 1.2.4 Compilation Details & Output

```
gcc -O3 -std=c99 stream.c -lnuma -fopenmp -DN=80000000 -DNTIMES=100 -o stream-gcc
.\stream-gcc
-------------------------------------------------------------
STREAM version $Revision: 5.9 $
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 80000000
Total memory required = 1831.1 MB.
Each test is run 100 times, but only
the *best* time for each is used.
-------------------------------------------------------------
Number of Threads requested = 80
Number of available nodes = 8
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 20711 microseconds.
(= 20711 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:       30344.0333      0.0429       0.0422       0.0441
Scale:      30521.8913      0.0424       0.0419       0.0440
Add:        30126.6472      0.0645       0.0637       0.0656
Triad:      30260.5691      0.0643       0.0634       0.0660
-------------------------------------------------------------
Solution Validates
-------------------------------------------------------------
```

# 2 Performance of kmeans Kernel

## 2.1 Compilation Details

The kmeans benchmark was compiled with the following flags apply to the executable and all c files:

```
CC_FLAGS= -DLIKWID_PERFMON -g -I/opt/likwid/include -L/opt/likwid/lib -llikwid -fopenmp -O2
```

GCC 7.3 was used as compiler:

```
e01426266@mars:~/NUMA-STREAM$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: .... <omitted for breviety>
Thread model: posix
gcc version 7.3.0 (Debian 7.3.0-5)
```

For performance measurements `likwid-4.3.1` was used.

## 2.2 Obtain AI for kmeans

We measured the AI of kmeans using likwid. The following schema was used for all invocations.

```
/opt/likwid/bin/likwid-perfctr -f -C <0-79/60-69> -g \
    FLOPS_SP ./kmeans_openmp/kmeans -n <threads> -i <datafile>
/opt/likwid/bin/likwid-perfctr -f -C <0-79/60-69> -g \
    MEM      ./kmeans_openmp/kmeans -n <threads> -i <datafile>
```

Table 1: Overview of AIs obtained for kmeans on Mars

| #cores | file | Total MFLOPS | Total MBytes | AI (FLOPS/Byte) |
|---|---|---|---|---|
| 10 | 100 | 1.33 | 73.44 | 0.02 |
| 10 | 204800.txt | 1517.92 | 476.26 | 3.19 |
| 10 | 819200.txt | 2225.72 | 734.17 | 3.03 |
| 10 | kdd_cup | 3427.63 | 1061.99 | 3.23 |
| 80 | 100 | 0.58 | 78.48 | 0.01 |
| 80 | 204800.txt | 731.82 | 981.66 | 0.75 |
| 80 | 819200.txt | 1164.44 | 1244.34 | 0.94 |
| 80 | kdd_cup | 1519.86 | 1420.36 | 1.07 |

## 2.3 Roofline plots for entire kmeans application

## 2.4 Roofline plots with realistic ceiling for kmeans kernel

We arrive at a realistic Ceiling as follows:

Start with the theoretical peak performance: 1280 GFlops

Take into account the lack of SIMD parallelism:

$$\frac{1280GFlops}{4} = 320GFlops$$

Take into account the imbalance of MUL vs ADD operations:

$$\frac{320GFlops}{2Flop/Cycle} * 1.5Flop/Cycle = 240GFlops$$

## 2.5 Obtain AI for kmeans with the LIKWID marker API

We add the -m switch to our invocations to activate the marker API. We marked the kernel of the main loop for performance measurement. This being the inner loop which iterates over all points of the dataset:

## 2.6 Roofline plots with realistic ceiling for kmeans kernel only (with marker API)

## 2.7 Comparison and Discussion

### 2.7.1 Empirical AI Data

Table 1 shows the recorded performance for kmeans on varius inputs. Table 2 improved on this data by only recording data for the kernel of the application.

