Performance Engineering
Assignment 1: The Roofline model
Assigned: April 4,
Due: April 13, 20:00

Daniël Boelman, Chaitanya Kumar, Kevin Kuurman, Dimitri Hooftman

1 Introduction

The goal of this assignment is to demonstrate how to build and use Roofline models, for both sequential and parallel code. To this end, we use matrix multiplication as an application of interest, and build three versions of the model: reference, sequential-optimized, and parallel (CPU and GPU).

This report aims to describe the process of building the Roofline models, and to reflect on this type of performance models. Specifically, we describe the general challenges of the Roofline model, then dive into the details of the matrix multiplication itself, and, further highlight the differences between sequential and parallel solutions and systems. Finally, we summarize our findings and lessons learned.

2 Background

In this section we discuss the background knowledge necessary to build the Roofline model.

The Roofline model combines system and application knowledge to provide accurate performance bounds for the given application. The following sections refer in more detail on what this knowledge entails, and how the data is taken into account in a unified model.

The system

Two performance aspects of the system used in the Roofline model are the computational throughput and memory bandwidth.

Q1 (1p): What are the ways to calculate/obtain these parameters? Which one(s) did you use and why?
There are various avenues available to find this information. One obvious start could be to go through the data sheets of your specific chip-set. Howerver, this means that you'd have to build the entire model by hand. A more automated method is by utilizing benchmarking software like likwid. This application will run various benchmarks to find the peak arithmetic intensity and the peak bandwidth. The advantage

of benchmarking is that it automatically generates the Roofline and could potentially include ceilings not available in the data sheet or documentation of the chip.

The application

To include knowledge about the application, the Roofline model uses a specific metric called arithmetic intensity (AI) - sometimes used interchangeably with operational intensity.

Q2 (1p): What are the ways to calculate/obtain an application's arithmetic intensity? Which one(s) did you use and why?
One way of calculating the AI would be to manually count the number of arithmetic operations and dividing it by the number of memory read/write instructions. This might lose accuracy due to missing or neglecting to count certain instructions in the code.

An alternative would be to use a profiler to measure the arithmetic intensity; the key advantage is that it gives a very accurate statistic of the actual kernel execution.

Q3 (1p): How does the application input data affect arithmetic intensity?
Cache behaviour can differ between input matrix dimensions. Furthermore, for SIMD applications a slight performance difference could be assumed if the input matrix is not properly dividable by the stride of a vector.

Putting it all together

Q4 (0.5p): Please explain how the two types of data (i..e, system and application, discussed in the previous two sections) are combined to build the model. What can the model tell, in general, about the given application in terms of performance?
The model consists out of two components. First a roof is constructed based on the memory bandwidth and the estimated peak computational intensity. Since there are multiple layers in the memory hierarchy (L1d cache, L2 cache, L3 cache and dynamic random access memory (DRAM)), and multiple instruction set architectures (ISAs) (scalar, AVX2, Fused Multiply Add (FMA)), every architecture will in general have multiple roofs for both memory bandwidth and peak flops [2].

Next a model is constructed for the kernel in consideration which will contain both the number of bytes loaded/stored and the number of arithmetic operations. Their ratio gives the AI. This gives the horizontal placement of the kernel under the roof, and the measured performance (flops/s) gives you the y component. If there is then a large gap between estimated performance (the roof) and actual measured performance the kernel might need to be optimised. The combination of hardware model (roofline) and software model (dot under the roof) then gives you the constraining factor preventing you from gaining more performance.

3 Modeling the sequential matrix multiply

In this section we explain how we have built the Roofline model for the sequential application.

The system(s) under investigation

Q5 (0.5p): What are the systems you considered for this version of the application, and what is the compute throughput and bandwidth data you found for them?
For this we first need to define how to theoratically calculate the throughput and bandwidth. First, throughput [GFLOPs] = chips * cores * vectorWidth * FLOPs/cycle * clockFrequency. Bandwidth is defined as memory bus frequency * bits per cycle * bus width.
For our experiments we used the DAS5 which has an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. The details as given by intel are listed in the table below.

