NVIDIA Jetson Platform Characterization

Hassan Halawa, Hazem A. Abdelhafez$^{(\boxtimes)}$, Andrew Boktor, and Matei Ripeanu

The University of British Columbia, Vancouver, Canada
{hhalawa,hazem,boktor,matei}@ece.ubc.ca

Abstract. This study characterizes the NVIDIA Jetson TK1 and TX1 Platforms, both built on a NVIDIA Tegra System on Chip and combining a quad-core ARM CPU and an NVIDIA GPU. Their heterogeneous nature, as well as their wide operating frequency range, make it hard for application developers to reason about performance and determine which optimizations are worth pursuing. This paper attempts to inform developers' choices by characterizing the platforms' performance using Roofline models obtained through an empirical measurement-based approach as well as through a case study of a heterogeneous application (matrix multiplication). Our results highlight a difference of more than an order of magnitude in compute performance between the CPU and GPU on both platforms. Given that the CPU and GPU share the same memory bus, their Roofline models' *balance points* are also more than an order of magnitude apart. We also explore the impact of frequency scaling: build CPU and GPU Roofline profiles and characterize both platforms' balance point variation, power consumption, and performance per watt as frequency is scaled.

The characterization we provide can be used in two main ways. First, given an application, it can inform the choice and number of processing elements to use (i.e., CPU/GPU and number of cores) as well as the optimizations likely to lead to high performance gains. Secondly, this characterization indicates that developers can use frequency scaling to tune the Jetson Platform to suit the requirements of their applications. Third, given a required power/performance budget, application developers can identify the appropriate parameters to use to tune the Jetson platforms to their specific workload requirements. We expect that this optimization approach can lead to overall gains in performance and/or power efficiency without requiring application changes.

1 Introduction

Optimizing software based on the underlying platform is non-trivial. This is due to complex interactions between the application code, the compiler, and the underlying architecture. Typically it is difficult to reason about the achievable application performance and decide on the best potential optimizations to apply. The problem becomes much harder for heterogeneous systems consisting of multiple processing elements of different types each with unique properties that make them suitable for different kinds of computing patterns and optimizations.

Thus, figuring out how best to distribute work over the different processing elements, what the best optimizations are, and how to manage the communication overhead due to data transfer between the processors, represent key challenges for the development of efficient heterogeneous applications.

NVIDIA introduced their own embedded heterogeneous systems in the form of the Jetson TK1 (in 2014) and, recently, the TX1 (in 2015) Platforms. Two characteristics make those systems stand out compared to most commodity heterogeneous architectures. First, their integrated nature: the various computing elements share the same memory bus; and second the wide range (one order of magnitude) of frequency scaling. These characteristics, as well as their low power consumption, makes them a great choice for today's embedded applications and outline a possible future path for tomorrow's high-performance platforms. However, their performance characteristics are still not well understood and optimizing applications to make full use of their heterogeneous capabilities is non-trivial.

One method to characterize the performance of a platform is through a bound-and-bottleneck analysis. Such an analysis aims to provide simple and insightful performance bounds on applications. An example of such an approach is the *Roofline model* [10] which ties together peak computation capability (e.g., floating-point compute rate) and the memory bandwidth of a platform with the observed application performance. The Roofline model provides a visual guideline which can help explain the observed application performance, relate it to the peak performance obtainable, help determine whether an application is compute- or memory-bound, and guide the reasoning of which/whether further investment in performance optimization is worthwhile (Sect. 5).

This paper presents a performance characterization of the NVIDIA Jetson TK1 and TX1 based on the Roofline model. We use an empirical measurement-based approach[1] (Sect. 2) with the aim of achieving a more reliable characterization as opposed to merely relying on back-of-the-envelope calculations based on theoretical peak performance. Moreover, due to the complexity of the underlying hardware, the theoretical peak may not even be achievable in practice (e.g., due to power caps, as we demonstrate using our empirical Roofline profiles). Additionally, given the wide operating frequency range offered, we investigate and characterize the impact of frequency scaling on floating-point performance, power consumption, balance point, and performance per watt (Sect. 3). Finally we present our experience with tuning a matrix multiplication kernel, a crucial component of many scientific and HPC applications, for both platforms. Our aim is to provide application developers with the necessary data to allow them to tune the Jetson Platforms to suit the requirements of their applications. Such an optimization approach could lead to increases in performance and/or energy efficiency without requiring any application changes as we discuss in Sect. 6.

