# Welcome to the Jungle

*In the twilight of Moore's Law, the transitions to multicore processors, GPU computing, and HaaS cloud computing are not separate trends, but aspects of a single trend – mainstream computers from desktops to 'smartphones' are being permanently transformed into heterogeneous supercomputer clusters. Henceforth, a single compute-intensive application will need to harness different kinds of cores, in immense numbers, to get its job done.*

*The free lunch is over. Now welcome to the hardware jungle.*

From 1975 to 2005, our industry accomplished a phenomenal mission: In 30 years, we put a personal computer on every desk, in every home, and in every pocket.

In 2005, however, mainstream computing hit a wall. In **"The Free Lunch Is Over"** (December 2004), I described the reasons for the then-upcoming industry transition from single-core to multi-core CPUs in mainstream machines, why it would require changes throughout the software stack from operating systems to languages to tools, and why it would permanently affect the way we as software developers have to write our code if we want our applications to continue exploiting Moore's transistor dividend.

2011 was special: it's the year that we completed the transition to parallel computing in all mainstream form factors, including tablets and smartphones

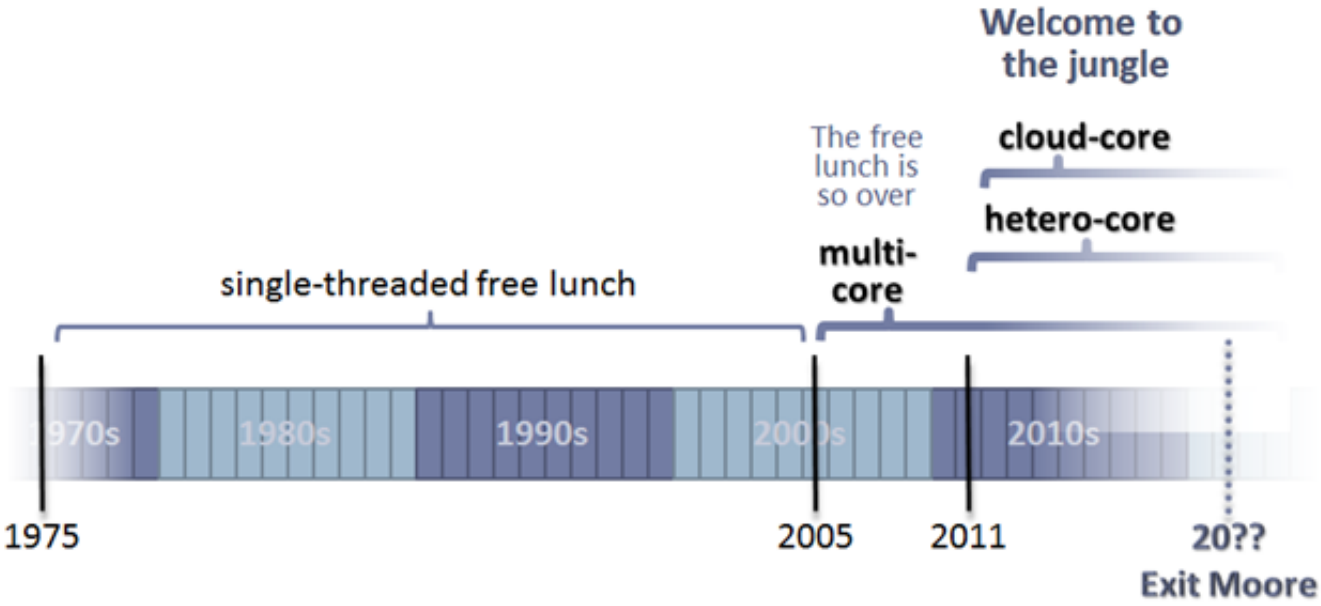
In 2005, our industry undertook a new mission: to put a *personal parallel supercomputer* on every desk, in every home, and in every pocket. 2011 was special: it's the year that we completed the transition to parallel computing in all mainstream form factors, with the arrival of multicore tablets (e.g., iPad 2, Playbook, Kindle Fire, Nook Tablet) and smartphones (e.g., Galaxy S II, Droid X2, iPhone 4S). 2012 will see us continue to build out multicore with mainstream quad- and eight-core tablets (as Windows 8 brings a modern tablet experience to x86 as well as ARM), and the last single-core gaming console holdout will go multicore (as

Nintendo's Wii U replaces Wii).

This time it took us just sixyears to deliver mainstream parallel computing in all popular form factors. And we know the transition to multicore is permanent, because multicore delivers compute performance that single-core cannot and there will always be mainstream applications that run better on a multi-core machine. There's no going back.

For the first time in the history of computing, mainstream hardware is no longer a single-processor von Neumann machine, and never will be again.

That was the first act.




## Overview: Trifecta

It turns out that multicore is just the first of three related permanent transitions that layer on and amplify each other.

**1. Multicore (2005-).** As above.

**2. Heterogeneous cores (2009-).** A single computer already typically includes more than one kind of processor core, as mainstream notebooks, consoles, and tablets all increasingly have both CPUs and compute-capable GPUs. The

open question in the industry today is not whether a single application will be spread across different kinds of cores, but only "how different" the cores should be – whether they should be basically the same with similar instruction sets but in a mix of a few big cores that are best at sequential code plus many smaller cores best at running parallel code (the Intel MIC model slated to arrive in 2012-2013, which is easier to program), or cores with different capabilities that may only support subsets of general-purpose languages like C and C++ (the current Cell and GPGPU model, which requires more complexity including language extensions and subsets).

Heterogeneity amplifies the first trend (multicore), because if some of the cores are smaller then we can fit more of them on the same chip. Indeed, 100x and 1,000x parallelism is already available today on many mainstream home machines – for programs that can harness the GPU.

We know the transition to heterogeneous cores is permanent, because different kinds of computations naturally run faster and/or use less power on different kinds of cores – including that different parts of the same application will run faster and/or cooler on a machine with several different kinds of cores.

**3. Elastic compute cloud cores (2010-).** For our purposes, "cloud" means specifically "hardware (or infrastructure) as a service" (HaaS) – delivering access to more computational hardware as an extension of the mainstream machine. This started to hit the mainstream with commercial compute cloud offerings from Amazon Web Services (AWS), Microsoft Azure, Google App Engine (GAE), and others.

Cloud HaaS again amplifies both of the first two trends, because it's fundamentally about deploying large numbers of nodes where each node is a mainstream machine containing multiple and heterogeneous cores. In the cloud, the number of cores available to a single application is scaling fast (e.g., in summer 2011, Cycle Computing delivered a 30,000-core cloud for

under $1,300/hour, using AWS) and the same heterogeneous cores are available in compute nodes (e.g., AWS already offers "Cluster GPU" nodes with dual nVIDIA Tesla M2050 GPU cards, enabling massively parallel and massively distributed CUDA applications).

In short, parallelism is not just in full bloom, but increasingly in full variety.

This article will develop four key points:

1. **Moore's End.** We can observe clear evidence that Moore's Law is ending, because we can point to a pattern that precedes the end of exploiting any kind of resource. But there's no reason to panic, because Moore's Law limits only one kind of scaling, and we have already started another kind.
2. **Mapping one trend, not three.** Multicore, heterogeneous cores, and HaaS cloud computing are not three separate trends, but aspects of a single trend: putting a *personal heterogeneous supercomputer cluster* on every desk, in every home, and in every pocket.
3. **The effect on software development.** As software developers, we will be expected to enable a single application to exploit a "jungle" of enormous numbers of cores that are increasingly different in kind (specialized for different tasks) and different in location (from local to very remote; on-die, in-box, on-premises, in-cloud). The jungle of heterogeneity will continue to spur deep and fast evolution of mainstream software development, but we can predict what some of the changes will be.
4. **Three distinct near-term stages of Moore's End.** And why "smartphones" aren't, really.

Let's begin with the end… of Moore's Law.

## Mining Moore's Law

We've been hearing breathless "Moore's Law is ending" announcements for years. That Moore's Law will end was never news; every exponential pro-

gression must. Although it didn't end when some prognosticators expected, its end is possible to forecast – we just have to know what to look for, and that is *diminishing returns*.

A key observation is that exploiting Moore's Law is like exploiting a gold mine or any other kind of resource. Exploiting a gold ore deposit never just stops abruptly; rather, running a mine goes through phases of increasing costs and diminishing returns until finally the gold that's left in that patch of ground is no longer commercially exploitable and operating the mine is no longer profitable.

*exploiting Moore's Law is like exploiting a gold mine, and has followed the same pattern*

Mining Moore's Law has followed the same pattern. Let's consider its three major phases, where we are now in transition from Phase II to Phase III. And throughout this discussion, never forget that the only reason Moore's Law is interesting at all is because we can transform its raw resource (more transistors) into a useful form (either greater computational throughput or lower cost).