Feature	Information
Microarchitecture	Haswell
Number of cores per socket	8
Threads per core	2
L1d Cache (per core)	32 kB
L1i Cache (per core)	32 kB
L2 Cache (per core)	256 kB
L3 Cache (shared)	20480K kB
Base Frequency	2.4 GHz
IPC	2

Table 1: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
https://ark.intel.com/content/www/us/en/ark/products/83356/intel-xeon-processor-e52630-v3-20m-cache-2-40-ghz.html

From this we can derive that the theoretical throughput for a sequential scalar application would be 1 * 1 core * 1 SP * 2 IPC * 2.4GHz = 4.8 GFLOPs. The bandwidth for this processor is 59 GB/s according to the specs.

The model for the reference implementation

Q6 (1.0p): Please describe how you modeled the reference application, and show the model(s) you obtained. Comment on the challenges you have encountered.
The reference implementation is defined in the snippet below. In this implementation we only consider the inner-most loop and neglect the iterators and branches. A total of four single-precision floats are transferred - three loads and one store. Each iteration has two flops for the multiply addition and two integer arithmetic operations per matrix indexing which thus gives 8 operations total. 8 flops per $4 \cdot 4 = 16$ bytes. The arithmetic intensity is therefore $8/16 = 0.5$ operations per byte. While the primary source on the cache-aware roofline model considers only flops, here the index calculations are integer FMAs and constitute a significant fraction of the compute load. Hence we choose to include them in the AI calculation.

Q7 (0.5p): How does the model compare against the observed performance?
The results visualized through the intel advisor is shown below. The measured arithmetic intensity is calculated to be 0.4 which is close to the estimated 0.5 found earlier.

Algorithm 1 Baseline Sequential Kernel

Input: matrices A[$m \times n$], B[$n \times p$] and C[$m \times p$]

for i in $0.. = (m-1)$ **do**
  for j in $0.. = (p-1)$ **do**
    $C[i,j] = 0$
    for k in $0.. = (n-1)$ **do**
      $C[i,j] \mathrel{+}= A[i,k] \cdot B[k,j]$
    end for
  end for
end for

However, the peak bandwidth and scalar peak intensity is quite different from what we found earlier which means either we misinterpreted the data-sheets, or we found incorrect information.

844.96GFLOPS(23.8x)