---

[1] Our benchmarks are available online at: https://bitbucket.org/nsl_europar17/benchmarks.

Table 1. Jetson TK1 and TX1 platform specifications as reported by the running OS and collected from relevant documentation such as [4,5]

	TK1 platform	TX1 platform
CPU	4+1-core 32-bit ARM Cortex-A15	4+4-core 64-bit ARM Cortex-A57
Architecture	ARMv7-A	ARMv8-A
L1/L2 cache	32 KB/512 KB	32 KB/2 MB
Main core frequency range	204 MHz → 2.3205 GHz	102 MHz → 1.734 GHz
Power-saving core frequency range	51 MHz → 1.092 GHz	N/A
Peak theoretical FLOPS	74.26 GFLOPS	55.48 GFLOPS
GPU	Kepler (192 CUDA Cores)	Maxwell (256 CUDA Cores)
L2 Cache	128 KB	256 KB
Frequency range	72 MHz → 852 MHz	76.8 MHz → 998.4 MHz
Peak theoretical FLOPS	327 GFLOP/s	511 GFLOP/s
DRAM	2 GB DDR3L RAM (933 MHz, 2 Channels)	4 GB LPDDR4 RAM (1.6 GHz, 2 Channels)
Data bus width	64 bit	64 bit
Peak theoretical bandwidth	14.93 GB/s	25.6 GB/s

2 Methodology

Table 1 presents in detail the TK1 and TX1 specifications. To construct the empirical Roofline profiles we developed two approaches: the first, similar to that proposed in [7], relies on hardware counters (available for the CPUs only), while the second obtains the same level of information for the GPUs by using microbenchmarking techniques similar to those used by Wong et al. [11]. We rely on an empirical measurement-based approach as opposed to one just based on peak performance calculations for two main reasons. First, the presented empirical results arguably provide a more realistic, and reliable characterization. Secondly, the theoretical peaks may not even be achievable in practice due to various complexities in the underlying hardware and power caps. We demonstrate this disparity between the two approaches as part of our evaluation in Sect. 3.

2.1 CPU Micro-Benchmarks

The ARM Cortex A15 and A57 both support various types of floating-point operations including fused multiply-add (FMA) and SIMD operations. Typical scalar floating-point operations are handled by the ARM core's Vector Floating-Point (VFP) unit whereas vector floating-point operations (SIMD) are handled

by the ARM core's NEON 128 bit SIMD engine. The VFP and NEON units share the same floating-point registers.

We designed a micro-benchmark in assembly that can operate with a user-defined variable operational intensity (i.e., a variable ratio between floating-point and memory operations). This micro-benchmark mixes SIMD double-word store instructions (using an unrolled loop of store instructions operating on 2 single-precision floating-point operands) as well as SIMD floating-point operations (using the NEON SIMD unit to compute an FMA of 4 single-precision floating-point operands). By varying the ratio between the store instructions and the SIMD floating-point instructions, we vary the operational intensity of the micro-benchmark to generate the Roofline profiles.

The Cortex-A15 and Cortex-A57 include a Performance Monitor Unit (PMU) that provides access to six hardware counters. We used the counters to estimate: the number of Floating-Point Instructions, the number of Vector (SIMD) Instructions, and the number of Loads and Stores. We then used this to derive the rate of Floating-Point Operations (FLOPS) as well as the memory bandwidth.

2.2 GPU Micro-Benchmarks

To obtain the Roofline profile for the GPU, where no hardware counters are available, we developed a benchmark with variable compute intensity (from 0.125 FLOPs/Byte to 1024 FLOPs/Byte). The benchmark loads 3 values from memory, performs a variable number of FMA operations on them, then stores the results. We disassembled the binary and verified that the benchmark contains exactly the intended number of memory and FMA operations.

We built two additional benchmarks: a memory bandwidth benchmark and a *FLOPS* benchmark. The memory bandwidth benchmark performs a vector add operation and is completely memory-bound. We use this benchmark to compute the memory bandwidth for the GPU and compare it to a similar benchmark on the CPU since memory is shared. The *FLOPS* benchmark is a high compute intensity benchmark, performing around a hundred thousand FMA instructions for each 4 float values loaded from memory. We use it to estimate the maximum achievable FLOPS. The results of those two benchmarks validate our variable intensity benchmark which approaches those asymptotes but does not cross them.