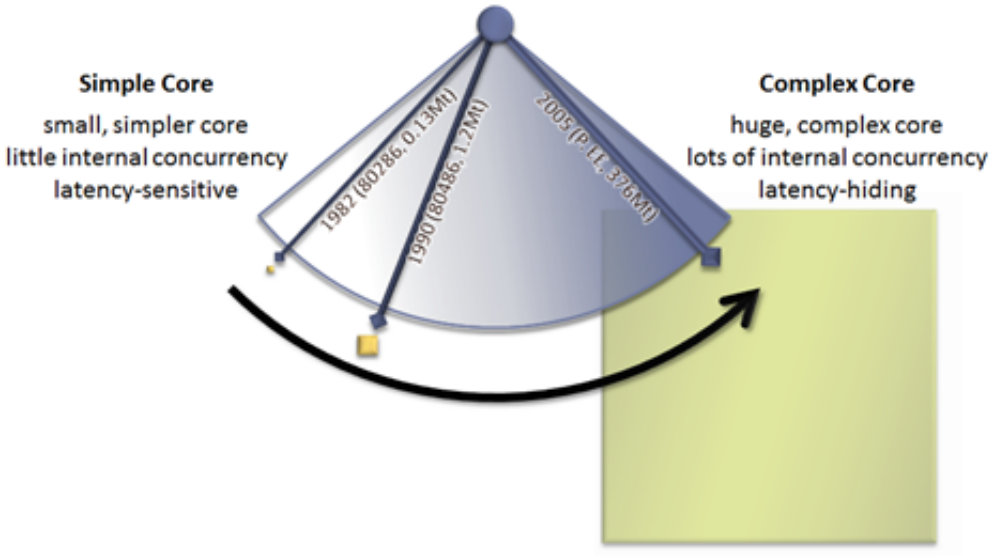
***Phase I, Moore's Motherlode = Unicore "Free Lunch" (1975-2005)***

When you first find an ore deposit and open a mine, you focus your efforts on the motherlode, where everybody gets to enjoy a high yield and a low cost per pound of gold extracted.

For 30 years, mainstream processors mined Moore's motherlode by using their growing transistor budgets to make a single core more and more complex so that it could execute a single thread faster. This was wonderful because it meant the performance was *easily exploitable* – compute-bound software would get faster with relatively little effort. Mining this motherlode in mainstream microprocessors went through two main subphases as the pendulum swung from simpler to increasingly complex cores:

- In the 1970s and 1980s, each chip generation could use most of the extra

transistors to add One Big Feature (e.g., on-die floating point unit, pipelining, out of order execution) that would make single-threaded code run faster.
- In the 1990s and 2000s, each chip generation started using the extra transistors to add or improve two or three smaller features that would make single-threaded code run faster, and then five or six smaller features, and so on.




The figure at right illustrates how the pendulum swung toward increasingly complex single cores, with three sample chips: the 80286, 80486, and Pentium Extreme Edition 840. Note that the chips' boxes are to scale by number of transistors.

By 2005, the pendulum had swung about as far as it could go toward the complex single-core model. Although the motherlode has been mostly exhausted, we're still scraping some ore off its walls in the form of some continued improvement in single-threaded code performance, but no longer at the historically delightful exponential rate.

### *Phase II, Secondary Veins = Homogeneous Multicore (2005-)*

As a motherlode gets used up, miners concentrate on secondary veins that are still profitable but have a more moderate yield and higher cost per

pound of extracted gold. So when Moore's unicore motherlode started getting mined out, we turned to mining Moore's secondary veins – using the additional transistors to make more cores per chip. Multicore let us continue to deliver exponentially increasing compute throughput in mainstream computers, but in a form that was *less easily exploitable* because it placed a greater burden on software developers who had to write parallel programs that could use the hardware.

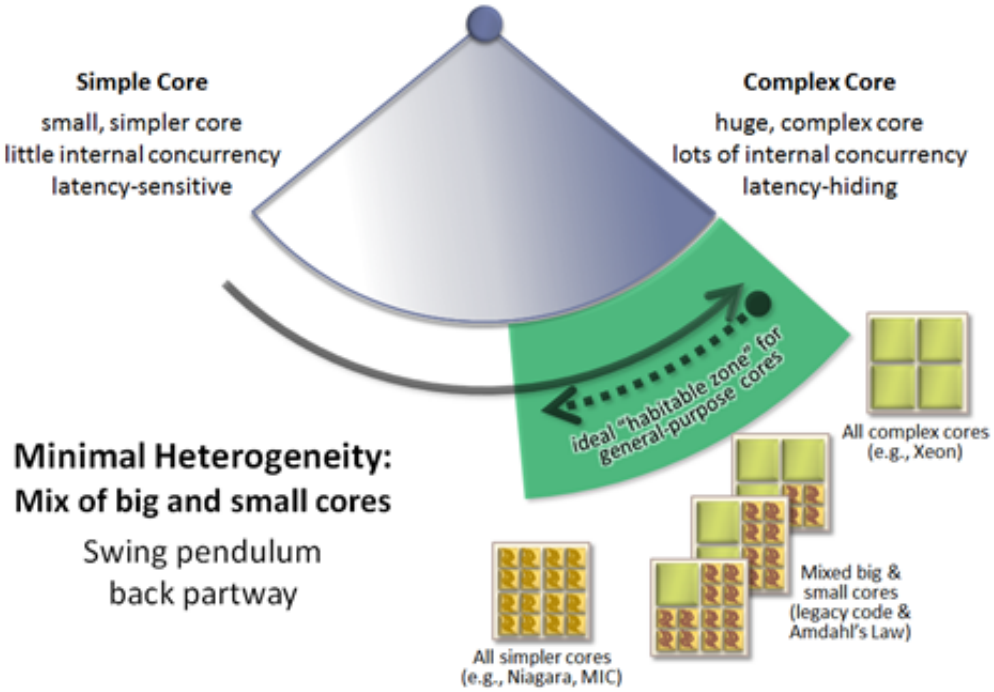
Moving into Phase II took a lot of work in the software world. We've had to learn to write "new free lunch" applications – ones that have lots of latent parallelism and so can once again ride the wave to run the same executable faster on next year's hardware, hardware that still delivers exponential performance gains but primarily in the form of additional cores. And we're mostly there – we have parallel runtimes and libraries like Intel Threading Building Blocks (TBB) and Microsoft Parallel Patterns Library (PPL), parallel debuggers and parallel profilers, and updated operating systems to run them all.

But this time the phase didn't last 30 years. We barely have time to catch our breath, because Phase III is already beginning.

### *Phase III, Tertiary Veins = Heterogeneous Cores (2011-)*

As our miners are forced to move into smaller and smaller veins, yields diminish and costs rise. Our intrepid miners are trying harder and harder, but for less reward, by turning to Moore's tertiary veins: Using Moore's extra transistors to make, not just more cores, but also different kinds of cores – and in very large numbers, because the different cores are often smaller and swing the pendulum back toward the left.

There are two main categories of heterogeneity.




**Big/fast vs. small/slow cores.** The smallest amount of heterogeneity is when all the cores are general-purpose cores with the same instruction set, but some cores are beefier than others because they contain more hardware to accelerate execution (notably by hiding memory latency using various forms of internal concurrency). In this model, some cores are big complex ones that are optimized to run the sequential parts of a program really fast, while others are smaller cores that are optimized to get better total throughput for the scalably parallel parts of the program. However, even though they use the same instruction set, the compiler will often want to generate different code; this difference can become visible to the programmer if the programming language must expose ways to control code generation. This is Intel's approach with Xeon (big/fast) and MIC (small/slow) which both run approximately the x86 instruction set.

**General vs. specialized cores.** Beyond that, we see systems with multiple cores having different capabilities, including that some cores may not be able to support all of a mainstream language like C or C++: In 2006-2007, with the arrival of the PlayStation 3, the IBM Cell processor led the way by incorporating different kinds of cores on the same chip, with a single general-purpose core assisted by eight or more special-purpose SPU cores. Since 2009,

we have begun to see mainstream use of GPUs to perform computation instead of just graphics. Specialized cores like SPUs and GPUs are attractive when they can run certain kinds of code more efficiently, both faster and more cheaply (e.g., using less power), which is a great bargain if your workload fits it.

