Gables: A Roofline Model for Mobile SoCs

Mark D. Hill*
Computer Sciences Department
University of Wisconsin—Madison
markhill@cs.wisc.edu

Vijay Janapa Reddi*
School Of Engineering And Applied Sciences
Harvard University
vj@eecs.harvard.edu

***Abstract*—Over a billion mobile consumer system-on-chip (SoC) chipsets ship each year. Of these, the mobile consumer market undoubtedly involving smartphones has a significant market share. Most modern smartphones comprise of advanced SoC architectures that are made up of multiple cores, GPS, and many different programmable and fixed-function accelerators connected via a complex hierarchy of interconnects with the goal of running a dozen or more critical software usecases under strict power, thermal and energy constraints. The steadily growing complexity of a modern SoC challenges hardware computer architects on how best to do early stage ideation. Late SoC design typically relies on detailed full-system simulation once the hardware is specified and accelerator software is written or ported. However, early-stage SoC design must often select accelerators before a single line of software is written. To help frame SoC thinking and guide early stage mobile SoC design, in this paper we contribute the Gables model that refines and retargets the Roofline model—designed originally for the performance and bandwidth limits of a multicore chip—to model each accelerator on a SoC, to apportion work concurrently among different accelerators (justified by our usecase analysis), and calculate a SoC performance upper bound. We evaluate the Gables model with an existing SoC and develop several extensions that allow Gables to inform early stage mobile SoC design.**

***Index Terms*—Accelerator architectures, Mobile computing, Processor architecture, System-on-Chip**

I. Introduction

Processor architecture continues to evolve from single-core microprocessors to multicore chips to complex systems on a chip (SoCs). The last transition is driven by both a technology push and an application pull. The push is the end of Dennard Scaling and slowing of Moore Law that encourages using accelerators that are more energy-efficient than CPU cores at some computations. The pull is application trends toward computations that are both "special purpose" and ubiquitous from video processing to deep neural network processing.

Nowhere is the trend toward heterogeneous SoCs more evident than in consumer electronics, such as SoCs for smartphones. On the one hand, over twenty billion SoC chips are shipped each year [1]. In 1934, Thomas J. Watson Sr.'s 1934 predicted the need for five computers for the world; today the need is closer to five per person. On the other hand, these SoCs have grown into powerful computer systems with multiple cores, GPUs, and many accelerators—often called intellectual property (IP) blocks, driven by the need for high performance in severely constrained battery and thermal power envelopes, and all-day battery life. These cores and IPs interact via rich interconnection networks, caches, coherence, 64-bit address spaces, virtual memory, and virtualization. Hence, consumer SoCs deserve the architecture community's attention.

Unlike the server marketplace, which is only now transitioning from CPUs and GPUs to add more special-purpose accelerators (e.g., tensor processing units [2]), consumer SoCs have long thrived on tight integration and extreme heterogeneity. A typical mobile SoC includes a camera image signal processor (ISP) for high-frame-rate, low-latency camera, a digital signal processor (DSP) for on-device AI/VR/AR, a high-speed modem for LTE and WiFi connectivity with diverse set protocols, and video encoders and decoders for video playback and video capture, etc. These specialized processing engines deliver an order of magnitude improvement in performance and power efficiency compared to the general-purpose application processor (AP). As a result, in most mobile SoCs, the AP occupies only between 15 to 30 percent of the total chip area [3], [4]. The rest of the area is dedicated to these specialized processing engines, and as a direct result mobile devices are capable of delivering desktop PC-like experiences under a tight 3 Watt thermal design point constraint [5].

While IP diversity and heterogeneity in a SoC provide a healthy dose of alternatives for the system integrator (i.e., a company integrating the SoC into a phone), it also presents them with several unique challenges. On the one hand, SoC designers struggle to determine what architectural design features are most likely to be useful in a future SoC—this must be done three years ahead of time. On the other hand, end-users (i.e., application designers) need to evaluate several different trade-offs between the different SoCs to determine which SoC best suits their performance, power and cost targets.

Addressing either of the challenges is not easy. There is no industry standard way to characterize, evaluate and compare mobile SoCs. Today's most widely used SoC performance evaluation tools are based on post silicon benchmarking, using tools such as Geekbench [6] and AnTuTu [7], which fall short of what is desired. These benchmarks focus on the CPU and memory subsystem and mostly ignore the rest of the SoC components. Therefore, they fall short in characterizing the rich heterogeneous capabilities of a SoC. Furthermore, existing benchmarking tools are geared toward post-silicon measurement and testing, not pre-silicon exploration.

*This work was conceived and largely done while the authors were embedded in a Google consumer product group as research "interns."

Designing, creating and selecting computer systems requires principled methods, and consumer mobile SoCs are no different. The primary mode in computer architecture has been the use of cycle-level simulation, and we expect this to continue in the later stages of SoC design. However, the early stages of SoC design have questions like, "Which IPs should my SoC include and roughly how big?" Also, SoC customers face a challenge of down-selecting among alternative already-designed SoCs with different IPs of different sizes.

It is not practical to use cycle-level simulation for the early-stage questions due to the substantial cost of "porting" SoC workloads among alternatives. Today, a consumer SoC must enable 10-20 important "usecases"—like making a phone call or watching a movie—to all run acceptably well. The average is immaterial. Moving current usecases among alternative designs is expensive and time-consuming as targeting different IPs is closer to re-writing aspects of code than to "porting." Moreover, one must plan for future usecases 2-3 years in advance of when the SoC is deployed in a product and when the software may not be completely be written. With cycle-level simulation not available for early decisions, one can turn to intuition, a few SoC parameters, or—as we advocate—using a SoC and workload usecase parameters to drive a model.

To facilitate early stage SoC design or selection, we repurpose the Roofline performance model [8]. Roofline applies bottleneck analysis [9] to a processor chip, originally a multicore chip. It models the chip hardware with peak computation performance (P_{peak}) and peak off-chip memory bandwidth (B_{peak}), and models software with operational intensity (I) that is the average number of operations (e.g., floating-point) per off-chip byte transferred to/from memory. Operational intensity reminds us that data reuse is critical to managing bandwidth use.[1] Figure 1 gives a plot of log maximum attainable performance (y-axis) that is upper-bounded by bandwidth (angled line) or peak performance (horizontal line) depending on software's log operational intensity (x-axis). Roofline plots can add *ceilings* that are the lesser bounds due to restrictions.

The Roofline model has not been applied to consumer SoCs. Part of the challenge is defining overall peak performance for a SoC with many different accelerators, rather than a single homogeneous multicore chip. For this reason, we instead apply Roofline to individual SoC accelerators where we see it as most appropriate. We then contribute the Gables SoC model, which is a generalization of Roofline's bottleneck analysis. In its base form, Gables targets a SoC for N IP blocks (e.g., CPU complex and accelerators) that can operate in parallel with each other and memory transfers. Gables models SoC hardware with a roofline for each IP and an equation of their shared memory bandwidth demand. Gables models a workload usecase with work fraction (like Amdahl's Law) and operational intensity at each IP and can then calculate the usecase's maximal attainable performance on the SoC. We evaluate Gables using a micro-benchmark on modern SoCs.

[1]Without reuse, double-precision multiply-accumulate can have operational intensity as low as $0.063 = \frac{1}{16} = \frac{2\ operations}{4*8\ bytes}$.

