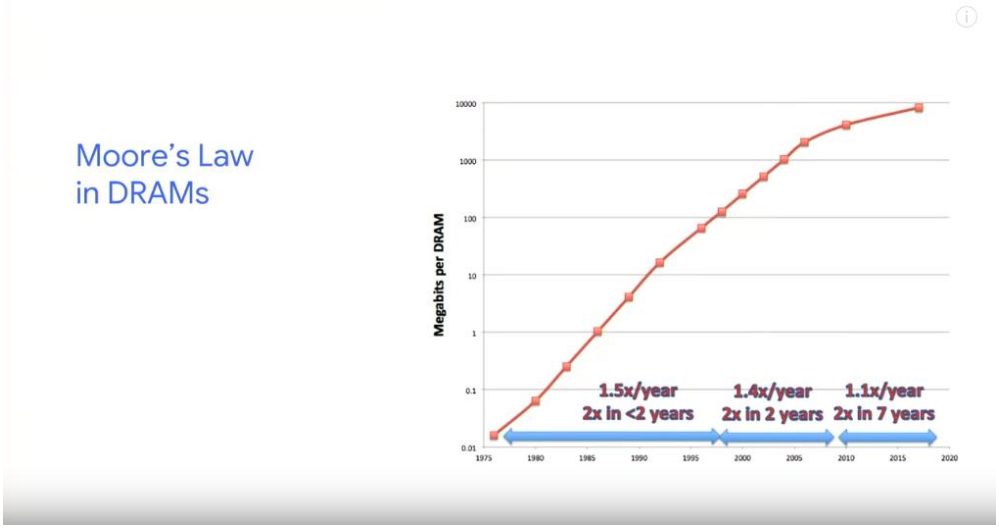
# The future of computing: a conversation with John Hennessy

(Google I/O '18)

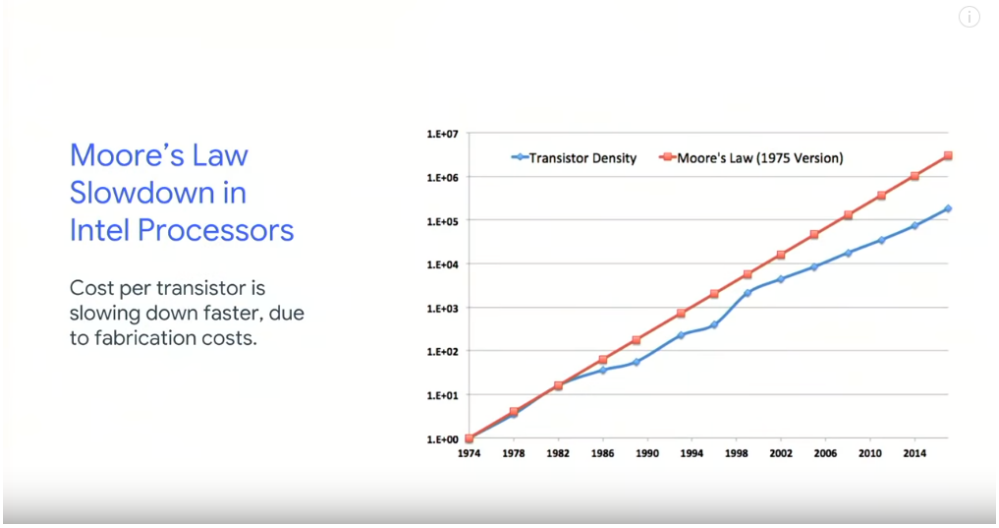
Boy, I'm delighted to be here today and have a chance to talk to you about what is one of the biggest challenges we faced in computing in 40 years, but also a tremendous opportunity to rethink how we build computers and how we move forward. You know, there's been a lot of discussion about the ending of Moore's law. The first thing to remember about the ending of Moore's law is something Gordon Moore said to me. He said, all exponentials come to an end. It's just a question of when. And that's what's happening with Moore's law.

오늘 이곳에 와서 40 년 만에 컴퓨팅에서 직면 한 가장 큰 도전 중 하나가 무엇인지에 대해 이야기 할 기회를 갖게 되어 기쁘게 생각합니다. 또한 컴퓨터를 구축하는 방법과 앞으로 나아갈 방법을 다시 생각해 볼 수 있는 엄청난 기회입니다. . 무어의 법칙 결말에 대해 많은 논의가 있었습니다. 무어의 법칙 결말에 대해 가장 먼저 기억해야 할 것은 고든 무어가 나에게 한 말이다. 그는 모든 지수가 끝나게 된다고 말했다. 그것은 언제의 문제일 뿐입니다. 그리고 그것은 무어의 법칙에서 일어나는 일입니다.



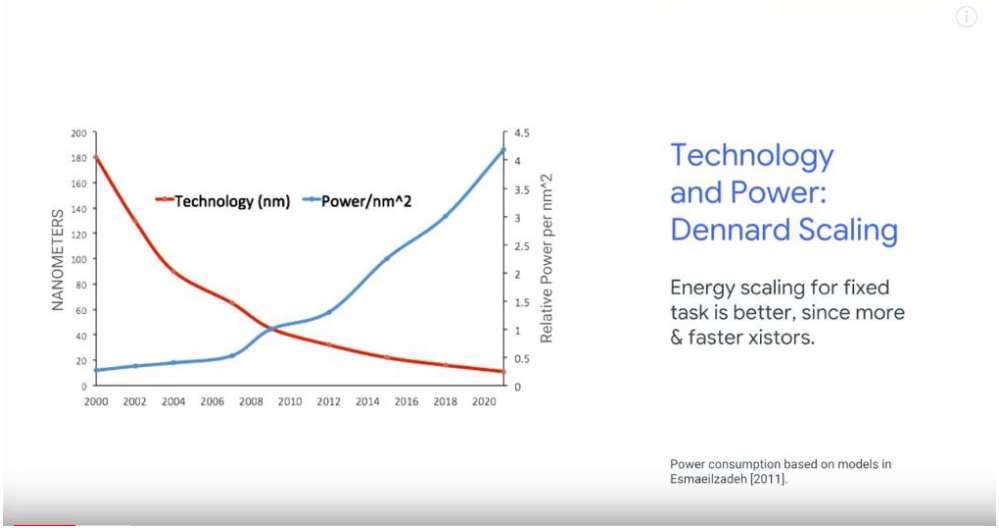

If we look at-- what does it really mean to say Moore's law is ending? What does it really mean? Well, look at what's happening in DRAMs. That's probably a good place to start because we all depend on the incredible growth in memory capacity. And if you look at what's happened in DRAMs, for many years we were achieving increases of about 50% a year. In other words, going up slightly faster even than Moore's law. Then we began a period of slowdown. And if you look what's happened in the last seven years, this technology we were used to seeing boom, the number of megabits per chip more than doubling every two years, is now going up at about 10% a year and it's going to take about seven years to double. Now, DRAMs are a particularly odd technology because they use deep trench capacitors, so they require a very particular kind of fabrication technology.

우리가 살펴 본다면, 무어의 법칙이 종말이라고 말하는 것은 무엇을 의미합니까? 그것은 정말로 무엇을 의미합니까? 글쎄, DRAM 에서 무슨 일이 일어나는지 보십시오. 그것은 우리가 모두 메모리 용량의 엄청난 성장에 의존하기 때문에 아마 시작할 수 있는 좋은 곳입니다. 그리고 DRAM 에서 일어난 일을 살펴 본다면, 수년 동안 우리는 일년에 약 50 %의 증가를 달성했습니다. 즉, 무어의 법칙보다 약간 더 빨리 올라간다는 것입니다. 그런 다음 우리는 침체기를 시작했습니다. 지난 7 년 동안 일어난 일을 살펴보면 2 년에 두 배 이상 증가한 칩당 메가 비트 수는 이제 매년 약 10 %씩 올라가고 있습니다. 두 배로 7 년. 이제 DRAM 은 딥 트렌치 커패시터를 사용하기 때문에 특히 이상한 기술이므로 매우 특정한 종류의 제조 기술이 필요합니다.



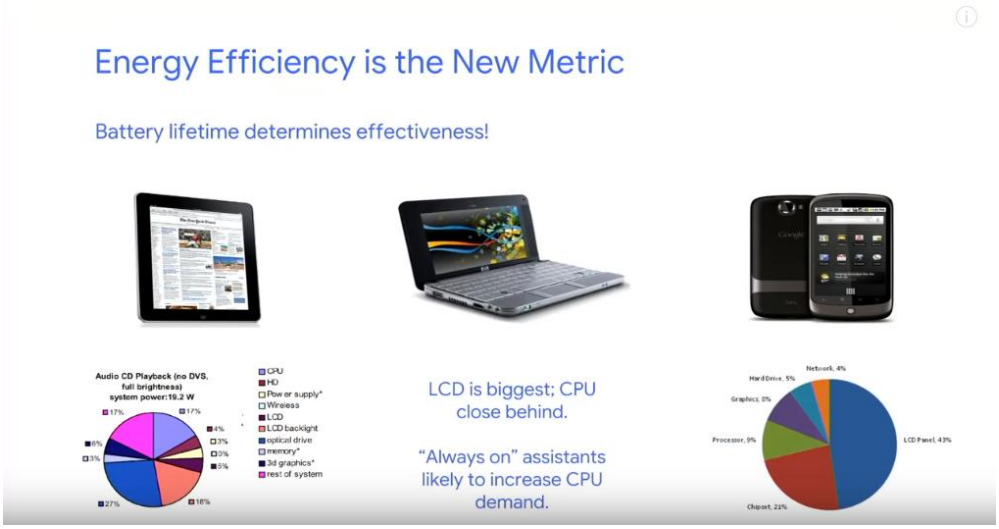

What's happening in processors, though? And if you look at the data in processors, you'll see a similar slowdown. Moore's law is that red line going up there on a nice logarithmic plot. Notice the blue line. That's the number of transistors on a typical Intel microprocessor at that date. It begins diverging, slowly at first. But look what's happened since in the last 10 years, roughly. The gap has grown. In fact, if you look at where we are in 2015, 2016, we're more than a factor of 10 off, had we stayed on that Moore's law curve. Now, the thing to remember is that there's also a cost factor in here. Fabs are getting a lot more expensive and the cost of chips is actually not going down as fast. So a result of that is that the cost per transistor is actually increasing at a worse rate. So we're beginning to see the effects of that as we think about architecture. But if the slowdown of Moore's law, which is what you see all the press about as one thing, the big issue is the end of what we call Dennard scaling.

그렇지만 프로세서에서 어떤 일이 벌어지고 있습니까? 프로세서의 데이터를 보면 비슷한 속도 저하를 볼 수 있습니다. 무어의 법칙은 빨간 선이 멋진 대수 구도에 올라가는 것입니다. 파란색 선을 확인하십시오. 이것은 그 날의 전형적인 인텔 마이크로 프로세서의 트랜지스터 수입니다. 처음에는 천천히 분기를 시작합니다. 그러나 지난 10 년 동안 무슨 일이 있었는지 살펴보십시오. 그 격차가 커졌습니다. 실제로, 2015 년, 2016 년에 우리가 어디에 있는지 보면, 우리는 그 무어의 법칙 곡선에 머물렀다면 10 분의 1 이상을 의미합니다. 이제 기억해야 할 것은 여기에 비용 요소가 있다는 것입니다. Fabs 는 훨씬 더 비싸고 칩 비용은 실제로 빠르지는 않습니다. 따라서 결과적으로 트랜지스터 당 비용은 실제로 더 나쁜 속도로 증가합니다. 그래서 우리는 건축에 대해 생각할 때 그 영향을 보기 시작했습니다. 그러나 모든 언론을 한 가지로 보는 무어의 법칙이 느려지면 큰 문제는 우리가 데나르드 스케일링 (Dennard scaling)이라고 부르는 것의 끝입니다.