GPGPU is especially interesting because we already have an *underutilized installed base*: A significant percentage of existing mainstream machines already have compute-capable GPUs just waiting to be exploited. With the June 2011 introduction of AMD Fusion and the November 2011 launch of NVIDIA Tegra 3, systems with CPU and GPU cores on the same chip is becoming a new norm. That installed base is a big carrot, and creates an enormous incentive for compute-intensive mainstream applications to leverage that patiently waiting hardware. To date, a few early adopters have been using technologies like CUDA, OpenCL, and more recently C++ AMP to harness GPUs for computation. Mainstream application developers who care about performance need to learn to do the same.

| Exploitability | Summary | Stages / Alternatives | Software Impact | Examples |
|---|---|---|---|---|
| Moore's motherlode:<br>Unicore | Make single core more complex to run single-threaded code faster | 1970s & 1980s: Add one big feature per chip generation<br>1990s & 2000s: Several smaller improvements/gen | The free lunch: Ship an EXE that will just run faster on new hardware | Single-core x86, SPARC, ARM |
| Secondary veins:<br>Multicore | Deliver more cores per chip | 2005-20??: Deliver several big cores<br>2012-20??: Deliver lots of smaller cores | Must write parallel code<br>Must write very parallel code | SPARC Niagara, x86<br>Intel MIC |
| Tertiary veins:<br>Hetero manycore | Deliver different kinds of cores<br>Because the cores are simpler, yields large one-time jump in #cores | Big/fast (complex) vs. small/slow (simpler)<br>General-purpose (traditional CPU core) vs. special-purpose (e.g., GPU core) | Still less exploitable: Must write heterogeneous and locally distributed parallel code | Cell (e.g., PS3)<br>Intel Xeon+MIC<br>AMD and NVIDIA GPUs, incl. on-die (Fusion and Tegra 3) |

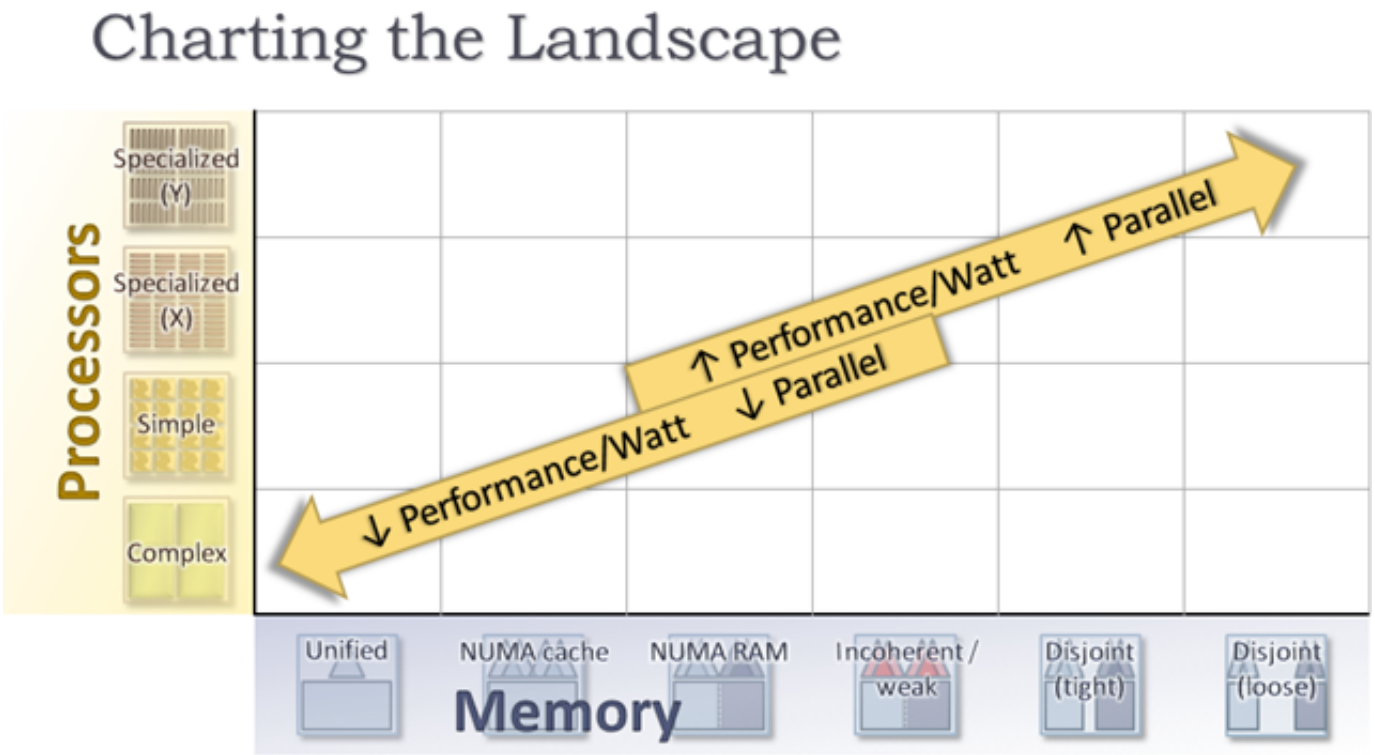
But that's pretty much it – we currently know of no other major ways to ex-

ploit Moore's Law for compute performance, and once these veins are exhausted it will be largely mined out.

We're still actively mining for now, but the writing on the wall is clear: "*mene mene* diminishing returns" demonstrate that we've entered the endgame.

## On the Charts: Not Three Trends, but One Trend

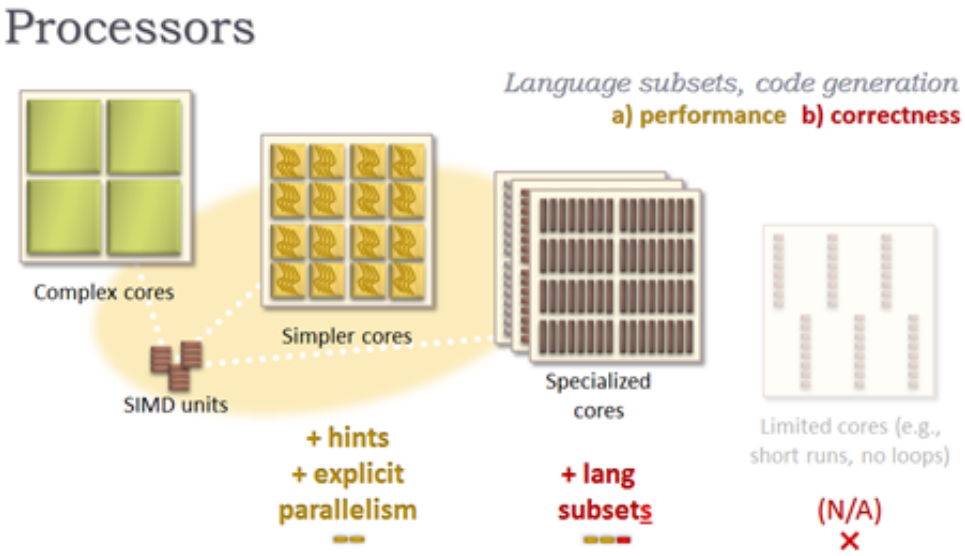
Next, let's put all of this in perspective by showing that multicore, hetero-core, and cloud-core are not three trends, but aspects of a single trend. To show that, we have to show that they can be plotted on the same map. Here is an appropriate map that lets us chart out where processor core architectures are going, where memory architectures are going, and visualize just where we've been digging around in the mine so far:




First we'll describe each axis, then map out past and current hardware to spot trends, and finally draw some conclusions about where hardware is likely to concentrate.

### *Processor Core Types*

The vertical axis shows processor core architectures. From bottom to top, they form a continuum of increasing performance and scalability, but also of increasing




restrictions on programs and programmers in the form of additional performance issues (yellow) or correctness issues (red) added at each step.

**Complex cores** are the "big" traditional ones, with the pendulum swung far to the right in the "habitable zone." These are best at running sequential code, including code limited by Amdahl's Law.

**Simple cores** are are the "small" traditional ones, toward the left of the "habitable zone." These are best at running parallelizable code that still requires the full expressivity of a mainstream programming language.

**Specialized cores** like those in GPUs, DSPs, and Cell's SPUs are more limited, and often do not yet fully support all features of mainstream languages (e.g., exception handling). These are best for running highly parallelizable code that can be expressed in a subset of a language like C or C++; for example, Xbox Kinect skeletal tracking requires using the CPU and the GPU cores on the console, and would be impossible otherwise.

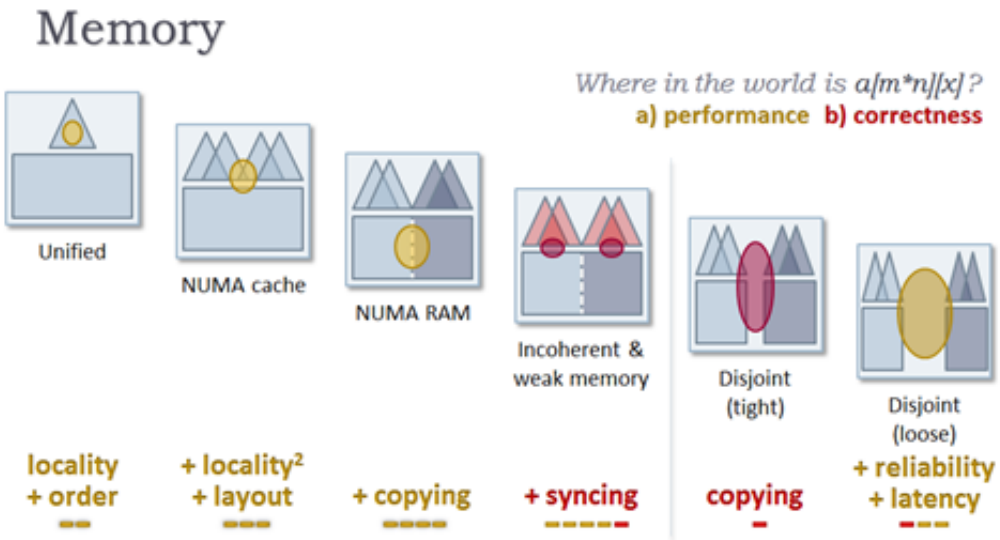
The further you move upward on the chart (to the right in the blown-up fig-

ure), the better the performance throughput and/or the less power you need, but the more the application code is constrained as it has to be more parallel and/or use only subsets of a mainstream language.