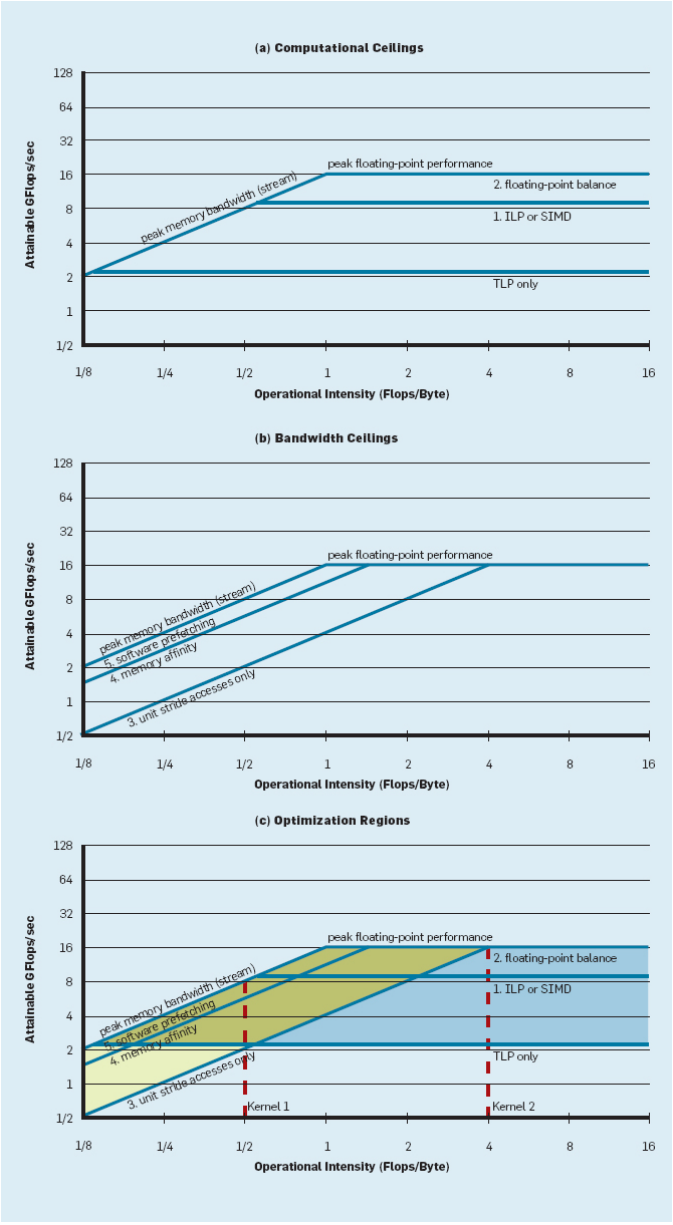
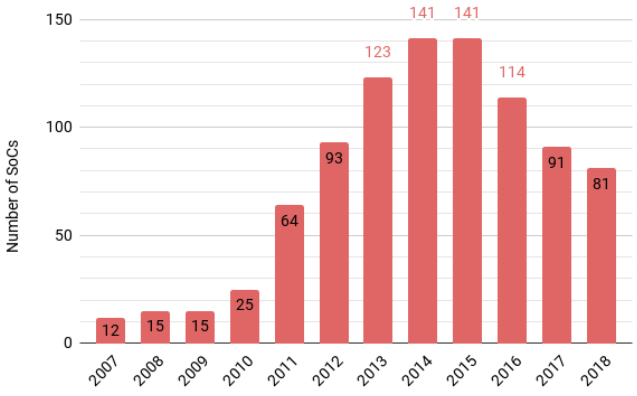


Figure 1: Roofline model [8]. Reprinted with permission.

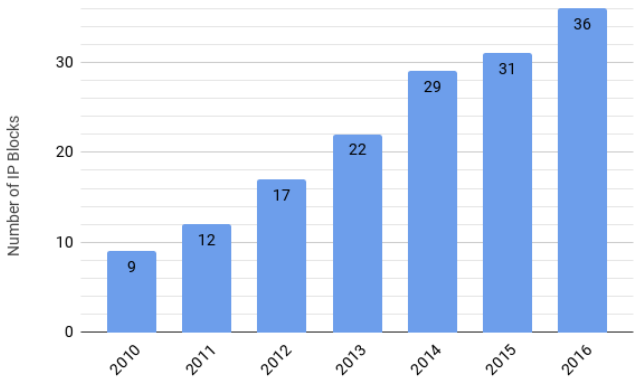
We contribute extensions of the base Gables model that are valuable in their own right and illustrate that many other extensions are possible. We develop extensions for a memory-side memory/scratchpad/cache and interconnect topologies and usecases where work is serialized among IPs. In addition, we provide an open-source Gables mobile app and an interactive visualization tool to facilitate deeper understanding [10].

In summary, we make the following contributions:

- We describe a real-world problem facing the industry, specifically in designing SoCs for future systems.
- We present the Gables performance model for complex SoC design and early stage design space exploration.
- We show how the insights derived from Gables apply to commercially available mobile SoCs.



(a) Total number of SoC chipsets found "in the wild."



(b) Increasing level of on-die heterogeneity for a modern SoC.

Figure 2: (a) We mined the data for the mobile chipsets from GSM arena [11], which encompasses over 9165 phone models and 109 different device brands. (b) The data for the number of IP blocks is based on the findings by Shao et al. [4].

II. Mobile SoC Background

Most modern SoCs, such as those found in smartphones, are costly and complex systems that contain multiple IPs. Each year, new SoCs, with growing complexity, are released into the consumer market to keep pace with ambitious user demands and expectations. As a result, SoCs have thrived over the past decade. Figure 2a shows the number of new SoCs that have been introduced into the market each year since 2007. Over the course of nearly ten years, SoC vendors have sought to identify critical differentiating factors that set their architecture apart from their competition. This has led to the healthy diversification of chipsets in the SoC consumer market with some recently evident consolidation that caused a decline following 2015.[2] At the same time, it has also led to growing chip complexity. Figure 2b shows the estimated number of IPs in a state of the art SoC over the past several generations. The number of IPs has steadily climbed to over 30 IPs. To better understand the Gables performance model that we present later, we next give background into mobile SoC hardware architecture and application software usecases.

A. SoC Architecture

Figure 3 shows an example block diagram of a typical modern SoC. Unsurprisingly, it consists of a CPU (AP) and GPU cluster complex. Most modern smartphones rely on asymmetric processing with big out-of-order cores handling user facing and computationally demanding tasks, while the little in-order cores handle background processing tasks, such as checking email and triggering notifications. Many of the SoCs contain only big and small CPUs, but, as of late, MediaTek introduced a new "middle" tier [12], creating three levels of the heterogeneity within the CPU complex alone.

In addition to the CPU and GPU, there are a plethora of other processing engines in a mobile SoC. In fact, the mobile CPU complex only takes about 25 to 30% of the total die area [4], [3], and the rest of it is dedicated to these specialized processing engines. Figure 3 includes a few to illustrate the point. Briefly, many of these IPs are responsible for doing the heavy lifting under computationally intensive scenarios.

The key to the success of the IPs is that they provide significant horsepower at relatively low power consumption compared to a CPU. For instance, the Pixel Visual Core (IPU) from Google [13] in a Pixel 3 smartphone is designed to accelerate High-Dynamic-Range (HDR) processing in camera applications. HDR is a computationally intensive task for extending the range of luminosity rather than what is possible using standard image processing techniques. The IPU is an 8-core processor (by itself) that can perform three trillion operations per second per core, allowing it to do HDR+ processing 5X faster than the main application processor at one-tenth of the power. Similarly, the Qualcomm Hexagon Digital Signal Processor (DSP) is a power-house for machine learning intelligence and computer vision (8X faster than the CPU, and 4X more quickly than the GPU) at nearly 8X and 25X more energy efficient than the CPU and GPU respectively [14].

Many, if not all of these hardware IPs, are clustered into a hierarchical network fabric(s). In Figure 3, we show multiple network fabrics, each with its speeds and feeds. Depending on the IPs bandwidth and latency requirements, the IPs are clustered together within different fabric hierarchy levels.

B. Usecases

Applications in the SoC world are commonly referred to as "usecases." These application usecases are best represented as

[2]Based upon further analysis of the data in Figure 2a we postulate that the drop in the number of SoCs following 2015 is due to many companies dropping out of the competitive, low-margin mobile consumer market. For instance, Texas Instruments (TI) stopped producing the OMAP processor line and Intel departed from the consumer smartphone market. Our analysis also causes us to postulate that vendors consolidated their chipset offerings to manage the complexity of supporting diverse offerings. For instance, in 2014, there were 49 Qualcomm chipsets, whereas, in 2017, there were only 27.

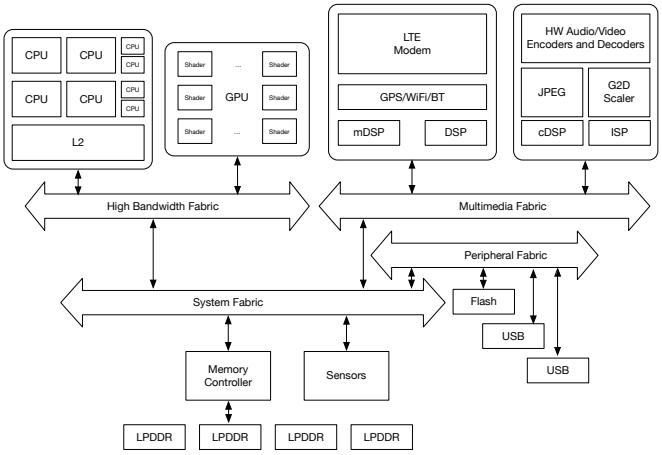


Figure 3: An example diagram of a mobile SoC, showing different compute engines that are clustered together across different interconnect fabrics. Picture is not drawn to scale.

application-level data flows from sensors to the processing engines. Figure 4 shows an example usecase for stream internet content over WiFi. The general usecase flow for this scenario is as such: IP packets streaming over WiFi are processed into a user/application-level buffer into insecure memory. The audio and video streams are separated, decrypted by the crypto block (or the CPU), and stored in secure memory. Both the video decoder and audio streams buffer some number of seconds of playback to account for network delays. The audio DSP (such as the Hexagon DSP from Qualcomm) initiates a Direct Memory Access (DMA) transfer into its SRAM, and the video decoder reads the video stream and starts generating frame buffers in memory that will be consumed by a display controller. The audio stream is encoded until it hits the Audio DSP. The CPU is only shown when it streams data. It may have control flow aspects which are not shown in this figure.