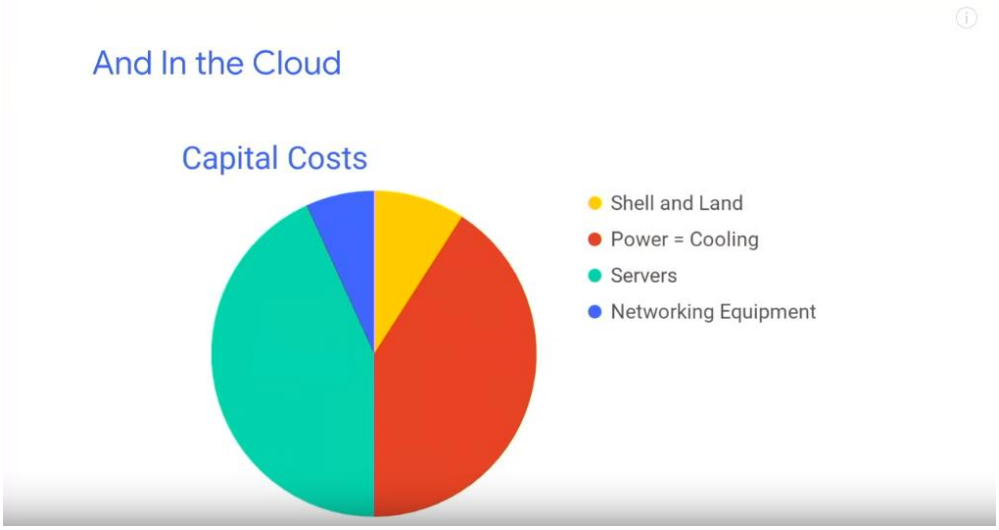

So Bob Dennard was an IBM employee, he was the guy who invented the one transistor DRAM. And he made a prediction many years ago that the energy, the power per square millimeter of silicon would stay constant, would stay constant because voltage levels would come down, capacitance would come down. What does that mean? If the energy, if the power stays constant and the number of transistors increases exponentially, then the energy per transistor is actually going down. And in terms of energy consumption, it's cheaper and cheaper and cheaper to compute. Well, what happened with Dennard scaling? Well, look at that blue line there. The red line shows you the technology improving on a standard Moore's law curve. The blue line shows you what's happening to power. And you all know. I mean, you've seen microprocessors now, right? They slow their clock down, they turn off cores, they do all kinds of things, because otherwise they're going to burn up. They're going to burn up. I mean, I never thought we'd see the day where a processor would actually slow itself down to prevent itself overheating, but we're there. And so what happens with Dennard scaling is it began to slow down starting about '97. And then since 2007, it's essentially halted. The result is a big change. All of a sudden, energy, power becomes the key limiter. Not the number of transistors available to designers, but their power consumption becomes the key limiter. That requires you to think completely differently about architecture, about how you design machines. It means inefficiency in the use of transistors in computing. Inefficiency in how an architecture computes is penalized much more heavily than it was in this earlier time. And of course, guess what?

그래서 Bob Dennard 는 IBM 직원이었습니다. 그는 한 트랜지스터 DRAM 을 발명한 사람이었습니다. 그리고 그는 수년 전에 에너지가, 평방 밀리미터의 실리콘당 전력은 일정하게 유지될 것이고, 전압 레벨이 낮아지기 때문에 정전 용량이 낮아질 것이라는 예측을 했습니다. 그게 무슨 뜻이죠? 전력이 일정하게 유지되고 트랜지스터의 수가 기하급수적으로 증가하면 에너지는 실제로 트랜지스터당 에너지를 떨어뜨립니다. 그리고 에너지 소비측면에서 볼 때, 저렴하고 저렴하며 저렴하게 계산할 수 있습니다. Dennard 스케일링은 어떻게 된거야? 음, 거기에 파란 선을 보세요. 빨간색 선은 표준 무어의 법칙 곡선에서 기술 향상을 보여줍니다. 파란 선은 어떤 일이 일어나고 있는지를 보여줍니다. 그리고 여러분 모두 알고 있습니다. 내 말은, 지금 마이크로 프로세서를 보았다는 거지? 그들은 시계를 늦추고, 핵을 끄고, 모든 종류의 일을 합니다. 그렇지 않으면 그들은 불타 버릴 것이기 때문입니다. 그들은 태워 버릴거야. 내 말은, 프로세서가 과열되는 것을 막기 위해 프로세서가 실제로 느려지는 날을 볼 수는 없다고 생각했지만, 우리는 거기에 있습니다. 따라서 Dennard 스케일링은 97 년부터 시작되기 시작합니다. 그리고 2007 년부터 본질적으로 중단되었습니다. 결과는 큰 변화입니다. 갑자기, 에너지, 권력이 핵심 리미터가 됩니다. 설계자가 사용할 수 있는 트랜지스터 수는 아니지만 전력 소비가 핵심 리미터가 됩니다. 이를 위해서는 아키텍처에 대한 완전히 다른 생각, 기계 설계 방법에 대한 생각이 필요합니다. 이는 컴퓨팅에서 트랜지스터 사용의 비효율을 의미합니다. 아키텍처가 어떻게 계산되는지에 대한 비효율은 이전보다 훨씬 심각하게 불이익을 받는다. 그리고 물론, 어떨까요?



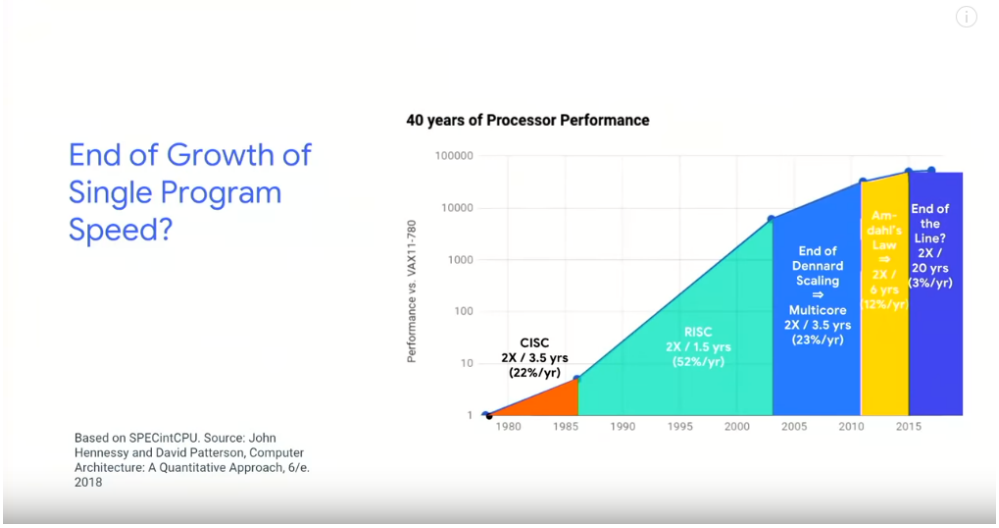

All the devices we carry around, all the devices we use are running off batteries. So all of a sudden, energy is a critical resource, right? What's the worst thing that happens is your cell phone runs out of power, your smartphone runs out of power. That's a disaster, right? But think about all the devices we walk around with. They're hooked up to battery. Think about the era, the coming era of IoT, where we're going to have devices that are always on and permanently on, which are expected to last 10 years on a single battery by using energy harvesting techniques. Energy becomes the key resource in making those things work efficiently. And as we move more and more to always on devices with things like Google Assistant, you're going to want your device on all the time or at least you're going to want the CPU on all the time, if not the screen. So we're going to have to worry more and more about power. But the surprising thing that many people are surprised by is that energy efficiency is a giant issue in large cloud configurations.

우리가 다니는 모든 장치, 우리가 사용하는 모든 장치는 배터리를 사용하지 않고 있습니다. 그래서 갑자기 모든 에너지가 중요한 자원입니다. 맞습니까? 최악의 상황은 휴대 전화의 전원이 끊어 지거나 스마트 폰의 전원이 끊어지는 경우입니다. 그게 재앙이야, 그렇지? 그러나 우리가 걸어 다니는 모든 장치에 대해 생각해 보십시오. 그들은 배터리에 연결되어 있습니다. 에너지 수확 기술을 사용하여 단일 배터리로 10 년 동안 지속될 것으로 예상되는 장치가 항상 있고 영구적으로 설치되는 IoT 의 시대와 시대를 생각해보십시오. 에너지가 효율적으로 작동하도록 하는 핵심 자원이 됩니다. Google Assistant 와 같은 장치를 사용하여 항상 장치를 계속 사용할 때마다 장치를 항상 켜 놓으려고 하거나 적어도 화면이 아닌 모든 시간에 CPU 를 원하게 됩니다. 그래서 우리는 점점 더 힘에 대해 걱정해야 할 것입니다. 그러나 많은 사람들이 놀라게 하는 놀라운 점은 대형 클라우드 구성에서는 에너지 효율성이 큰 문제라는 것입니다.




This shows you what the typical capital cost would be like for a Google data center. You'll notice that green slice there, those are the servers. But look at the size of that red slice. That red slice is the cost of the power plus cooling infrastructure. Spending as much on power and cooling as you're spending on processors. So energy efficiency becomes a really critical issue as we go forward. And the end of Dennard scaling has meant that there's no more free lunch. For a lot of years, we had a free lunch. It was pretty easy to figure out how to make computation more energy efficient. Now, it's a lot harder. And you can see the impact of this.

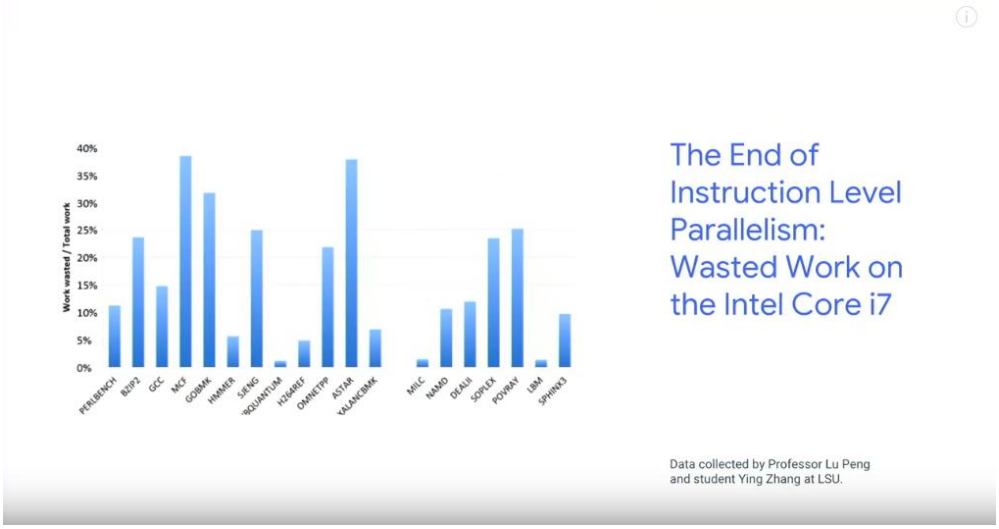
이것은 Google 데이터 센터에 대한 일반적인 자본 비용을 보여줍니다. 거기에 녹색 조각이 있음을 알게 될 것입니다. 그것들은 서버입니다. 그러나 그 붉은 조각의 크기를 보십시오. 이 빨간색 조각은 전력 및 냉각 인프라 비용입니다. 프로세서에 소비하는 전력 및 냉각량에 많은 돈을 지출합니다. 따라서 에너지 효율은 우리가 진보 할 때 정말로 중요한 문제가 됩니다. 그리고 Dennard 스케일링의 끝은 자유로운 점심이 더 이상 없다는 것을 의미합니다. 몇 년 동안 우리는 무료 점심 식사를 했습니다. 계산을 보다 에너지 효율적으로 만드는 방법을 알아내는 것은 꽤 쉬웠습니다. 이제 훨씬 더 어려워졌습니다. 그리고 당신은 이것의 영향을 볼 수 있습니다.




This just shows you 40 years of processor performance, what's happened to uniprocessor, single processor performance, and then multiprocessor performance. So there were the early years of computing, the beginning of the microprocessor era. We were seeing about 22% improvement per year. The creation of risk in the mid-1980s, a dramatic use of instruction level parallelism, pipelining, multiple issue. We saw this incredible period of about 20 years, where we got roughly 50% performance improvement per year. 50%. That was amazing. Then the beginning of the end of Dennard scaling. That caused everybody to move to multi-core. What did multi-core do? Multi-core shoved the efficiency problem from the hardware designer to the software people. Now, the software people had to figure out how to use those multi-core processors efficiently. But Amdahl's law came along, reared its ugly head. I'll show you some data on that. And now, we're in this late stage period where it looks like we're getting about 3% performance improvement per year. Doubling could take 20 years. That's the end of general purpose processor performance as we know it, as we're used to for so many years.