2.3 Other Methodology Notes

Scaling. The Jetson platforms offer a wide range of configuration options that include: the number of operational CPU cores, the CPU cores' operating frequency, as well as the number of application threads to execute. Each configuration point changes the performance characteristics in terms of single-precision floating-point performance, memory bandwidth, and power consumption.

Power Consumption. We use a Watts Up? PRO Power Meter [2] connected via USB to the Jetson development boards. This allows us to collect power

consumption statistics (at 1 Hz) during our benchmarks. As a baseline for the subsequent power consumption measurements, we measured the idle power consumption with the default dynamic frequency scaling and power-saving profiles. The idle power draw was ≈3.1 W for TK1 and ≈3.8 W for TX1.

3   Platform Characterization

3.1   CPU Characterization

***Roofline Profiles.*** Figure 1 shows the theoretical peak as well as the measured Roofline profiles for both platforms (at peak frequency and with the number of application threads equal to the number of cores). For the TK1, the maximum single-precision FLOPS rate achieved was 73.02 GFLOPS. This represents ≈98.3% of the theoretical peak and defines the upper limit for a compute-bound application. The maximum observed memory bandwidth was 13.72 GByte/Sec which represents ≈91.9% of the theoretical peak and defines the upper bound for memory-bound applications. The intersection of the two bounds defines the *balance point* (i.e., the point where an application spends the same amount of time fetching data from memory and computing on it) at 5.32 FLOPS/Byte. For the TX1, the maximum FLOPS rate achieved was 54.31 GFLOPS (≈97.8% of the theoretical peak), the maximum observed memory bandwidth was 20.2 GB/s (≈78.9% of the theoretical peak), and the balance point is at 2.68 FLOPS/Byte.

Figure 1 highlights that, on the one hand, the measured peak TX1 FLOPS rate is lower than that of the TK1 (by ≈25.6%). This can be attributed to its lower maximum frequency while using the same 128-bit wide SIMD engine. On the other hand, the measured peak TX1 memory bandwidth is higher than that for the TK1 (by ≈47.2%). This difference is caused by the higher DRAM frequency for the TX1 (1.6 GHz) compared to the TK1 (933 MHz) given that both utilize dual channel DRAM with the same 64-bit wide data bus. These differences cause the balance point to shift from 5.32 FLOPS/Byte for the TK1 to 2.68 FLOPS/Byte for the TX1. Thus, the TX1 CPU is more suitable for memory-bound applications (given its higher memory bandwidth and lower balance point).

The Impact of Frequency Scaling on Power Consumption and Roofline Profiles. We investigated the performance characteristics of both platforms for all CPU frequency scaling configurations. Due to space constraints we present only a subset of the results for the TK1 (Fig. 2). There are two important observations: firstly, frequency scaling has a larger impact on the FLOPS rate achieved than on memory bandwidth, and, secondly, the platform has a large dynamic power range (3.2x from 3.4 W to 10.8W). Using the power-saving core (labelled **0c** in the figure) increases the power range to 3.6x. It can also be observed that the power-saving core does not offer a good power/performance trade-off: when running the TK1's power-saving core at 204 MHz compared to all 4 high-performance cores at the same frequency, a power-saving of ≈11.7% is achieved but at the cost of a ≈78.4% decrease in performance.
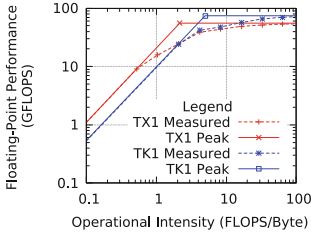
Fig. 1. CPU Roofline profiles: theoretical peak and measured CPU performance for the TK1 (blue) and TX1 (red). (Color figure online)
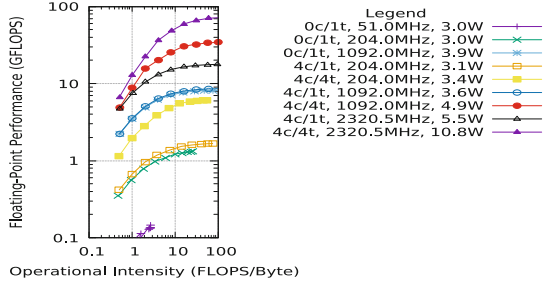
Fig. 2. TK1 Roofline profiles for the power-saving core (labelled **0c**) and *all* normal cores (labelled **4c**). We also vary the number of threads (labels **1t** vs. **4t**). Each line label includes measured power consumption.

