ACCELERATOR-LEVEL PARALLELISM

A PREPRINT

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

With the slowing of technology scaling, the only known way to further improve computer system performance under energy constraints is to employ specialized hardware engines, or more commonly known as accelerators. Already today, many chips in mobile, edge and cloud computing concurrently employ multiple accelerators in what we call **accelerator-level parallelism (ALP)**. For the needed benefits of ALP to spread to computer systems more broadly, we herein charge the community to develop better "best practices" for: targeting accelerators, managing accelerator concurrency, choreographing inter-accelerator communication, and productively programming accelerators.

Keywords Accelerators · Parallelism · Programmability · Concurrency · Mobile

1 Introduction

Programmable, interconnected computers are essential in modern society. Spanning from mobile smartphones to warehouse-scale machines, these marvels of the digital computing age continue to shape our interactions with one another and transform our lives at a planetary scale. Smartphones have changed the way we interact with machines using taps, swipes, and clicks. Warehouse-scale machines have made the world smaller, and more integrated, through computing solutions and distributed technologies that allow us to summon over 15 billion search results in a fraction of a second. Going forward, the potential for artificial intelligence based on machine learning, augmented and virtual reality, etc., to change our lives seems virtually unbounded.

Underlying much of the success of our rich end-user experiences and planetary-scale applications are the dramatic improvements in the often-forgotten programmable hardware [1]. Hardware improvements deliver performance that unlocks new application capabilities. So, performance is king. However, unlike in the 1990s and early 2000s, performance must now be obtained under severe power and resource constraints. Those who deliver performance, at a given power or area, will create more value at the same cost or deliver the equal value at less cost.

For the past few decades, hardware performance improvements have been driven by numerous advancements using faster devices (especially transistors) in parallel (billions today) at nested levels to manage complexity. In the 20th Century, hardware designers first harnessed these transistors with **bit-level parallelism (BLP)** to manipulate more bits concurrently, often 64 bits today. As device budgets repeatedly doubled–a challenge all of us would like to have with our incomes–designers turned to **instruction-level parallelism (ILP)** that has a processor (core) execute many instructions concurrently (dozens today), while still making instructions appear to execute sequentially. These changes retained easy programmability because BLP is straightforward and ILP is transparent to software.

The early 21st Century has seen the concurrent explosion of two more levels of parallelism: **Thread-level parallelism (TLP)** and **Data-level parallelism (DLP)**. TLP uses multiple sequential processors and is currently embodied in multicore chips. This parallelism is explicitly exposed for programming. TLP has had mixed success on the desktop and dramatic success in warehouse-scale computers (WSC), so much so that effective TLP exploitation is a factor in computing migrating toward WSCs and the cloud. DLP concurrently performs the same (or similar) operations on many data elements. DLP's success is mainly due to the rise of general-purpose graphics processing units (GP-GPUs) using the Single Instruction Multiple Thread (SIMT) programming model. While the roots of TLP and DLP go back a half-century, only now do people widely benefit from them, thanks to advances in software programming models.

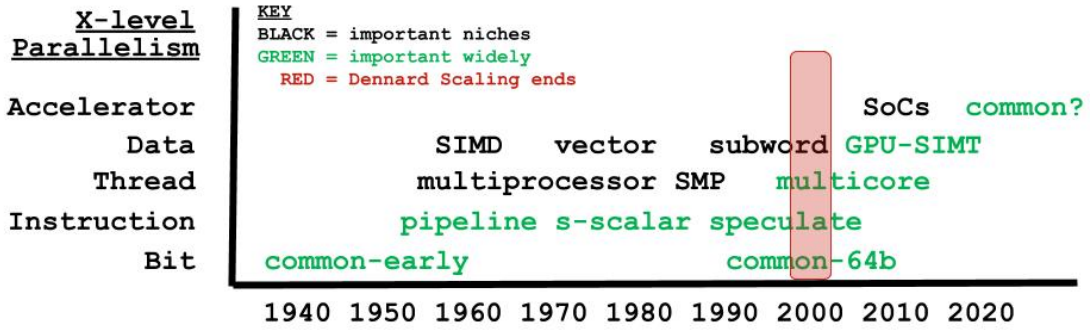


Figure 1: An approximate timeline of x-level parallelism.

