RICHARD WIRT

Female Voice: Ladies and gentlemen, please welcome Richard Wirt.

[Applause]

Richard Wirt: Good morning. Thank you. I want to talk something about software today. Steve set the pace for hardware and where we're going there. I want to bring you up-to-date with some of the software that we're working on. We're going to focus primarily on parallelization and threading. And most of this will be focused at the core level, as opposed to the big cluster level. A lot of the techniques are applicable across both the single CPU as well as the large cluster. Now in the cluster there are standards, open MPI and products like that, that we do support. So if you have questions at the end, we'll take those also.

At Intel, as you've seen, we're rapidly moving to more and more cores per die. We just introduced not only the dual processor Woodcrest, but we've now moved to Cloverton, which has quad-core. So automatically if you put two of those in a system with quad-core, you've got the possibility of eight threads that you want to use. The biggest problem for software is, how do you take advantage of these? And if you think about it, this is Intel's scaling strategy; to go more and more core. So if we don't get the software to go threaded and have a lot more of that available, ultimately the end user is not going to see a lot of benefit.

There is benefit in multiple processes running at the same time. I'm sure all of you now have your browser open, and simultaneously you may have your email. Those are separate processes; they both can share those cores and take advantage of them. And so there is a lot of benefit in those type of applications. Some of the applications that we're seeing, though, are very innovative and I think you're going to see a lot more of that as we move forward. So how can I simultaneously take conversation, translate it, maybe even put it in Word text, and data mine on that Word text as I'm running. Now this kind of application is easy to break up into multiple functions that each can run on a separate thread and, therefore, on a separate core.

The other thing that you do is you cut movies -- you watch a movie, you download it, you burn it to a CD. You get a lot of functional type things that you can split apart and make threads work here. One of the things that we did is we took advantage of virtualization and began to build a system. So we have just released, and it will ship out through some of the OEMs, software that virtualizes -- because of the virtual support in the CPU, you virtualize the NIC driver. Therefore, you can begin putting manageability type functions on that virtual machine while you're running your other applications on the other virtual machine. So you have a way now of protecting the NIC driver and looking at all the data that's coming into your machine to look for viruses or management or whatever.

So another functional splitting on the virtual machine, each virtual machine runs on its own core and therefore you can bring more value to that. Seeing a lot in the way of gains; really driving quickly to an immersion type of virtualization where you are immersed in the scene. You're also seeing that in Google Earth, Google Maps, you're beginning to fly around, go right down and a lot of innovative things happening here that you're bringing the power that you can do.

Give you a little update where we're at as work with ISVs and get the software threaded. Last year when we talked to you, roughly 60 percent level of the ISVs that we deal with were beginning to thread their applications. Doesn't say all their applications, that they now are trained or they've got major applications in progress; some of those are shipping, some of them are starting to. That number's come up, and you see it's come up significantly in some countries -- APAC running ahead of others. But now it's well over 80-85 percent of the ISVs are beginning to move their apps. So that's good.

Here is a list of some of those ISVs. You'll see on this list most major ISVs that you're building applications around now are in this threading exercise. Now, what we're doing with most of these is we have AEs that we've trained on our own processes and tools that we're implanting in these companies; training them, but also helping them as they begin to thread those applications or rework those applications. So that's progress that we've made.

In looking at scaling your performance, there are a number of things that you want to look at. And I want to just quickly highlight those and what they are. A lot of performance can be gotten out micro-architecture tuning. Now this is basically taking advantage of the micro-architecture, knowing information about cache size, all of those kind of things, the number of buffers to the bus and write-back buffers and all those things, and then being able to tune around those or utilizing them most effectively.

This level of optimization is typically the last thing you do to get the last 30 percent performance out of that application. This is also the part that we do well in tuning our compiler to automatically do this for you. That's the advantage of using some of the tools that Intel puts out. So whether it's our compiler or our hand-tuned libraries such as our math library or our video codecs, much of the effort we put in there is along these lines.