The Impact of Frequency Scaling on the Balance Point. For each frequency, we compute the balance point. Surprisingly, in Fig. 3, we observe that the TK1 can be configured to cover a wider range of balance points, a potentially useful feature when attempting to match hardware capabilities to the application demand as we discuss in Sect. 6.

The Impact of Frequency Scaling on Power-Normalized Computational Rate (FLOPS/Watt). Figure 4 examines the power consumption (left y-axis) and performance per watt (right y-axis) for all possible CPU frequencies for a compute intensive application. It can be seen that the TK1 typically consumes less power across the entire frequency range compared to the TX1. This, in addition to its higher floating-point performance, results in a higher energy efficiency (performance per watt) across the entire frequency range.
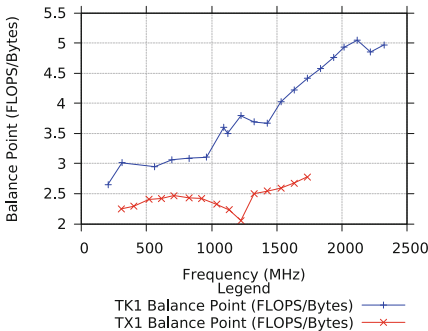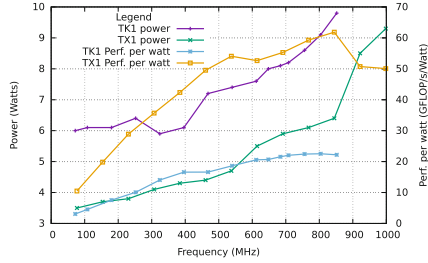

Fig. 3. The impact of frequency scaling on the balance point (4 cores, 4 threads) for TK1 (blue) and TX1 (red). (Color figure online)
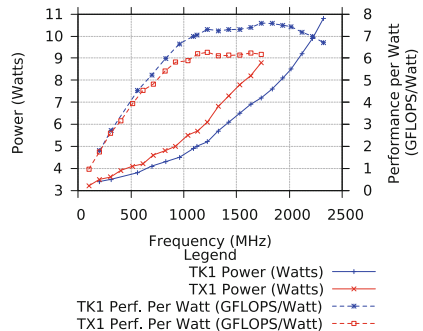
Fig. 4. Power and performance per Watt while scaling CPU frequency (4 cores, 4 threads) for TK1 (blue) and TX1 (red). (Color figure online)
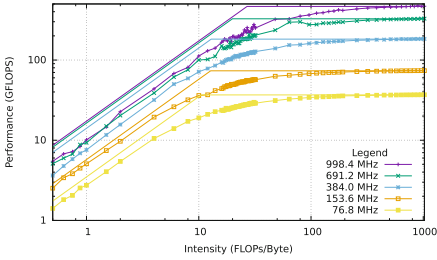
**Fig. 5.** TX1 Maxwell GPU Roofline: theoretical (solid line) and achieved (dots). (Color figure online)

Fig. 6. GPU power and performance per watt for a compute-bound benchmark.

3.2 GPU Characterization

Roofline Profiles. Figure 5 shows the Roofline profiles constructed for the TX1 GPU (TK1 profiles are similar, not shown here to conserve space). We find that, in some cases, the theoretical peak is unattainable even with highly tuned benchmarks. We therefore use our benchmarks to estimate the Roofline bounds (solid lines in the figures). The Kepler GPU on the TK1 is able to achieve up to 218 GFLOP/s while the Maxwell GPU on the TX1 achieves 465 GFLOP/s. The memory bandwidth is at 12 GB/s and 17.3 GB/s respectively (similar to the CPU results). For better readability, we only show a subset of the frequencies.

The Impact of Frequency Scaling: Power-Normalized Computational Rate (FLOPS/Watt). NVIDIA focused on power efficiency when designing the TX1's Maxwell GPU. Our results confirm this: TX1 provides 3x higher performance per watt compared to TK1 (Fig. 6). Note the impressive 6x (TK1) and 20x (TX1) higher FLOPS/Watt for GPUs compared to CPUs.