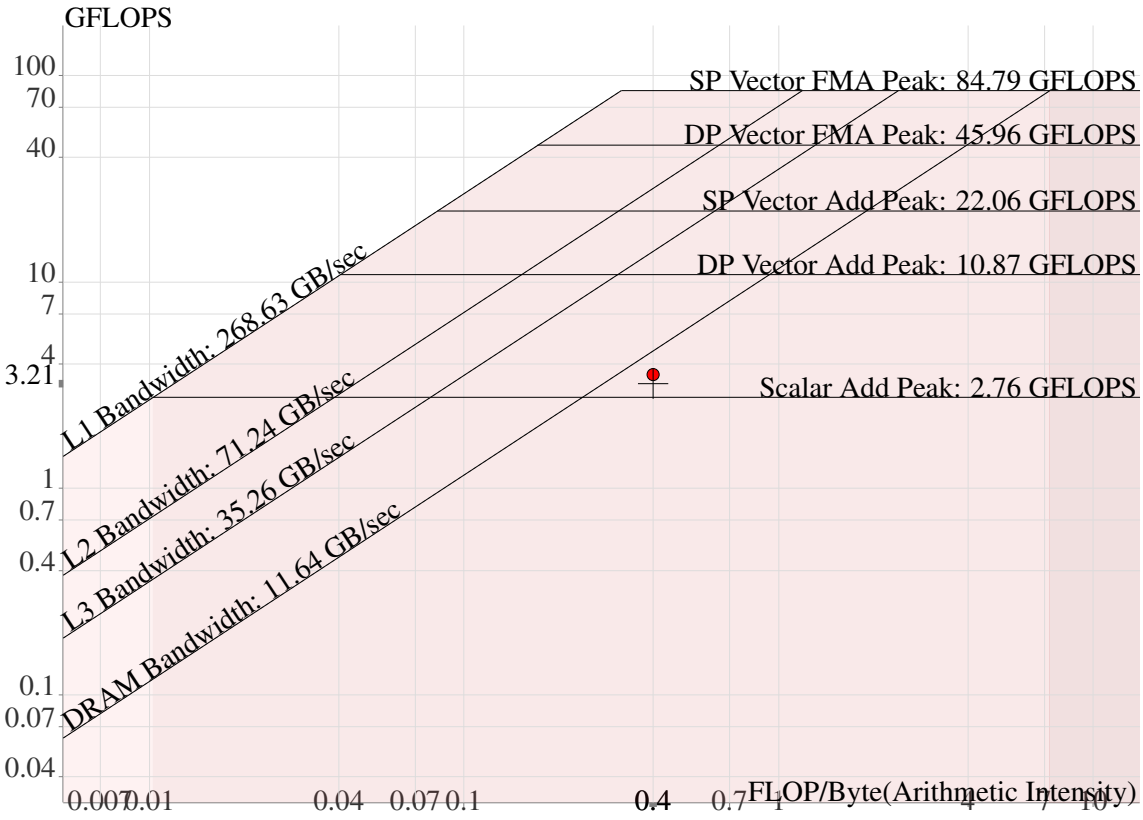

Figure 1: Roofline model for the reference implementation

A better sequential version

Q8 (0.5p): How did you use the Roofline model to determine what (kind of) optimizations to apply?
An arithmetic intensity of 1/2 shows that the application is bandwidth bound and close to the DRAM bandwidth ceiling. This means that the application should do more arithmetic operations on the fetched/stored bytes in order to make it compute bound. After the application is compute bound a possible optimization is to make use of vectorization. This would increase the peak performance of the core which in turn allows for better performance.
The next step is parallelization such as OpenMP. This would theoretically multiply the peak performance by the number of cores available, but does incur some overhead.
Finally CUDA is considered as it allows for a significantly higher degree of parallelization compared to the CPU.

Q9 (0.5p): Explain briefly the optimizations you applied to improve the performance of the sequential version.
In order to make the kernel more cache friendly, we introduce tiling - the matrices A, B, and C are partitioned into submatrices. The matrix multiplication is then carried out in the form of block matrix multiplications. Pseudocode for the same is given in algorithm 2. We let the tile width and height be equal to the cache line width, so that each tile row occupies one cache line. A more generalised implementation would parameterise the tile dimensions and optimising that would require performing a parameter sweep.

Algorithm 2 Optimised Sequential Implementation

Input: matrices A[$m \times n$], B[$n \times p$] and C[$m \times p$]

$b = W/F$ ▷ Calculate tile width
for i_b in $0 : b : (m-1)$ **do** ▷ Iterate through tiles of matrix using step size b
  for j_b in $0 : b : (p-1)$ **do**
    for k_b in $0 : b : (n-1)$ **do**
      `microkernel`$(A[i_b, k_b], B[k_b, j_b], C[i_b, j_b])$ ▷ Perform algorithm 3 to tile
    end for
  end for
end for

Algorithm 3 The tile microkernel

Input: matrices A[$m_b \times n_b$], B[$n_b \times p_b$] and C[$m_b \times p_b$]

for i in $i_b..(i_b + b)$ **do**
  for j in $j_b..(j_b + b)$ **do**
    $d = 0$
    for k in $k_b..(k_b + b)$ **do**
      $d \mathrel{+}= A[i, k] \cdot B[k, j]$
    end for
    $C[i, j] = d$
  end for
end for

The microkernel in line 5 implements essentially the same thing as algorithm 1. An advantage of this class of methods (i.e. a macrokernel iteratively calling a microkernel) is that further optimisations can be made in a modular fashion. E.g. another optimised version can be produced by optimising just the microkernel while letting the macrokernel remain the same.

Note that practical implementations of algorithm 2 and algorithm 3 would take into matrices of arbitrary dimensions and recalculate the block size for the fringe blocks. Here, we effectively calculate amortised costs of data movement by assuming that the matrix dimensions are dividable by the block size.

The estimated number of flops remains the same as the baseline implementation. Each microkernel invocation accesses $2Fb^3 + Fb^2$ bytes, and the microkernel is invoked $\frac{mnp}{b^3}$ times. Therefore, the total number of bytes read/written becomes

$$\frac{F}{b}mnp(2b+1) \tag{1}$$

The operational intensity is then

$$\frac{2mnpb}{Fmnp(2b+1)} = \frac{2b}{2b+1} \tag{2}$$

Our CPU has a cache line width of $W = 64$ bytes, and $b = W/F = 64/4 = 8$. Therefore, the arithmetic intensity for this kernel is a constant value

$$\frac{2b}{2b+1} = \frac{16}{17} \approx 0.94 \tag{3}$$

Q10 (0.5p): How did the optimizations affect the arithmetic intensity of the application? Demonstrate in an updated Roofline model.