Future mainstream hardware will likely contain all three basic kinds of cores, because many applications have all these kinds of code in the same program, and so naturally will run best on a heterogeneous computer that has all these kinds of cores. For example, most PS3 games, all Kinect games, and all CUDA/OpenCL/C++AMP applications available today could not run well or at all on a homogeneous machine, because they rely on running parts of the same application on the CPU(s) and other parts on specialized cores. Those applications are just the beginning.

### *Memory Architectures*

The horizontal axis shows six common memory architectures. From left to right, they form a continuum of increasing performance and scalability, but (except for one important discontinuity) also increasing work for programs and programmers to deal with performance issues (yellow) or correctness issues (red). In the blown-up figure, triangles represent cache and lower boxes represent RAM. A processor core (ALU) sits at the top of each cache "peak."



**Unified memory is tied to the unicore motherlode** and the memory hierarchy is wonderfully simple – a single mountain with a core sitting on top. This describes essentially all mainstream computers from the dawn of computing until the mid-2000s. This delivers a simple programming model: Every pointer (or object reference) can address every byte, and every byte is equally "far away" from the core. Even here, programmers need to be conscious of at least two basic cache effects: *locality*, or how well "hot" data fits into cache; and *access order*, because modern memory architectures love sequential access patterns. (For more on this, see my Machine Architecture talk.)

**NUMA cache** retains a single chunk of RAM, but adds multiple caches. Now instead of a single mountain, we have a mountain range with multiple peaks, each with a core on top. This describes today's mainstream multi-core devices. Here we still enjoy a single address space and pretty good performance as long as different cores access different memory, but programmers now have to deal with two main additional performance effects: *locality* matters in new ways because some peaks are closer to each other than others (e.g., two cores that share an L2 cache vs. two cores that share only L3 or RAM), and *layout* matters because we have to keep data physically close together if it's used together (e.g., on the same cache line) and apart if it's not (e.g., to avoid the ping-pong game of false sharing).

**NUMA RAM** further fragments memory into multiple physical chunks of RAM, but still exposes a single logical address space. Now the performance valleys between the cores get deeper, because accessing RAM in a chunk not local to this core incurs a trip across the bus. Examples include bladed servers, symmetric multi-processor (SMP) desktop computers with multiple sockets, and newer GPU architectures that provide a unified address space view of the CPU's and GPU's memory but leave some memory physically closer to the CPU and other memory closer to the GPU. Now we add another item to the menu of what a performance-conscious programmer needs to think about: *copying*. Just because we can form a pointer to anything doesn't

mean we always should, if it means reaching across an expensive chasm on every access.

**Incoherent and weak memory** makes memory be by default unsynchronized, in the hope that allowing each core to have its own divergent view of the state of memory can make them run faster, at least until memory must inevitably be synchronized again. As of this writing, the only remaining mainstream CPUs with weak memory models are current PowerPC and ARM processors (popular despite their memory models rather than because of them; more on this below). This model still has the simplicity of a single address space, but now the programmer further has to take on the burden of *synchronizing* memory himself.

*weak memory models are a performance experiment that is in the process of failing in the marketplace*

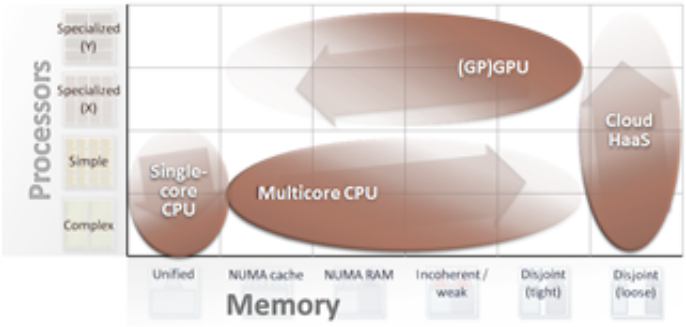
Clarification: By "weak (hardware) memory model" CPUs I mean specifically ones that do not natively support efficient sequentially consistent atomics, because on the software side programming languages have converged on the strong "sequential consistency for data-race-free programs" (SC-DRF, roughly aka DRF0 or RCsc) as the default (C11, C++11) or only (Java 5+) supported software memory model for software. Hardware that supports weaker memory models than that are permanently disadvantaged and will either become stronger (as ARMv8 is now doing by adding SC acquire/release instructions) or atrophy. The two main hardware architectures with what I called "weak" memory models were ARMv7 and POWER. ARMv8 is upgrading to SC acquire/release, as predicted, and it remains to be seen whether POWER will upgrade or atrophy. I've seen some call x86 "weak", but x86 has always been the poster child for a *strong* hardware memory model in all of our software memory model discussions for Java, C, and C++ during the 2000s. Therefore it's clear that "weak" and "strong" are not useful terms because they mean different things for software and hardware memory models, and I've updated the text to clarify this.

**Disjoint (tightly coupled)** memory bites the bullet and lets different cores

see different memory, typically over a shared bus, while still running as a tightly-coupled unit that has low latency and whose reliability is still evaluated as a single unit. Now the model turns into a tightly-clustered group of mountainous islands, each with core-tipped mountains of cache overlooking square miles of memory, and connected by bridges with a fleet of trucks expediting goods from point to point – bulk transfer operations, message queues, and similar. In the mainstream, we see this model used by 2009-2011 vintage GPUs whose on-board memory is not shared with the CPU or with each other. True, programmers no longer enjoy having a single address space and the ability to share pointers, but in exchange we have removed the entire set of programmer burdens accumulated so far and replaced them with a single new responsibility: *copying* data between islands of memory.

**Disjoint (loosely coupled) is the cloud** where cores spread out-of-box into different rooms and buildings and datacenters. This moves the islands farther apart, and replaces the bus "bridges" with network "speedboats" and "tankers." In the mainstream, we see this model in HaaS cloud computing offerings; this is the commoditization of the compute cluster. Programmers now have to arrange to deal with two additional concerns, which often can be abstracted away by libraries and runtimes: *reliability* as nodes can come and go, and *latency* as the islands are farther apart.

### *Charting the Hardware*

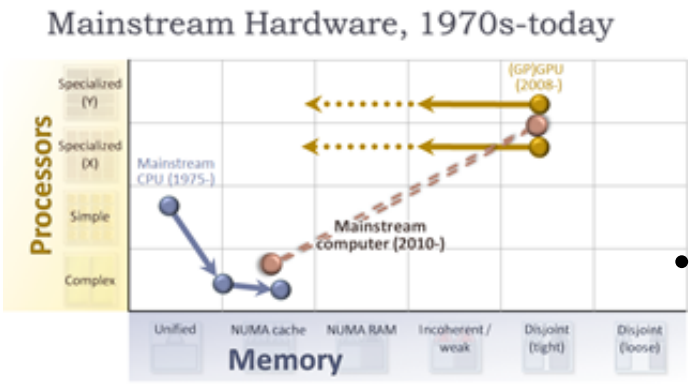



All three trends are just aspects of a single trend: filling out the chart and enabling heterogeneous parallel computing. The chart wants to be filled out because there are workloads that are naturally suited to each of these boxes, though some boxes are more popular than others.

To help visualize the filling-out process more concretely, why not check to

see how mainstream hardware has progressed on this chart? The easiest place to start is the long-standing mainstream CPU and more recent GPU:

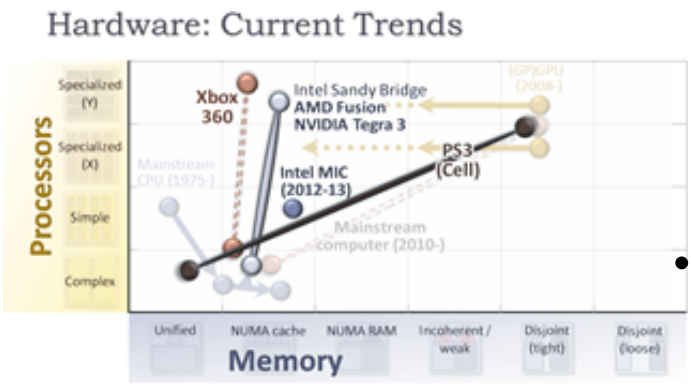
- From the 1970s to the 2000s, CPUs started with simple single cores and then moved downward as the pendulum swung to increasingly complex cores. They hugged the left side of the chart by staying single-core as long as possible, but in 2005 they ran out of room and turned toward multi-core NUMA cache architectures.




- Meanwhile, in the late 2000s, mainstream GPUs started to be capable of handling computational workloads. But because they started life in an add-on discrete GPU card format where graphics-specific cores and memory were physically located away from the CPU and system RAM, they started further upward and to the right (Specialized / Disjoint (local)). GPUs have been moving leftward to increasingly unified views of memory, and slightly downward to try to support full mainstream languages (e.g., add exception handling support).
- Today's typical mainstream computer includes both a CPU and a discrete or integrated GPU. The dotted line in the graphic denotes cores that are available to a single application because they are in the same device, but not on the same chip.