40 년 동안의 프로세서 성능, 단일 프로세서, 단일 프로세서 성능, 다중 프로세서 성능에 어떤 변화가 있었는지를 보여줍니다. 그래서 초기 마이크로 컴퓨팅 시대, 마이크로 프로세서 시대의 시작이 있었습니다. 우리는 연간 약 22%의 개선을 보았습니다. 1980 년대 중반에 위험 요소가 창출되었는데, 이는 명령어 수준의 병렬 처리, 파이프 라이닝, 다중 문제의 극적인 사용이었습니다. 우리는 이 놀라운 기간을 약 20 년 동안 보았습니다. 여기서 우리는 연간 약 50%의 성능 향상을 보았습니다. 50%. 놀랍습니다. 다음 Dennard 스케일링의 끝 부분. 이로 인해 모든 사람들이 멀티 코어로 이동했습니다. 멀티 코어는 무엇을 했습니까? 멀티 코어는 하드웨어 디자이너에서 소프트웨어 사용자에 이르는 효율성 문제를 해결했습니다. 이제 소프트웨어 사용자는 멀티 코어 프로세서를

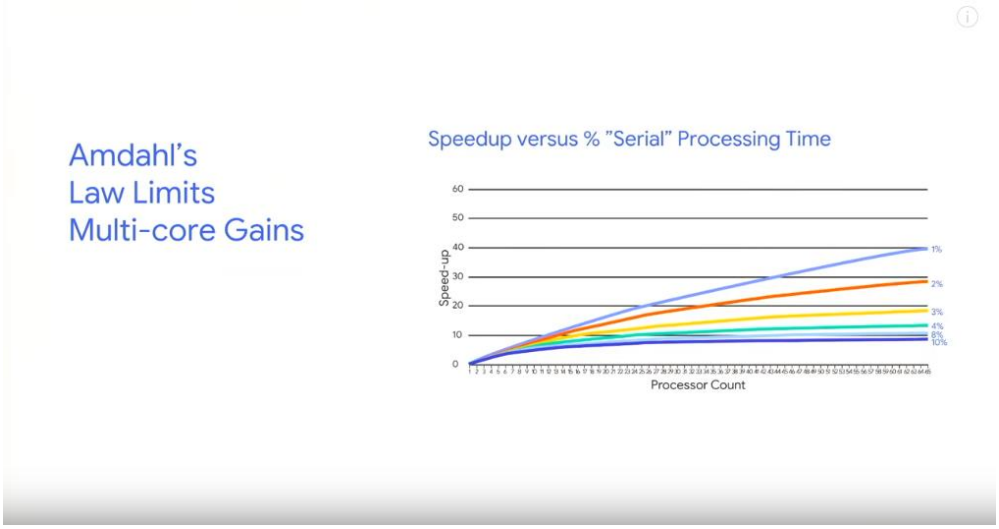
효율적으로 사용하는 방법을 찾아야 했습니다. 그러나 Amdahl 의 법칙이 따라와서 추악한 머리를 길렀습니다. 그것에 대한 데이터를 보여 드리겠습니다. 이제 우리는 1 년에 약 3%의 성능 향상을 얻는 것처럼 보이는 이 후기 단계에 있습니다. 이중화는 20 년이 걸릴 수 있습니다. 이것이 우리가 알고 있는 범용 프로세서 성능의 끝입니다. 오랜 세월 동안 그랬던 것처럼.




Why did this happen? Why did it grind to a halt so fast? Well, think about what was happening during that risk era where we're building these deeply pipelined machines. 15, 16, 17 stages deep pipelines, four issues per clock. That machine needs to have 60 instructions that it's working on at once. 60 instructions. How does it possibly get 60 instructions? It uses speculation. It guesses about branches, it yanks instructions and tries to execute them. But guess what happens? Nobody can predict branches perfectly. Every time you predict a branch incorrectly, you have to undo all the work associated with that missed prediction. You've got to back it out, you've got to restore the state of the machine. And if you look inside a typical Intel Core i7 today, on integer code roughly 25% of the instructions that get executed end up being thrown away. Guess what? The energy still got burnt to execute all those instructions. And then, I threw the results away and I had to restore the state of the machine. A lot of wasted energy. That's why the single processor performance curve ended, basically.

왜 이런 일이 일어 났습니까? 왜 그렇게 빨리 멈추는가? 글쎄, 우리가 파이프 라인이 깊은 기계를 만드는 위험한 시대에 무슨 일이 일어 났는지 생각해 보십시오. 15, 16, 17 단계 딥 파이프 라인, 클록 당 4 개의 이슈. 그 기계는 그것이 일하는 60 가지 지시 사항을 가지고 있어야합니다. 60 지침. 어떻게 60 가지 지시사항을 얻을 수 있습니까? 그것은 추측을 사용합니다. 그것은 지점에 대해 추측하고 지시사항을 잡아 당겨 실행합니다. 그러나 어떻게 될지 생각해 보십시오. 아무도 지점을 완벽하게 예측할 수는 없습니다. 분기를 잘못 예측할 때마다 누락된 예측과 관련된 모든 작업을 실행취소 해야 합니다. 당신은 그것을 되찾아야 합니다, 당신은 기계의 상태를 복원해야 합니다. 그리고 전형적인 인텔 코어 i7 을 살펴보면 정수 코드에서 실행되는 명령어의 대략 25%가 버려지게 됩니다. 맞춰봐? 모든 지시 사항을 실행하기 위해 여전히 에너지가 소모되었습니다. 그리고 나서 결과를 버리고 컴퓨터의 상태를

복원해야 했습니다. 많은 에너지 낭비. 이것이 단일 프로세서 성능 곡선이 기본적으로 끝난 이유입니다.

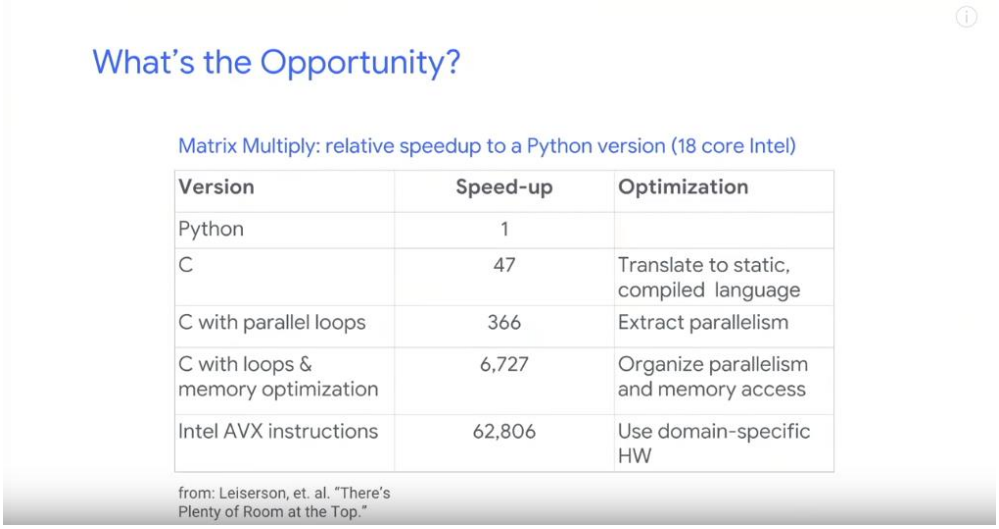



But we see similar challenges when you begin to look at multi-core things. Amdahl's law, Gene Amdahl wrote Amdahl's law more than 40 years ago. It's still true today. Even if you take large data centers with heavily parallel workloads, it's very hard to write a big complicated piece of software and not have small sections of it be sequential, whether it's synchronization or coordination or something else. So think about what happens. You've got a 64 processor multi-core in the future. Suppose 1%, just 1% of the code is sequential. Then that 64 processor multi-core only runs at the speed of a 40 processor core. But guess what? You paid all the energy for a 64 processor core executing all the time and you only got 40 processors out of that, slightly more than half. That's the problem. We've got to breakthrough this efficiency barrier. We've got to rethink how we design machines.

그러나 우리가 멀티 코어의 것들을 보기 시작할 때 비슷한 문제가 발생합니다. Amdahl 의 법칙 Gene Amdahl 은 Amdahl 의 법칙을 40 년 전에 작성했습니다. 오늘도 여전히 사실입니다. 심하게 병렬 작업 부하가 있는 대형 데이터 센터를 가져가더라도 복잡한 소프트웨어를 작성하고 동기화나 조정 또는 기타 다른 작업을 수행하지 않고 작은 부분을 순차적으로 처리하는 것은 매우 어렵습니다. 그래서 무슨 일이 일어날 지 생각해보십시오. 당신은 미래에 64 프로세서 멀티 코어를 가지고 있습니다. 1%라고 가정하면 코드의 1%만 순차적입니다. 그런 다음 64 프로세서 멀티 코어는 40 프로세서 코어의 속도로만 실행됩니다. 하지만 그거 알아? 항상 실행중인 64 프로세서 코어에 대한 모든 에너지를 지불 했으므로 절반 이상을 차지하는 프로세서가 40 개 밖에 없습니다. 그것이 문제이다. 우리는 이러한 효율성 장벽을 획기적으로 극복해야 합니다. 우리는 기계를 설계하는 방법에 대해 다시 생각해야 합니다.


So what's left? Well, software-centric approaches. Can we make our systems more efficient? It's great that we have these modern scripting languages, they're interpreted, dynamically-typed, they encourage reuse. They've really liberated programmers to get a lot more code written and create incredible functionality. They're efficient for programmers. They're very inefficient for execution, and I'll show you that in a second. And then there are hardware-centric approaches, what Dave Patterson and I call domain-specific architectures. Namely, designing an architecture which isn't fully general purpose, but which does a set of domains, a set of applications really well, much more efficiently.

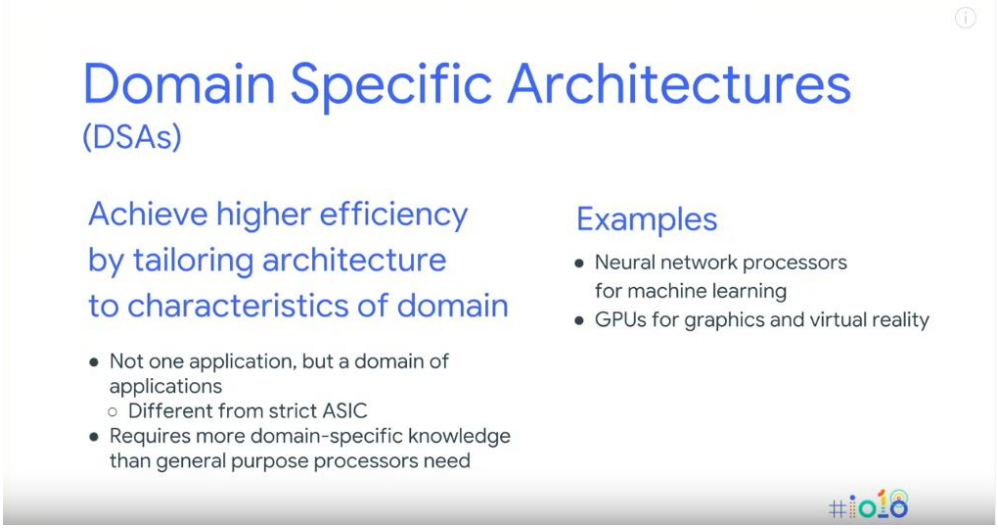
그럼 남은 건 뭐야? 음, 소프트웨어 중심 접근법. 시스템을 보다 효율적으로 만들 수 있습니까? 우리가 이러한 최신 스크립팅 언어를 가지고 있고, 해석되고, 동적으로 타입화되고, 재사용을 권장하는 것은 대단한 일입니다. 그들은 프로그래머에게 많은 코드를 작성하고 놀라운 기능을 만들 수 있게 해줬습니다. 그것들은 프로그래머에게 효율적입니다. 실행에 매우 비효율적인데, 잠시 후에 보여 드리겠습니다. 그리고 Dave Patterson 과 내가 도메인 특정 아키텍처라고 부르는 하드웨어 중심의 접근 방식이 있습니다. 즉, 완전히 일반적인 목적은 아니지만 일련의 도메인을 이루는 아키텍처를 설계하는 것입니다. 애플리케이션 세트는 훨씬 효율적입니다.




So let's take a look at what the opportunity is. This is a chart that comes out of a paper by Charles Leiserson and a group of colleagues at MIT, called "There's Plenty of Room at the Top." They take a very simple example, admittedly, matrix multiply. They write it in Python. They run it on an 18 core Intel processor. And then they proceed to optimize it. First, rewrite it in C. That speeds it up 47 times. Now, any compiler in the world that can get a speed up of 47 would be really remarkable, even a speed up of 20. Then they rewrite it with parallel loops. They get almost a factor of nine out of that. Then they rewrite it by doing memory optimization. That gives them a factor of 20. They block the matrix, they allocate it to the caches properly. That gives them a factor of 20. And then finally, they rewrite it using Intel AVX instructions, using the vector instructions in the Intel Core, right, domain-specific instructions that do vector operations efficiently. That gives them another factor of 10. The end result is that final version runs 62,000 times faster than the initial version. Now admittedly, matrix multiply is an easy case, small piece of code. But it shows the potential of rethinking how we write this software and making it better.