After running the tiling version with intel advisor the roofline below is obtained.

66246GFLOPS(35.5x)

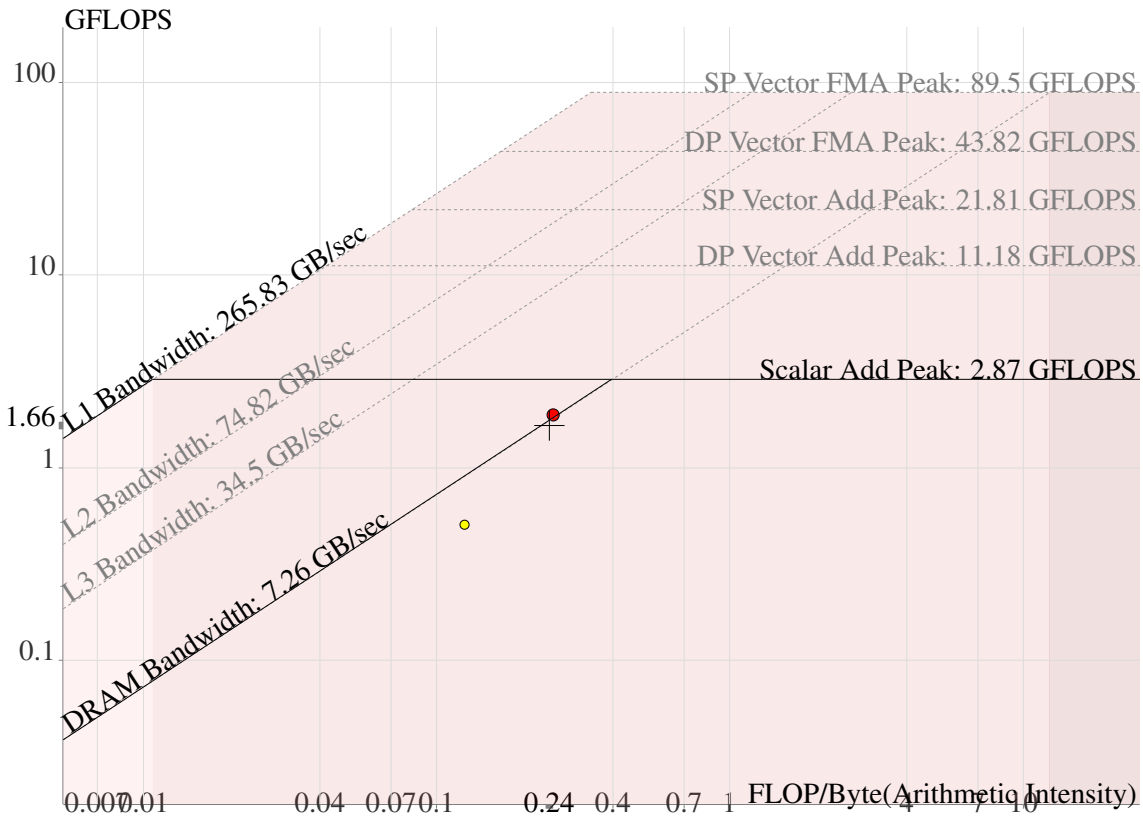

Figure 2: Roofline model for the tiling implementation

This unfortunately tells us that the arithmetic intensity went down, and same goes for the performance. This could be due to the extra loops which add another 3 integer additions and the tiling which adds even more operations along with branches.

Q11 (0.5p): How does the new model compare against the observed performance of the optimized version? Overall, when we compare our model (operational intensity of 0.94 flops/byte) with the measured results (0.24 flops/byte) it is clear that our model is not quite right. This could be because we overestimated the potential gain from the caching improvement, or that the added loops add a lot of unexpected overhead.

4 Modeling the parallel matrix multiply

In this section we explain how we have built the Roofline model for the parallel version(s) of the application.

The system(s) under investigation revisited

Q12 (0.5p): What are the systems you considered for this version of the application, and what is the compute throughput and bandwidth data you found for them?
We considered two methods for parallelization. The first method is to use OpenMP to run the sequential version in parallel on the CPU. This should give a potential speedup of up to 16 times as there are 8 cores which support 2 threads each.