Now we are seeing a trend to use CPU and specialized (currently GPU) cores with very tightly coupled memory, and even on the same die:

- In 2005, the Xbox 360 sported a multi-core CPU and GPU that could not only directly access the same RAM, but had the very unusual feature that they could share even L2 cache.
- In 2006 and 2007, the Cell-based PS3 console sported a single processor having both a single general-purpose core and eight special-purpose

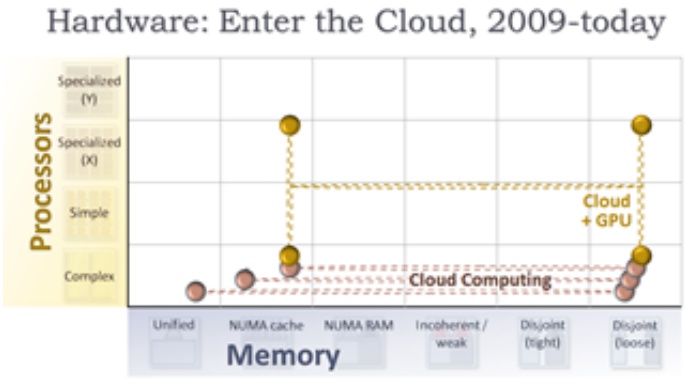



SPU cores. The solid line in the graphic denotes cores that are on the same chip, not just in the same device.

- In June 2011 and November 2011, respectively, AMD and NVIDIA launched the Fusion and Tegra 3 architectures, multi-core CPU chips that sported a compute-class GPU (hence extending vertically) on the same die (hence well to the left).
- Intel has also shipped the Sandy Bridge line of processors, which includes an integrated GPU that is not yet as compute-capable but continues to grow. Intel's main focus has been the MIC effort of more than 50 simple general-purpose x86-like cores on the same die, expected to be commercially available in the near future.

Finally, we complete the picture with cloud HaaS:

- In 2008 and 2009, Amazon, Microsoft, Google, and other vendors began rolling out their cloud compute offerings. AWS, Azure, and GAE support an elastic cloud of nodes each of which is a traditional computer ("big-core" and loosely coupled, therefore on the bottom right corner of the chart) where each node in the cloud has a single core or multiple CPU cores (the two lower-left boxes). As before, the dotted line denotes that all of the cores are available to a single application, and the network is just another bus to more compute cores.

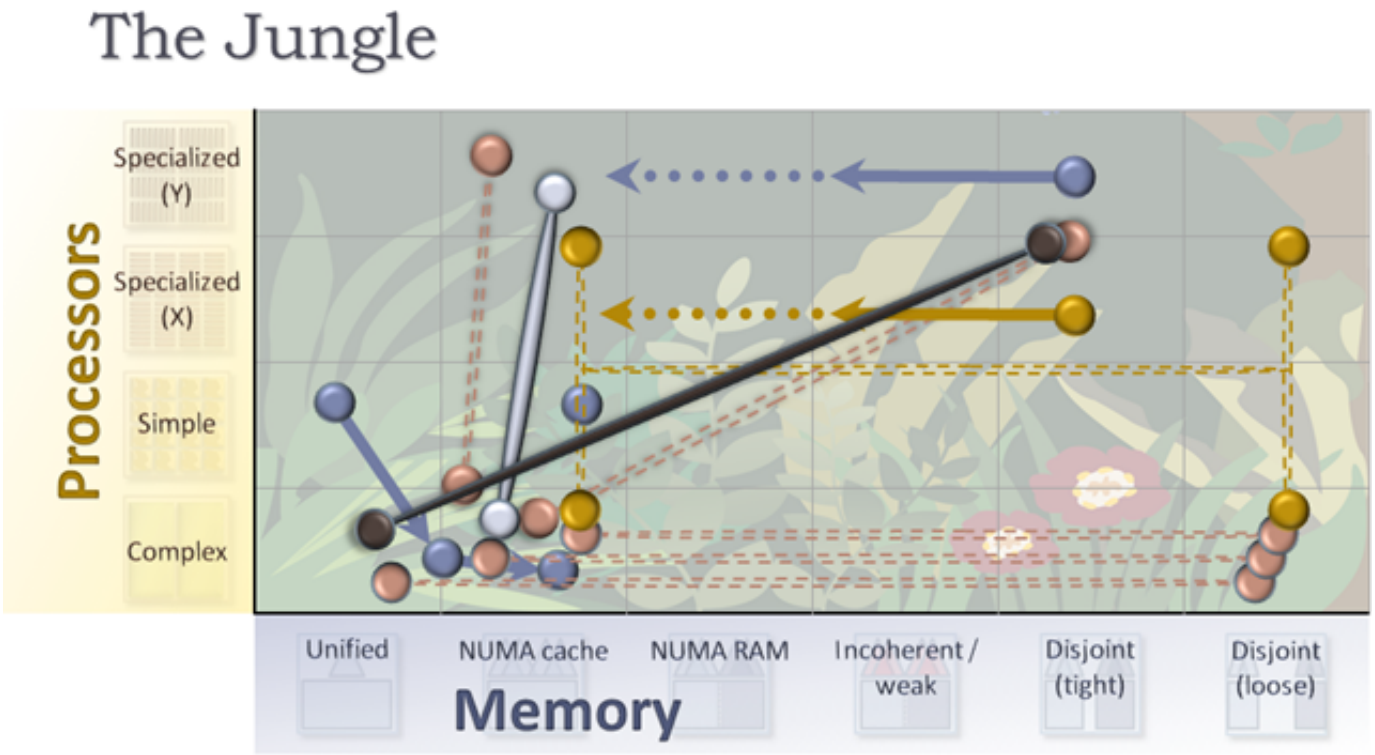



- Since November 2010, AWS also supports compute instances that contain both CPU cores and GPU cores, indicated by the H-shaped virtual machine where the application runs on a cloud of loosely-coupled nodes with disjoint memory (right column) each of which contains both CPU

and GPU cores (currently not on the same die, so the vertical lines are still dotted).

### *The Jungle*

Putting it all together, we get a noisy profusion of life and color:




This may look like a confused mess, so let's notice two things that help make sense of it.

First, every box has a workload that it's best at, but some boxes (particularly some columns) are more popular than others. Two columns are particularly less interesting:

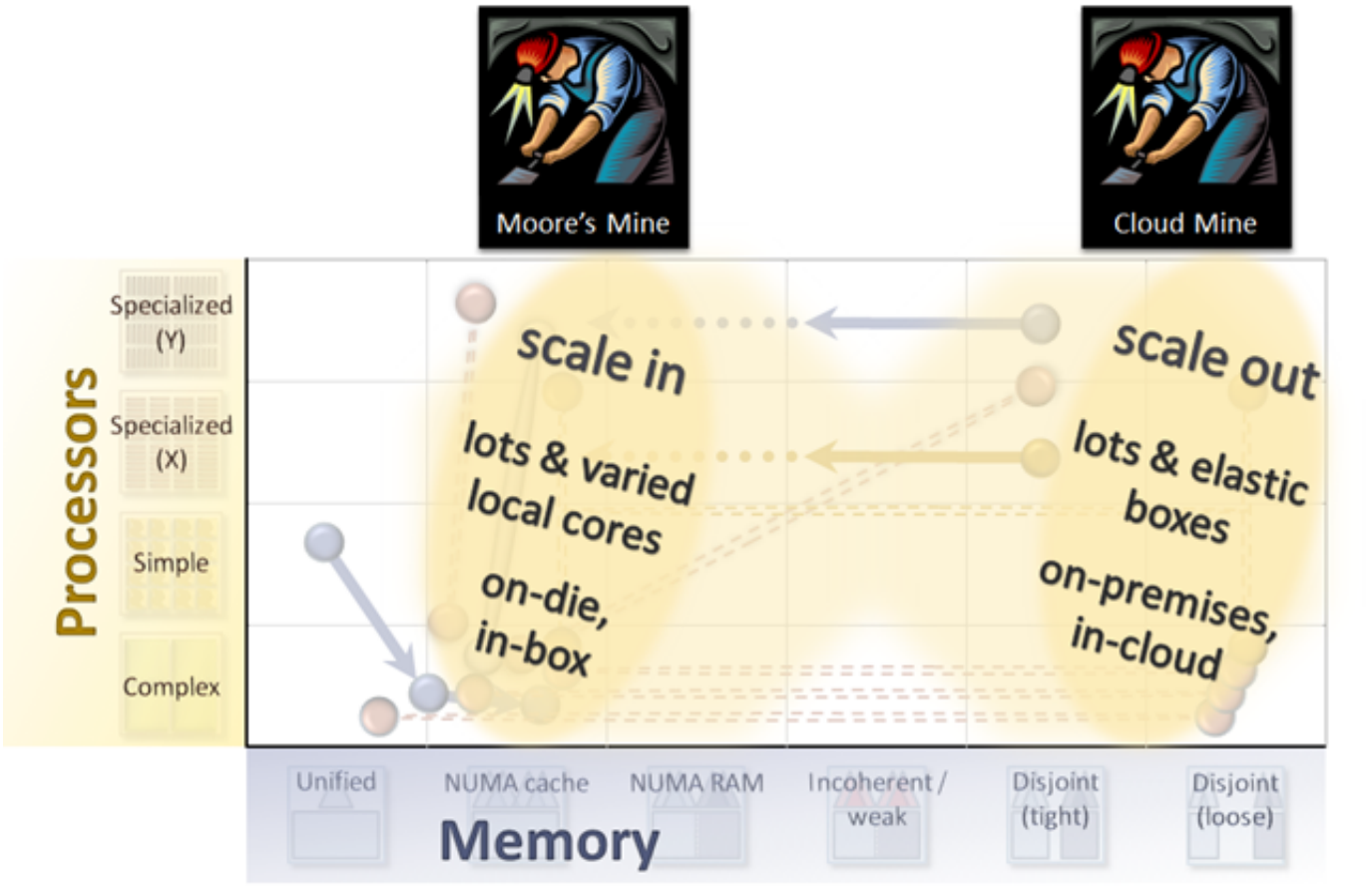
*every box has a workload that it's best at, but some boxes (particularly some columns) are more popular than others*