In a mobile SoC, it is typical that the usecase dataflows exercise multiple IPs *concurrently*. Table I shows five different usecases and the set of different IPs that are commonly applied inside the SoC. In the interest of space, we only show five usecases. All five usecases correspond to the camera application. The camera application is one of the most heavily used apps in a smartphone, and it has strict real-time performance guarantees to maintain. Otherwise, the user's experience can suffer. Hence, we discuss it in detail here. Across all of the camera usecases in Table I, at least half of all IPs are concurrently active. Hence, the performance of a usecase can be primarily determined by all of the concurrent activity that is going on inside the chip. This contrasts with Amdahl's Law where work at different IPs is serialized.

The performance of a usecase is strongly dependent upon three fundamental aspects. The first of these is the computational performance of the individual IPs, which is affected by the isolated capability of the execution engine, as well as the IP-internal memory hierarchy and interconnect that supports

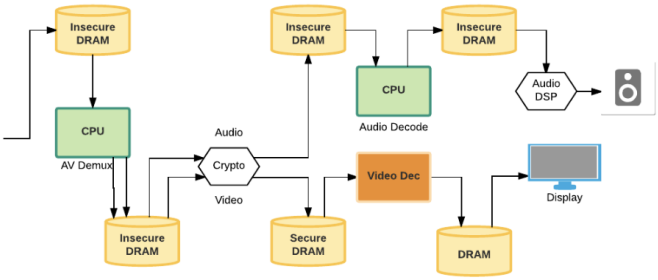


Figure 4: Streaming Internet content over WiFi usecase.

that engine. Any part of the concurrent application data flow can become a bottleneck if the overall IP's performance is not fast enough to maintain a steady stream in the overall processing rate.

The second is IP-external data movement. Camera applications, in particular, can consume a significant amount of memory bandwidth at high frame rates (HFR). Let us consider the example of a "4K" Video Recording at 240 frames per second (FPS). A 4K image is 3840x2160 pixels per frame. Assuming a YUV420 color encoding system, which uses 6 bytes per 4 pixels, the frame size approximately amounts to 12 MB. Processing such a large image at high frame rates, with data moving between the different IPs, such as the Image Signal Processor (ISP) for wavelet noise reduction (WNR) and temporal noise reduction (TNR), while keeping track of as many as five reference frames, through the DRAM, can cause the memory bandwidth of a mobile SoC (around 30 GB/s) to become the bottleneck.

The third major usecase bottleneck is the coordination overhead between the IPs, which by and large today are routed through the CPU. Since the IPs are exposed as individual devices, at least in Android, the CPU gets an explicit interruption whenever the IP finishes processing. And the CPU must handle the task of signaling the consumer of that data to start.

Understanding which of the aforementioned reasons can cause the system to suffer from a bottleneck is a challenging task for any SoC integrator. Hence, it is useful to have early stage tools for guiding the overall architecture of the SoC.

III. Gables: A Multi-IP Roofline Model

The key challenge with holistically characterizing and understanding mobile SoC performance with many accelerators is that it is difficult to gather insight into the performance of the SoC as a whole. Unlike in CPU and GPU systems, where the characteristics of the system and an application running on that system can be studied by isolating the specific IP, the performance of a mobile SoC and its usecase is the result of concurrent activity across multiple IPs. Therefore, the goal of our approach is to design a model that can provide intuition and insight for SoCs, much like Roofline does for multicore chips and Amdahl's Law does for parallel systems in general.

In this section, we present the Gables SoC performance model where hardware has a Roofline for each IP, and a

Usecases	AP	Display	G2DS	GPU	ISP	JPEG	IPU	VDEC	VENC	DSP
HDR+	✓	✓		✓	✓	✓	✓			
Videocapture	✓	✓		✓	✓				✓	
Videocapture (HFR)	✓	✓		✓	✓				✓	
Videoplayback UI	✓	✓	✓	✓				✓		
Google Lens	✓	✓	✓	✓						✓

Table I: The table shows a variety of applications usecases that are typically exercised on a mobile smartphone SoC. For each usecase, the table indicates which of the IPs are exercised *concurrently*. Not all IPs are shown, but it suffices to show that in all of the usecases multiple different IPs are exercised concurrently. Moreover, different usecases use different IPs simultaneously.

workload usecase apportions work among the different IPs. The model can be used to determine the critical limitation for performance in a mobile SoC and can be visualized via multiple rooflines on a single plot. Although we specifically focus on mobile systems, which are at the forefront of extremely heterogeneous computing, the model is generalizable for application to any SoC design.

A. SoC Model and High-level Assumptions

Gables targets the SoC shown in Figure 5, typical of a modern SoC, that has N IP blocks, IP[i], i=0, N-1, an interface to external DRAM memory, and high-bandwidth on-chip interconnect. For easy reference, Table II provides all the Gables parameters that this section will introduce.

The baseline for the Gables model makes several hardware assumptions. First, each IP[i] has a roofline defined by its peak performance and bandwidth in and out of the IP. Second, all IPs on the SoC operate concurrently (as rationalized in Section II-B) and share bandwidth to off-chip memory. Third, the on-chip interconnect has sufficient bandwidth to be modeled by just the bandwidths that connect to it from the IPs and off-chip memory. Fourth, all substantial inter-IP communication occurs via DRAM memory where multiple-megabyte buffering/rate-matching is possible.

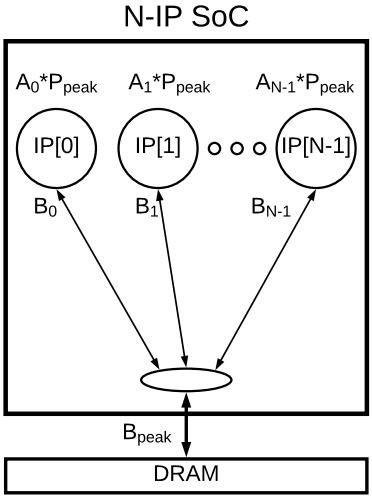


Figure 5: N-IP SoC with Gables.

The base model also makes software usecase assumptions. First, a usecase is divided into concurrent non-negative work at each IP[i], following the usecase observations of Section II-B. Second, a usecase's operational intensity at each IP may be different, either because various aspects of the work are assigned to different IPs and/or the IPs have different internal caches or scratchpads, e.g., for latency reduction versus latency tolerance. Third, we assume that the software concurrently exercises multiple IPs, also as per the rationale we presented earlier in Section II-B and summarized in Table I.

B. Two-IP Model: A Primer

We first introduce Gables with a two-IP SoC as it is notionally more straightforward and all concepts extend to an N-IP SoC. An example two-IP SoC might have IP[0] as the CPU (Application Processor) complex and IP[1] as a GPU. Let the SoC's peak off-chip memory bandwidth be B_{peak}.

The Gables model assumes that both IPs have a roofline defined by hardware and an operating point picked by the usecase. IP[0] hardware has peak computation performance P_{peak} and bandwidth in and out of B_0. IP[1] hardware has peak computation performance $A \cdot P_{peak}$, where A is its acceleration, and bandwidth in and out of B_1. A given usecase will assign (1-f) work to IP[0] with operational intensity I_0 and f work to IP[1] with operational intensity I_1, where $0 \leq f \leq 1$. At a broad level, this follows Amdahl's Law except that the work occurs in parallel rather than sequentially.

Runtime. Gables assumes that the time for the SoC to complete a usecase can be limited by the time at IP[0], IP[1], or the DRAM memory interface. First, IP[0]'s computation time is its work divided by its peak performance: $C_0 = (1-f)/P_{peak}$. The data capacity (bytes) IP[0] needs to do this work is that work divided by its operational intensity: $D_0 = (1-f)/I_0$. The minimum time to transfer IP[0]'s needed data is this data capacity divided by IP[0]'s bandwidth to the interconnect: D_0/B_0. Finally, IP[0]'s minimum time to execute the usecase is the maximum of its data transfer and computation times:

$$T_{IP[0]} = \max(\frac{D_0}{B_0}, C_0) \quad (1)$$

Second, IP[1]'s equations follow a similar pattern with the slightly different inputs of peak performance $A \cdot P_{peak}$, IP

Parameter	Description
HW Inputs	
P_{peak}	Peak performance of CPUs (ops/sec)
B_{peak}	Peak off-chip bandwidth (bytes/sec)
A_i	Peak acceleration of IP[i] (unitless)
B_i	Peak bandwidth to/from IP[i] (bytes/sec)
SW Inputs	
f_i	Fraction of usecase work at IP[i] (ops)
I_i	Operational intensity of usecase at IP[i] (ops/byte)
Tmp Values	
C_i	Compute time at IP[i] (sec)
D_i	Data transferred for IP[i] (bytes)
$T_{IP[i]}$	Time at IP[i] (sec)
T_{memory}	Time on chip memory interface (sec)
Output	
$P_{attainable}$	Upper bound on SoC Performance (ops/sec)