The second option we applied is to use CUDA to perform the operations on the GPU.

Feature	Information
Microarchitecture	Turing
Number of cuda cores	4352
Number of streaming multiprocessors (SM)	68
L1 Cache	64 kB per SM
L2 Cache	5.5 mB
Base Frequency	1.35 GHz

Table 2: NVIDIA RTX 2080Ti @ 1.35GHz
https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305

The model(s) for the parallel implementation

OpenMP

Q13 (1.0p): Please describe how you modeled the parallel application(s), and show the model(s) you obtained. Comment on the challenges you have encountered.
The OpenMP version uses the tiling version as a base and parallelizes the outer 3 loops. This means that effectively the model remains the same as the tiling version but distributes the operations over multiple cores.

Q14 (0.5p): How did parallelization affect the arithmetic intensity of the application? Demonstrate in the updated Roofline model(s).
Here we use data-parallelism using OpenMP's parallel for loop pragmas. We also use `collapse(3)` since the macrokernel is a triple loop. A consequence of this is that there can be dependencies between threads as multiple threads can update the same element of C, which is why the final update in line 7 in algorithm 3 is done atomically.

Q15 (0.5p): How does the model compare against the observed performance of the parallel versions?
Under the cache-aware roofline model framework, barring overhead of threads management and the synchronisation (due to the atmomic update), the arithmetic intensity remains the same, but the ceilings all move up roughly by a factor of the number of cores or threads.

There is a caveat to the ceiling height increase because of hyperthreading (also known as simultaneous multithreading (SMT)). When there are multiple hardware threads per core, some hardware like floating point units (FPUs), registers etc are replicated per thread, but they share the same caches. If two multiple threads of the same kernel are scheduled on the same core, then each thread effectively ends up with smaller

caches.

Another possible pitfall specific to some architectures like X86-64 is false sharing, when multiple threads (on the same core or otherwise) update different elements in the same cache line. However, we do not expect this to be a major problem in this particular kernel as the data is neatly partitioned.

1039.94GFLOPS(16.3x)