- Fully unified memory models are only applicable to single-core, which is being essentially abandoned in the mainstream.
- Incoherent/weak hardware memory models (those that do not efficiently support sequential consistency for data race free programs, roughly

aka DRF0 or RCsc) are a performance experiment that is in the process of failing in the marketplace. On the hardware side, the theoretical performance benefits that come from letting caches work less synchronously have already been largely duplicated in other ways by mainstream processors having stronger memory models. On the software side, all of the mainstream general-purpose languages and environments (C, C++, Java) have largely rejected weak memory models, and require a coherent model that is technically called "sequential consistency for data race free programs" as either their only supported memory model (Java) or their default memory model (ISO C++11, ISO C11). Nobody is moving toward the middle vertical incoherent/weak memory strip of the chart; at best they're moving through it to get to the other side, but nobody wants to stay there. (Note: x86 has always been considered a "strong" hardware memory model and supports sequentially consistent atomics efficiently, as does the recently-announced ARMv8 architecture with its new *ldra* and *strl* instructions; POWER and ARMv7 notoriously do not support SC atomics efficiently.)

But all other boxes, including all rows (processors), continue to be strongly represented, and we realize why that's true – because different parts of even the same application naturally want to run on different kinds of cores.

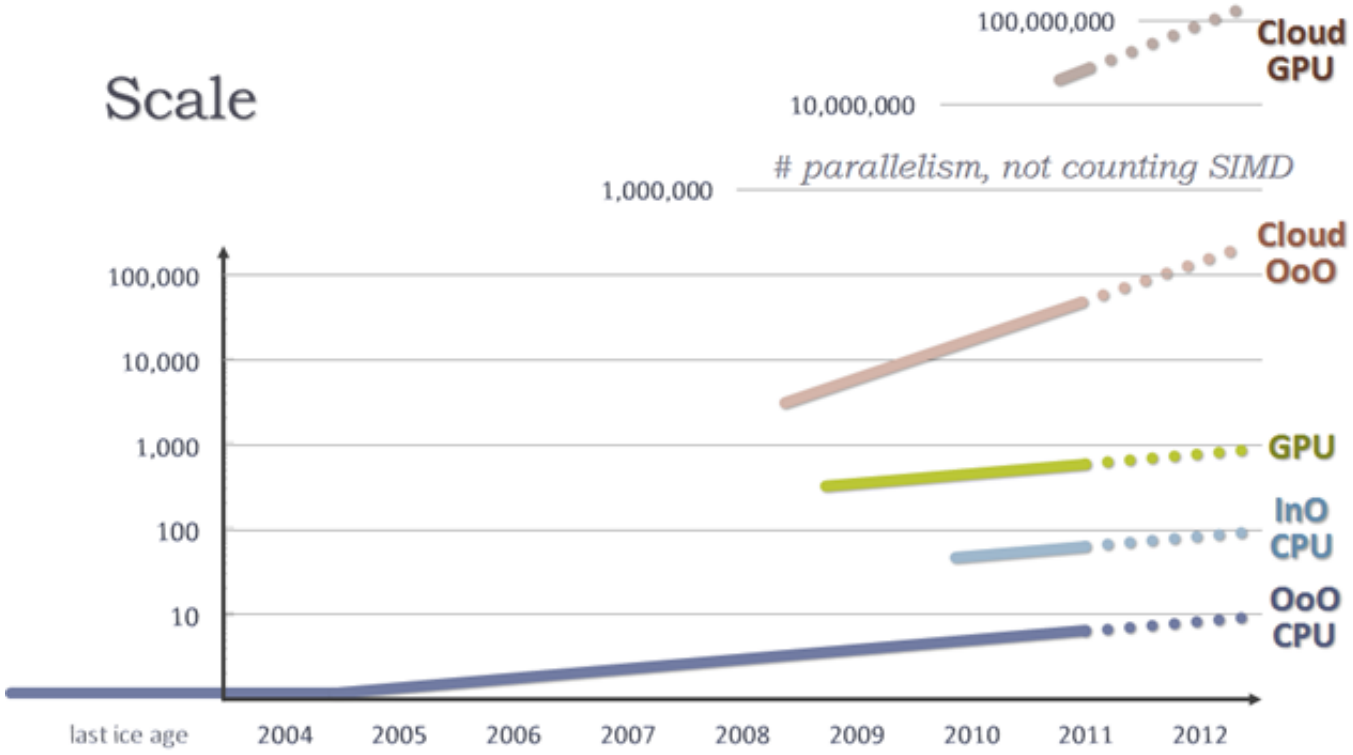
Second, let's clarify the picture by highlighting and labeling the two regions that hardware is migrating toward:




Here again we see the first and fourth columns being deemphasized, as hardware trends have begun gradually coalescing around two major areas. Both areas extend vertically across all kinds of cores – and the most important thing to note is that these represent *two mines*, where the area to the left is the Moore's Law mine.

- **Mine #1: "Scale in" = Moore's Law.** Local machines will continue to use large numbers of heterogeneous local cores, either in-box (e.g., CPU with discrete GPU) or on-die (e.g., Sandy Bridge, Fusion, Tegra 3). We'll see core counts increase until Moore's Law ends, and then stabilize core counts for individual local devices.
- **Mine #2: "Scale out" = distributed cloud.** Much more importantly, we will continue to see a cornucopia of cores delivered via compute clouds, either on-premises (e.g., cluster, private cloud) or in public clouds. This is a brand new mine directly enabled by the lower coupling of disjoint memory, especially loosely coupled distributed nodes.

The good news is that we can heave a sigh of relief at having found another mine to open. The even better news is that the new mine has a far faster growth rate than even Moore's Law. Notice the slopes of the lines when we graph the amount of parallelism available to a single application running on various architectures:




The bottom three lines are mining Moore's Law for "scale-in" growth, and their common slope reflects Moore's wonderful exponent, just shifted upward or downward to account for how many cores of a given size can be packed onto the same die. The top two lines are mining the cloud (with CPUs and GPUs, respectively) for "scale-out" growth – and it's even better.

If hardware designers merely use Moore's Law to deliver more big fat cores, on-device hardware parallelism will stay in double digits for the next decade, which is very roughly when Moore's Law is due to sputter, give or take about a half decade. If hardware follows Niagara's and MIC's lead to go back to simpler cores, we'll see a one-time jump and then stay in triple digits. If we all learn to leverage GPUs, we already have 1,500-way parallelism in modern graphics cards (I'll say "cores" for convenience, though that word

means something a little different on GPUs) and likely reach five digits in the decade timeframe.

But all of that is eclipsed by the scalability of the cloud, whose growth line is already steeper than Moore's Law because we're better at quickly deploying and using cost-effective networked machines than we've been at quickly jam-packing and harnessing cost-effective transistors. It's hard to get data on the current largest cloud deployments because many projects are private or secret, but the largest documented public cloud apps (which don't use GPUs) are already harnessing over 30,000 cores *for a single computation*. I wouldn't be surprised if undocumented projects are exceeding 100,000 cores today. And that's general-purpose cores; if you add GPU-capable nodes to the mix, add two more zeroes.

*all of that is eclipsed by the scalability of the cloud, whose growth line is already steeper than Moore's Law because we're better at quickly deploying and using cost-effective networked machines than we've been at quickly jam-packing and harnessing cost-effective transistors*

Such massive parallelism, already available for rates of under $1,300/hour for a 30,000-core cloud, is game-changing. If you doubt that, here is a boring example that doesn't involve advanced augmented reality or spook-level technomancery: How long will it take someone who's stolen a strong password file (which we'll assume is correctly hashed and salted and contains no dictionary passwords) to retrieve 90% of the passwords by brute force using a publicly available GPU-enabled compute cloud? Hint: An AWS dual-Tesla node can test on the order of 20 billion passwords per second, and clouds of 30,000 nodes are publicly documented (of course, Amazon won't say if it has that many GPU-enabled nodes for hire; but if it doesn't now, it will soon). To borrow a tired misquote, 640 trillion affordable attempts per second should be enough for anyone. But if that's not enough for you, not to worry; just wait a small number of years and it'll be 640 quadrillion affordable attempts per second.