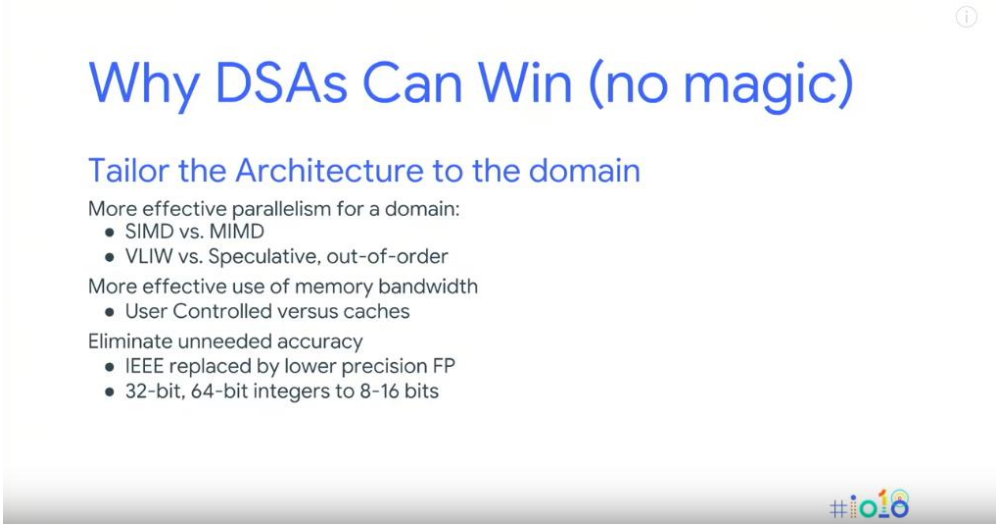
그래서 기회가 무엇인지 살펴 보겠습니다. 이것은 Charles Leiserson 과 MIT 의 한 그룹의 연구원이 발표한 "The Top of Room"에 나오는 도표입니다. 그들은 아주 간단한 예를 들고 있습니다. 그들은 파이썬으로 그것을 씁니다. 그들은 18 코어 Intel 프로세서에서 실행합니다. 그런 다음 최적화를 진행합니다. 먼저 C 로 다시 작성하십시오. 47 번 속도가 빨라집니다. 자, 47 의 속도를 얻을 수 있는 세계에서 모든 컴파일러는 정말 놀라운 것입니다 심지어 최대 20 의 속도. 그럼 그들은 병렬 루프로 다시 작성합니다. 그것들은 거의 9 점을 얻습니다. 그런 다음 메모리 최적화를 수행하여 다시 작성합니다. 그것들은 20 의 인수를 줍니다. 그들은 행렬을 차단하고, 그것을 캐시에 적절히 할당합니다. 그러면 최종적으로 벡터 작업을 효율적으로 수행하는 Intel 코어의 벡터 명령어를

사용하여 인텔 AVX 명령어를 사용하여 다시 작성합니다. 결과적으로 최종 버전은 초기 버전보다 62,000 배 빠릅니다. 이제는 행렬 곱셈이 쉬운 경우, 작은 코드 조각입니다. 그러나 이 소프트웨어를 작성하고 개선하는 방법을 다시 생각할 가능성이 있음을 보여줍니다.




So what about these domain-specific architectures? Really what we're going to try to do is make a breakthrough in how efficient we build the hardware. And by domain-specific, we're referring to a class of processors which do a range of applications. They're not like, for example, the modem inside the cell phone, right? That's programmed once, it runs modem code. It never does anything else. But think of a set of processors which do a range of applications that are related to a particular application domain. They're programmable, they're useful in that domain, they take advantage of specific knowledge about that domain when they run, so they can run much more efficiently. Obvious examples, doing things for neural network processors, doing things that focus on machine learning. One example. GPUs are another example of this kind of thinking, right? They're programmable in the context of doing graphics processing.

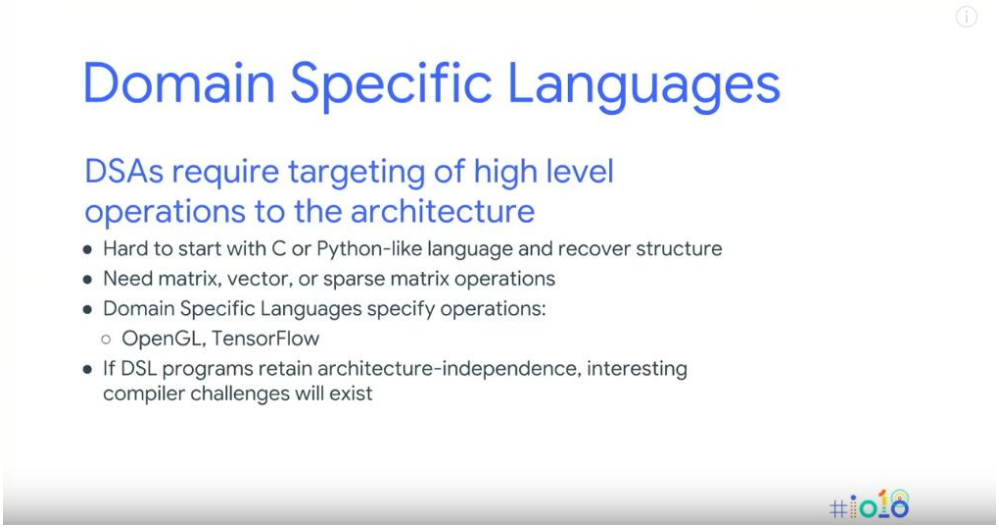
그렇다면 이러한 도메인 별 아키텍처는 어떻습니까? 실제로 우리가 하려고 하는 것은 하드웨어를 얼마나 효율적으로 구축하는지에 대한 돌파구를 만드는 것입니다. 그리고 도메인에 따라, 우리는 다양한 애플리케이션을 수행하는 프로세서 클래스를 언급하고 있습니다. 그들은, 예를 들어, 휴대 전화 내부의 모뎀과 같지 않습니까? 일단 프로그래밍하면 모뎀 코드가 실행됩니다. 그것은 결코 다른 것을 하지 않습니다. 그러나 특정 응용 프로그램 도메인과 관련된 일련의 응용 프로그램을 처리하는 프로세서 집합을 생각해 보십시오. 프로그래밍이 가능하고 해당 도메인에서 유용하며 실행 시 해당 도메인에 대한 특정 지식을 활용하므로 훨씬 효율적으로 실행할 수 있습니다. 명백한 예, 신경망 프로세서에 대한 작업, 기계 학습에 초점을 맞춘 작업. 한 가지 예. GPU 는 이런 종류의 생각의 또 다른 예입니다. 맞습니까? 그래픽 처리와 관련하여 프로그래밍이 가능합니다.




So for any of you who have ever seen that any of the books that Dave Patterson and I wrote, you know that we like quantitative approaches to understand things and we like to analyze why things work. So the key about domain-specific architectures is there is no black magic here. Going to a more limited range of architectures doesn't automatically make things faster. We have to make specific architectural changes that win. And there are three big ones. The first is we make more effective use of parallelism. We go from a multiple instruction, multiple data world that you'd see on a multi-core today to a single instruction multiple data. So instead of having each one of my cores fetch separate instruction streams, have to have separate caches, I've got one set of instructions and they're going to a whole set of functional units. It's much more efficient. What do I give up? I give up some flexibility when I do that. I absolutely give up flexibility. But the efficiency gain is dramatic. I go from speculative out-of-order machines, what a typical high-end processor from ARM or Intel looks like today, to something that's more like a VLIW, that uses a set of operations where the compiler has decided that a set of operations can occur in parallel. So I shift work from runtime to compile time. Again, it's less flexible. But for applications when it works, it's much more efficient. I move away from caches. So caches are one of the great inventions of computer science, one of the truly great inventions. The problem is when there is low spatial and low temporal locality, caches not only don't work, they actually slow programs down. They slow them down. So we move away from that to user control local memories. What's the trade-off? Now, somebody has to figure out how to map their application into a user controlled memory structure. Cache does it automatically for you, it's very general purpose. But for certain applications, I can do a lot better by mapping those things myself. And then finally, I focus on only the amount of accuracy I need. I've move from IEEE to the lower precision floating point or from 32 and 64-bit integers to 8-bit and 16-bit integers. If that's all the accuracy I need, I can do eight integer operations, eight

8-bit operations in the same amount of time that I can do one 64-bit operation.

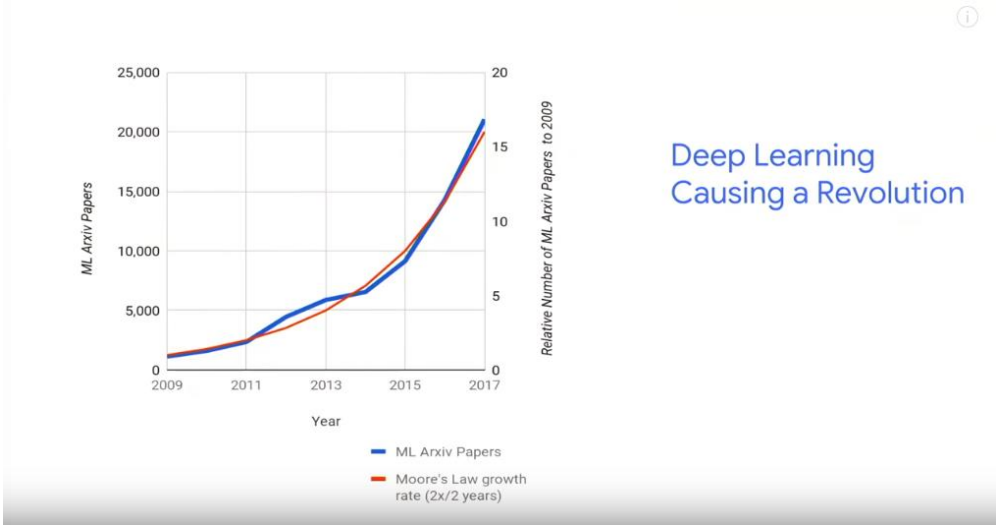
따라서 Dave Patterson 과 제가 쓴 책을 본 적이 있는 여러분 중 어떤 것을 이해하기 위해 양적 접근법을 선호한다는 것과 우리가 왜 작동하는지 분석하고 싶다는 것을 알고 있습니다. 따라서 도메인 별 아키텍처의 핵심은 여기에 흑마술이 없다는 것입니다. 좀 더 제한된 범위의 아키텍처로 가더라도 자동으로 작업이 빨라지지는 않습니다. 우리는 특정한 건축적 변화를 이루어야 합니다. 그리고 세 가지 큰 것들이 있습니다. 첫 번째는 병렬 처리를 보다 효과적으로 사용한다는 것입니다. 우리는 오늘날 다중 코어에서 볼 수 있는 다중 명령어, 다중 데이터 환경에서 단일 명령어 다중 데이터로 전환합니다. 따라서 각 코어에 별도의 명령어 스트림을 가져 오지 않고 별도의 캐시가 있어야 하며 명령어 집합이 하나뿐이므로 기능 단위 집합 전체로 갈 것입니다. 훨씬 더 효율적입니다. 나는 무엇을 포기 하는가? 내가 그렇게 할 때 유연성을 포기한다. 나는 절대적으로 융통성을 포기합니다. 그러나 효율성은 극적으로 향상됩니다. ARM 이나 Intel 의 전형적인 하이 엔드 프로세서가 오늘날의 VLIW 와 비슷한 개념의 비 순차적인 시스템에서 컴파일러가 일련의 작업을 결정한 작업 집합을 사용합니다. 병렬로 발생할 수 있습니다. 그래서 저는 작업을 런타임에서 컴파일 시간으로 바꿉니다. 다시 말하지만 유연성이 떨어집니다. 그러나 작동하는 응용 프로그램의 경우 훨씬 효율적입니다. 나는 캐시에서 벗어난다. 따라서 캐시는 컴퓨터 과학의 위대한 발명품 중 하나이며 진정으로 위대한 발명품 중 하나입니다. 문제는 낮은 공간적 및 시간적 지역성이 낮고 캐시가 작동하지 않을 뿐만 아니라 실제로 프로그램이 느려 지는 경우입니다. 그들은 천천히. 그래서 우리는 그것을 사용자 제어 로컬 메모리로 옮깁니다. 절충점은 무엇입니까? 이제 누군가가 자신의 응용 프로그램을 사용자 제어 메모리 구조로 매핑하는 방법을 찾아야 합니다. 캐시가 자동으로 수행하므로 매우 일반적인 목적입니다. 그러나 특정 응용 프로그램의 경우 이러한 것들을 직접 매핑하여 훨씬 잘 수행 할 수 있습니다. 그리고 나서 마침내 필자는 필요한 정확도에 집중한다. 저는 IEEE 에서 저 정밀도 부동 소수점으로, 또는 32 및 64 비트 정수에서 8 비트 및 16 비트 정수로 이동했습니다. 이것이 내가 필요한 모든 정확도라면, 나는 하나의 64 비트 연산을 수행 할 수 있는 동일한 시간 내에 8 개의 정수 연산, 8 개의 8 비트 연산을 수행 할 수 있다.