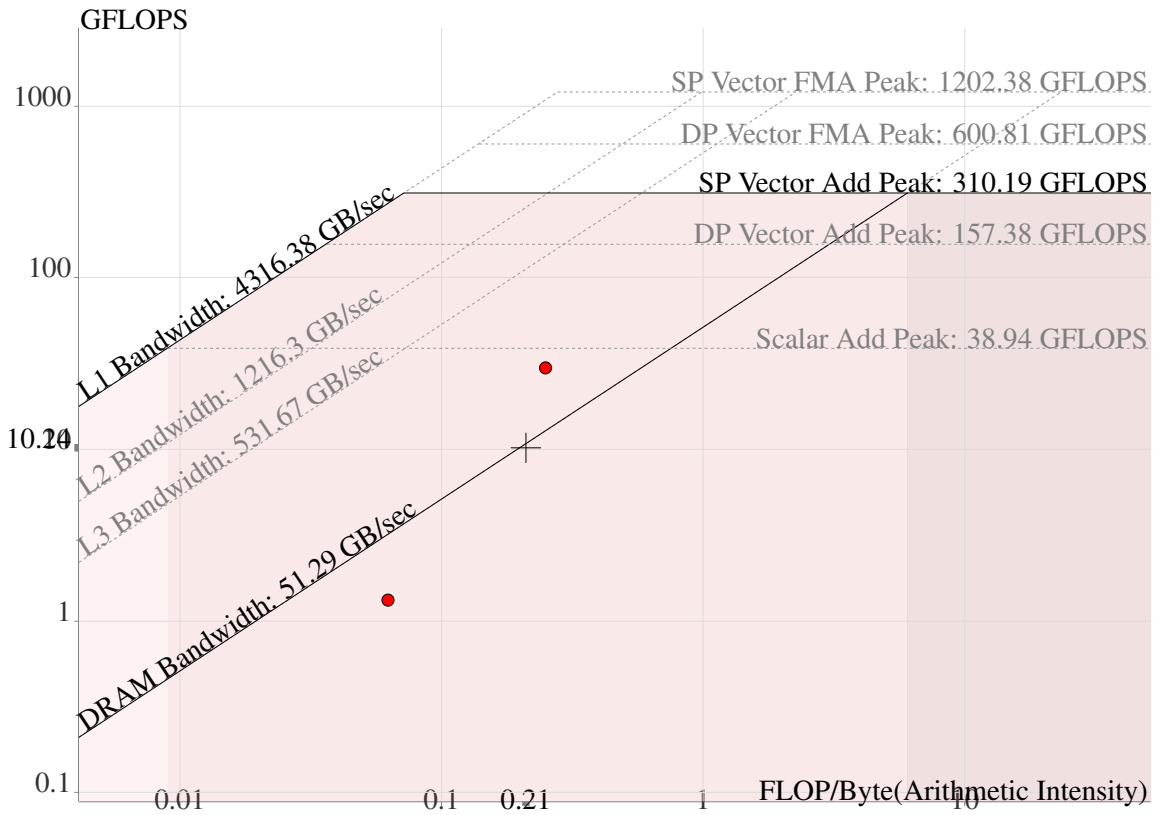

Figure 3: Roofline model for the tiled OpenMP implementation

CUDA

Q13 (1.0p): Please describe how you modeled the parallel application(s), and show the model(s) you obtained. Comment on the challenges you have encountered.

We implement a tiled matrix multiplication in CUDA. The tiling is done in shared memory - elements from global memory are staged via the shared memory to reduce memory traffic. Compared to a naive implementation that uses only the global memory without touching the shared memory, the amount of traffic to global memory reduces by a factor of the tile width.

We use NVIDIA NSight Compute to profile the kernel and get a tool-assisted roofline chart. While Intel Advisor uses the cache-aware roofline model, we think that NSight Compute uses the original roofline

Algorithm 4 CUDA parallelised version (bound-checks elided for brevity)

Input: matrices A[$m \times n$], B[$n \times p$] and C[$m \times p$]

```
1:  TILEDIM = 16                                          ▷ Tile dimension is 16
2:  A_TILE[TILEDIM][TILEDIM]                              ▷ Create shared memory tiles
3:  B_TILE[TILEDIM][TILEDIM]
4:  bx = blockIdx.x
5:  by = blockIdx.y
6:  tx = threadIdx.x
7:  ty = threadIdx.y
8:  row = by * TILEDIM + ty
9:  col = bx * TILEDIM + tx
10: for f in 0.. = (n + TILEDIM − 1)/TILEDIM do    ▷ Iterate through tiles of matrix using step size b
11:     A_TILE[ty, tx] = A[row * n + f * TILEDIM + tx]
12:     B_TILE[ty, tx] = B[(f * TILEDIM + ty) * p + col]
13:     for k in 0 : TILEDIM do
14:         result += A_TILE[ty, k] * B_TILE[k, tx]
15:     end for
16: end for
17: C[row * p + col] = result
```

model.

Q14 (0.5p): How did parallelization affect the arithmetic intensity of the application? Demonstrate in the updated Roofline model(s).

While the original roofline model measures the bytes transferred between the DRAM and the last level cache (LLC), we approximate it as the bytes transferred between the DRAM and the shared memory.

So, we calculate the number of flops per byte loaded into shared memory. The loop at line 1 in algorithm 4 is executed by each thread. In each iteration of the loop, the thread in the block loads two floats, one element of A and another of B, into the shared memory corresponding to that block. It then performs $2 * \texttt{TILEDIM}$ flops. Finally after the outer loop in line 1 is done, the thread block stores the results back into the global memory.

We thus estimate the arithmetic intensity as

$$\frac{2 \cdot \texttt{TILEDIM}}{2 * 4} = \frac{2 \cdot 16}{8} = 4\text{flops/byte} \tag{4}$$