Table II: Glossary of Gables model parameters.

bandwidth B_1, work f, and operational intensity I_1 to yield $C_1 = f/(A \cdot P_{peak})$, $D_1 = f/I_1$ and:

$$\boxed{T_{IP[1]} = \max(\frac{D_1}{B_1}, C_1)} \tag{2}$$

Third, the minimum time to transfer data on/off chip is the total data to be transferred divided by memory bandwidth:

$$\boxed{T_{memory} = \frac{D_0 + D_1}{B_{peak}}} \tag{3}$$

Fourth, a SoC's maximal attainable performance is inversely proportional to the maximum of times at each component:

$$\boxed{P_{attainable} = \frac{1}{\max(T_{IP[0]},\ T_{IP[1]}, T_{memory})}} \tag{4}$$

Performance/Roofline. Alternately, via algebra and re-expanding terms, the following performance equations provide the dual of the above time equations when $0 < f < 1$:

$$\boxed{\frac{1}{T_{IP[0]}} = \frac{\min(B_0 \cdot I_0, P_{peak})}{(1-f)}} \tag{5}$$

$$\boxed{\frac{1}{T_{IP[1]}} = \frac{\min(B_1 \cdot I_1, A \cdot P_{peak})}{f}} \tag{6}$$

$$\boxed{\frac{1}{T_{memory}} = B_{peak} \cdot I_{avg}} \tag{7}$$

where $I_{avg} = 1/((1-f)/I_0) + (f/I_1))$, the harmonic mean of I_0 and I_1 weighted by fraction of work at each IP.

$$\boxed{P_{attainable} = min(\frac{1}{T_{IP[0]}}, \frac{1}{T_{IP[1]}}, \frac{1}{T_{memory}})} \tag{8}$$

Note that (a) Equations 5 and 6 are just IP rooflines scaled by dividing by fraction of work, (b) Equation 7 is memory's slanted-only roofline, and (c) Equation 8 picks the smallest roofline. The disadvantage of these performance equations is that one must remove the IP[1] term if $f = 0$ and IP[0] term if $f = 1$ to avoid divide-by-zero exceptions while handling all of $0 \leq f \leq 1$. Their key advantage is that they enable the multi-roofline visualizations, which we develop next.

C. Visualizing Gables via Scaled Rooflines

Here we develop multi-roofline plots for Gables that:

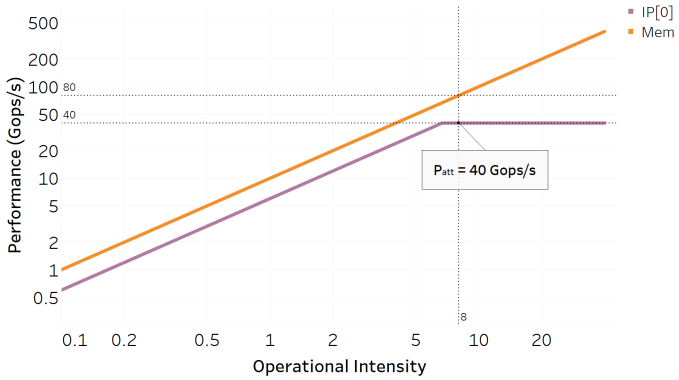
- use axes following Roofline (x-axis log operational intensity I and y-axis log performance attainable P_{att}),
- display three scaled rooflines from Equations 5-7 by varying their operational intensities over the x-axis,
- add "drop lines" where operation intensities I_0, I_1, and I_{avg} select performance on each roofline, and
- reveal performance attainable as the lowest selected point among the rooflines, following Equation 8.

Figure 6 illustrates an example sequence of two-IP Gables where our example posits that IP[0] is a CPU complex with caches that support data reuse, while IP[1] is a GPU designed for latency tolerance, not bandwidth reduction. We begin with hardware inputs $P_{peak} = 40$ Gops/s, $B_{peak} = 10$ Gbytes/s, $A_1 = 5$, $B_0 = 6$ and $B_1 = 15$. For the software usecase, we assume $I_0 = 8$ operations/byte on IP[0], $I_1 = 0.1$ for IP[1], and $f = 0$. As Figure 6a illustrates I[0] limits performance to 40 Gops/s, memory at 80 Gops/s, while IP[1] is not shown since it is assigned no work ($f = 0$). Overall peformance is 40 Gop/s, equal to IP[0]'s lesser performance, and is disappointing because unused IP[1] can do 200 Gops/s.

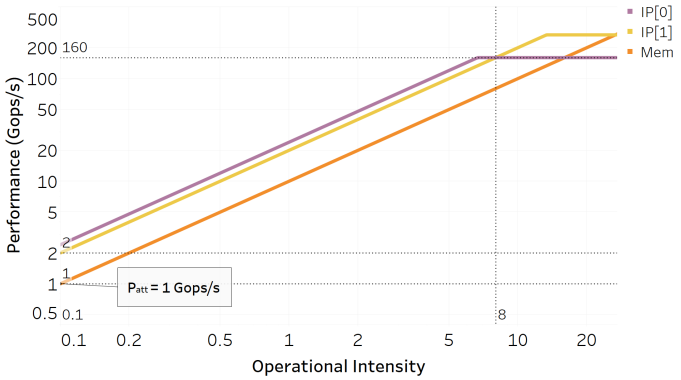
As illustrated in Figure 6b, we seek more performance with the obvious change of assigning work to IP[1] by setting $f = 0.75$. However, performance drops to 1.3 Gops/s (rounded to 1). *Why?* The answer is that the memory bandwidth we apportioned is inadequate due to the low data reuse at IP[1] ($I_1 = 0.1$) that makes IP[1] and memory rooflines provide their bounds to the far left as operational intensities near 0.1.

If memory bandwidth is problem, perhaps it makes sense to increase B_{peak} from 10 to 30 GB/s with the result illustrated in Figure 6c. This increases performance from 1.3 to 2 Gops/s, but this is still disappointing. Moreover, it moves the memory roofline well above the other two bounds, incurring additional expense without benefit (for this usecase).

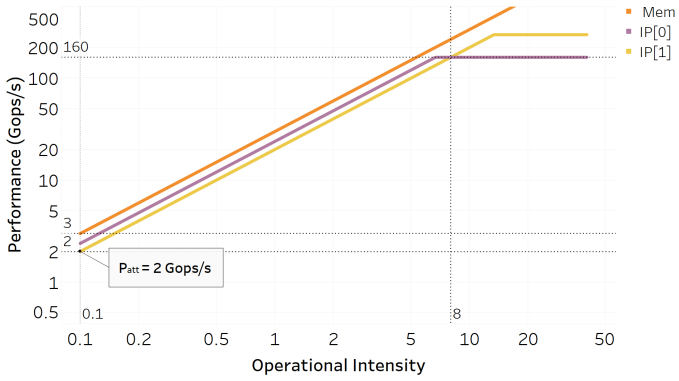
Figure 6d shows our final SoC with two changes. First, we increase I_1 from 0.1 to 8 (like IP[0]) by adding memory (registers/scratchpads/caches) to IP[1] and ensuring that the usecase reuses data from it (easier said than done). Second, we reduce B_{peak} from 30 down to a sufficient 20 Gops/s. This results in overall performance of 160 Gops/s with all three rooflines equal at $I = 8$, a perfectly balanced design.



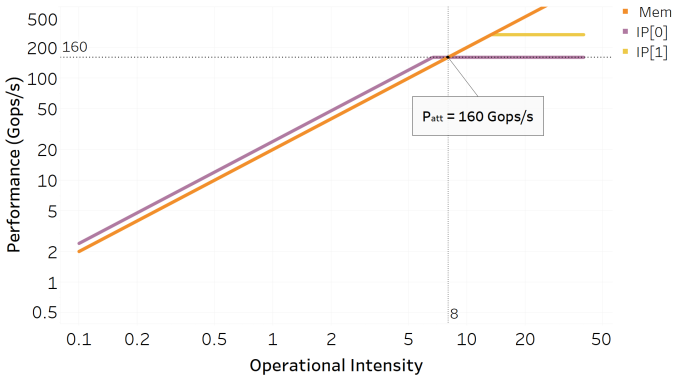
(a) With initial parameters, performance matches CPU's 40 Gops/s at assumed operational intensity $I_0 = 8$, as the GPU is not used.



(b) Changing f from 0 to 0.75 to use the GPU drops performance to 1.3 Gop/s (rounded to 1) as the GPUs poor data reuse (operational intensity $I_1 = 0.1$ at far left) makes memory bandwidth inadequate.



(c) Increasing memory bandwidth B_{peak} from 10 to 30 GB/s increases performance to only 2 Gops/s.