Figure 1 and Figure 2 put the levels of parallelism just discussed–as well as a new accelerator-level parallelism–in context two ways. First, Figure 1 depicts historically when (x-axis) a certain type of parallelism (y-axis) became important under certain niches (black) or more broadly (green) with significant change due to the end of Dennard Scaling (red). In Figure 1, the label "S-scalar" stands for superscalar, "SMP" for symmetric multiprocessor, and "subword" for parallel SIMD like MMX. Second, Figure 2 shows a lattice for how the different levels of parallelism relate to one another when they are present together in a modern system. These figures illustrate the point that many of the performance improvements we know of today have been achieved through considerable use of levels of parallelism.

Over the coming decade, we will embark on a new form of parallelism to continue delivering hardware performance improvements. With the end of Dennard scaling and the slowing down of Moore's Law to a crawl, it is our thesis that the future of computer systems performance will have to rely on **accelerator-level parallelism (ALP)**.

We define accelerator-level parallelism (ALP) as the parallelism of different workload components concurrently executing on multiple accelerators. In turn, each accelerator is a hardware component that executes a targeted computation class faster and usually with (much) less energy. Accelerators are attractive when computations are both specialized and frequently used (the "opportunity"), while at the same time goals (timing) and resources (power/energy) are aggressive (the "need"). Examples include audio, video, and image processing. As new application domains emerge, e.g., machine learning, automotive computing, new accelerators continue to be designed.

ALP is already commonplace in today's smartphone chipsets, primarily because smartphones are energy-constrained. We assert that ALP will need to spread to computer systems more broadly as it is the only means by which we know how to improve performance under power and energy constraints. To this end, we foresee some issues. At the hardware level, we only have point solutions. These point solutions range from how we (rapidly) design new accelerators to how we orchestrate execution on them. In our judgment, point solutions are inadequate for broad ALP adoption. At the software level, ALP requires deep re-thinking regarding programming models for individual accelerators. The challenge of effectively programming ALP is daunting, in part, because it adds another level of parallelism to Figure 2. If we have learned anything from the past, it is that efficiency improvements in hardware performance can only be harnessed when the hardware capabilities are aligned with suitable programming models.

In this article, we charge the community to address these challenges by examining ALP in smartphones as a harbinger of a broad ALP future. Our views are informed by experiences as 2017-18 research interns with Google's mobile silicon group but do not necessarily represent any opinions of Google. While our perspectives are written from the perspective of mobile chipsets, it is worth stating that the industry has been adopting ALP for a while across a variety of different and important application domains. For instance, Qualcomm and ARM design accelerators for IoT. MobilEye exploits ALP for automotive electronics. Even large companies like Amazon, Google, and Microsoft are exploring ALP for video transcoding, machine learning, networking, etc. But despite numerous advances and efforts, many fundamental challenges continue to persist. It is our hope that this article paves the way for new research in enabling pervasive ALP.

2 Smartphone Chips as the Harbinger of Accelerator-level Parallelism

Most of us carry devices exploiting ALP around in our pockets or purses, as ALP is already heavily exploited in smartphone chips. Smartphones include a chip that implements a System on a Chip (SoC) that includes several traditional processor cores–often with high-performance and low-power variants, GPUs, and dozens of specialized accelerators. Figure 3 shows three generations of Apple smartphone chips with approximate accelerator count and chip area. As the process technology scales, the portion of the die area occupied by the CPU and GPU blocks diminishes

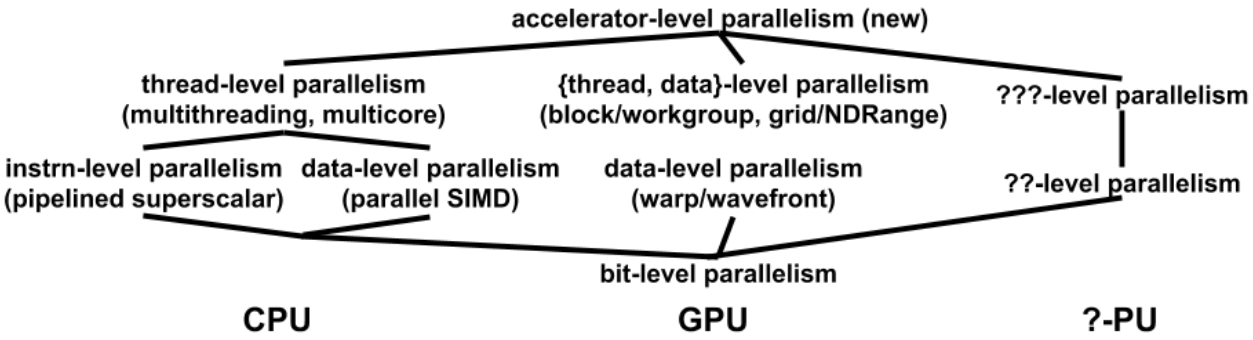


Figure 2: A parallelism lattice.

as more area is allocated for specialized accelerator units that support ALP. The white boxes delineate individual accelerators. The Apple A4 has four accelerators, which grew dramatically to 28 accelerators in the A8 to over 40 in the A12.