The Impact of Frequency Scaling on the GPU Balance Point, Peak FLOPS Rate, and Peak Bandwidth. We observe similar behaviors on both platforms in Fig. 7: at low GPU frequencies, both memory and compute bandwidth increase linearly as the frequency is scaled. The memory bandwidth at
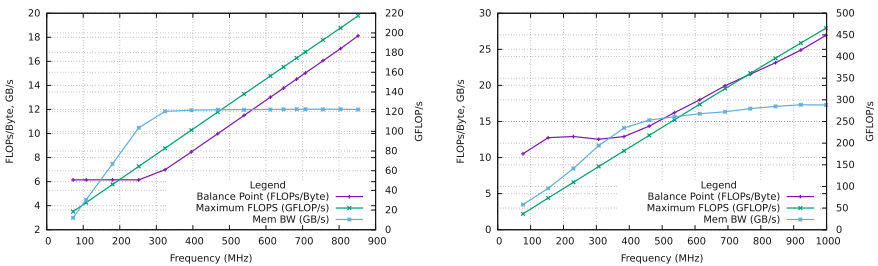
Fig. 7. The impact of frequency scaling on the TK1 (left) and TX1 (right) balance point (left y-axis), peak bandwidth (left y-axis), and peak FLOPS rate (right y-axis). Note the different scales on the y-axes.

low frequencies is bottlenecked by the ability of the processor to issue instructions fast enough to saturate the memory bus. After $\approx$300–400 MHz, the peak FLOPS rate continues to increase linearly with frequency, however, the memory bandwidth stops increasing linearly and becomes constant. As such, the balance point is constant for the lower frequencies while for higher frequencies the balance point increases linearly as the frequency is scaled.

It is worth noting that the platforms offer a different range for the balance point: on the TK1, the GPU balance point ranges between 6 and 18 FLOPs/Byte while on the TX1 it ranges from 11 to 27. This is largely due to the superior performance of the Maxwell architecture. In contrast with the CPU results, the TK1's Kepler GPU can be better tuned for applications with lower intensity while the TX1's Maxwell GPU can be better tuned for applications with higher intensity. However, due to the TX1's higher memory bandwidth and higher compute rate we find that its GPU provides equal or better performance regardless of the application's arithmetic intensity. In other words, the TX1's GPU provides better performance than that on the TK1 for all intensities, despite the fact that it is operating sub-optimally at the lower ones.

4   Case Study: Matrix Multiplication

To study the effect of frequency scaling on performance/energy beyond microbenchmarks, we developed a tuned matrix multiplication kernel that can be run on the CPU only, the GPU only, or be partitioned on the heterogeneous platform. We chose matrix multiplication as it is an operation that is essential in many scientific and HPC applications.

Our CPU implementation is based on the OpenBLAS matrix multiplication single precision subroutine (SGEMM). For OpenBLAS, we enable optimization flags for the ARMV8 architecture to support the advanced features present in the processor. This enables the use of NEON SIMD instructions for performing vector floating point operations. The library also implements matrix tiling (blocking) optimized for the multi-level caches of each processor and loop unrolling. For the NVIDIA GPUs we use the cuBLAS [6] library, a BLAS implementation optimized for NVIDIA GPUs. Our developed heterogeneous matrix multiplication kernel can partition the matrices between the CPU and the GPU to take advantage of the optimizations by OpenBLAS and cuBLAS.

Collecting Results: In each run we measure the performance in terms of FLOPS and power consumed. The number of floating point operations in a matrix multiplication routine is independent of the underlying hardware or the processor used. Assuming square matrices of size $n \times n$, then matrix multiplication generates $2 \times n^3$ FLOPS. We measure the running time which we then use to obtain the computation rate (GFLOPS).

Configurations: We used different matrix sizes to cover the fit-in-cache and non-fit in cache cases, this shows the effect of matrix tiling (blocking) on performance. In the heterogeneous case, for space limitations, we selected the lowest

and highest frequencies for each of the CPU and GPU. Moreover, we fixed the matrix size to 4096 and limited the *cpuColumns* to be a value out of (16, 128, 512) columns. From our experience, increasing the *cpuColumns* to more than 512 degrades the performance as the computation is heavily unbalanced between the CPU and the GPU.

CPU/GPU Only Experiments: Figures 8 and 9 present the energy efficiency (GFLOPS/Watt) while scaling frequency. The figures show a sample of the results that covers fit-in-cache (matrix sizes 32 and 64) and non-fit in cache (matrix sizes 128, 512 and 4096) cases. We note that efficiency saturates (or even decreases) above 1428 MHz for the CPU and 691 MHz for the GPU: scaling up frequency, while improving runtime, leads to higher power consumption and thus lower energy efficiency.