So considerably faster. But to go along with that, I also need a domain-specific language. I need a language that will match up to that hardware configuration. We're not going to be able to take code written in Python or C, for example, and extract the kind of information we need to map to a domain-specific architecture. We've got to rethink how we program these machines. And that's going to be high-level operations. It's going to be vector-vector multiply or a vector-matrix multiply or a sparse matrix organization, so that I get that high-level information that I need and I can compile it down into the architecture. The key in doing these domain-specific languages will be to retain enough machine independence that I don't have to recode things, that a compiler can come along, take a domain-specific language, map it to maybe one architecture that's running in the cloud, maybe another architecture that's running on my smartphone. That's going to be the challenge. Ideas like TensorFlow and OpenGL are a step in this direction, but it's really a new space. We're just beginning to understand it and understand how to design in this space.

매우 빨라졌습니다. 그러나 그와 함께 가기 위해, 나는 또한 도메인 특정 언어가 필요하다. 해당 하드웨어 구성과 일치하는 언어가 필요합니다. 예를 들어 Python 이나 C 로 작성된 코드를 가져올 수 없으며 특정 도메인 아키텍처에 매핑하는 데 필요한 정보를 추출 할 수 있습니다. 우리는 이 기계들을 어떻게 프로그래밍 할 것인지 다시 생각해야합니다. 그리고 그것은 높은 수준의 운영이 될 것입니다. 벡터 벡터 곱셈 또는 벡터 행렬 곱셈 또는 희소 행렬 구성이므로 필요한 고급 정보를 얻고 아키텍처로 컴파일 할 수 있습니다. 이러한 도메인 특정 언어를 수행하는 데 있어 핵심은 내가 물건을 다시 코딩 할 필요가 없는 충분한 기계 독립성을 유지하는 것, 컴파일러가 도메인 특정 언어를 사용하여 이를 실행시킬 수 있는 하나의 아키텍처에 매핑 할 수 있다는 것입니다. 클라우드, 내 스마트 폰에서 실행중인 다른 아키텍처 일 수 있습니다. 그것은

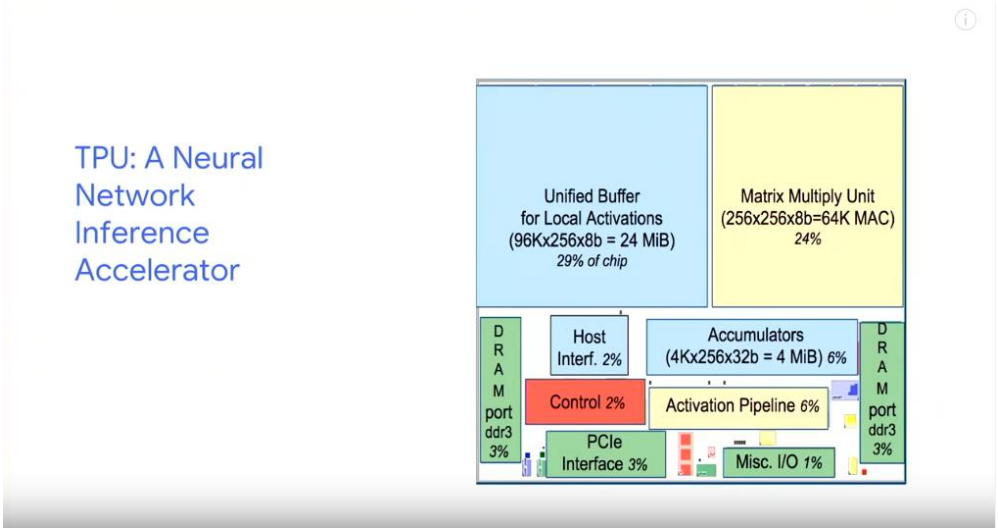
도전이 될 것입니다. TensorFlow 및 OpenGL 과 같은 아이디어는 이 방향의 한 걸음이지만 실제로는 새로운 공간입니다. 우리는 이제 그것을 이해하고 이 공간에서 어떻게 디자인 할 것인지 이해하기 시작했습니다.




You know, I built my first computer almost 50 years ago, believe it or not. I've seen a lot of revolutions in this incredible IT industry since then--the creation of the internet, the creation of the World Wide Web, the magic of the microprocessor, smartphones, personal computers. But the one I think that is really going to change our lives is the breakthrough in machine learning and artificial intelligence. This is a technology which people have worked on for 50 years. And finally, finally, we made the breakthrough. And the basis of that breakthrough? We needed about a million times more computational power than we thought we needed to make the technology work. But we finally got to the point where we could apply that kind of computer power. And the one thing-- this is some data that Jeff Dean and David Patterson and Cliff Young collected-- that shows there's one thing growing just as fast as Moore's law-- the number of papers being published in machine learning. It is a revolution. It's going to change our world. And I'm sure some of you saw the Duplex demo the other day. I mean, in the domain of making appointments, it passes the Turing test in that domain, which is an extraordinary breakthrough. It doesn't pass it in the general terms, but it passes it in a limited domain. And that's really an indication of what's coming. So how do you think about building a domain-specific architecture to do deep neural networks?

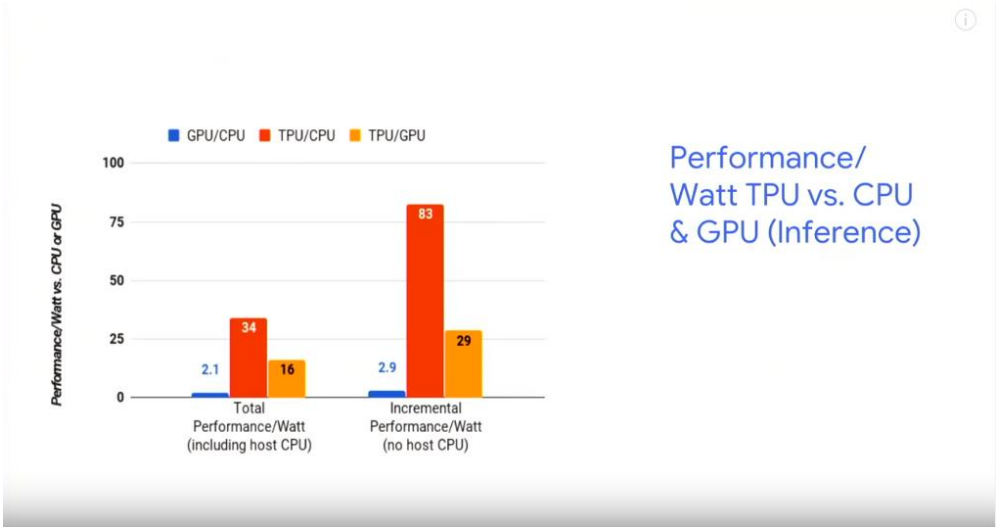
나는 거의 50 년 전에 처음으로 컴퓨터를 만들었고, 믿거나 말거나. 그 이후로 인터넷의 창출, 월드 와이드 웹의 창조, 마이크로 프로세서, 스마트 폰, 개인용 컴퓨터의 마법과 같은, 이 놀라운 IT 산업에서 많은 혁명을 겪었습니다. 그러나 실제로 우리 삶을 변화시킬 것이라고 생각하는 것은 기계 학습과 인공 지능의 돌파구입니다. 이것은 사람들이 50 년 동안 일해온 기술입니다. 그리고 마지막으로, 우리는 획기적인 변화를 만들었습니다. 그리고 그 돌파구의 기초? 우리는 기술력을 향상시키는 데 필요한 것보다 약 백만 배 더 많은 계산 능력이

필요했습니다. 그러나 마침내 우리는 그런 종류의 컴퓨터 성능을 적용 할 수 있게 되었습니다. 한 가지는 - Jeff Dean 과 David Patterson 및 Cliff Young 이 수집한 데이터입니다. 이는 무어의 법칙과 마찬가지로 빠르게 성장하는 것으로 나타났습니다. 기계 학습에 게시되는 논문의 수입니다. 혁명이다. 그것은 우리의 세상을 변화시킬 것입니다. 얼마 전 Duplex 데모를 보신 분도 계실 것입니다. 제 말은 약속을 정하는 영역에서 그 영역에서 Turing 테스트를 통과한다는 것입니다. 이것은 획기적인 돌파구입니다. 일반적인 용어로는 전달하지 않지만 제한된 도메인에서만 전달합니다. 그리고 그것은 실제로 무엇이 올 것인가의 표시입니다. 그렇다면 심층 신경 네트워크를 수행하기위한 도메인 별 아키텍처를 어떻게 구축 할 것입니까?




Well, this is a picture of what's inside a tensor processing unit. The point I want to make about this is if you look at this what uses up the silicon area, notice that it's not used for a lot of control, it's not used for a lot of caching. It's used to do things that are directly relevant to the computation. So this processor can do 256 by 256--that is 64,000 multiply accumulates, 8-bit multiply accumulates every single clock. Every single clock. So it can really crunch through, for inference things, enormous amounts of computational capability. You're not going to run general purpose C code on this. You're going to run something that's a neural network inference problem.

음, 이것은 텐서 처리 장치 내부에 있는 것의 그림입니다. 내가 이것에 관해서 지적하고자 하는 점은 실리콘 영역을 사용하는 것을 보았을 때 많은 제어를 위해 사용되지 않는다는 것을 알아 차리고 많은 캐싱에 사용되지 않는다는 것입니다. 계산과 직접적으로 관련이 있는 작업을 수행하는 데 사용됩니다. 따라서 이 프로세서는 256 x 256 을 수행할 수 있습니다. 즉, 64,000 곱셈이 누적되고, 8 비트 곱셈이 매 클럭마다 누적됩니다. 모든 단일 시계. 따라서 추론을 위해 엄청난 양의 계산 기능을 제공합니다. 이것에 대해 범용 C 코드를 실행하지 않을 것입니다. 당신은 신경망 추론 문제를 일으킬 것입니다.




And if you look at the performance and you look at-- here we've shown performance per watt. Again, energy being the key limitation. Whether it's for your cell phone and you're doing some kind of machine learning on your cell phone or it's in the cloud, energy is the key limitation. So what we plotted here is the performance per watt. And you see that the first generation tensor processing unit gets roughly more than 30 times the performance per watt compared to a general purpose processor. It even does considerably better than a GPU, largely by switching from floating point to lower density integer, which is much faster. So again, this notion of tailoring the architecture to the specific domain becomes really crucial.

퍼포먼스를 살펴보면 여기에 와트 당 성능이 나타났습니다. 다시 말하지만, 에너지가 핵심 한계입니다. 휴대 전화 용이든 휴대 전화로 학습한 일종의 기계를 사용하든 클라우드에 있든 상관없이 에너지가 핵심 한계입니다. 그래서 여기에 우리가 계획한 것은 와트당 성능입니다. 또한 1 세대 텐서 프로세싱 유닛은 범용 프로세서에 비해 와트당 성능이 대략 30 배 이상이 됩니다. GPU 보다 훨씬 더 뛰어납니다. 부동 소수점에서 저밀도 정수로 전환하는 것이 훨씬 더 빠릅니다. 다시 말해, 아키텍처를 특정 도메인에 맞추는 개념이 정말로 중요합니다.


So this is a new era. In some sense, it's a return to the past. In the early days of computing, as computers were just being developed, we often had teams of people working together. We had people who were early applications experts working with people who were doing the beginning of the software environment-- building the first compilers and the first software environment-- and people doing the architecture. And they're working as a vertical team. That kind of integration, where we get a design team that understands how to go from application to representation in some domain-specific language to architecture and can think about how to rebuild machines in new ways to get this, it's an enormous opportunity and it's a new kind of challenge for the industry to go forward. But I think there are enough interesting application domains like this where we can get incredible performance advantages by tailoring our machines in a new way. And I think if we can do that, maybe it will free up some time to worry about another small problem, namely cybersecurity and whether or not the hardware designers can finally help the software designers to improve the security of our system. And that would be a great problem to focus on. Thank you for your attention and I'm happy to answer any questions you might have. Thanks.