However, Apple SoCs may paint an optimistic ALP future, as Apple's tightly-controlled and vertically-integrated ecosystem makes hardware-software co-design easier to coordinate than in the broader Android ecosystem. Using the OS as a proxy, as of May 2019 Android's market share is 75% versus Apple iOS's 23%. Android's decoupling of software and hardware leads to software flexibility and chipset diversity that provides competitive benefits but increases accelerator and ALP co-design coordination challenges. There is no free lunch. As it is not clear which market model will prevail as ALP spreads to other markets, we must prepare for both.

Hence, to anticipate the challenges we are likely to face when ALP adoption goes mainstream, we take a look at how accelerators are used in today's commodity mobile SoCs. In a typical modern SoC, not all the accelerators are operating at all times. Instead, they invoked as needed. To illustrate this point, we present a few common smartphone usecases in Table 1. The rows give example usecases, where a usecase is something a smartphone user might do for some time. The columns identify a subset of the accelerators and an 'X' marks which usecases use which accelerators. The last column, labeled, "Dozens More" reminds us that dozens of more accelerators are unused (no 'X') by these five example usecases. We do not claim that this mapping between usecases and accelerators is ideal or perfectly representative of what the industry does at large. However, we are confident that it is sufficiently representative of industry practice to make the fundamental point that even ignoring the details shown in Table 1, there are four important observations:

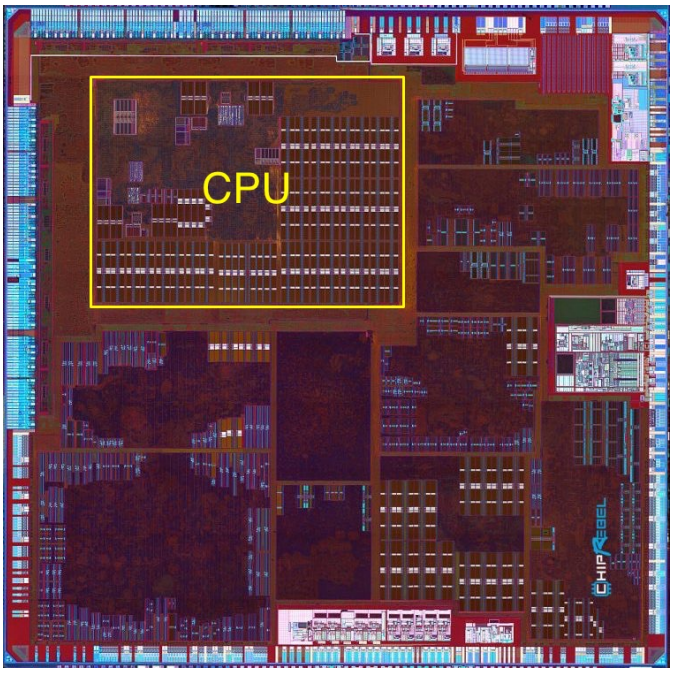
- All of the smartphone usecases rely heavily on accelerator specialization.
- Many individual usecases exercise multiple accelerators concurrently.
- The accelerators communicate heavily via the CPU and DRAM.
- Many of the accelerators rely on domain-specific programming languages or custom programming interfaces.

Table 1 shows how ALP is already prevalent in today's smartphone usecases. As new devices and usecases emerge, such as Hololens and AR/VR applications, respectively, ALP will continue to grow in prevalence. To that end, Table 1 helps raise the point that as ALP permeates mainstream computing, new challenges will arise that we must be prepared to face. We discuss these in the context of specialization, concurrency, communication, and programmability challenges.