Heterogeneous (CPU and GPU): The CUDA toolkit v8.0, deployed on the TX1, offers multiple software-level mechanisms to use the shared global physical memory between the CPU and the GPU such as the Unified Memory Architecture (UMA). Using UMA allows us to allocate and initialize the matrix on the shared memory with no need for explicit memory transfers. At first glance, one would think that with UMA and shared memory, an efficient heterogeneous matrix multiplication can be easily implemented. However, NVIDIA's documentation [1, J2.2] states that for GPUs with compute capability less than 6.x (the TX1's is 5.3), it is not possible for the CPU and GPU to access (read or write) a memory location allocated using UMA simultaneously.[2]

After evaluating the alternatives, we settled on the following solution: to compute $A \times B = C$, we split matrices B and C by columns. We specify the number of columns that the CPU will process as *cpuColumns* and the rest is processed on the GPU. The allocation is performed using UMA to avoid copying
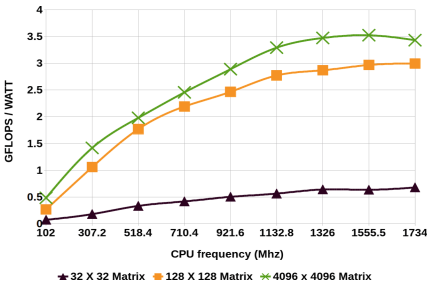

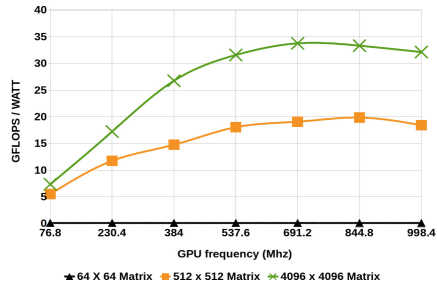**Fig. 8.** CPU power efficiency values for different matrix sizes

Fig. 9. GPU power efficiency for different matrix sizes

---

[2] We tried several alternative techniques such as using *mprotect()* which changes memory access permissions on a specific memory range. The NVIDIA driver locks the memory accessed by the GPU kernels until they complete. Therefore, it is not possible to have a shared matrix object accessed at the same time by the CPU and GPU even when we use UMA, even if all accesses are read-only.
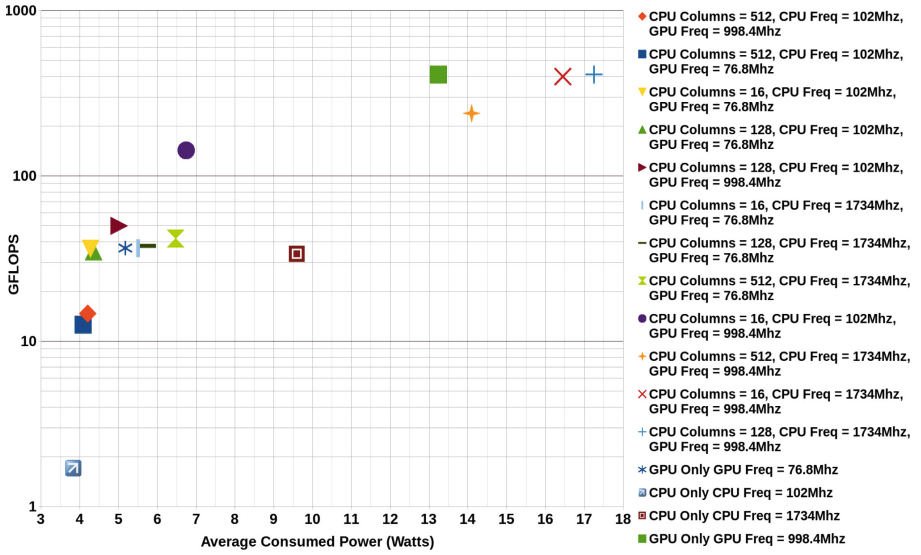
Fig. 10. Run-time configurations effect on performance and power consumption

the results back from the GPU. With this approach, we eliminated any write conflict between the GPU and the CPU on the shared memory, at the cost of wasting memory by duplicating matrix A (imposed by the limited compute capability).

Figure 10, plots a chart that highlights the value of the configurable frequency scaling of the TX1. The figure highlights the multiple criteria that can be used to select a configuration (CPU/GPU frequencies, cpuColumns values). For example, in situations where capping power is the limiting factor one can determine the best frequency configuration that meets the power cap. Alternatively, in situations where meeting a runtime constraint is important, one can select the most energy efficient configuration that meets the imposed runtime deadline. It is worth noting that that the observed operational space varies over a wide range along the performance and power dimensions (over one order of magnitude on performance and 4x on power).