The other area is the application of self-tuning. Do I have the right algorithms? Do those algorithms effectively utilize the architecture? Do they effectively share data? How about bus bandwidth? Am I keeping the processor fed? So, typically, what you do here is you use a tool like VTune, you go through and analyze, you look at the critical path through the application, where it's spending most of it's time, and you want to shorten that path. The shorter that path, the faster you get through that code and the faster the application runs. So that's a lot of what you're doing here.

In this case, we have provided the number of algorithms, particularly if you're into our math library algorithms -- a lot of uses of those in the talk that Steve gave. Most of

those major players are using libraries like MKL; a lot of matrix solvers in that. An example of an interesting one that we've been working with is Johns Hopkins University. They're trying to do brain modeling to help a surgeon. As they do surgery on a brain, they want to be able to model the brain and get dynamic feedback of that. So, getting an application like that fast enough, that they can use it real-time is what it's all about. We're working with them on their equation solvers and trying to get highly tuned equation solvers that will speed that up. On an application like that, you can typically get 3x, 5x, maybe even 10x performance gain by choosing the right type of algorithm and then adapting your application to that algorithm.

Another type of tuning is system tuning. Now this is an area that's very important to look at, not only the network but the I/O sub-system. Can I get enough information in and out of that system to feed those CPU's? This is not between the memory and the main processor, but this is basically I/O in and out. A lot of work in the database area goes into this. You may be surprised to see that some of these benchmarks that we run, TPC at the very top, the high-end machines, we're running as many as 5,000 to 6,000 disk drives on those to get rid of the latency of the spinning disk and to get a CPU balance so you can really tune it. So these are the types of things you're worrying about there -- tuning that I/O subsystem and really getting the full performance out of it.

Today, I want to focus primarily on giving you an update of where we're at on our support for parallelization and some of the techniques we're looking at and working and actually delivering here. So if you think about it, this is where you take the application and you split it up so that it can take advantage of those multiple cores. So you should think of the core as the CPU's set of elements, registers, and that you run the applications on, ALU, and you should think of the thread as the part of the code that's running on a given core. So that terminology is a little bit loose, but that's conceptually what you have.

So what we want to do is speed up that application when it's running on those multiple cores. Now there are a couple of things that you need to look at here. First of all, we introduced hyper-threading, and hyper-threading basically was the first concept where you're actually duplicating a set of registers, but you're not duplicating the full chip. And you probably ask, "Well, why did we do that?" That's basically an easy way, an extra cost to do that, somewhere 5 to 10 percent in transistors, and is fairly low in power. So, that gave us -- keeps the power down, get performance, and we're seeing now, typically, we're after using that, somewhere in the range of 30 percent performance gain. It's the sharing of the cache, the sharing of some of the registers, ALU's, that limits that.

Now as we go to multiple core on a die, you've got the complete chip reduplicated, as opposed to just the register sets. This case you should expect the parallelism to be basically somewhere, theoretically, 1x for each additional core. So two cores should be 2x. Practically, you don't get that kind of performance, because you still have the memory and you've got the bandwidth of the bus coming from memory.

When you think about threading, there's two ways of doing it. You can think of dividing up a task into subtasks and farming those out, or basically adding new

functionality to the application. So, now if you're like me at home, your wife says, "You go do the dishes, and I'll mop the floors." That's two tasks, okay? They're independent. I can do mine when I get done watching TV; she can do hers before she goes to bed. One's not dependent on the other. So that's the way you want to think about it. Now if it's creating new capability, you're trying to bring innovation, then those tasks are different. It's "you go build a new addition to the house while I do the normal work." And you're bringing new capability and functionality there. So that's a way.