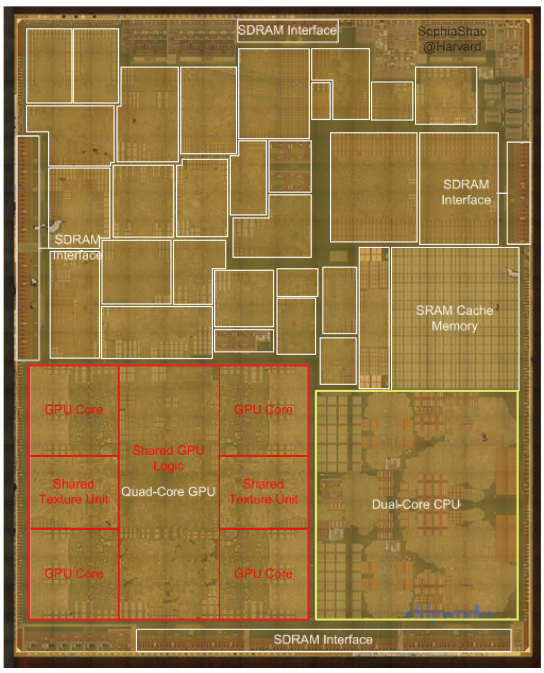
3 Challenge #1: The Accelerator Specialization Design Space

As Table 1 shows, different applications use different accelerators. For instance, the Photo Enhancing application uses six different macro-level hardware accelerator blocks, whereas the video playback usecase uses only five of them. Many of these hardware blocks have been hardened because they proved computationally challenging for the CPU, or the algorithms have matured to the point that baking them into silicon made sense for energy-efficiency. It may also be worthwhile for the reader to note that many of the accelerator blocks are internally comprised of several distinct accelerator sub-blocks. For example, a commercial Image Signal Processor (ISP) may be made up of as many as 15 different sub-blocks for specific tasks such as pixel correction, demosaicing, white balancing, noise reduction, shading, geometric correction, color transforms, tone adjustment, and edge enhancement.

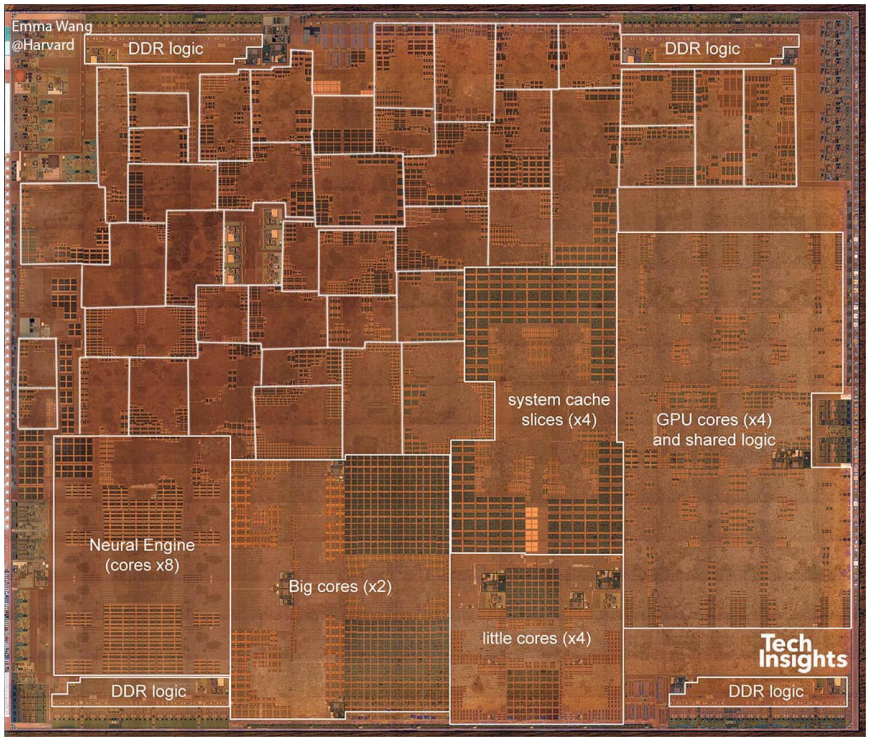
[1]Emma Wang and Sophia Shao, http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis



(a) **2010 Apple A4**
65 nm Samsung 53 mm^2
4 accelerators



(b) **2014 Apple A8**
20 nm TSMC 89 mm^2
20+ accelerators



(c) **2019 Apple A12**
7 nm TSMC 83 mm^2
40+ accelerators

Figure 3: Three Apple iPhone SoCs with accelerators highlighted.[1]

As future usecases emerge, we will have a "sea of accelerators" to choose from, where every accelerator is designed to handle one specific usecase task. But on any chip, there is limited room for hardware given a fixed area, power, and budget constraints. Hence, the number of accelerators in future chips may plateau along with the well-understood technology scaling plateauing, thus putting even more pressure on having the "right" accelerators. Unless usecases have a lot of overlap, architects will struggle to make room for all the accelerators.