5   Related Work

The Roofline Model [10], is a visual model that makes it easier to reason about bounds to attainable performance. It combines the operational intensity (FLOPS per Byte), floating-point performance (FLOPS) and memory bandwidth (Bytes per Second) together into a two-dimensional plot that outlines the performance bounds of the platform (defined by a Roofline *profile* which acts as an envelope). Moreover, such a bound-and-bottleneck characterization approach provides insights into the primary factors affecting the performance of individual applications based on their *position* on the plot with respect to the Roofline

profile (e.g., whether they are compute-bound or memory-bound depending on their operational intensities) and thus allows application developers to prioritize which optimizations to pursue to improve performance.

Roofline Model Uses. Ofenbeck et al. [7] take a practical approach to applying the Roofline model. They use measured data based on benchmarks to reason about the performance in a way similar to the methodology we employ to generate our CPU Roofline. However, this study focuses on an Intel architecture with access to fine-grained event counters through the Performance Monitoring Unit. Wong et al. [11] use a carefully crafted set of benchmarks to discover the microarchitecture of GPUs. Their approach is to craft microbenchmarks that amplify the different microarchitecture parameters and make them visible at runtime to uncover detailed information about GPU internals. We use similar benchmark design to compute the GPU Roofline. Lo et al. [3] developed the Empirical Roofline Tool and use it to empirically construct Roofline models for a variety of accelerated architectures (including multicore, manycore, and GPU-accelerated architectures). The toolkit makes use of instrumented microbenchmarks implemented in MPI, OpenMP, and CUDA. Our methodology is similar to that used in the Empirical Roofline Tool to construct the GPU Roofline, however, we rely on hardware performance counters to generate the CPU Roofline.

NVIDIA Jetson Platform Characterization. Although there are many applications that use the unique capabilities of the studied platforms (particularly TK1 such as [8,9]), to the best of our knowledge, we are the first to carry out a complete characterization. In [8], the authors employ an application: a distributed MPI-based neural network simulation, to compare a distributed embedded platform (based on several interconnected TK1s) with a server platform (based on an Intel quad-core dual socket system). The authors show that the distributed embedded platform's instantaneous power consumption is 14.4x lower despite the server platform being 3.3x faster (in terms of execution time). Another study [9] evaluates the TK1 in an HPC context as a cloud offload unit for a discrete Tesla K40 GPU. The study shows that such a cluster approach offers superior power efficiency compared to using a separate discreet GPU, while offering substantially better performance than using the TK1 by itself.

6   Summary and Discussion

We characterized the performance of the NVIDIA Jetson TK1 and TX1 Platforms by presenting Roofline profiles for both the CPU and the GPU on each platform. When comparing the CPU vs. GPU performance, our Roofline profiles showed a difference of more than an order of magnitude on compute performance suggesting that the GPU on the Jetson Platforms is preferable for compute intensive applications. Since the CPU and GPU share the same memory bandwidth, the balance points are also more than an order of magnitude apart. Additionally, we explored the impact of frequency scaling on floating-point performance, balance point, power consumption, and efficiency (GFLOPS/watt).

The data provided by this study offers application developers a starting point when tuning the platforms to their applications' requirements (by choosing the optimal operational frequency on the CPU/GPU) and indicates that net gains in performance and/or power efficiency without any modifications to the applications can be obtained.

We discuss below the key implications of our observations for application developers and device manufacturers.

6.1 Implications for Application Developers

Modular Application Design and the Division of Work. The asymmetric nature of the CPU and GPU can be harnessed by application developers during runtime for optimum performance and energy efficient computing. Highly parallelizable portions of the application are more suited for deployment on the GPU with its larger number of SIMD units while the less parallelizable parts (or those requiring a more complex processor pipeline) can be executed on the CPU. As such, developers should design their applications in a modular way so as to allow for the efficient distribution of work across the available asymmetric cores.

A Free Lunch? Reducing Power Consumption without Performance Degradation. If an application's computational intensity is below the platform's balance point, then the application could potentially be able to save energy without sacrificing performance by scaling the frequency down until the balance point is equal to its required intensity. This works only down to the point where the system becomes bottle-necked on the memory bandwidth. Beyond this point, further reduction of frequency will result in performance degradation.