그래서 이것은 새로운 시대입니다. 어떤 면에서는 과거로 돌아온 것입니다. 컴퓨팅 초기에는 컴퓨터가 개발됨에 따라 팀원들이 함께 작업하는 경우가 많았습니다. 첫 번째 컴파일러와 첫 번째 소프트웨어 환경을 구축하는 소프트웨어 환경의 시작과 아키텍처를 수행하는 사람들과 일하는 초기 애플리케이션 전문가인 사람들이 있었습니다. 그리고 그들은 수직 팀으로 일하고 있습니다. 이러한 종류의 통합은 특정 도메인 언어에서 아키텍처로 응용 프로그램을 표현하는 방법을 이해하고 이를 새로운 방식으로 다시 작성하는 방법을 생각할 수 있는 디자인 팀이 있는 곳에서 엄청난 기회이며 업계가

앞으로 나아갈 새로운 종류의 도전. 그러나 우리는 새로운 방식으로 장비를 조정하여 놀라운 성능 이점을 얻을 수 있는 이와 같은 흥미로운 응용 프로그램 도메인이 충분하다고 생각합니다. 우리가 그렇게 할 수 있다면, 사이버 보안과 하드웨어 디자이너가 소프트웨어 디자이너가 마침내 우리 시스템의 보안을 향상시키는 데 도움이 될 수 있는지 여부와 같은 또 다른 작은 문제에 대해 걱정할 시간을 갖게 될 것입니다. 그리고 그것은 초점을 맞추는 큰 문제가 될 것입니다. 관심을 가져 주셔서 감사 드리며 궁금한 점에 대해 기꺼이 답변 해 드리겠습니다. 감사합니다.

# Q & A

AUDIENCE: Can you talk about some of the advances in quantum and neuromorphic computing?

양자 및 신경 모폴 컴퓨팅의 진보에 대해 이야기 해 주시겠습니까?

JOHN HENNESSY: Yeah. So quantum-- that's a really good question. So my view of this is that we've got to build a bridge from where we are today to post-silicon. The possibilities for post-silicon, there are a couple. I mean there's organic, there's quantum, there's carbon nanofiber, there's a few different possibilities out there. I characterize them as technology of the future. The reason is the people working on them are still physicists. They're not computer scientists yet or electrical engineers, they're physicists. So they're still in the lab. On the other hand, quantum, if it works, the computational power from a reasonably modest sized qubit, let's say 128 corrected qubits, 128 corrected qubits, meaning they're accurate, that might take you 1,000 qubits to get to that level of accuracy. But the computational power for things that make sense, protein folding, cryptography, of 128-bit qubit is phenomenal. So we could get an enormous jump forward there. We need something post-silicon. We need something post-silicon. We've got maybe, as Moore's law slows down, maybe another decade or so before it comes to a real halt. And we've got to get an alternative technology out there, because I think there's lots of creative software to be written that wants to run on faster machines.

네. 그래서 양자는 정말 좋은 질문입니다. 그래서 내 생각은 우리가 현재의 위치에서 포스트 실리콘까지 다리를 놓아야 한다는 것입니다. 포스트 실리콘의 가능성은 몇 가지 있습니다. 유기체가 있고, 양자가 있고, 탄소 나노 섬유가 있으며, 거기에는 몇 가지 가능성이 있습니다. 나는 그것들을 미래의 기술로서 특성화한다. 그(것)들에 종사하는 사람들은 아직도 물리학자이다. 그들은 아직 컴퓨터 공학자가 아니고 전기 기술자도 아니며 물리학자입니다. 그래서 그들은 실험실에 있습니다. 다른 한편, 양자가 작동한다면 합리적으로 적당한 크기의 큐 비트의 계산력을 말하자면, 128 개의 수정된 큐 비트, 128 개의 수정된 큐 비트가 정확하다는 것을 말하면, 1,000 개의 큐 비트가 정확도 수준에 도달 할 수 있습니다 . 그러나 128 비트 퀴 비트의 단백질 접힘, 암호화, 감각을 만드는 것들에 대한 계산 능력은 경이롭습니다. 그래서 우리는 거기서 엄청난 도약을 할 수 있었습니다. 포스트 실리콘이 필요합니다. 포스트 실리콘이 필요합니다. 우리는 아마도 무어의 법칙이 진정되기 전에 어쩌면 또 다른 10 년이 될 수도

있습니다. 그리고 빠른 기술을 사용하여 실행하고자 하는 많은 창조적인 소프트웨어가 있다고 생각하기 때문에 대안 기술을 도입해야 합니다.

AUDIENCE: I just-- at the end of your presentation, you briefly mentioned how we could start using hardware to increase security. Would you mind elaborating on that?

방금 프리젠테이션이 끝날 때 보안을 강화하기 위해 하드웨어를 어떻게 사용할 수 있는지 간단히 언급했습니다. 그걸 좀 더 자세히 생각해 주시겠습니까?

JOHN HENNESSY: Sure. Sure. OK, so here's my view with security. Everybody knows about Meltdown and Spectre? First thing about Meltdown and Spectre is to understand what happened is an attack that basically undermined architecture in a way that we never anticipated. I worked on out-of-order machines in the mid-1990s. That's how long that bug has been in those machines, since the 1990s. And we didn't even realize it. We didn't even realize it. And the reason is that basically what happens is our definition of architecture was there is an instruction set. Programs run. I don't tell you how fast they run, all I tell you is what the right answer is. Side channel attacks that use performance to leak information basically go around our definition of architecture. So we need to rethink about architecture. You know, in the 1960s and 1970s, there was a lot of thought about how to do a better job of protection. Rings and domains and capabilities. They all got dropped. And they got dropped because two things. First of all, we became convinced that people were going to verify their software and it was always going to be perfect. Well, the problem is that the amount of software we write is far bigger than the amount of software we ever verify, so that's not going to help. I think it's time for architects to begin to think about how can they help software people build systems which are more secure? What's the right architecture support to make more secure systems? How do we build those? How do we make sure they get used effectively? And how do we together-- architects and software people working together-- create a more secure environment? And I think it's going to mean thinking back about some of those old ideas and bringing them back in some cases.

확실한. 확실한. 좋아, 보안에 대한 나의 견해가 있다. 멜트 다운과 스펙터에 대해 모두 알고 있습니까? Meltdown and Spectre 의 첫 번째 일은 우리가 결코 예상하지 못한 방식으로 아키텍처를 근본적으로 손상시키는 공격이 무엇인지 이해하는 것입니다. 나는 1990 년대 중반에 주문형 기계를 사용했다. 1990 년대 이래로 그 기계에 벌레가 얼마나 오래 있었습니까. 그리고 우리는 그것을

깨닫지도 못했습니다. 우리는 그것을 깨닫지도 못했습니다. 그 이유는 기본적으로 아키텍처의 정의에는 명령어 세트가 있다는 것입니다. 프로그램이 실행됩니다. 나는 그들이 얼마나 빨리 달리는 지 말하지 않고, 나는 당신에게 옳은 대답이 무엇인지 말해 준다. 정보를 누설하기 위해 성능을 사용하는 사이드 채널 공격은 기본적으로 아키텍처에 대한 정의를 둘러싼 것입니다. 그래서 우리는 건축에 관해 재고할 필요가 있습니다. 아시다시피, 1960 년대와 1970 년대에 더 나은 보호 방법을 생각했습니다. 반지와 도메인 및 기능. 그들 모두는 떨어졌다. 그리고 두 가지 때문에 떨어졌습니다. 우선, 우리는 사람들이 자신의 소프트웨어를 검증 할 것이며 항상 완벽 할 것이라고 확신하게 되었습니다. 글쎄, 문제는 우리가 작성하는 소프트웨어의 양이 우리가 검증 한 소프트웨어의 양보다 훨씬 많기 때문에 도움이 되지 않는다는 것입니다. 건축가가 소프트웨어 사람들이 어떻게 더 안전한 시스템을 구축 할 수 있도록 도울 수 있을지 생각해보십시오. 보다 안전한 시스템을 만들기 위해 올바른 아키텍처 지원은 무엇입니까? 어떻게 그걸 만들까요? 우리는 어떻게 그들이 효과적으로 사용되는지 확인합니까? 그리고 함께 협력하는 건축가와 소프트웨어 사람들은 어떻게 보다 안전한 환경을 조성합니까? 그리고 나는 이 낡은 아이디어에 대해 생각해보고 어떤 경우에는 되돌아 오는 것을 의미할 것이라고 생각합니다.

AUDIENCE: After I took my processor architecture class, which used your book—

귀하의 서적을 사용했던 프로세서 아키텍처 클래스를 사용한 후,

JOHN HENNESSY: I hope it didn't hurt you.

나는 그것이 당신을 해치지 않았으면 좋겠다.

AUDIENCE: Hopefully not. I had a real appreciation for the simplicity of a risk system. It seems like we've gone towards more complexity with domain-specific languages and things. Is that just because of performance or has your philosophy changed? What do you think?

바라기를 바랍니다. 나는 위험 시스템의 단순함에 대해 정말로 감사했다. 도메인 별 언어 및 사물에 대한 복잡성이 증가한 것처럼 보입니다. 그것은 단지 성과 때문입니까 아니면 철학이 바뀌었습니까? 어떻게 생각해?

JOHN HENNESSY: No, I actually think they're not necessarily more complicated. They have a narrower range of applicability. But they're not more complicated in the sense that they are a better match for what the application is. And the key thing to understand about risk, the key

insight was we weren't targeting people writing assembly language anymore. That was the old way of doing things, right? In the 1980s, the move was on. Unix was the first operating system ever written in a high level language, the first ever. The move was on from assembly language to high level languages. And what you needed to target was the compiler output. So it's the same thing here. You're targeting the output of a domain-specific language that works well for a range of domains. And you design the architecture to match that environment. Make it as simple as possible, but no simpler.

아니요, 사실 그들이 반드시 더 복잡하지는 않다고 생각합니다. 그들은 적용 범위가 더 좁습니다. 그러나 애플리케이션이 무엇인지 더 잘 일치한다는 점에서 그들은 더 복잡하지 않습니다. 위험에 대해 알아야 할 핵심 사항은 더 이상 어셈블리 언어 작성자를 목표로 하지 못했다는 것입니다. 그게 옛 일을 하는 방식 이었지, 그렇지? 1980 년대에 이 움직임이 시작되었습니다. 유닉스는 처음으로 높은 수준의 언어로 작성된 최초의 운영체제였습니다. 이 작업은 어셈블리 언어에서부터 고급 언어까지 진행되었습니다. 그리고 당신이 필요로 했던 것은 컴파일러 출력이었습니다. 그래서 여기서도 마찬가지입니다. 도메인 범위에서 잘 작동하는 도메인별 언어의 결과물을 타겟팅 합니다. 그리고 그 환경에 맞는 아키텍처를 설계하십시오. 가능하면 단순하게 만드십시오. 그러나 더 간단하지는 마십시오.

AUDIENCE: With the domain-specific architectures, do you have examples of what might be the most promising areas for future domain-specific architectures?

도메인 별 아키텍처의 경우 향후 도메인 별 아키텍처에서 가장 유망한 영역이 무엇인지에 대한 예가 있습니까?

JOHN HENNESSY: So I think the most obvious one are things related to machine learning. I mean, they're computationally extremely intensive, both training as well as inference. So that's one big field. Virtual reality. Virtual reality and augmented reality environments. If we really want to construct a high-quality environment that's augmented reality, we're going to need enormous amounts of computational power. But again, it's well-structured kinds of computations that could match to those kinds of applications. We're not going to do everything with domain-specific architectures. They're going to give us a lift on some of the more computationally-intensive problems. We're still going to have to advance and think about how to push forward general purpose, because the general purpose machines are going to drive these domain-specific machines. The domain-specific machine will not do everything for us. So we're going to have to figure out ways to go

forward on that front as well.

가장 확실한 것은 기계 학습과 관련된 것입니다. 내 말은, 그들은 계산적으로 매우 집중적인데, 훈련뿐 아니라 추론입니다. 그래서 그것은 하나의 커다란 분야입니다. 가상 현실. 가상 현실 및 증강 현실 환경. 현실을 증대시키는 고품질의 환경을 만들고자 한다면 엄청난 양의 계산 능력이 필요합니다. 그러나 다시 한 번, 이러한 종류의 응용 프로그램과 일치 할 수 있는 잘 구조화된 계산 유형입니다. 도메인 별 아키텍처로 모든 것을 수행하지는 않습니다. 그들은 더 많은 계산 집중적 인 문제들에 대해 우리에게 관심을 가질 것입니다. 범용 컴퓨터가 이러한 도메인 특정 컴퓨터를 구동 할 것이기 때문에 범용 목적을 추진하는 방법을 계속 발전시키고 생각해야합니다. 도메인 특정 시스템은 우리를 위해 모든 것을 수행하지 않습니다. 그래서 우리는 앞으로 나아갈 길을 찾아야 할 것입니다.