Table 2: Overview of AIs obtained for kmeans on Mars for marked region.

| #cores | file | Total MFLOPS | Total MBytes | AI (FLOPS/Byte) |
|---|---|---|---|---|
| 10 | 100 | 385.08 | 150.01 | 2.57 |
| 10 | 204800.txt | 12738.94 | 2248.92 | 5.66 |
| 10 | 819200.txt | 12814.52 | 3500.45 | 3.66 |
| 10 | kdd_cup | 12995.59 | 3451.19 | 3.77 |
| 80 | 100 | 7.46 | 1292.02 | 0.01 |
| 80 | 204800.txt | 4017.84 | 2807.56 | 1.43 |
| 80 | 819200.txt | 6579.41 | 3312.47 | 1.99 |
| 80 | kdd_cup | 5940.10 | 3230.91 | 1.84 |

When recording performance for the whole application we end up also recording parts of the program not relevant to roofline. So we record for example during IO (reading of the input file) and initialization (allocating buffers).

As these tasks involve syscalls and possibly external hardware the are not suitable for analysis through the roofline model. We can improve on this by only recording information about the application kernel as presented in the second table.

We can improve on our analysis even further by taking into consideration maximum performance for the given code as discussed in subsubsection 2.7.2.

The maximum performance recorded for the marked region in a benchmark is 13GFLOPS. According to the roofline model this would require bandwith close to

$$13GFLOPS * 0,31Flop/Byte = 4,03GB/s$$

However actual bandwith used for the benchmark is at $3,45GB/s$.

So the performance of our kernel seems limited by FP performance. But we are well below the ceiling there as well.

This is likely due to reasons outside of FP performance like branch mispredics, control flow overhead, synchronisation overhead or others. Given that performance actually goes down when using 80 Threads compared to 10 it is an reasonable assumption that a lot of performance is lost in overhead caused by parallelization.

### 2.7.2 Maximum performance of the kmeans kernel

The kernel of the `kmeans` algoirthm is in the `kmeans_clustering` function.

The function is split into three main parts:

- Initialization (happens once)
- In the main loop:
  - Calculating membership to existing clusters for each point.
  - Update clusters.

We can safely ignore initialization for our performance consideration which leaves the main loop.

The main loop itself can be further split into three distinct parts.

- For each point:

  - Find the nearest cluster(`find_nearest_point`).
  - Collect partial updates on the position of clusters.
- Reduce partial information and update positions of clusters

For our analysis we assume 5 clusters and 34 features which matches the default of kmeans and our input datasets.

The only significant contributors here in terms of computation are the loops to find the nearest cluster and updating the cluster information. For this reason we will look only at these inner loops and not at the rest of the main loop.

Listing 1: Inlined representation of find_nearest_point

```
for (i=0; i<nclusters; i++) {
        float dist=0;
        for (j=0; j<nfeatures; j++)
                dist += (pt1[j]-clusters[i][j]) * (pt1[j]-clusters[i][j]);
        if (dist < min_dist) {
                min_dist = dist;
                index    = i;
        }
}
```

The kernel here is: `dist += (pt1[j]-pt2[j]) * (pt1[j]-pt2[j])`.

We get 8 byte of memory bandwith for three floating point operations. Memory bandwith consists of 2 loads for each iteration:

- Read: `pt1[j]`
- Read: `clusters[i][j]`

Operations are two additions and one multiplication. [1] [2]

With one call to `find_nearest_point` for each data point per iteration of the main loop this gives us:

$$iterations = n * \texttt{nclusters} * \texttt{nfeatures} = n * 170$$

Even for our smallest data sample which has 100 points this already gives us 17000 iterations and will dominate the computational demands for any reasonably large size of n.

If we stop here our Arithmethic Intensity is 3 FP operations per 8 byte. Giving an AI of $\frac{0{,}375 Ops}{Byte}$ [3]

Also for each point we will update the partial cluster information:

Listing 2: Updating (partial) cluster information

```
for (j=0; j<nfeatures; j++)
partial_new_centers[tid][index][j] += feature[i][j];
```

[1] Treating subtraction as addition.

[2] While there are two subtractions in the code gcc will eliminate one through common subexpression elimination.

[3] Although in practice caches will usually hold both pt1 and the clusters after the first run of the loop body.

We get for the number of iterations:

$$iterations = n * nfeatures = 34$$

While this is not as big an contributor as the loop finding the nearest cluster it still contributes a noteworthy amount of computational demands.