(d) Increasing GPU data reuse I_1 from 0.1 to 8 and decreasing B_{peak} to a sufficient 20 GB/s achieves 160 Gops/s performance with all three rooflines equal at $I = 8$, achieving a balanced design.

Figure 6: Two-IP SoC model. IP[0] is the CPU. IP[1] is assumed to be the GPU. The parameters for the figures progress from (a) to (d). So, for instance, to interpret the results in (c) one must understand the parameters as they were setup in (a) and (b).

This example shows some (teaching) value for the two-IP Gables. The GPU can be substituted with any other accelerator with respect to the CPU, and the same set of conclusions would hold. But the full power of Gables awaits handling more-complex N-IP SoCs, which we develop next.

D. Multi-IP Model

Here we generalize the Two-IP model to a SoC with N different IP[i], i = 0, N-1 of Figure 5. The SoC continues to have P_{peak} computation performance at IP[0] and off-chip memory bandwidth B_{peak}. Each IP[i] has performance $A_i \cdot P_{peak}$ and bandwidth B_i, were A_0 must be 1. A usecase assigns each IP[i] concurrent work f_i at operational intensity I_i where the f_i's are non-negative and sum to 1. At each IP[i], computation time $C_i = f_i/(A_i \cdot P_{peak})$, data transferred $D_i = f_i/I_i$, and minimum time:

$$T_{IP[i]} = \max(\frac{D_i}{B_i}, C_i) \tag{9}$$

The minimum time at the memory interface is the total data transferred divided by memory bandwidth:

$$T_{memory} = \frac{\sum_{i=0}^{N-1} D_i}{B_{peak}} \tag{10}$$

And the usecase's maximal attainable performance on the N-IP Soc is:

$$P_{attainable} = \frac{1}{\max(T_{IP[0]},\ ...,\ T_{IP[N-1]},\ T_{memory})} \tag{11}$$

The dual performance/roofline equations follow with IP[i] terms omitted whenever $f_i = 0$ (to avoid dividing by zero) and using weighted harmonic mean $I_{avg} = 1/(\sum_{i=0}^{N-1} f_i/I_i)$.

$$\boxed{\frac{1}{T_{IP[i]}} = \frac{\min(B_i \cdot I_i, A_i \cdot P_{peak})}{f_i}} \quad (12)$$

$$\boxed{\frac{1}{T_{memory}} = B_{peak} \cdot I_{avg}} \quad (13)$$

$$\boxed{P_{attainable} = min(..., \frac{1}{T_{IP[i]}}, ..., \frac{1}{T_{memory}})} \quad (14)$$

The above equations complete the base Gables model, but extensions and variations are possible (Section V).

Fortunately, the two-IP Gables plots also generalize to visualize N IPs with a scaled roofline for each IP used plus a memory roofline. See the Gables home page [10] for interactive visualizations for both two-IP and three-IP SoCs, as well as a pointer to the Gables open-source Android app.

IV. Experimental Evaluation

To explore whether the Gables SoC roofline model provides insight, we use it to empirically examine existing consumer smartphone SoCs that are commercially available off-the-shelf. The exact roofline of a black-box chip or IP is hard to ascertain correctly. An optimistic estimate of a roofline uses manufacturer's specifications (if available). These might multiply the speed and number of functional units to get a number that cannot be exceeded but may not be attainable. Alternatively, a pessimistic estimate of a roofline uses varying micro-benchmarks to seek the best achievable performance. These estimates yield a roofline that is attainable but may not be the best performance possible (i.e., it may be the ceiling). In this section, we use the latter empirical method (i.e., the ceiling) to see whether Gables provides insight regarding a SoC even if the ultimate peak performance is not unearthed.

A. Methods

We determine the rooflines for the three most common general purpose compute engines found in mobile SoCs: the CPU, GPU, and the DSP. The goal is to evaluate how accurately the Gables model can assess real hardware behavior. Gables's performance predictions as parameters change should at the very least have the correct shape and reasonable relative error. Note, however, that Gables leaves the goal of high-quality absolute, yet almost never attainable, accuracy to a cycle-level simulation of executing the usecase software.

We conduct our evaluation on two commercially available Qualcomm SoCs, the Snapdragon 835 [15], and the Snapdragon 821 [16]. We ran experiments on the three most programmable engines on these mobile SoCs: the CPU, GPU, and DSP. Our findings hold true for both systems, hence here forth we discuss the results only for the latest of the two chipsets (i.e., the Qualcomm Snapdragon 835).

Algorithm 1 The pseudocode for how to vary the operational intensity for the CPU, GPU and DSP (and other IPs).

```
1:  procedure GABLES_ROOFLINE_KERNEL(trials, size) ▷
    alpha, beta: variables for dummy work
2:      alpha ← 0.5;
3:      for i ← 0, trials do
4:          for n ← 0, size do
5:              beta ← 0.5;
6:              #if FLOPS_PER_BYTE == 2
7:              beta ← beta * A[i] + alpha
8:              #elif FLOPS_PER_BYTE == 4
9:              beta ← beta * A[i] + alpha
10:             beta ← beta * A[i] + alpha
11:             #elif ...
12:             ...
13:             #endif
14:             A[i] ← beta;
15:         end for
16:     end for
17: end procedure
```

The Snapdragon 835 includes the Qualcomm Kryo ™ CPU, an Adreno ™ 540 GPU, a Hexagon ™ 682 Digital Signal Processor (DSP), in addition to several other domain-specific accelerators, such as the modem and audio/video encoders. The CPU has eight cores (up to 1.9 GHz). The GPU focuses on graphics (OpenGL, DX12) with limited general-purpose support and it boasts a maximum of 540 GFLOPS/s at slightly over 700 MHz. The DSP processes very wide (1024 bits) integer vectors with limited single precision (SP) support using a dedicated scalar processor with four threads, and it can be clocked at 920 MHz for peak operation.

Algorithm 1 illustrates the micro-benchmark we run on each IP. The basic idea is to have all of an IP processing elements load each word in an array of a certain size and perform some number of operations on it. We vary the array size to see how performance changes for different memory footprints. We modify the number of operations per word to control the operational intensity, and repeatedly benchmark this kernel on different IPs to determine their bandwidth (GB/s) and computational limits (GFLOPS/s) of the system. The structure of this computational kernel was initially conceived by the authors of the Empirical Roofline Toolkit [17].

By default, the word size we assume in our algorithm is 32-bits, and the operation is a single-precision floating-point multiply. Single-precision is a compromise between double-precision historically preferred by scientific computation and the half-precision (or less) favored by emerging algorithms (e.g., machine learning inference). All three engines (CPU, GPU, and DSP) support IEEE compatible single-precision, hence they are comparable. To indeed achieve peak performance, one should leverage the full SIMD capabilities of the

compute engines, which we discuss later on.

Processor overheating and throttling is a significant issue since the code is severely floating point (FP) intensive. Therefore, we benchmark the devices in a thermally controlled unit. Otherwise, performance can vary significantly from one run to another. Furthermore, many vendor-specific knobs are used to disable performance and power monitoring governors to ensure repeatable and sustained high performance.

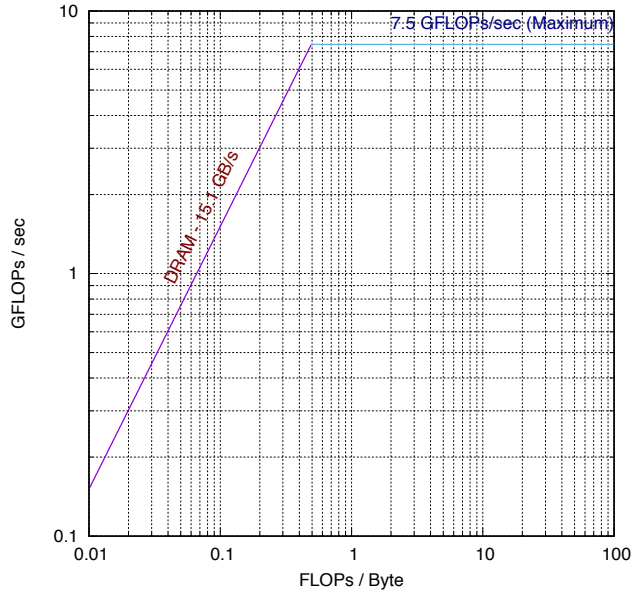
We developed an open-source Gables Android mobile application, which can be used to evaluate different mobile chipsets efficiently. The Android app uses the Java Native Interface (JNI) to implement the CPU, GPU, and DSP kernels to ensure high performance and no interruptions during program execution. We program the CPU kernel using C++ and OpenMP [18] pragmas to facilitate multithreading necessary to load all the cores. CUDA support is virtually nonexistent on consumer smartphones and OpenCL is not widely adopted by ARM and Qualcomm. Therefore, we program the Adreno GPU using OpenGL ES 3.1 [19], re-purposing the graphics shader pipeline for general purpose computing. We program the Qualcomm Hexagon DSP using the Qualcomm Interface Definition Language (IDL) toolchain [20].