When you think about threading, though, really the problem is somewhat different. And the best way to look at is from a data viewpoint. I have this amount of data. I have to do work on that data. How do I divide my task to get that work done? We're going to show you some demos in a little bit. An example would be a video codec. Suppose I'm writing a video codec. I can break it up into blocks for each frame, and I can have a thread work on each block. Or, another way of dividing the data up is, I do frame one, you do frame two. I do frame three, you do frame four. You split it that way. So those are ways.

But when you think about it, you really want to think about taking the data, splitting it up. Now the problem when you do that is you can have data dependencies on the boundary. When I break it up into blocks, how about that little line that comes down the boundary? Who writes that? Do you write it, or do I write it? And there could be different results on that line. So how do you handle the data dependency areas, is where you have to spend a lot of time and thought. So that's one technique for threading. That's usually the way that you want to think about it if you're writing new code.

Another is functional. So this is like I mentioned where we split up the tasks into two independent things. And a lot of existing applications that you're doing, this approach to threading is where you end up, because you don't want to rewrite the full application.

So let me look a little bit at a couple examples. If I'm writing to the native threads of the operating system, or even a POSIX package of thread user level, there's a lot of replication I have to do. I have to start off by declaring the threads. I have to then create the threads. I have to then begin managing those threads and what they're going to do. So a lot of repetitious work. And the programmer, in his own head, has to keep track of this data does have -- or these two tasks have data dependencies and I've got to lock that data so that one guy doesn't change the value until it's correct, you know, if you're both trying to write at the same time, that's called a race condition.

Now there's one thing in this program I don't like. They hard-coded the number of threads in there. You see up at the top, constant integer, number of threads equals four. We really encourage the ISVs, don't do that. You basically want to use CPU ID, ask how many threads are available in the system, and then use that many dynamically. That way, as you go to the next-generation processor that we bring out in six months or eight months, you automatically can take advantage of those threads, as opposed to being locked in.

So, that's traditional, a lot of handwork, a lot of keeping in your own head the dependencies and worrying about it. Why can't the compiler do this for you? It's meant to do the busywork. It's meant to check on those data dependencies. So, the holy grail of compilers for the last 25 years -- and Dave Kuck is probably in the audience; he's the father of this -- was to produce automatic parallelization in the compilers. So what you want to think about here is [having] the compiler take that data, it figure out how to split it up, and then it looks for data dependencies, and it really then does all the work for you. That's done. There are compilers out there, including our own, that have auto-parallelization. Twenty-five years of research has improved it. You can get scalability that reasonable. Some applications have been rewritten to help that even do better. And we continue to invest in that area.

Now a lot of what we've done there, the infrastructure we've built, can be helped if the programmer gives some hints to the compiler. And that's what OpenMP is all about. It's an industry standard. It's in our compilers; it's in SGIs compilers; it's in Sun's compilers; it's in IBM's compilers; Microsoft's adding it to their compilers; it's in GCC. So it's an open standard that everybody is working together. [redacted] Now, what it's telling the compiler is, "Here is the section of code that you can parallelize." And now what the compiler does is automatically figures out how to do a transformation on those loops, change the order around. So in this case you see, changes the order around so that you're operating on a column of that vector at a time. And it creates a thread for that column. So that's what OpenMP is all about. Let the compiler do as much of the busywork as possible, but the programmer gives it some hints. This works very well for loopy code that has a lot of loops and that the data-dependencies, you can analyze it and figure it out. And if you know you're going to use this, you can write code better – even to make it better.

Now our compilers today hold the record. On SGI, I think it's 128-way SGI now, and we took SPEC – and there's a version the industry has of SPEC for OpenMP where it has these pragmas in it and how much parallelization are you getting out of it. So most of the scientific code, a lot of the multimedia code, most of our libraries we produce utilize this and try to get as much parallelism out of it as possible.

What I want to talk about next is where we're at on our tools and what's available. I talked to you last February and we had a number of things then in a beta program and now they've moved forward and they're actually moving into product. So the first new things are some basic building blocks. Now, this approach I'll go into some detail about, but at a high level, these are templates -- C++ templates -- and some existing algorithms that we've parallelized that allow you to write your code using our templates and that does a lot of the busy work for you.