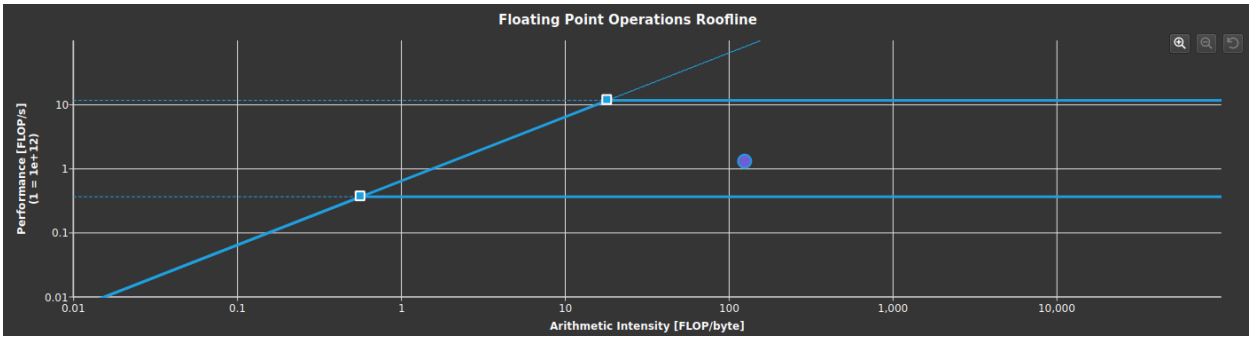

Figure 4: Roofline model for the NVIDIA RTX 2080Ti with the CUDA kernel placed

Q15 (0.5p): How does the model compare against the observed performance of the parallel versions?
The roofline model constructed by NVIDIA's NSight Compute is shown in fig. 4. It would appear that our estimate (4) is a long shot from what the profiler calculates. This could be because NSight compute could be measuring the traffic between the global memory and the L2 cache instead of the shard memory.

While we (and Intel Advisor) use the cache-aware roofline model, NSight Compute appears to be using the original roofline model [3] as the arithmetic intensity is a lot larger than what would be calculated in the cache-aware roofline model. This happens because tiling reduces the number of DRAM accesses by resuing values stored in the shared memory, and the original roofline model measures the bytes transferred between the LLC and DRAM.

5 Conclusion and Future Work

This section presents the lessons learned from this assignment. We specifically focus on the construction and usage of Roofline models.

Lessons learned

Q16 (0.5p): What are the challenges and uses of Roofline models?
Accurate manual calculation of bandwidths and peakflops is non-trivial; it requires incorporating the microarchitecture details like instruction latency, different base, turbo and AVX frequencies, pipeline structure, number of functional units and ports etc.

Matrix-matrix multiplication is a somewhat low operational intensity kernel with respect to floating point operations. This means that other operations like integer arithmetic required for indexing into memory locations in the matrix start having non-trivial contributions. We feel that we do not have a proper understanding of how equal or unequal their contributions to the overall AI is.

Q17 (0.5p): What are your findings about the validity and accuracy of the models you created?
The baseline kernel model's inner-most loop's arithmetic intensity was estimated by us to be ≈ 0.5, while Intel Advisor reports it around 0.4. This would indicate that including the integer arithmetic performed in the `lea` instructions might be necessary for an accurate analysis of the kernel. This is corroborated by the

fact that we mispredicted the change in arithmetic intensity for the tiled kernel as we only considered flops and not the integer arithmetic instructions in the operational intensity calculation; the arithmetic intensity decreased instead of increasing per Intel Advisor.

Future work

Q18 (1p): Given more time, is there anything you would do to improve your models and/or the matrix multiplication application? What, how, why?

An improvement to the sequential execution of algorithm algorithm 1 could be applied by optimizing for for efficient cache usage. The current implementation has an compulsory miss for the reading of the second matrix. By switching the innermost two loops the resulting implementation is expected to have fewer cache misses based on row-major order.

An improvement is also possible for the tiled algorithm algorithm 2 to Layer the tile-tile product as rank-1 updates (transposed because of row-major ordering) instead of dot-products to take full advantage of the higher ISAs like AVX2 and FMAs [1].

While the baseline kernel was just underneath the DRAM bandwidth roof and optimising for better cache-reuse and locality should ideally have been a step in the right direction per the cache-aware roofline model, We might have been better off by starting the tiling from the L3 cache. So, the order of optimisations would look something like so.

1. Find the optimum loop ordering
2. Tile for caches, starting with the LLC
3. Design the microkernel around AVX2's broadcast and fmadd instructions

For the CUDA implementation we used a arbitrary size for the tiling: 16. It would be valuable to look into what the best tiling size would be to get a better performance.

Given a more in depth understanding of the device architecture, we could have been able to extend the creates models to predict an arithmetic intensity closer to NSight's value. The limited amount of time led to the current amount of detail in the model, this could be extended upon for more specific calculations.

References

[1] Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

[2] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2013.

[3] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.