B. CPU and GPU Rooflines

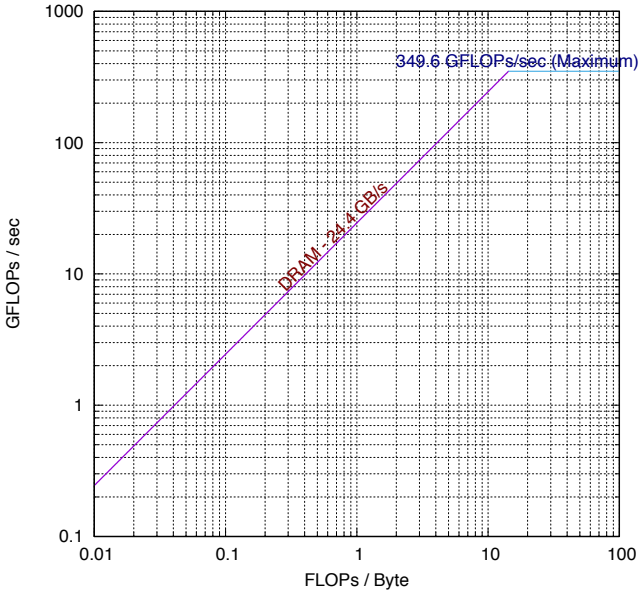
As we did in Section III-C, we begin our exploration with two IPs: the CPU (IP[0]) and GPU (IP[1]). Figure 7a shows our empirically derived roofline estimates for CPU IP[0]. The peak performance is 7.5 GFLOPS/s, which can be seen as low. We are not running a NEON (SIMD) instruction set optimized microbenchmark. When we apply vectorization to the code with compiler support we can achieve in excess of 40 GFLOP/s (not shown). The exact benchmark we use does not affect our later analysis, as long as we stay consistent with one benchmark. Hereonforward, we only refer to the non-NEON peak performance. The bandwidth to DRAM is 15.1 GB/s, which is only 50% of the peak. The stated theoretical peak bandwidth is 30 GB/s. The bandwidth is lower, in part, because we perform both read and write operations in Algorithm 1. We use this as our basis for our analysis, because this is more common in useful programs than read-only accesses.[3]

From this test alone, one cannot tell whether that bandwidth is limited by the bandwidth from IP[0] (B_0), the bandwidth off-chip (B_{peak}), or some other entity (e.g., interconnect). The CPU can obtain higher bandwidth from its internal L1 and L2 caches by using smaller micro-benchmark array sizes.

Figure 7b shows our empirically derived roofline estimates for the Adreno GPU IP[1]. In the case of the GPU, since it is generally more of a streaming workload used mainly for rendering graphics, rather than general purpose compute, we apply a slight variant of the kernel. We perform a stream read from an array and update another array, much like in the CPU STREAM kernel, which allows the GPU to maximize its read bandwidth and compute capability; such a scenario is more typical of GPU applications than CPU applications.



(a) CPU.



(b) GPU.

Figure 7: Two-IP Roofline for the CPU and GPU.

The theoretical peak performance for the Adreno 540 ™ is 567 GFLOPS. The peak performance we were able to obtain is 349.6 GFLOPS/s. This enables us to estimate its acceleration with respect to the CPU as $A_1 = 349.6/7.5 = 46.6 \approx 47X$. This 47X diminishes down to less than an order of magnitude when aggressive SIMD optimization is enabled, which is in line with prior conclusions [23]. The peak DRAM bandwidth

[3] As a sanity check, we also ran a read-only version of the micro-benchmark (not shown) that achieves close to 20 GB/s and is consistent with long-standing benchmarks STREAM [21] and lmbench [22].

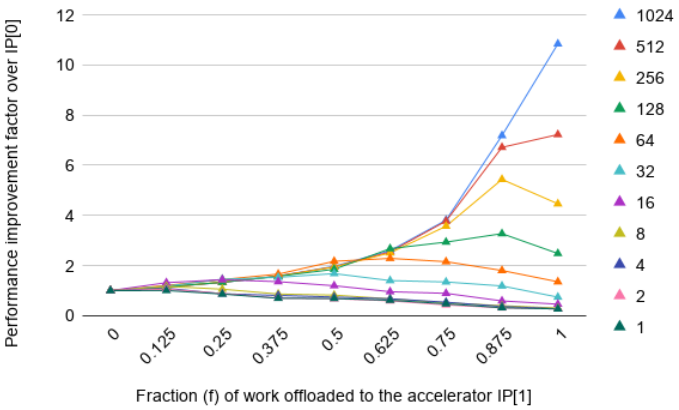


Figure 8: Performance improvement as different amount of work is offloaded to the accelerator IP[1].

is higher at 24 GB/s, as one would expect. The GPU is concurrently running 1024 workgroups with 256 threads each.

C. Gables for Two IPs

In this section we empirically examine the Snapdragon chip running our microbenchmark on CPUs and GPUs following the work fraction and operational intensity assumptions of Gables to show that Gables provides some insights.

Figure 8 displays results. The y-axis gives performance normalized to all work on the CPU IP[0] ($f = 0$) with operational intensity $I_0 = 1$. The x shows the fraction of work f at the GPU from 0 (all work at CPU) to 1 (all at GPU) in increments of 1/8. All runs do the same total amount of work (number of single-precision ops) and IPs operate in parallel when $0 < f < 1$. Different lines show different operational intensities from 1 to 1024 ops per byte.

It is hard to draw absolute conclusions from the data, nevertheless there are general trends we can observe. First, we find that when operational intensity is low, offloading work from the CPU to the GPU results in a performance slowdown (but one as bad as the terrible performance of Figure 6b). This result indicates that one should not offload low operational intensity work to the GPU. Second, when operational intensity is high, offloading work from the CPU to the faster GPU results in substantial speedup, e.g., 39.4 for $I_0 = I_1 = 1024$.

Hence, acceleration can work (no surprises there) but the important piece to understand is that the benefits, which we can achieve through hardware accelerators is strongly a function of the inherent workload characteristics, which includes not only the fraction of work that is being offloaded but also the operational intensity of that fraction that is being offloaded.

D. Toward N-IP SoC

As a step toward applying Gables for more than two IPs, we also constructed the roofline for the Hexagon DSP and proceeded with "mixing" analysis like in the previous section. One can think of the Hexagon DSP as having two components: (a) a low-power scalar unit designed to be (almost) always

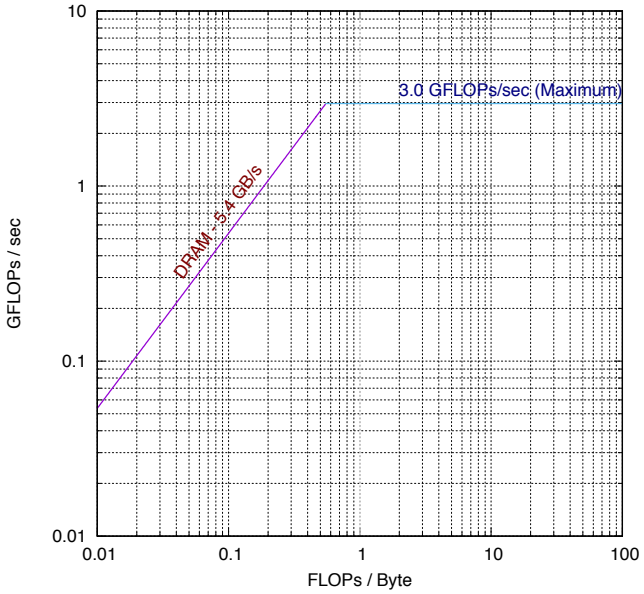


Figure 9: DSP roofline for the scalar threads. The y-axis scale here is smaller than Figure 7 because we use the HVX scalar engine, and not the high-performance (non-IEEE compliant FP) vector engine (see details in Section IV-D).

on and (b) a high-performance integer-only vector unit (4096 bits per cycle). We chose to focus on the scalar unit as it can execute the single-precision floating-point operations of our micro-benchmark and allows us to compare and contrast.

Figure 9 shows the roofline of Hexagon DSP's scalar unit. We obtain performance of 3.0 GFLOPS/s on Algorithm 1 that is somewhat less than the maximum 3.6 GFLOPS/s predicted for four threads by the spec. While the acceleration relative to the CPU is low, the scalar DSP provides value for low-power offload, leaving acceleration the vector units. The DSP's bandwidth is limited to 12.5 GB/s, much less than the CPU and GPU and likely due to using a different interconnect fabric, as illustrated for a generic SoC by Figure 3.

We performed some "mixing" analysis by having the DSP scalar unit do work in parallel with CPU and GPU. We do not show results here, because the scalar DSP was too wimpy to substantially perturb CPU-GPU behavior. We will examine using the DSP vector unit, but this will require method changes as the DSP operates only on integer vectors.