The other the thread-checker, thread-profiler. These now are both up to version 3.0. We've gotten a lot of feedback from you on your applications and we've made improvements. Now one thing is, I pushed [on] my own compiler team, "Why don't you parallelize your compiler? Why don't you try to use your tools to parallelize the compiler?" "Bad idea. Why would I ever want to do that?" "No. Go do it." So they did.

Guess what? It started off with somewhere in the range of 300,000 violations. Now, a violation is a data-violation; it's where you're trying to parallelize and you have those dependencies that you haven't fixed. It could be through shared variables; it could be through global variables. So they went through and began working on those. And now we have used this tool to get rid of all those violations. And we've got the basic part of the compiler now parallelized so that it will take large programs, break out functions, do those all in parallel, and begin getting some speedups by taking advantage of multiple cores at once to do that.

So we're trying to eat our own dog food here. We've got our AEs using them, feeding back improvements as they work with you. We've got our own teams trying to use them on threading some of our own applications; seeing some good results. Those are now both on Linux and Windows and some of these tools we are also moving to Apple. We'll talk about [that].

The third place -- we've done a lot of work on libraries. We can give you these heavy math libraries that are highly tuned, not only for the micro-architecture, but threaded; allows you to get your application to market sooner. We now are extending that to include XML libraries. These libraries are in progress. We're beginning beta. We're at the stage we would love your feedback. You've got a chance to influence us. We're trying to follow industry standards, [JAXBE], other standard Open Source libraries, so that you can just have a drop-in replacement for those interfaces.

We'll come back at the end and talk about those. So if you think about the entire tool set, here's the way we would like to think of the process to tune your application. First thing, go through and run VTune and the thread profiler and analyze your program and it's match to the architecture. Then you go through and there's those tools, and the version number tells you our latest shipping version, so if you're using them, you know whether you're on the latest version or not. Then you go through and introduce your threads.

Now when you introduce your threads, you probably aren't going to keep all those data dependencies in your head. So you're going to expect some programming errors. The next step then, comes along and helps you check those, the confidence, correctness, the thread checker. It helps you to see if you do have those data dependencies, that you need to put locks around, or if you have race conditions and you fix those. And then you come back after you've reduced all those problems, and you start working on optimizing and tuning, much like I talked about there, the system level, the architecture level, and the micro-architecture level.

So with that, that's the compilers and the tools that are available now. What I'd like to do is go into a little bit more depth on the thread building blocks. These are new. We've gotten some very good feedback on these, and we're looking at innovative ways to get more people using them, working together.

Now the concept here comes from a concept in C++. C++ handles what's called templates, and there are already some classes out there that take advantage and give you templates to build your applications. So things like caching, sorting, those kinds of things, are standard application templates that you can utilize. And those have been around as long as C++ and they're classes that go along with C++, and they've been standardized through the committees.

So what we said is, "Why don't we take the same concept, build templates, but go in and parallelize a lot of algorithms and provide lower-level capability to reduce your busywork." Why should a programmer have to initialize threads? Why should a programmer have to do a lot of management functions? Can we raise up to a higher-level capability much like OpenMP does, hence to the compiler, things like parallel for, reduce or scan, these kind of operations that you want to do in parallel. So, we've done that. [Parameter] is also for locks in that.

And then we took a number of standard algorithms and we parallelized them, so that we even provide that level. So let me give you an example here around the sort, and it's a quick sort. If you think about it, the concept I said, take your data and focus on parallelizing it and getting multiple tasks to work on that data. So the first thread in this goes through, picks a random spot, and does a quick sort on that random spot.