AUDIENCE: Professor, what do we think about some emerging memory technology? How will it impact the future computer architecture? Thank you.

교수님, 떠오르는 메모리 기술에 대해 어떻게 생각합니까? 미래의 컴퓨터 아키텍처에 어떤 영향을 미칩니까? 고맙습니다.

JOHN HENNESSY: Yeah, that's a really great question. So as we get to the end of DRAMs, I think some of the more innovative memory technologies are beginning to appear. So-called phase change technologies, which have the advantage that they can probably scale better than DRAM and probably even better than Flash technologies. They have the advantage that lifetimes are better, too, than Flash. The problem with Flash is it wears out. Some of these phase change memories or memristor technologies have the ability to scale longer. And what you'll get is probably not a replacement for DRAM. You'll probably get a replacement for Flash and a replacement for disks. And I think that technology is coming very fast. And it'll change the way we think about memory hierarchies and I/O hierarchy, because you'll have a device that's not quite as fast as DRAM, but a lot faster than the other alternatives. And that will change the way we want to build machines.

그래, 정말 좋은 질문이네. 그래서 우리가 DRAM 을 끝내고 나면 좀 더 혁신적인 메모리 기술이 등장하기 시작했다고 생각합니다. 소위 상 변화 기술로, 아마 DRAM 보다 확장 가능하고 플래시 기술보다 뛰어날 수 있다는 이점이 있습니다. 그들은 플래시보다 수명이 더 좋습니다 장점이 있습니다. 플래시 문제는 지치고 있습니다. 이러한 상 변화 메모리 또는 멤 리스터 기술 중 일부는 더 오랜 시간 확장 할 수 있습니다. 그리고 아마도 당신은 DRAM 을 대신 할 수 있는 것은 아닙니다. 플래시 대체품과 디스크 교체품이 나옵니다. 그리고 저는 기술이 매우

빠르게 발전하고 있다고 생각합니다. 그리고 메모리 계층과 I/O 계층 구조에 대한 생각을 바꾸게 될 것입니다. 왜냐하면 당신은 DRAM 만큼 빠르지만 다른 대안보다 훨씬 빠른 장치를 갖기 때문입니다. 그리고 그것은 우리가 기계를 만들고 싶은 방식을 바꿀 것입니다.

AUDIENCE: As a person, you think about education quite often. We all saw Zuckerberg having a conversation with Congress. And I'm excited to see children getting general education around computing and coding, which is something that a lot of us didn't have the opportunity to have. Where do you see education, not only for K-12, grad, post-grad, et cetera, but also existing people in policy-making decisions, et cetera?

사람으로서, 당신은 교육에 관해 자주 생각합니다. 우리 모두 주커 버그가 의회와 대화하는 것을 보았습니다. 아이들이 컴퓨팅 및 코딩에 관한 전반적인 교육을 받는 것을 보게 되어 기쁩니다. 많은 사람들이 가질 기회가 없었습니다. K-12, 졸업생, 졸업생 등의 교육 뿐만 아니라 정책 결정 과정에서 기존 사람들에 대한 교육은 어디에서 볼 수 있습니까?

JOHN HENNESSY: Yeah. Well, I think first of all, education has become a lifelong endeavor. Nobody has one job for a lifetime anymore. They change what they're doing and education becomes constant. I mean, you think about the stuff you learned as an undergrad and you think how much technology has already changed, right? So we have to do more there. I think we also have to make more-- society needs to be more technology-savvy. Computing is changing every single part of the world we live in. To not have some understanding into that technology, I think, limits your ability to lead an organization, to make important decisions. So we're going to have to educate our young people at the beginning. And we're going to have to make an investment in education so that as people's careers change over their lifetime, they can go back and engage in education. Not necessarily going back to college, it's going to have to be online in some way. But it's going to have to be engaging. It's going to have to be something that really works well for people.

네. 우선, 저는 교육이 평생의 노력이 되어 왔다고 생각합니다. 아무도 일생 동안 한 가지 직업을 가지고 있지 않습니다. 그들은 그들이 하는 일을 바꿔 교육이 일정 해집니다. 내 생각에, 당신은 학부생으로서 배운 것들에 대해 생각하고 얼마나 많은 기술이 이미 바뀌었는지 생각할 것입니다, 그렇죠? 그래서 우리는 더 많은 것을 해야 합니다. 사회를 더 기술에 정통시켜야 합니다. 우리가 살고있는 세계의 모든 부분이 컴퓨팅으로 바뀌고 있습니다. 이 기술에 대해 어느 정도 이해하지 못하면 조직을 이끌고 중요한 결정을 내릴 수 있는 능력이 제한됩니다. 그래서 우리는 처음에는 젊은 사람들을 교육해야 할 것입니다.

그리고 우리는 교육에 투자해야 할 것입니다. 그래서 사람들의 직업이 평생 동안 바뀌면 돌아와 교육에 참여할 수 있습니다. 꼭 대학에 가지 않고, 어떤 면에서는 온라인 상태여야 합니다. 그러나 매력적이어야 합니다. 사람들을 위해 실제로 잘 작동하는 무언가가 되어야 할 것입니다.

AUDIENCE: Hi. Olly [INAUDIBLE] from BBC. Just wondered what your view is on the amount of energy being used on Bitcoin mining and other cryptocurrencies and that sort of thing.

안녕. BBC 의 Olly [INAUDIBLE]. Bitcoin 마이닝 및 기타 크립토크루스(cryptocurrencies)와 그런 종류의 일에 사용되는 에너지의 양에 대한 귀하의 견해가 궁금합니다.

JOHN HENNESSY: Yeah. So I could build a special purpose architecture to mine Bitcoins. That's another obvious example of a domain-specific architecture for sure. So I'm a long-term believer in cryptocurrency as an important part of our space. And what we're going to have to do is figure out how to make it work, how to make it work efficiently, how to make it work seamlessly, how to make it work inexpensively. I think those are all problems that can be conquered. And I think you'll see a bunch of people that have both the algorithmic heft and the ability to rethink how we do that, and really make cryptocurrencies go quite quick. And then we can also build machines which accelerate that even further, so that we can make--a cryptocurrency transaction should be faster than a cash transaction and certainly no slower than a credit card transaction. We're not there yet. But we can get there. We can get there with enough work. And I think that's where we ought to be moving to.

네. 그래서 나는 비트 코인 광산을 위한 특별한 목적의 아키텍처를 구축할 수 있었습니다. 이것이 도메인 별 아키텍처의 확실한 예입니다. 그래서 저는 우리 공간의 중요한 부분으로서 암호 해독에 대한 장기적인 신자입니다. 우리가 해야 할 일은 그것이 어떻게 작동하게 하는지, 어떻게 효율적으로 작동하게 하는지, 원활하게 작동하게 만드는 방법, 그리고 그것이 값 싸게 작동하게 만드는 방법입니다. 나는 그 모든 것이 정복 될 수 있는 문제라고 생각한다. 그리고 나는 알고리즘의 중요성과 능력을 재고 할 수 있는 사람들이 많이 있다는 것을 알게 될 것입니다. 그리고 크립토 통화를 아주 빠르게 만들어 줄 것입니다. 그리고 우리는 이를 더욱 가속화 시켜서 우리가 만들 수 있는 기계를 만들 수 있습니다. 암호 해독 트랜잭션은 현금 거래보다 빠르며 신용 카드 거래보다 느리지 않습니다. 우리는 아직 거기에 없다. 그러나 우리는 거기에 갈 수 있습니다. 우리는 충분한 노력으로 거기에 갈 수 있습니다. 그리고 그것이 우리가 움직여야 하는 곳이라고 생각합니다.

AUDIENCE: What do you think the future operating system has to have to cope with this?

미래의 운영 체제가 이 문제에 어떻게 대처해야 한다고 생각하십니까?

JOHN HENNESSY: Yeah. The future of operating system, you said, yes? Yeah. So I think operating systems are really crucial. You know, way back when in the 1980s, we thought we were going to solve all our operating system problems by going to kernel-based operating systems. And the kernel would be this really small little thing that just did the core functions of protection and memory management. And then, everything else around it would be protected, basically. And what happened was kernel started out really small and then they got bigger and then they got bigger and then they got bigger. And all of a sudden, almost the entire operating system was in the kernel, primarily to make it performance-efficient. And the same thing happen with hypervisors. They started really small in the very beginning and then they got bigger. We're going to have to figure out how we structure complex operating systems so that they can deal with the protection issues, they can deal with efficiency issues, they can work well. We should be building operating systems which, from the beginning, realize that they're going to run on large numbers of processors, and organize them in such a way that they can do that efficiently. Because that's the future, we're going to have to rely on that.

네. 운영 체제의 미래, 당신이 말했다, 네? 네. 그래서 운영 체제가 정말로 중요하다고 생각합니다. 아시다시피, 1980 년대에 우리는 커널 기반 운영 체제로 이동하여 모든 운영 체제 문제를 해결할 것이라고 생각했습니다. 그리고 커널은 방금 보호 및 메모리 관리의 핵심 기능을 수행한 이 작은 작은 것입니다. 그리고 나서, 그 주변의 모든 것은 기본적으로 보호될 것입니다. 그리고 무슨 일이 있었는지는 커널이 정말로 작게 시작한 다음 커졌다가 커졌다가 커지고 커지면 더 커졌습니다. 그리고 갑자기, 거의 모든 운영체제가 커널에 있었고 주로 성능을 효율적으로 사용했습니다. 하이퍼 바이저에서도 마찬가지입니다. 그들은 처음부터 아주 작게 시작했고 그 다음에는 더 커졌습니다. 복잡한 운영 체제를 어떻게 구성하여 보호 문제를 처리할 수 있는지, 효율성 문제를 처리 할 수 있는지, 제대로 작동하는지 파악해야 합니다. 우리는 처음부터 많은 수의 프로세서에서 실행되고 효율적으로 그렇게 할 수 있는 방식으로 구성 할 수 있는 운영 체제를 구축해야 합니다. 그것이 미래이기 때문에 우리는 그것에 의존해야 할 것입니다.

AUDIENCE: In your intro video, you mentioned this chasm between concept and practice. And also in your talk, you've mentioned that hardware is vital to the future of computing. Given that most investors are very hardware-averse, especially this day and age, where do you expect that money to come from? Is that something that will come from governments or private investing? How are we going to fund the future of computing is really what my question is.

소개 비디오에서 개념과 실천 사이의 틈을 언급했습니다. 또한 여러분의 이야기에서 하드웨어는 컴퓨팅의 미래에 필수적이라고 언급했습니다. 대부분의 투자자들이 하드웨어 혐오, 특히 오늘날과 같은 경우, 돈이 어디에서 오는 것으로 예상하십니까? 그것은 정부나 민간 투자에서 나오는 것입니까? 우리는 어떻게 컴퓨팅의 미래에 대한 기금을 마련 할 것인가가 실제로 내 질문입니다.

JOHN HENNESSY: Yeah, it's a good question. I mean, I think the answer is both. You know, certainly Google's making large investments in a lot of these technologies from quantum to other things. I think government remains a player. So government, you look at how many of the innovations we're used to. The internet, risk, the rise of VLSI, modern computer-aided design tools. All had funding basically coming from the government at some point. So I think the government should still remain a player in thinking about--what's the one area the government has probably funded longer than anybody else? Artificial intelligence. They funded it for 50 years before we really saw the breakthrough that came. Right? So they're big believers. They should be funding things long-term. They should fund things that are out over the horizon that we don't yet really understand what their practical implications may be. So I think we're going to have to have that and we're going to have to have industry playing a big role. And we're going to have to make universities work well with industry, because they complement one another, right? They do two different kinds of things but they're complementary. And if we can get them to work well, then we can have the best of both worlds.