V. Model Extensions

The Gables model provides a platform from which further extensions are possible, including the three that we discuss in this section. We present extensions to address scratch-pad/caches, on-chip interconnects, and serial/exclusive work.

A. Memory-Side Memory/Scratchpad/Cache

The Gables model assumes a SoC where all substantial inter-IP communication occurs via DRAM memory where

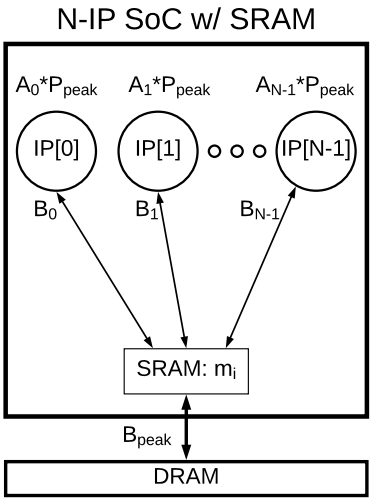


Figure 10: N-IP SoC with on-chip memory extension.

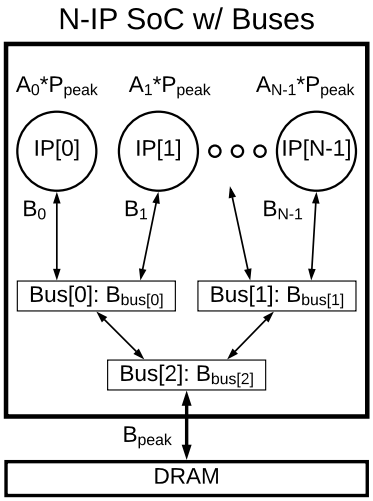


Figure 11: N-IP SoC with detailed interconnect extension.

multiple-megabyte buffering/rate-matching is possible. An alternative is to add memory to the SoC or package that is shared so that it can buffer inter-IP communication and other IP data. Options include an on-chip "memory side" scratchpad or cache, as well as a high-bandwidth memory attached to the SoC with many through-silicon vias and organized as a software-managed memory or cache.

Figure 10 illustrates an N-IP SoC with memory added to reduce DRAM bandwidth use. This extension assumes that IP[i]'s memory references now go to DRAM with probability m_i (e.g., misses) and are reused from the new memory with complementary probability $(1 - m_i)$ where good reuse has $m_i \ll 1$. Values for m_i's depend on properties of both the SoC (e.g., memory size) and the usecase (e.g., reuse by IP[i]'s references). This new memory reduces IP[i]'s off-chip traffic to D "prime"—$D'_i = m_i \cdot D_i$—and total off-chip demand also uses primes:

$$T_{memory} = \frac{\sum_{i=0}^{N-1} D'_i}{B_{peak}} \tag{15}$$

The maximal attainable performance of the N-IP Soc with this extension is uses Section III-C's Equation 11 with the above T_{memory} term.

B. On-Chip Interconnect

The base Gables model assumes a SoC where the on-chip interconnect has sufficient bandwidth to be modeled by the bandwidths that connect to it from the IPs (B_i's) or memory (B_{peak}), but without modeling its internal topology.

This extension models the interconnect in more detail as some topology of Q interconnection networks that we colloquially denote as Bus[j], j = 0, Q-1. Each bus contributes the diagonal part of a roofline due to its bandwidth $B_{Bus[j]}$ that proceeds unbounded, as a bus has no computational limit. Following bottleneck analysis, we assume that buses operate concurrently with each other, IPs, and the memory interface. Figure 11 illustrates an N-IP SoC with interconnection networks (buses) modeled in more detail.

The data that flows over a bus depends on assumptions. Here we continue base Gable's assumption that inter-IP data travel via memory (or memory-side scratchpad/cache of the previous extension) and add the assumption that each IP[i] has one bus path to/from memory. Let Use(i,j) be 1 if IP[i] uses Bus[j] and 0 if it doesn't use the bus. Then the time each Bus[j] requires is the data that uses the bus divided by the bus's bandwidth:

$$T_{Bus[j]} = \frac{\sum_{i=0}^{N-1} D_i \cdot Use(i,j)}{B_j} \tag{16}$$

The N-IP SoC formula must add terms for each potential bus bottleneck:

$$\begin{aligned} P_{attainable} = 1/(\max(&T_{memory}, \\ &T_{IP[0]},\ ...,\ T_{IP[N-1]}), \\ &T_{Bus[0]},\ ...,\ T_{Bus[Q-1]}) \end{aligned} \tag{17}$$

Further extensions to richer topologies (e.g., multiple alternative bus paths) and/or richer flows (e.g., directly among IPs) are straightforward at the cost of more assumptions, parameters, and notation. However, we often find that "less is more" when it comes to models for providing early insights.

C. Exclusive/Serialized Work

The base Gables model assumes a usecase is divided into concurrent work at each IP[i], consistent with the usecase discussion of Section II-B.

It is straightforward to model exclusive or serialized work, where only one IP is active at a time. This generalizes the

computational assumptions of Amdahl's Law and matches those of MultiAmdahl [24] (see Section VI). Neither of these models, however, include data transfer times.

We also assume that each IP transfers its needed data concurrently with its own execution. For this reason, we modify the equation for the time at IP[i] (Equation 9) to include a new term for the time transferring its needed data from/to offchip (D_i/B_{peak}) with the original two terms (data transfer time from IP[i], and IP[i] execution time) to get:

$$T'_{IP[i]} = \max(\frac{D_i}{B_{peak}}, \frac{D_i}{B_i}, C_i) \tag{18}$$

Finally, the equation for $P_{attainable}$ changes two ways from the base Gables Equation 11. First, T_{memory} is omitted since off-chip data transfer are included in $T'_{IP[i]}$. Second, exclusive work uses the sum of $T'_{IP[i]}$ rather than the maximum used for concurrent work:

$$P_{attainable} = \frac{1}{T'_{IP[0]} + \ ... + \ T'_{IP[N-1]}} \tag{19}$$

More complex combinations of parallel and serialized work are possible with more assumptions, parameters, and notation. At some point, however, this complexity is too much for early decisions. As statistician George Box said in 1987, *Essentially, all models are wrong, but some are useful.*

VI. Prior Work on Modeling

Models have long been important to computer architecture. Some of these include the following. First, Amdahl's Law [25] relates the impact of speeding up or parallelizing part of a computation to its impact on whole computation's speedup. It reminds us that we must beware of the aspects that are not sped up. Second, the Iron Law [26] partitions program execution time on a processor core into $\frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle}$. It reminds us to focus on the product of all three terms rather than a subset, e.g., clock frequency only. Third, the 3C Model [27] partitions cache misses into compulsory, capacity, and conflict misses. It reminds us of the source of cache misses and led to victim caches and stream buffers [28].

Roofline, discussed earlier, and Gables, introduced in this paper, are both special cases of bottleneck analysis [9]. Bottleneck analysis models the maximum throughput (or bandwidth) of a system by recursively combining component throughputs with two simple rules. First, the throughput of a subsystem of components in parallel is the sum of the component throughputs. Second, the throughput of a subsystem of components in series is the minimum of the component throughputs.

MultiAmdahl [24] is the model most closely related to Gables. MultiAmdahl also models an N-IP SoC, computes each IP's performance as a function of resources used (e.g., area), divides work sequentially (i.e., exclusively) among IPs, and computes an optimal resource allocation among the different IPs. The most important difference between the two models is that Gables models bandwidth bounds both leaving each IP[i] to the on-chip interconnect (B_i's) and leaving the SoC for DRAM (B_{peak}). This follows Roofline's view that data movement is a first-order consideration, as is true for consumer SoCs that process video, audio, and other streams. A secondary difference is that base Gables assumes concurrent rather than sequential work (Section II-B), while the Gables extension of Section V-C eliminates this difference.

Both MultiAmdahl and Gables can also be placed in the tradition of adapting Amdahl's Law to new architectures, with other examples targeting large parallel processors [29], multicore chips [30], and data centers [31]. Gables, however, builds on Roofline and Amdahl's Law together.

While Gables models each IP with a simple roofline, future work could incorporate more-sophisticated sub-models, regarding on-chip memory trade-offs [32], IP interaction overheads [33], specific IPs like GPUs [34], etc.

VII. Summary and Conjectures

To frame SoC thinking and aid early SoC design, this paper contributes to the *Gables* model. Gables retargets the Roofline model to model each accelerator on a SoC, apportions work concurrently among different accelerators, and calculates a SoC performance upper bound. We evaluate the Gables model with an existing SoC and develop several extensions that allow Gables to guide early stage SoC design.

We end this paper with conjectures regarding Gables that appear true but will require future work to establish robustly. First, SoC models, such as Gables, will be valuable for SoC design and selection before cycle-level simulation is feasible. As the computer architecture industry increasingly leans on designing accelerators for delivering future performance, we will see a strong need to develop systematic methodologies for understanding balanced design and accelerator selection at an early stage with relatively few parameters. Subsequent conjectures discuss the value of Gables's specific parameters.