Now if you're not a programmer, it's pretty simple. I go through, I've got a random spot -- it was pretty close to middle; that's good enough. And that number turns out to be 37 [blade], yes. Now, once you have that, that first thread then, starts walking the list, serially. Is the number bigger than 37 or less than 37? If it's less than 37, it leaves it in the bucket it's in, your left side; if it's bigger than 37, it just throws it in the other bucket. So you've got an initial bucket of those two.

Now the interesting thing is, the algorithm divides the work and conquers. Now it has a second list it can sort, the yellow. So it spawns off another thread automatically and it says repeat the same process on that thread, or on that data. So, it goes through, finds a random spot and walks the list. If it's less than that number, it keeps on your left; if it's bigger, it throws it in the other side. And it just repeats this process down through, dividing the data and creating more threads.

Now, because we wrote the templates, we've put the busywork in to find out how many threads, how many cores are available and making it utilize all those cores that are available. So you don't have to do that; it does it for you and keeps track of that. The other thing is, we've built a scheduler in and basically, you have a pool of work to get done and you have cores that are available. So what the scheduler does is, it goes and grabs from the pool of cores, a core, assigns it to a thread, do some work. And you're going to see that concept used quite frequently in threading. And that's all taken care of for you. So you're able to utilize this algorithm on your sort with different data, and it will effectively go through. Using these templates will do that for you. So, the process is on through until it's actually done.

So the whole concept of building blocks is built around this kind of concept. Provide templates for algorithms as well as some of the management-type functions and it allows you to go through fairly quickly. We've seen, in a lot of the applications that we've actually written, a reduction -- about a fourth as much code by the programmer needs to be written as if they do everything by hand by themselves. Also, the other advantage is the correctness when it's done. Somebody else is worried about that for you, as opposed to you having to go through.

So, I'd like to invite Charlotte Lawrence up to the stage and actually give you a visual demo of this on another system. Welcome, Charlotte.

Charlotte Lawrence: Good morning, Richard. I am so excited about our new product, Intel's Threading Building Blocks for C++. As Richard said earlier, it's a template-based set of libraries for multi-core performance and scalability development efforts.

So what we're going to do for this demonstration this morning is, I'm going to take the Tachyon application. And I have the native serial version. And, of course, I have a threaded version built using the -- what else? -- Intel Threading Building Blocks for C++. So, let me go ahead and get the serial version launched right now.

Richard Wirt: Can you tell us how many processors in each of these now you're going to use as you go through?

Charlotte Lawrence: Well, okay, the system that we're running this demonstration on is XeonMP dual-core platform. And this one has four processors in it, totaling eight cores.

Richard Wirt: Okay.

Charlotte Lawrence: So let me go ahead and launch the serial version, as I said earlier.

Richard Wirt: So being serial, this one's effectively only on one of those cores. So you can see the time on that.

Charlotte Lawrence: And, you know, when you look at this rendering, it's a very familiar rendering. Now let me launch the threaded version. Wow! Wasn't that just awesome?

Richard Wirt: Now I noticed there are a lot of black spots.

Charlotte Lawrence: Yeah. So that is actually what Richard spoke of earlier where you have the blocked rendering. And each of one those are the data blocks that Richard spoke of.

Richard Wirt: Effectively then, you've created blocks. And you noticed they were quite random in their pattern, so the library figured out, obviously, some stuff there, and

assigned a thread to a block. And that block processed that data, and it did it in parallel, so you're breaking up a lot there.

Charlotte Lawrence: I had to just run it again. This stuff makes me really giddy. It is so neat. And, you know, Richard did say that what it does is it just goes ahead and does great performance. It scales well, but I think the really great news here is for the programmers. And, again, without going too much into detail, because, you know, Richard already covered it, is that you can actually deliver and achieve this application, and have the performance and scalability with much less code, definitely. We're looking at three-quarters less code using the Intel C++ Thread Builder than the native code threading.

Richard Wirt: Good. Understand you've got another demo. What's that?

Charlotte Lawrence: Yeah, you were speaking about the Intel C++ compiler. Let's get over here. Now we talked about this as a beta a year ago was it?