Thus, the question is, how do we identify which accelerators matter the most, and how do we automatically "generate" and size those accelerators so that there is likely to be significant accelerator reuse? State of the art for generating accelerators is a "whack-a-mole" approach: Domain experts identify the important application usecases, understand the (in)efficiency on the CPU, and work their way toward custom hardware blocks (e.g., ISP) that can accelerate each specific task. This is a slow, tedious, and costly process. Moreover, the whack-a-mole process works well only when the usecase(s) and its tasks are well established. It is much harder to accelerate tasks in hardware that are still emerging.

We require tools and frameworks that can look at a large codebase, and automatically make recommendations for accelerator designs. These recommendation systems will likely need to come in two flavors. The first is given a large pool of code the system must automatically recommend what portions of the code are amenable for acceleration and how to best design those accelerators. The second is given an existing pool of accelerators the system must automatically determine which parts of the code are most amenable for (re)mapping to the existing accelerator suite. Either of these recommendation systems would be a considerable advance in coping with the impending surge of accelerators.

There is one other approach that is just as important to address: the need to raise the level of abstraction in the design process. As accelerators are integrated into the system, we must manage integration complexity as it will grow. Hence, we need new methodologies and frameworks where accelerators can be readily put into place without the need to overly scrutinize the overall SoC architecture every time a new accelerator is introduced [2].

4 Challenge #2: Accelerator Concurrency

Table 1 shows that there are multiple accelerators per usecase. Many of these accelerators are running concurrently. In fact, in all of the usecases, the accelerators are running simultaneously. Therefore, they tend to impose severe shared resource demands from the underlying architecture (e.g., DRAM/network-on-chip bandwidth). Hence, it is essential to understand how concurrent execution affects the overall system performance in a large SoC.

Shared resource contention is not a new problem for computer system architects. Recall the early days of multicore CPUs when techniques such as cache-fair thread scheduling at the OS for multicore processors was an important topic? Multicore processors introduced concurrently running threads but shared a global last level cache that was "transparent" to a traditional single-core OS that was oblivious to core count. The OS scheduler made simplistic time-multiplexing assumptions about single-core CPUs that did not directly apply to multicore CPUs. Consequently, this led to performance degradation when the single-core OS scheduler was given a multicore CPU with no new awareness or adaptations. It was only after the OS was made aware of multiple cores that the resource contention problem alleviated.

We are on the cusp of a similar situation with accelerators. Today, the execution of these accelerators is often hidden from the users through hardware abstraction layers. In Android, the Hardware Abstraction Layer (HAL) API provides an interface between the hardware drivers and the Android OS framework. The HAL defines an interface for hardware vendors to implement interfaces to their devices, thus enabling Android to be agnostic of the lower-level driver details. The interface is defined by Android and implemented by the hardware vendors.

While HAL-like abstraction layers serve us well to implement low-level functionality without affecting or modifying the higher-level system, such an approach comes with drawbacks. Guaranteeing real-time performance, orchestrating smooth execution when there are multiple competing usecases, becomes challenging since the OS is "blind" to the underlying hardware concurrency. In many usecases, there is an implicit real-time constraint. So, accelerators cannot be too slow. At the same time, there is little to no benefit in being too fast either, since any performance that is not perceivable to the end-user is generally a waste of energy. But this sort of contextual knowledge is often lost behind the abstraction layers. Hence, as ALP proliferates, the OS must have more fine-grained knowledge over its accelerators' execution.

In the future, we must explicitly think about cooperative scheduling mechanisms between the hardware and software/OS mechanisms to orchestrate accelerator concurrency. This may mean providing hooks at the hardware level to enable more intelligence at the software level, or developing policies at the software level to orchestrate the accelerator computation flows.