## What It Means For Us: A Programmer's View

How will all of this change the way we write our software, if we care about harnessing mainstream hardware performance? The basic conclusions echo and expand upon ones that I proposed in "The Free Lunch is Over":

- **Applications will need to be at least massively parallel, and ideally able to use non-local cores and heterogeneous cores,** if they want to fully exploit the long-term continued exponential growth in compute throughput being delivered both in-box and in-cloud. After all, soon the vast majority of compute cores available to a mainstream application will be non-local.
- **Efficiency and performance optimization will get more, not less, important.** We're being asked to do more (new experiences like sensor-based UIs and augmented reality) with less hardware (constrained mobile form factors and the eventual plateauing of scale-in when Moore's Law ends). In December 2004 I wrote: "Those languages that already lend themselves to heavy optimization will find new life; those that don't will need to find ways to compete and become more efficient and optimizable. Expect long-term increased demand for performance-oriented languages and systems." This is still true; witness the resurgence of interest in C++ in 2011 and onward, primarily because of its expressive flexibility and performance efficiency. A program that is twice as efficient has two advantages: it will be able to run twice as well on a local disconnected device especially when Moore's Law can no longer deliver local performance improvements in any form; and it will always be able to run at half the power and cost on an elastic compute cloud even as those continue to expand for the indefinite future.
- **Programming languages and systems will increasingly be forced to deal with heterogeneous distributed parallelism.** As previously predicted, just basic homogeneous multicore has proved to be a far bigger event for languages than even object-oriented programming was, because some languages (notably C) could get away with ignoring objects while still remaining commercially relevant for mainstream software development. No mainstream language, including the just-ratified C11

standard, could ignore basic concurrency and parallelism and stay relevant in even a homogeneous-multicore world. Now expect all mainstream languages and environments, including their standard libraries, to develop explicit support for at least distributed parallelism and probably also heterogeneous parallelism; they cannot hope to avoid it without becoming marginalized for mainstream app development.

Expanding on that last bullet, what are some basic elements we will need to add to mainstream programming models (think: C, C++, Java, and .NET)? Here are a few basics I think will be unavoidable, that must be supported explicitly in one form or another.

- **Deal with the processor axis' lower section by supporting compute cores with different performance (big/fast, slow/small).** At minimum, mainstream operating systems and runtimes will need to be aware that some cores are faster than others, and know which parts of an application want to run on which of those cores.
- **Deal with the processor axis' upper section by supporting language subsets, to allow for cores with different capabilities including that not all fully support mainstream language features.** In the next decade, a mainstream operating system (on its own, or augmented with an extra runtime like the Java/.NET VM or the ConcRT runtime underpinning PPL) will be capable of managing cores with different instruction sets and running a single application across many of those cores. Programming languages and tools will be extended to let the developer express code that is restricted to use just a subset of a mainstream programming language (e.g., the restrict() qualifiers in C++ AMP; I am optimistic that for most mainstream languages such a single language extension will be sufficient while leveraging existing language rules for overloading and dispatch, thus minimizing the impact on developers, but experience will have to bear this out).
- **Deal with the memory axis for computation, by providing distributed algorithms that can scale not just locally but also across a compute**

**cloud.** Libraries and runtimes like OpenCL and TBB and PPL will be extended or duplicated to enable writing loops and other algorithms that run on large numbers of local and non-local parallel cores. Today we can write a parallel_for_each call that can run with 1,000x parallelism on a set of local discrete GPUs and ship the right data shards to the right compute cards and the results back; tomorrow we need to be able to write that same call that can run with 1,000,000,000x parallelism on a set of cloud-based GPUs and ship the right data shards to the right nodes and the results back. This is a "baby step" example in that it just uses local data (e.g., that can fit in a single machine's memory), but distributed computation; the data subsets are simply copied hub-and-spoke.

- **Deal with the memory axis for data, by providing distributed data containers, which can be spread across many nodes.** The next step is for the data itself to be larger than any node's memory, and (preferably automatically) move the right data subsets to the right nodes of a distributed computation. For example, we need containers like a distributed_array or distributed_table that can be backed by multiple and/or redundant cloud storage, and then make those the target of the same distributed parallel_for_each call. After all, why shouldn't we write a single parallel_for_each call that efficiently updates a 100 petabyte table? Hadoop enables this today for specific workloads and with extra work; this will become a standard capability available out-of-the-box in mainstream language compilers and their standard libraries.
- **Enable a unified programming model that can handle the entire chart with the same source code.** Since we can map the hardware on a single chart with two degrees of freedom, the landscape is unified enough that it should be able to be served by a single programming model in the future. Any solution will have at least two basic characteristics: First, it will cover the Processor axis by letting the programmer express language subsets in a way integrated holistically into the language. Second, it will cover or hide the Memory axis by abstracting the location of data, and copying data subsets on demand by default, while also providing a

way to take control of the copying for advanced users who want to optimize the performance of a specific computation.

Perhaps our most difficult mental adjustment, however, will be to learn to think of the cloud as part of the mainstream machine – to view all these local and non-local cores as being equally part of the target machine that executes our application, where the network is just another bus that connects us to more cores. That is, in a few years we will write code for mainstream machines assuming that they have million-way parallelism, of which only thousand-way parallelism is guaranteed to always be available (when out of WiFi range).

*perhaps our most difficult mental adjustment will be to learn to think of the cloud as part of the mainstream machine – to view all these local and non-local cores as being equally part of the target machine that executes our application, where the network is just another bus that connects us to more cores*

Five years from now we want to be delivering apps that run well on an isolated device, and then just run faster or better when they are in WiFi range and have dynamic access to many more cores. The makers of our operating systems, runtimes, libraries, programming languages, and tools need to get us to a place where we can create compute-bound applications that run well in isolation on disconnected devices with 1,000-way local parallelism… and when the device is in WiFi range just run faster, handle much larger data sets, and/or light up with additional capabilities. We have a very small taste of that now with cloud-based apps like Shazam (which function only when online), but yet a long way to go to realize this full vision.

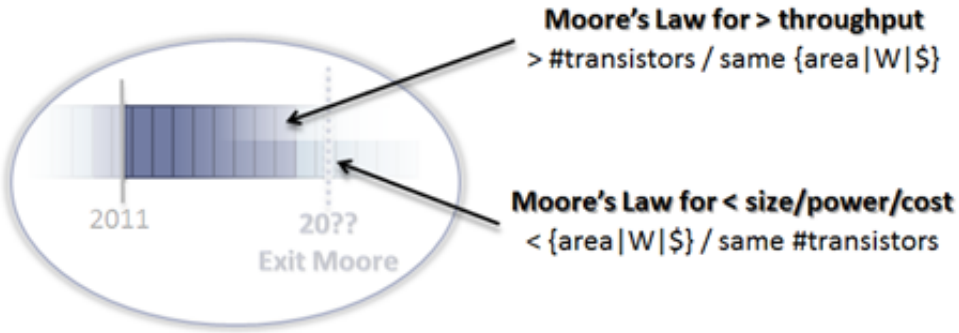
## Exit Moore, Pursued by a Dark Silicon Bear

Finally, let's return one more time to the end of Moore's Law to see what awaits us in our near future, and why we will likely pass through three distinct stages as we navigate Moore's End.

Eventually, our tired miners will reach the point where it's no longer economically feasible to operate the mine. There's still gold left, but it's no

longer commercially exploitable. Recall that Moore's Law has been interesting only because we have been able to transform its raw resource of "more transistors" into one of two useful forms:

- **Exploit #1: Greater throughput.** Moore's Law lets us deliver more transistors, and therefore more complex chips, at the same cost. That's what will let us continue to deliver more computational performance per chip – as long as we can find ways to harness the extra transistors for computation.
- **Exploit #2: Lower cost/power/size.** Alternatively, Moore's Law lets us deliver the same number of transistors at a lower cost, including in a smaller area and at lower power. That's what will let us continue to deliver powerful experiences in increasingly compact and mobile and embedded form factors.




The key thing to note is that we can expect these two ways of exploiting Moore's Law to end, not at the same time, but one after the other and in that order.

Why? Because Exploit #2 only relies on the basic Moore's Law effect, whereas the first relies on Moore's Law *and* the ability to use all the transistors at the same time.

Which brings us to one last problem down in our mine…

### *The Power Problem: Dark Silicon*

Sometimes you can be hard at work in a mine, still productive, when a small disaster happens: a cave-in, or striking water. Besides hurting miners, such disasters can *render entire sections of the mine unreachable*. We are now starting to hit exactly those kinds of problems.

One particular problem we have just begun to encounter is known as "dark silicon." Although Moore's Law is still delivering more transistors, *we are losing the ability to power them all at the same time*. For more details, see Jem Davies' talk "Compute Power With Energy-Efficiency" and the ISCA'11 paper "Dark Silicon and the End of Multicore Scaling" (alternate link).

This "dark silicon" effect is like a Shakespearian bear chasing our doomed character offstage. Even though we can continue to pack more cores on a chip, if we cannot use them at the same time we have failed to exploit Moore's Law to deliver more computational throughput (Exploit #1). When we enter the phase where Moore's Law continues to give us more transistors per die area, but we are no longer able to power them all, we will find ourselves in a transitional period where Exploit #1 has ended while Exploit #2 continues and outlives it for a time.