Richard Wirt: That's a silver notebook. I kind of remember most of our notebooks are black. What's that?

Charlotte Lawrence: Yeah, this is pretty hot. This is your Apple MacBook Pro, which is your Core Duo-based processor. What I'm going to show on this demonstration is, I'm going to use an Open Source, 3D ray-tracer application. And it's a binary that I built using the Intel C++ compiler. I also built a second binary using the GCC compiler. And I want all of you to know that I did use the best optimization that was available for both binaries. So let me go ahead and get this one launched.

Richard Wirt: Now, those of you who are not familiar with ray tracers, this is another way to give a 3D effect to a scene. And most 3D effects are through what we call rasterization, and it's going through and creating little triangles and coloring the triangles from the model. A ray tracer actually takes a ray of light and traces it through the scene. Now, that's a good example where, because it's a ray of light, you can have a single thread do a single ray. So it's very natural to parallelize that type of application.

Charlotte Lawrence: So as you view the rendering, you notice that the Intel binary, the binary that we built using the Intel compiler outperformed the GCC version, and we are looking at, I would say, easily a 30 percent performance advantage there. And I think that's just great.

Richard Wirt: So, thank you.

Charlotte Lawrence: All right. Thank you.

Richard Wirt: Thank you, Charlotte.

Charlotte Lawrence: So much.

[Applause]

Richard Wirt: This shows you some of the work that we've done to tune to that microarchitecture and really get the most performance out of that as we can. We also have people who are working on GCC and feeding back information there to keep it highly tuned.

So we've talked a lot about threading and how to go about it. The problem is most of you have existing applications and you want to take advantage of those cores. So one way to do this is through some of the third-party products that are available in the market. So I'd like to invite Geva Perry up -- I'm ahead of myself, just a moment.

One other library I did want to talk about is the XML, very quickly. So, much like the math libraries, we've threaded these libraries, and we're in the process of getting into final product. Now, in order to help you take advantage of these, we've tried to maintain as many of the industry-standard library interfaces for XML that are out there. So, Java AXP is a good example of an interface to Java for XML and allows you to process it.

Some of the results we're seeing -- these will be available over the course of the next six months to a year; they're in design now, as well as some of the early ones are in beta. Some of the results we're seeing on these benchmarks show you the 3x to 5x performance gain. Now, one of the things that we're after here is really beginning to establish these so we can put hardware under them. Most of you are aware we've bought a couple XML companies that are working in this space. This work is part of the Sarvega team that came on board. Also have a hardware team that we're beginning to really analyze how to speed up the hardware parsing, because a lot of data handling now is through XML.

So, I started to say, I wanted to welcome Geva Perry. Geva is an ISV from GigaSpaces. And basically they're focusing on existing applications, how you can take those and take advantage of the threads. Welcome Geva.

Geva Perry: Thank you, Richard. [Applause] Thank you. Thanks for the opportunity to let me speak here. We at GigaSpaces takes a somewhat different approach. It's a middleware approach that allows you, basically, to benefit from parallelization, whether within a multi-core, multi-CPU machine and across machines, without having to explicitly program for threading or for any aspect of the parallelization. We do it with an approach called space-based architecture.

So for those who aren't familiar with space-based architecture, I think I should start by defining what is the space in this context. That term applies to a distributed shared memory space. So I allocate a memory address space within a RAM of a machine, and I allow different processes and threads, whether they're local on the same box or remote, to share that memory space. So in other words, I can have one process, write an object into that memory space, another one read it, and essentially they share it. And I

can cluster that memory space for high availability and for partitioning and load balancing.

Now how do I talk to this memory address space? And this is a key point. It's via standard API's, whether it's J2E type API's, like JDBC, so I can do SQL queries on this memory space, or JMS, or C++ objects, C-sharp objects, Java objects.