This loop has for each iteration two loads (8 Byte) and one addition. [4]

Past this what is left to do is the combination of partial cluster information and calculating the new clusters.

This happens in a fixed (relative to the number of points) number of iterations so no AI intensity calculation was done on this code.

Listing 3: Reduction pt1

```
/* let the main thread perform the array reduction */
for (i=0; i<nclusters; i++) {
        for (j=0; j<nthreads; j++) {
                new_centers_len[i] += partial_new_centers_len[j][i];
                partial_new_centers_len[j][i] = 0.0;
                for (k=0; k<nfeatures; k++) {
                        new_centers[i][k] += partial_new_centers[j][i][k];
                partial_new_centers[j][i][k] = 0.0;
                }
        }
}
```

The amount of iterations of the above loop is static in regards to the number of items we work on. It is given by $iterations = clusters * threads * features$. In our case when using all cores of the Mars system we get:

$$iterations = 5 * 80 * 34 = 13600$$

However this is neglible for nontrivial datasets so we exclude it from the AI caluclation.

Similar we have the loop calculating the new clusters at the end of the main loop:

Listing 4: Reduction pt2

```
/* replace old cluster centers with new_centers */
for (i=0; i<nclusters; i++) {
        for (j=0; j<nfeatures; j++) {
                if (new_centers_len[i] > 0)
                        clusters[i][j] = new_centers[i][j] / new_centers_len[i];
                new_centers[i][j] = 0.0;    /* set back to 0 */
        }
        new_centers_len[i] = 0;    /* set back to 0 */
}
```

With this loop having at most iterations of:

$$clusters * features = 5 * 34 = 170 = iterations$$

[4] Although it is fair to assume that partial_new_centers[tid][index][j] will be cached.

This loop has no significant impact on overall performance and can safely be ignored.

Table 3 gives us the AI for the inner main loop based on it's two essential parts according to the roofline model.

According to the roofline model this gives us a demand of

$$1280GFlops * 0,31Flop/Byte = 396,8Gb/s$$

on the memory system for 80 Threads if we want to reach the FP Performance Ceiling.

It is however important to point out this does assume NO caching. In practice much of the memory pressure might be absorbed by the cache.

Table 3: Arithmethic Intensity according to the Roofline Model

| | Find nearest Cluster | Update cluster information | Total |
|---|---|---|---|
| Iterations | $n * 170$ | $n * 34$ | - |
| Memory (Byte) | $8 * 170$ | $12 * 34$ | 1768 |
| Operations (Flop) | $3 * 170$ | $1 * 34$ | 544 |
| Arithmetic Intensity (Flop/Byte) | 0,375 | 0,08 | 0,31 |

## 2.8 kmeans: Maximum Performance

When considering the maximum performance of the given code there are multiple things to consider:

- The kernel does not use SIMD instructions. This immediatly brings down maximum performance by a factor of 4!
- The kernel is not balanced. There is actually only a single relevant multiplication in the inner loop! Which can happen in parallel to the additions.

In `find_nearest_point` we have two additions and one multiplication per result. This mean on average we will execute an addition every cycle and a multiplication every second. So we execute 1,5 Instructions per cycle instead of the maximum of 2 in this kernel because of the mul/add imbalance.

We could additionally take into account the smaller loop Listing 2 which is limited purely by addition. However this would bring down flops/cycle only slightly so we will keep using 1.5Flops/Cycle for further analysis.

Maximum FP performance under these considerations is:

$$1280GFlops * 0.25(\text{no SIMD}) * 0.75(\text{imbalance}) = 240GFlops$$

.

Which gives us for the demand on the memory subsystem for 80 Threads:

$$240GFlops * 0,31Flop/Byte = 74,4GB/s$$

And 10 Threads:

$$30GFlops * 0,31Flop/Byte = 18,6GB/s$$

According to our bandwith measurements this means we will be computation bound on 10 cores and bandwith bound on 80 cores.

However since many of the inputs to the kernel might be cached this is not representative of the exact demands on the memory system as our measurements show.