This means that we will likely see the following major phases in the "scale-in" growth of mainstream machines. (Note that these apply to individual machines only, such as your personal notebook and smartphone or an individual compute node; they do not apply to a compute cloud, which we saw belongs to a different "scale-out" mine.)

- **Exploit #1 + Exploit #2: Increasing performance (compute throughput) in all form factors (1975 – mid-2010s?).** For a few years yet, we will see continuing increases in mainstream computer performance in all form factors from desktop to smartphone. As today, the bigger form factors will still have more parallelism, just as today's desktop CPUs and GPUs are routinely more capable than those in tablets and smartphones – as long as Exploit #1 lives, and then…

- **Exploit #2 only: Flat performance (compute throughput) at the top end, and mid and lower segments catching up (late 2010s – early 2020s?).** Next, if problems like dark silicon are not solved, we will enter a period where mainstream computer performance levels out, starting at the top end with desktops and game consoles and working its way down through tablets and smartphones. During this period we will continue to use Moore's Law to lower cost, power, and/or size – delivering the same complexity and performance already available in bigger form factors also in smaller devices. Assuming Moore's Law continues long enough beyond the end of Exploit #1, we can estimate how long it will take for Exploit #2 to equalize personal devices by observing the difference in transistor counts between current mainstream desktop machines and smartphones; it's roughly a factor of 20, which will take Moore's Law about eight years to cover.
- **Democratization (early 2020s? – onward).** Finally, this democratization will reach the point where a desktop computer and smartphone have roughly the same computational performance. In that case, why buy a desktop ever again? Just dock your tablet or smartphone. You might think that there are still two important differences between the desktop and the mobile device: power, because the desktop is plugged in, and peripherals, because the desktop has easier access to a bigger screen and a real keyboard/mouse – but once you dock the smaller device, it has the same access to power and peripherals and even those differences go away.

### *Speaking of Smartphones Pocket Tablets and Democratization*

Note that the word "smartphone" is already a major misnomer, because a pocket device that can run apps is not primarily a phone at all. It's primarily a general-purpose personal computer that happens to have a couple of built-in radios for cell and WiFi service – making the "traditional cell phone" capability just an app that happens to use the cell radio, and the Skype "IP phone" capability on the same device just another similar app that happens

to use the WiFi radio instead.

The right way to think about even today's mobile landscape is that there are not really "tablets" and "smartphones"; there are just page-sized tablets and pocket-sized tablets, both already available with or without cellular radios, and that they run different operating systems today is just a point-in-time effect.

*there are just page-sized tablets and pocket-sized tablets, both already available with or without cellular radios, and that they run different operating systems today is just a point-in-time effect*

This is why those people who said an iPad is just a big iPhone without the cellular radio had it exactly backwards – the iPhone (3G or later, which allows apps) is a small iPad that fits in your pocket and happens to have a cellular radio in order to obsolete another pocket-sized device. Both devices are primarily tablets – they minimize hardware chrome and "turn into" the full-screen immersive app, and that's the closest thing you can get today to a morphing device that turns into a special-purpose device on demand. (Aside: It'll be great when we figure out how to get past the flat-glass-pane model to let the hardware morph too, initially just raised bumps so we can feel where the keys and controls are, and then eventually more; but hardware morphing is a separate topic and flat glass is plenty fine for now.) Many of us routinely use our "phones" mostly as a small tablet – spending most of our time on the device running apps to read books, browse news, watch movies, play games, update social networks, and surf the net. I already use my phone as a small tablet far more often than I use it as a phone, and if you have an app-capable phone then I'll bet you already do that too.

Well before the end of this decade, I expect the most likely dominant mainstream form factor to be "page-sized and pocket-sized tablets, plus docking" – where "docking" means any means of attaching peripherals like keyboards and big screens on demand, which today already encompasses physical docks and Bluetooth and "Play To" connections, and will only continue to get more wireless and more seamless.

This future shouldn't be too hard to imagine, because many of us have already been working that way for a while now: For the past decade I've routinely worked from my notebook as my primary and only environment; usually I'm in my home office or work office where I use a real keyboard and big screens by docking the notebook and/or using it via a remote-desktop client, and when I'm mobile I use it as a notebook. In 2012, I expect to replace my notebook with an x86-based modern tablet and use it exactly the same way.

We've seen it play out many times:

- Many of us used to carry around both a PalmPilot and a cell phone, but then the smartphone took over the job of the dedicated PalmPilot and eliminated a device with the same form factor.
- Lots of kids (or their parents) carry a hand-held gaming device and a pocket tablet (aka "smartphone"), and we are seeing the decline of the dedicated hand-held gaming device as the pocket tablet is taking over more and more of that job.
- Similarly, today many of us carry around a notebook and a dedicated tablet, and convergence will again let us eliminate a device with the same form factor.

Computing loves convergence. In general-purpose personal computing (like notebooks and tablets, not special-purpose appliances like microwaves and automobiles that may happen to use microprocessors), convergence always happily dooms special-purpose devices in the long run, as each device either evolves to take over the other's job or gets taken over. We will continue to have distinct pocket-sized tablets and page-sized tablets for a time because they are different form factors with different mobile uses, but even that may last only until we find a way to unify the form factors (fold them?) so that they too can converge.

## Summary and Conclusions

Mainstream hardware is becoming permanently parallel, heterogeneous, and distributed. These changes are permanent, and so will permanently affect the way we have to write performance-intensive code on mainstream architectures.

The good news is that Moore's "local scale-in" transistor mine isn't empty yet; it appears the transistor bonanza will continue for about another decade, give or take a half decade or so, which should be long enough to exploit the lower-cost side of the Law to get us to parity between desktops and pocket tablets. The bad news is that we can clearly observe the diminishing returns as the transistors are decreasingly exploitable – with each new generation of processors, software developers have to work harder and the chips get more difficult to power. And with each new crank of the diminishing-returns wheel, there's less time for hardware and software designers to come up with ways to overcome the next hurdle; the motherlode free lunch lasted 30 years, but the homogeneous multicore era lasted only about six years, and we are now already overlapping the next two eras of hetero-core and cloud-core.

*the cloud wave is already scaling enormously quickly – faster than the Moore's Law wave that it complements, and that it will outlive and replace*

But all is well: When your mine is getting empty, you don't panic, you just open a new mine at a new motherlode, operate both mines for a while, then continue to profit from the new mine long-term even after the first one finally shuts down and gets converted into a museum. As usual, in this case the end of one dominant wave overlaps with the beginning of the next, and we are now early in the period of overlap where we are standing with a foot in each wave, a crew in each of Moore's mine and the cloud mine. Perhaps the best news of all is that the cloud wave is already scaling enormously quickly – faster than the Moore's Law wave that it complements, and that it will outlive and replace.

If you haven't done so already, now is the time to take a hard look at the de-

sign of your applications, determine what existing features – or, better still, what potential and currently-unimaginable demanding new features – are CPU-sensitive now or are likely to become so soon, and identify how those places could benefit from local and distributed parallelism. Now is also the time for you and your team to grok the requirements, pitfalls, styles, and idioms of hetero-parallel (e.g., GPGPU) and cloud programming (e.g., Amazon Web Services, Microsoft Azure, Google App Engine).

To continue enjoying the free lunch of shipping an application that runs well on today's hardware and will just naturally run faster or better on tomorrow's hardware, you need to write an app with lots of juicy latent parallelism expressed in a form that can be spread across a machine with a variable number of cores of different kinds – local and distributed cores, and big/small/specialized cores. The filet mignon of throughput gains is still on the menu, but now it costs extra – extra development effort, extra code complexity, and extra testing effort. The good news is that for many classes of applications the extra effort will be worthwhile, because concurrency will let them fully exploit the exponential gains in compute throughput that will continue to grow strong and fast long after Moore's Law has gone into its sunny retirement, as we continue to mine the cloud for the rest of our careers.

## Acknowledgments

I would like to particularly thank Jeffrey Barr, David Callahan, Olivier Giroux, Yossi Levanoni, Henry Moreton, and James Reinders, who graciously made themselves available to answer questions to provide background information, and who shared their feedback on appropriately mapping their companies' products on the processor/memory chart.

## Update History

2012-08-02: Updated to clarify that by "weak (hardware) memory model"

CPUs I mean specifically ones that do not natively support efficient sequentially consistent (SC) atomics, because on the software side programming languages have converged on the strong "sequential consistency for data-race-free programs" (SC-DRF, roughly aka DRF0 or RCsc) as the default (C11, C++11) or only (Java 5+) supported software memory model for software. Hardware that supports weaker memory models than that are permanently disadvantaged and will either become stronger (as ARMv8 is now doing by adding SC acquire/release instructions) or atrophy. The two main hardware architectures with what I called "weak" memory models were ARMv7 and POWER. ARMv8 is upgrading to SC acquire/release, as predicted, and it remains to be seen whether POWER will upgrade or atrophy. I've seen some call x86 "weak", but x86 has always been the poster child for a *strong* hardware memory model in all of our software memory model discussions for Java, C, and C++ during the 2000s. Therefore it's clear that "weak" and "strong" are not useful terms because they mean different things for software and hardware memory models, and I've updated the text to clarify this.