Table 1: How usecases (rows) concurrently use accelerators (columns)

	CPUs (AP)	Display	Media Scaler	GPU	Image Signal Proc.	JPEG	Pixel Visual Core	Video De-coder	Video En-coder	Dozens More
Photo Enhancing (HDR+)	X	X		X	X	X	X			
Video Capture	X	X		X	X				X	
Video Capture (HDR)	X	X		X	X				X	
Video Playback	X	X	X	X			X			
Image Recognition	X	X	X	X						

5 Challenge #3: Accelerator Communication

The accelerators in Table 1 do not operate in isolation. To execute a usecase, the accelerators must communicate with one another. But they do not communicate with one another directly. Instead, they communicate via main memory (or shared cache) coordinated by the CPU. In Android, each accelerator is a separate device, interfaced through a device driver. All communication is coordinated by the CPU, which acts as a master, first setting up the accelerators and then delegating work to them, and after which, the accelerator raises an interrupt to tell the CPU it is done. In most systems, the accelerator/CPU would then copy the data out to some other portion of main memory from where the CPU either re-copies the data to another accelerator's memory-mapped region and/or sets up a new DMA transfer; there are research solutions to avoid this. This leads to a vast amount of inefficient data movement.

Handshakings between the CPU, accelerators, and memory leads to performance slowdowns and added energy consumption. An analysis of Google consumer workloads on a mobile device reveals that data movement between the main memory system and computation units (e.g., CPUs, GPU, special-purpose accelerators) is a significant contributor to the total system energy [3]. As much as 62.7% of the overall system energy, on average, is spent on data movement between the main memory and computing units. As ALP rises and a system's accelerator count increases, the overheads will accumulate and degenerate performance.

Hence, efforts are needed to reduce data movement and improve communication. Are there lessons we can borrow from the past to mitigate the data movement problem? The short answer is a resounding yes. Data movement is not a new problem. For example, system architects have faced it before in the context of heterogeneous CPU+GPU systems. Initially, GPUs were discrete processing engines with their private memories into which the CPU copied data back and forth. Computation was offloaded to the GPUs, just as we do today with accelerators on a mobile SoC. But system architects quickly learned the severe penalty of offloading and innovated several new solutions to mitigate the overheads. They developed shared memory strategies and innovative hardware and software managed coherence protocols between CPUs and GPUs to improve the read and write bandwidths, and so forth.

For ALP, due to the large-scale accelerator integration challenge, we need to explore new caching/coherency, scratchpad, or queuing solutions, which often stand in the way of performance and power efficiency improvements. We must think of how to orchestrate dataflows (i.e., point-to-point communication between accelerators) directly in hardware to remove the CPU from the critical path. As ALP prevails, the CPU will become the bottleneck because the cost of taking interrupts, performing context switches, etc. is high. Future research must enable the OS to manage data flows at a high level (like the control plane in networking) without intimate involvement with most data transfers (the data plane).

6 Challenge #4: Accelerator Programmability

Another major challenge that lurks in the dark is the accelerator programming model. Today, many of the accelerators must be programmed in their preferred framework. For instance, programming a DSP often requires one to use a specialized Software Development Kit (SDK) that does not port well across SoCs. Hence, in the context of mobile, many application developers resist the use of specialized accelerators. Not only is programming domain-specific accelerators hard, but it also cripples code portability across different SoC vendors. Preferred frameworks also compromise performance predictability because there is no guarantee that all SoC vendors will provide an implementation for the accelerator. This situation is worse than that that preceded MPI's 1990's unification of message-passing libraries on supercomputers and clusters. Moreover, across the world's population of over two billion mobile devices, there is a long tail of legacy devices that cannot be retrofitted with accelerators [4].