Tuning the Platform, an Alternative Method to Optimize Application Performance. Typically application developers apply various optimizations to their code in order to try to attain the maximum performance possible on the target platform. We propose that application developers can alternatively tune the platform to the operational intensity of the developed application in order to achieve optimum compute performance and/or power-saving. One way to do this is for application developers to statically determine the appropriate frequency scaling for their applications and then set the CPU/GPU to this frequency at runtime. Our analysis suggests that application developers can tune the Jetson Platforms to the applications' requirements to achieve net gains in application performance and/or energy efficiency without the need to modify the application itself.

Full System Power. In addition to the other benefits provided by the shared memory architecture on the Jetson platform, we find that the emphasis on low full system power has important implications. The platform was designed for embedded applications, thus, emphasis was placed on optimizing its idle power as well as the power consumed by supporting components. At idle, we find that both platforms consume less than 3W. This is negligible when compared to traditional machines that host a CPU and/or a GPU. As a result, under load,

close to all of the power consumed goes to the active components (CPU, GPU and DRAM), and makes the full system's power efficiency much better compared to other platforms.

6.2   Implications for Device Manufacturers

Simplify Application Development. In order to increase the adoption of heterogeneous compute platforms, device manufacturers should focus on simplifying the application development process as well as the tools available to developers. The promise of such platforms is higher performance and better energy efficiency but, in our experience, this potential is not currently attainable without significant effort by application developers.

This is currently a major drawback of the NVIDIA Jetson platform. In order to provide implementations optimized for the CPU and the GPU, application developers need to rewrite their applications specifically for each processor. For the CPU, almost any general purpose programming language can be utilized but typically a low-level programming language such as C or Assembly is used to extract the maximum performance possible. While for the GPU, CUDA must be used in order to make full use of the features and libraries provided by NVIDIA. There is little reuse of application code between optimized CPU and GPU implementations with this development approach. The significant development effort and costs involved represent a high barrier to entry.

Better Dynamic Frequency Scaling. In theory, manufacturers could potentially instrument the hardware to dynamically estimate the running application's arithmetic intensity. Based on the computed intensity, the device can apply dynamic frequency scaling to reduce the consumed power even under 100% utilization. A good guess for the frequency that would work best can be based on trying to match the hardware balance point with the application's intensity. Following such an approach could potentially lead to reduced power consumption as well as increased performance per watt without any changes to the running applications.

Memory Bandwidth at Lower Frequencies. Based on our findings, the memory bandwidth becomes a performance bottleneck at lower operating frequencies. Device manufacturers can try to avoid this bottleneck by designing the hardware to support the full memory bandwidth even at the lowest frequencies. This would allow a wider range of achievable balance points and, in turn, lead to larger power-savings for applications with low computational intensity.

References

1. NVIDIA   CUDA   toolkit   v8.0:   https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd. Accessed 16 Feb 2017

2. Watts-up: https://www.wattsupmeters.com/. Accessed 23 Aug 2016
3. Lo, Y.J., Williams, S., Van Straalen, B., Ligocki, T.J., Cordery, M.J., Wright, N.J., Hall, M.W., Oliker, L.: Roofline model toolkit: a practical tool for architectural and program analysis. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 129–148. Springer, Cham (2015). doi:10.1007/978-3-319-17248-4_7
4. NVIDIA: Technical brief NVIDIA Jetson TK1 development kit: bringing GPU-accelerated computing to embedded systems. Technical report, April 2014
5. NVIDIA: Tegra X1: NVIDIA's new mobile superchip. Technical report, January 2015
6. NVIDIA: CUBLAS library. Technical report, September 2016
7. Ofenbeck, G., et al.: Applying the Roofline model. In: ISPASS 2014, pp. 76–85, March 2014
8. Paolucci, P.S., et al.: Power, energy and speed of embedded and server multi-cores applied to distributed simulation of spiking neural networks: ARM in NVIDIA Tegra vs Intel Xeon quad-cores. CoRR abs/1505.03015 (2015)
9. Ukidave, Y., et al.: Performance of the NVIDIA Jetson TK1 in HPC. In: 2015 IEEE International Conference on Cluster Computing (CLUSTER), pp. 533–534, September 2015
10. Williams, S., et al.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
11. Wong, H., et al.: Demystifying GPU microarchitecture through microbenchmarking. In: ISPASS 2010, pp. 235–246. IEEE (2010)