Second, there is value in determining the rooflines of individual SoC IPs, i.e., their acceleration A_i and bandwidth $B_{[i]}$. We found, for example, that this gave us insight into Snapdragon chips with more effort than we expected but much less effort than porting full usecases.

Third, it is critical to estimate the fraction of work f_i at each IP for important usecases. Doing this can illuminate whether an IP is over-designed to provide more acceleration (A_i) than is justified by the work assigned to it. Amdahl's Law again.

Fourth, operation intensity I_i bears careful thought, as data reuse is critical to reducing the bandwidths required. High operational intensity requires considerations from both hardware——providing sufficient registers/scratchpad/cache within an IP—and software——algorithmic changes to use the local memory well. Operational intensity can also illuminate pitfalls, such as adding more IP-local memory even when important usecases don't/can't use the added capacity to increase reuse (e.g., when the next reuse opportunity requires much larger local memory).

VIII. Acknowledgements

We thank Google for hosting our academic sabbatical visits and the gChips team members for enhancing this work, including Benjamin Dodge, Ali Iranli, Allan Knies, Xiaoyu Ma, Albert Meixner, Ofer Shacham and Hongil Yoon. Gables's scaled roofline plots were conceived by Penporn Koanantakool as a replacement for our original 3D graphical version and the scaled plots were implemented by Nikhita Kunati. At Wisconsin, Hill is supported by NSF CCF-1617824, NSF CNS-1815656, and John P. Morgridge Endowed Chair.

References

[1] "ARM financial results." https://www.arm.com/-/media/global/company/investors/Financial\%20Result\%20Docs/Arm_SB_Q4_2017_Roadshow_Slides_Final.pdf?revision=97c97bdd-76f3-4c98-a106-538680bb8d68&la=en. Accessed: 2018-07-30.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12, IEEE, 2017.

[3] "ARM AND QUALCOMM: Enabling the Next Mobile Computing Revolution with Highly Integrated ARMv8-A based SoCs." https://www.arm.com/files/pdf/ARM_Qualcomm_White_paper_Final.pdf. Accessed: 2018-08-3.

[4] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks, "The Aladdin Approach to Accelerator Design and Modeling," *IEEE Micro*, vol. 35, pp. 58–70, May 2015.

[5] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile Cpu's Rise to Power: Quantifying the Impact of Generational Mobile Cpu Design Trends on Performance, Energy, and User Satisfaction," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 64–76, IEEE, 2016.

[6] "Primate labs geekbench." https://www.primatelabs.com. Accessed: 2018-08-2.

[7] "Antutu benchmark." http://www.antutu.com/en/index.htm. Accessed: 2018-08-2.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[10] "Gables Home Page." http://research.cs.wisc.edu/multifacet/gables/.

[11] "GSMArena." https://www.gsmarena.com. Accessed: 2018-08-2.

[12] "MediaTek helio x30." http://i.mediatek.com/heliox30. Accessed: 2018-07-30.

[13] "Pixel visual core: image processing and machine learning on pixel 2." https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/. Accessed: 2018-08-2.

[14] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications," *IEEE Micro*, no. 2, pp. 34–43, 2014.

[15] "Qualcomm snapdragon 835." https://www.qualcomm.com/media/documents/files/snapdragon-835-mobile-platform-product-brief.pdf. Accessed: 2018-07-30.

[16] "Qualcomm snapdragon 821." https://www.qualcomm.com/media/documents/files/snapdragon-821-processor-product-brief.pdf. Accessed: 2018-07-30.

[17] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pp. 129–148, Springer, 2014.

[18] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[19] D. Shreiner, B. T. K. O. A. W. Group, *et al.*, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Pearson Education, 2009.

[20] "Hexagon V60 HVX Programmer's Reference Manual." https://developer.qualcomm.com/download/hexagon/hexagon-v60-hvx-programmers-reference-manual.pdf. Accessed: 2018-08-2.

[21] J. D. McCalpin, "STREAM Benchmark," *Link: www. cs. virginia. edu/stream/ref. html# what*, vol. 22, 1995.

[22] L. W. McVoy, C. Staelin, *et al.*, "Lmbench: Portable Tools for Performance Analysis.," in *USENIX annual technical conference*, pp. 279–294, San Diego, CA, USA, 1996.

[23] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, *et al.*, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 451–460, 2010.

[24] I. Keslassy, U. Weiser, and T. Zidenberg, "Multiamdahl: How should i divide my heterogenous chip?," *IEEE Computer Architecture Letters*, vol. 11, pp. 65–68, 07 2012.

[25] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2019.

[27] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989.

[28] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[29] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[30] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008.

[31] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Communications of the ACM*, vol. 61, no. 8, pp. 65–72, 2018.

[32] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, no. 1, pp. 25–28, 2009.

[33] M. S. B. Altaf and D. A. Wood, "LogCA: A High-Level Performance Model for Hardware Accelerators," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 375–388, ACM, 2017.

[34] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of Gpu Memory System for Multi-Application Execution," in *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 223–234, ACM, 2015.

APPENDIX

The appendix provides specific numbers used to create Figures 6a–6d. "(NEW)" indicates a changed input parameter.

Formulae

1 / $T_{IP[0]}$ = MIN(B_0 * I_0, P_{peak}) / (1 – f), $f \neq 1$
1 / $T_{IP[1]}$ = MIN(B_1 * I_1, A_1 * P_{peak}) / f, $f \neq 0$
1 / T_{memory} = B_{peak} * I_{avg} where I_{avg} = 1/[(1 - f) / I_0) + (f / I_1)]

$P_{attainable}$ = MIN(1/$T_{IP[0]}$, 1/$T_{IP[1]}$, 1/T_{memory})

Figure 6a

P_{peak} = 40 $Gops/s$, B_{peak} = 10 $Gbytes/s$, A_1 = 5, B_0 = 6 and B_1 = 15.
I_0 = 8 $operations/byte$ at IP[0], I_1 = 0.1 at IP[1], and f = 0.00.

1 / $T_{IP[0]}$ = MIN(6 * 8, 40) / 1.0 = 40, $f \neq 1$
1 / $T_{IP[1]}$ = MIN(B_1 * I_1, A_1 * P_{peak}) / f, $f \neq 0$, Moot since f = 0
1 / T_{memory} = 10 * 8 = 80 where I_{avg} = 8 since f = 0

$P_{attainable}$ = MIN(40 , –, 80) = 40 $Gops/s$

Figure 6b

P_{peak} = 40 $Gops/s$, B_{peak} = 10 $Gbytes/s$, A_1 = 5, B_0 = 6 and B_1 = 15.
I_0 = 8 $operations/byte$ at IP[0], I_1 = 0.1 at IP[1], and f = 0.75 (NEW).

1 / $T_{IP[0]}$ = MIN(6 * 8, 40) / 0.25 = 40/0.25 = 160
1 / $T_{IP[1]}$ = MIN(15 * 0.1, 5 * 40) / 0.75 = 1.5/0.75 = 2
1 / T_{memory} = 10 * I_{avg} where I_{avg} = 1/[(0.25/ 8) + (0.75 / 0.1)] = 0.13278
1 / T_{memory} = 10 * 0.13278 = 1.3

$P_{attainable}$ = MIN(160, 2, 1.3) = 1.3 $Gops/s$

Figure 6c

P_{peak} = 40 $Gops/s$, B_{peak} = 30 $Gbytes/s$ (NEW), A_1 = 5, B_0 = 6 and B_1 = 15.
I_0 = 8 $operations/byte$ at IP[0], I_1 = 0.1 at IP[1], and f = 0.75.

1 / $T_{IP[0]}$ = MIN(6 * 8, 40) / 0.25 = 40/0.25 = 160
1 / $T_{IP[1]}$ = MIN(15 * 0.1, 5 * 40) / 0.75 = 1.5/0.75 = 2
1 / T_{memory} = 30 * I_{avg} where I_{avg} = 1/[(0.25/ 8) + (0.75 / 0.1)] = 0.13278
1 / T_{memory} = 30 * 0.13278 = 3.98

$P_{attainable}$ = MIN(160, 2, 3.98) = 2.0 $Gops/s$

Figure 6d

P_{peak} = 40 $Gops/s$, B_{peak} = 20 $Gbytes/s$ (NEW), A_1 = 5, B_0 = 6 and B_1 = 15.
I_0 = 8 $operations/byte$ at IP[0], I_1 = 8 (NEW) at IP[1], and f = 0.75.

1 / $T_{IP[0]}$ = MIN(6 * 8, 40) / 0.25 = 40/0.25 = 160
1 / $T_{IP[1]}$ = MIN(15 * 8, 5 * 40) / 0.75 = 120/0.75 = 160
1 / T_{memory} = 20 * 8 = 160

$P_{attainable}$ = MIN(160 , 160, 160) = 160 $Gops/s$