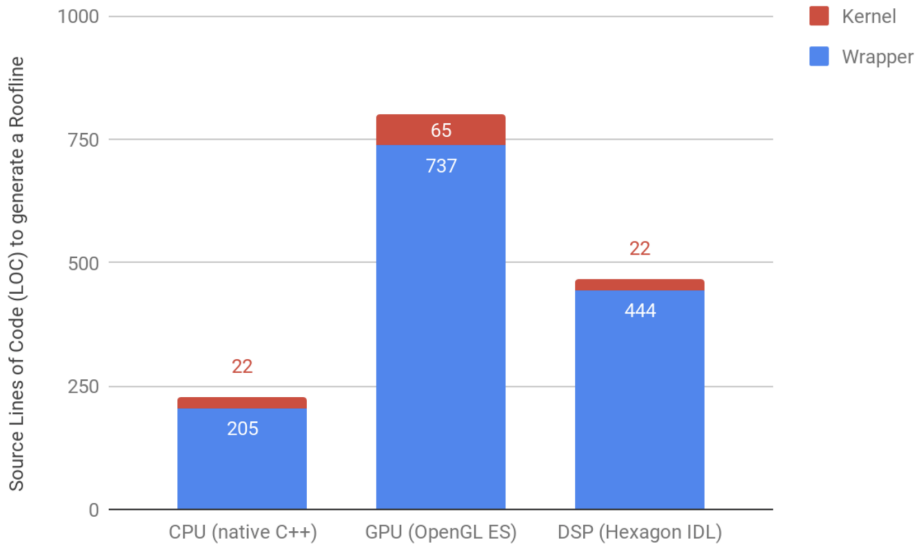


Figure 4: Roofline "Lines of Code" for CPU, GPU, or DSP.

Let's take a concrete example. Recently, we wrote a paper modeling the performance of mobile SoCs [5]. It required us to develop a microbenchmark to generate "rooflines" for the CPU, GPU, DSP, etc. on a smartphone. The Roofline kernel itself is a simple 30 lines of assembly/C++ application code. However, current accelerator programming interfaces require 100s of lines of non-obvious "wrapper" code that acts as substantial friction on using a single accelerator, much less multiple accelerators. While CPU programming has (relatively) easy, programming the GPU required using OpenGL ES including "shader storage buffer objects," and programming the DSP included setting up its operating modes with "FastRPC." An important aside that is not reflected in Figure 4 directly is the expertise and time investment needed to be productive in multiple languages and environments. All of this, combined with the fact that there is no guarantee that common mobile chipsets have the same set of accelerators, causes developers to cringe at the idea of specializing code for a chipset. These challenges are not impossible to overcome. Nonetheless, they stand in the way ALP progress.

Programmability is an inherently tough problem. But it is also an opportunity for all of us to innovate. We need new programming models and software libraries and packages that can ease us all into widespread and broad exploitation of ALP. Simply put, focusing on programmability is as necessary as the accelerators themselves, since ALP is the only way to continue improving in (cost-)performance for future workloads. The existing solutions are too ad hoc for widespread use, and we need more portable and robust software to support ALP.

All is not lost. Our past experiences can inform this quest. For instance, exploiting TLP was hard with low-level techniques like pThread primitives. It is not for the faint-hearted. Medium success occurred with middle-level approaches like OpenMP and MPI. Broad success hid TLP from most programmers with SQL above parallel database management systems, Map-Reduce for clusters, and, importantly, cloud services in general. Data-level parallelism also had limited success with vectorizing compilers, SIMD intrinsics, and GPU shader programming. Broad DLP success on GPUs required the SIMT programming model that enabled programmers to think of DLP as a thread per data item, democratizing the data-parallel vision after a quarter-century [6]. Domain-specific languages like Halide can help bridge some of the existing voids by targeting specific application domains like computer vision and creating an abstraction between the language and hardware optimizations. But more is needed!

7 A Call to Action

This paper identifies challenges and perhaps paths to solutions. We see these challenges as beyond the scope of any one individual or research groups' reach to conquer. They likely will require work from a community of researchers, who span across architecture, programming languages and operating systems, to work collaboratively. We must collectively work toward developing better "best practices" for targeting the right set of accelerators, managing accelerator concurrency, choreographing inter-accelerator dataflow/communication, and productively programming accelerators. So go forth and invent the future of ALP!

References

[1] J. L. Hennessy and D. A. Patterson. Computer architecture: A quantitative approach. *Elsevier*, 2019.

[2] Carloni L.P. The case for embedded scalable platforms. In *53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. ACM, 2016.

[3] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *ACM SIGPLAN Notices*, volume 53, pages 316–331. ACM, 2018.

[4] C.-J. et al Wu. Machine learning at facebook: understanding inference at the edge. In *High-Performance Computer Architecture (HPCA), 2019 IEEE 25th International Symposium on*. IEEE, 2019.

[5] M. D. Hill and V. Janapa Reddi. Gables: A roofline model for mobile socs. In *High-Performance Computer Architecture (HPCA), 2019 IEEE 25th International Symposium on*. IEEE, 2019.

[6] W. D. Hillis and Steele Jr. G. L. Data parallel algorithms. In *Communications of the ACM, vol. 29, no. 12*, page 1170–1183. ACM, 1986.