So, given that concept of a space, now what is a space-based architecture? So, this sort of gives you an example, you know, the orange rectangles at the bottom are basically boxes. A couple of them are multi-core boxes. One of them is a single-core. It's called multi-CPU, but the concept is the same. Essentially what I do is, I write objects as they come in and what are these objects? It's things that need to get done, whether it's data that needs to be processed, service requests that need to responded to, messages that need to be responded to, and so on. I write them into this clustered distributed memory space. And essentially it then, by itself, launches what we call worker threads. What is a worker thread? Well, a worker thread is your business logic, your code, that you wrote as if you're writing a single box, single CPU, single core, and it is launched as threads and can dynamically grow and shrink based on service-level agreements that you require. And those are the little orange rectangles at the bottom; those are the worker threads that basically work dynamically.

So just to give an example of what the effect of this is from a traditional approach to how you would build an application. In this case it's for a very large global bank, a foreign exchange trading application. So the basic logic of this application is, if I want to buy a million yen and sell for pounds, it has to match, find another trader that wants to do the opposite. In real life, it would probably be more complex, because it would have to find combinations in order to do those tradings. So it's a matching agent, for those familiar with financial services systems. Essentially it has to go through three things. It has to validate the order as it comes. Am I a legitimate customer? Is the order prepared okay, etcetera, etcetera. Then it has to do the actual matching logic. And then eventually you route it to different systems to execute the orders.

The way, traditionally, people on Wall Street and other places have built this is, as the orders come in from the clients, there is some kind of a validation logic that goes on in order to scale. What you would do is you would use a large, expensive SMP box and have to write proprietary code, etcetera. And then you probably want to write it into a database and that's how you maintain the state of the transaction. And then you would have another expensive box that needs to do the matching logic and work with the database. So there's a lot of latency there, inefficiency, complexity, and so on. And then eventually it will do the routing.

With this kind of parallel approach, I can do two things. One is I can have these three pieces of logic that are part of a sequence of a single transaction happen as thread and share that memory, share that data in memory so I get very fast performance. And it maintains sequence of the transaction. But what I can do now is I can parallelize that transaction, so I can have many of these transactions happening simultaneously because

there's no dependency among them, as Richard was talking about before. There is no challenge there; each of them is independent. So I can do them across -- it views across cores and across boxes.

This is not science fiction. This is being used today in mission-critical applications in Wall Street and other financial services areas, in TelCos, in government agencies, and so on. It's real and it's happening now, and you can see some of the big customer names that I'm allowed to say who are using it.

So, basically to sum up, some of the key points about this is one, it allows you a way to parallelize your applications and run multiple threads and benefit from the advantages of a multi-core system without having to explicitly code for it. You get huge latency benefits because everything is talking to memory locally. It's very simple, I mean, you use standard APIs, either for your existing application or if you're building a new one. And it's a different, much more simple, elegant model.

Richard Wirt: Thank you, Geva. It looks like a very interesting software that you have there that takes advantage of existing applications today and helps them get parallel very quickly. Thank you.

Geva Perry: Thank you, Richard.

[Applause]

Richard Wirt: So in summary, I think you see that Intel is working with the industry. We've made good progress, as we move forward, at getting more applications threaded. We're providing a very rich ecosystem of tools, libraries around this, and we'll continue to do that.

Parallel, I wanted to give you a quick update. We're continuing to work with the universities -- we've got 50, 60 of them now we're working with worldwide -- to begin to impact their curriculum to teach more about threading and take advantage of multi-core architectures and this shift in the industry and trending. In parallel, we have our own training programs through what we call Software College. We put all our AEs through it and we're now in the process of going out and making that material available to universities as well as software parks, or within companies if you want training in-house.

I think you can see that ISVs and end users are now significantly engaged. A lot of new innovation is happening around this. We're very excited about the trend we're seeing in the industry and the capability that Intel is doing to continue Moore's law in going to more cores per die and the fact that the ISVs are responding, bringing innovation to those, as well as scaling their applications.