그래, 좋은 질문이야. 제 대답은 둘 다 생각합니다. 알다시피, 확실히 Google 은 양자 기술에서부터 다른 기술에 이 기술에 많은 투자를 하고 있습니다. 나는 정부가 여전히 선수라고 생각한다. 그래서 정부, 우리는 지금까지 얼마나 많은 혁신이 있었는지 봅니다. 인터넷, 위험, VLSI 의 부상, 현대적인 컴퓨터 지원 설계 도구. 모두는 근본적으로 어떤 시점에서 정부로부터 오는 자금을 가지고 있었습니다. 그래서 정부는 여전히 생각할 선수로 남겨져 있어야 한다고 생각합니다. 정부가 아마도 다른 어느 나라보다 더 오래 투자 한 부분이 무엇일까요? 인공 지능. 그들은 우리가 실제로 얻은 돌파구를 실제로 보기 전에

50 년간 자금을 지원했습니다. 권리? 그래서 그들은 큰 신자입니다. 그들은 장기적으로 자금을 조달해야 합니다. 그들은 지평선 너머에 있는 것들을 기금으로 모아야한다. 우리는 실제로 그들의 실질적인 의미가 무엇인지 이해하지 못한다. 그래서 우리는 그렇게 해야 한다고 생각합니다. 업계에서 큰 역할을 해야 합니다. 우리는 대학들이 서로 보완하기 때문에 업계와 잘 어울리는 대학을 만들어야 합니다. 맞습니까? 그들은 두 가지 종류의 일을 하지만 보완적입니다. 우리가 잘 작동하도록 만들 수 있다면, 우리는 두 세계의 장점을 모두 누릴 수 있습니다.

AUDIENCE: You talked a little bit about the difference between the memory hierarchy and storage that is coming up with these new memory technologies. Have you seen any applications where the compute and the storage get combined, kind of more like the brain?

이 새로운 메모리 기술이 등장할 메모리 계층 구조와 스토리지의 차이점에 대해 조금 이야기했습니다. 컴퓨팅과 스토리지가 결합된 응용 프로그램을 보았습니까? 더 두뇌와 비슷합니다.

JOHN HENNESSY: Yeah, I think increasingly we'll see things move towards that direction where the software takes care of the difference between what is in storage and-- "storage," quote unquote, right, because it may actually be Flash or some kind of next generation memory technology-- and what's in DRAM. What you need to tell me is what's volatile and when do I have to ensure that a particular operation is committed to nonvolatile storage. But if you know that, we've got log base file systems, you've got other ideas which move in the direction of trying to take advantage of a much greatly different memory hierarchy, greatly different storage hierarchy than we're used to. And we may want to continue to move in that direction, particularly when you begin to think about--if you think about things like networking or I/O and they become major bottlenecks in applications, which they often do, then rethinking how we could do those efficiently and optimize the hardware, but also the software. Because the minute you stick an operating system transaction in there, you've added a lot of weight to what it costs to get to that storage facility. So if we can make that work better and make it more transparent without giving up protection, without giving up a guarantee that once something is written to a certain storage unit it's permanently recorded, then I think we can make much faster systems.

예, 소프트웨어가 저장 장치에 있는 것과 저장 장치의 차이를 처리하는 방향으로 이동하는 것을 점점 더 볼 수 있다고 생각합니다. "저장 장치"는 따옴표로 묶지 마십시오. 실제로 플래시 또는 다음 종류일 수 있습니다. 세대

메모리 기술 - 그리고 DRAM 에 무엇이 있는가? 당신이 말해야 할 것은 변동성이 크며 특정 작업이 비 휘발성 저장 장치에 맡겨져 있는지 확인해야 할 때입니다. 그러나 로그 기반 파일 시스템이 있다는 것을 알게 되면 훨씬 다른 메모리 계층 구조, 이전에 사용하던 것과는 크게 다른 저장소 계층 구조를 활용하려는 방향으로 나아가는 다른 아이디어를 얻게 됩니다. 특히 네트워킹에 대한 생각이나 I/O 에 대해 생각하고 응용 프로그램에서 주요 병목 현상을 일으키고 종종 우리가 할 수 있는 일을 생각해 보면 어떻게 생각하느냐에 따라 계속 나아갈 수 있습니다. 이를 효율적으로 수행하고 하드웨어를 최적화할 뿐 아니라 소프트웨어도 최적화하십시오. 운영 체제 트랜잭션을 사용하는 순간부터 해당 저장 장치에 들어가는 데 드는 비용에 많은 도움을 줍니다. 따라서 우리가 보호 장치를 포기하지 않고 더 나은 작업을 수행하고 투명하게 만들 수 있다면 어떤 저장 장치에 기록 된 내용이 영구적으로 기록되면 보증을 포기하지 않고 훨씬 빠른 시스템을 만들 수 있다고 생각합니다.

AUDIENCE: So do you see the implementation of a domain-specific architecture being implemented as hetero type or do you see it off-die, off-chip type implementations, or both?

그렇다면 도메인 특정 아키텍처의 구현이 헤테로 유형으로 구현되는 것을 보시겠습니까? 아니면 오프 다이 (off-die), 오프 칩 유형 구현 또는 둘 다 보입니까?

JOHN HENNESSY: I think both. I mean, I think it's a time of great change. The rise of FPGAs, for example, gives you the opportunity to implement these machines, try them out. Implement them in FPGA before you're committed to design a custom silicon chip. Put it in an FPGA. Unleash it on the world. Try it out, see how it works, see how the applications map to it. And then, perhaps, decide whether or not you want to freeze the architecture. Or you may just want to build another next generation FPGA. So I think we'll see lots of different implementation approaches. The one thing we have to do--you know, there was a big breakthrough in how hard it was to design chips that occurred from about the mid-'80s to about 1995 or 2000. Things have kind of ground to a halt since then. We haven't had another big--we need a big breakthrough because we're going to need many more people designing processors targeting particular application domains. And that's going to mean we need to make it much easier and much cheaper to design a processor.

나는 둘 다 생각한다. 나는 이것이 큰 변화의 시기라고 생각한다. 예를 들어, FPGA 의 등장으로 이 기계를 구현하고 시험해 볼 기회가 생깁니다. 커스텀 실리콘 칩을 설계하기 전에 FPGA 로 구현하십시오. 그것을 FPGA 에 넣으십시오.

그것을 세계에 공개하십시오. 그것을 시험해 보고, 어떻게 작동하는지 보고, 응용 프로그램이 그것에 어떻게 매핑되는지 보십시오. 그런 다음 아키텍처를 고정할지 여부를 결정합니다. 또는 차세대 FPGA 를 구축하고 싶을 수도 있습니다. 그래서 우리는 다양한 구현 방법을 보게 될 것이라고 생각합니다. 우리가 해야 할 한 가지는 - 당신이 아는 바로는, 80 년대 중반에서 1995 년 또는 2000 년 사이에 발생한 칩을 설계하는 것이 얼마나 어려운지에 대한 획기적인 돌파구가 있었습니다. 그 이후로는 일종의 정지가 있었습니다 . 우리에게는 또 다른 큰 특징이 없었습니다. 특정 응용 프로그램 도메인을 대상으로 하는 프로세서를 설계하는 사람들이 더 필요하기 때문에 큰 발전이 필요합니다. 이는 프로세서를 설계하는 것이 훨씬 쉽고 저렴해질 필요가 있음을 의미합니다.

AUDIENCE: I'm wondering, as a deep learning engineer for a private enterprise, what is my role in pushing forward DSA?

민간 기업을 위한 심층적인 학습 엔지니어로서 DSA 를 추진하는데 있어 나의 역할은 무엇일까요?

JOHN HENNESSY: Yeah. Well, I think your role is vital because we need people who really understand the application space. And that's really critical. And this is a change. I mean, if you think about how much architects and computer designers, hardware designers have had to think about the applications, they haven't had to think about them. All of a sudden, they're going to have to develop a bunch of new friends that they can interact with and talk to and colleagues they can work with, to really get the insights they need in order to push forward the technology. And that's going to be a big change for us, but I think it's something that's absolutely crucial. And it's great for the industry too, because all of a sudden we get people who are application experts beginning to talk people who are software domain experts or talk to hardware people. That's a terrific thing.

네. 자, 저는 응용 프로그램 공간을 실제로 이해하는 사람들이 필요하기 때문에 여러분의 역할이 중요하다고 생각합니다. 그리고 그것은 정말로 중요합니다. 그리고 이것은 변화입니다. 제 말은, 얼마나 많은 건축가와 컴퓨터 디자이너가 하드웨어 디자이너가 어플리케이션에 대해 생각해 보아야 하는지 생각해 본다면, 그것에 대해 생각할 필요가 없다는 것입니다. 갑자기 그들은 상호 작용하고 대화 할 수 있는 새로운 친구들을 사귈 수 있게 되었고, 동료와 이야기를 나눠서 기술을 발전시키기 위해 필요한 통찰력을 얻게 되었습니다. 그리고 그것은 우리에게는 커다란 변화가 될 것입니다. 그러나 저는 이것이 절대적으로 중요한 것이라고 생각합니다. 그리고 갑자기 우리는 응용 전문가 인

사람들이 소프트웨어 도메인 전문가이거나 하드웨어 사용자와 대화하기 시작하기 때문에 산업계에서도 좋습니다. 그것은 대단한 것입니다.

AUDIENCE: You mentioned the performance enhancements of domain-specific languages over Python, for instance, but they're also much harder to use. So do you think software engineering talent can keep up in the future?

예를 들어 Python 에 비해 도메인 특정 언어의 성능이 향상되었다고 언급했지만 사용하기가 훨씬 어려웠습니다. 그렇다면 소프트웨어 엔지니어링 인재가 앞으로도 계속 유지할 수 있다고 생각하십니까?

JOHN HENNESSY: Yeah. I think the challenge will be--the gain we've gotten in software productivity in the last 20 or 30 years is absolutely stunning. It is absolutely stunning. I mean, a programmer now can probably write 10 to 100 times more code than they could 30 years ago, in terms of functionality. That's phenomenal. We cannot give that up because that's what's created all these incredible applications we have. What we need to do is figure out--all of a sudden, we need a new generation of compiler people to think about how do we make those run efficiently. And by the way, if the gap is a factor of 25 between C and Python, for example, if you get only half that, that's a factor of 12 times faster. Any compiler writer that can produce code that runs 12 times faster is a hero in my book. So we have to just think about new ways to approach the problem. And the opportunity is tremendous.

네. 나는 도전이 될 것이라고 생각합니다. 지난 20~30 년 동안 소프트웨어 생산성에서 얻은 이득은 절대적으로 놀라운 것입니다. 그것은 절대적으로 충격적입니다. 즉, 프로그래머는 이제 기능면에서 30 년 전보다 10~100 배 더 많은 코드를 작성할 수 있습니다. 그것은 놀랍습니다. 그것이 우리가 가진 이 놀라운 응용 프로그램을 모두 만들었기 때문에 우리는 그것을 포기할 수 없습니다. 우리가 해야 할 일은 무엇인지 알아내는 것입니다. 갑자기 우리는 이들을 효율적으로 운영하는 방법에 대해 생각하는 새로운 세대의 컴파일러가 필요합니다. 그런데 만약 C 와 Python 사이의 간격이 25 정도라면, 예를 들어 그 절반 정도면 12 배 빠릅니다. 12 배 빠르게 실행되는 코드를 생성 할 수 있는 컴파일러 작성자는 저의 영웅입니다. 따라서 우리는 문제에 접근하는 새로운 방법에 대해서 생각해야 합니다. 기회는 엄청납니다.

AUDIENCE: Are there any opportunities still left in x86 as far as, like, lifting the complexity of the ISA into software and exposing more microarchitecture to the compiler?

x86 의 복잡성을 소프트웨어로 끌어 올리고 더 많은 마이크로 아키텍처를 컴파일러에 노출시키는 것과 같은 기회가 있습니까?

JOHN HENNESSY: It's tough. I mean, I think the Intel people have spent more time implementing x86s than anybody's ever spent implementing one ISA, one instruction set ever. They've mined out almost all the performance. And in fact, if you look at the tweaks that occur, for example, they do aggressive prefetching in the i7. But you look at what happens with prefetching, some programs actually slow down. Now on balance, they get a little bit of speed up from it, but they actually slow down other programs. And the problem right now is it's very hard to turn that dial in such a way that we don't get overwhelmed with negative things. And I see my producer telling me it's the end of the session. Thank you for the great questions and for your attention.

힘들어요. 내 말은, 인텔의 사람들이 x86 을 구현하는 데 더 많은 시간을 소비했다고 생각합니다. 그들은 거의 모든 성과를 채굴했습니다. 실제로 발생하는 조정을 보면 i7 에서 적극적인 프리 페치를 수행합니다. 그러나 프리 페치 (prefetching)로 어떤 일이 일어나는지 살펴보면서 실제로 느려지는 프로그램도 있습니다. 이제는 균형을 이루기 때문에 속도가 약간 빨라지지만 실제로는 다른 프로그램의 속도가 느려집니다. 그리고 문제는 바로 지금 우리가 부정적인 것들로 압도 당하지 않도록 다이얼을 돌리기가 매우 어렵다는 것입니다. 그리고 저는 프로듀서가 세션의 끝이라고 말했습니다.
위대한 질문과 관심에 감사드립니다.