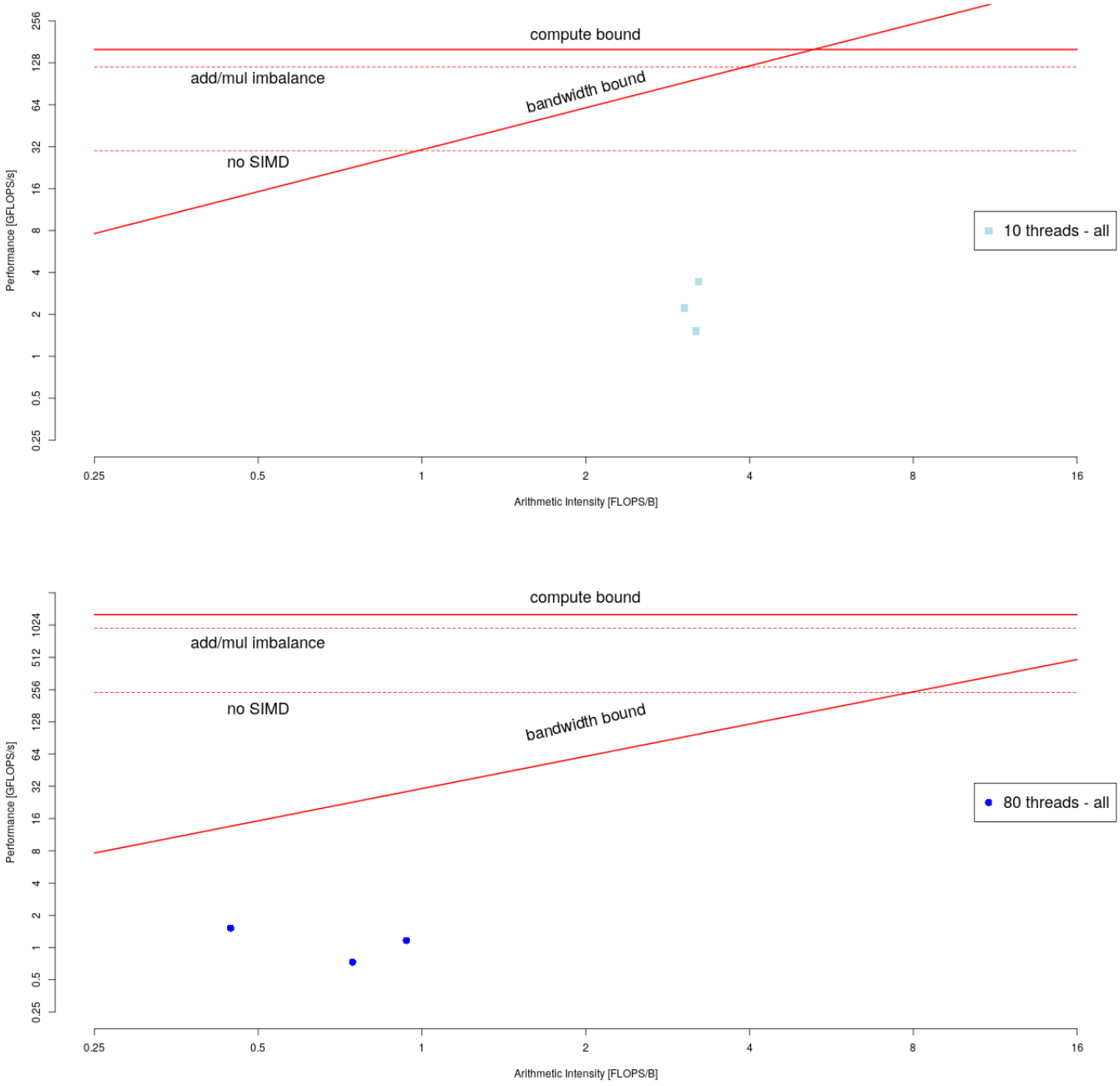






Figure 2: Roofline for complete kmeans application

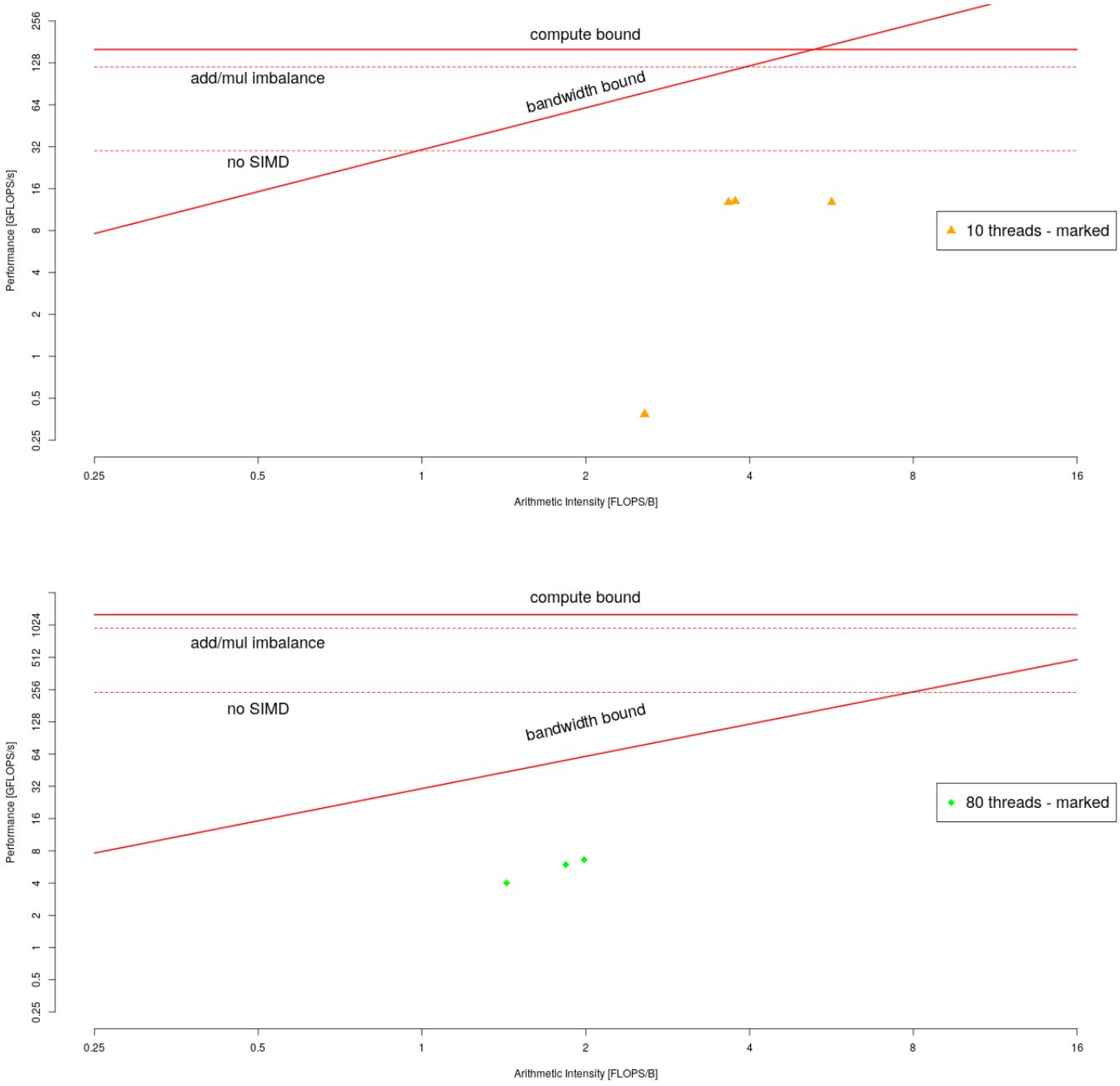



Figure 3: Roofline including realistic ceilings for Mars

## References

[1] Agner Fog. Software optimization resources - The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `http://www.agner.org/optimize/microarchitecture.pdf`.

[2] Michael L. Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Springer, 3 edition, July 2008. doi: 10.1007/978-1-4614-2361-4.

[3] Various. Xeon e7-8850 - intel. `https://en.wikichip.org/wiki/intel/xeon_e7/e7-8850`.