PROFESSOR: So, what I'll talk about here is how to actually understand the performance of your application, and what are some of the things you can do to actually improve your performance. You're going to hear more about automated optimizations, compile the optimizations on Monday. There will be two talks on that. You'll get some cell-specific optimizations that you can do, so some cell-specific tricks on Tuesday in the recitation.

So here, this is meant to be a more general purpose talk on how can you debug performance anomalies and performance problems. Then what are some way that you can actually improve the performance. Where do you look after you've done your parallelization.

So just to review the key concepts to parallelism. Coverage -- how much parallelism do you have in your application. All of you know, because you all had perfect scores on the last quiz. So that means if you take a look at your program, you find the parallel parts and that tells you how much parallelism that you have. If you don't have more than a certain fraction, there's really nothing else you can do with parallelism. So the rest of the talk will help you address the question of well, where do you go for last frontier.

The granularity. We talked about how the granularity of your work and how much work are you doing on each processor affects your load balancing, and how it actually affects your communication costs. If you have a lot of things colocate on a single processor, then you don't have to do a whole lot of communication across processors, but if you distribute things at a finer level, then you're doing a whole lot of communication. So we'll look at the communication costs again and some tricks that you can apply to optimize that.

Then the last thing that we had talked about in one of the previous lectures is locality, in locality of communication versus computation, and both of those are critical. So we'll have some examples of that.

So just to review the communication cost model, so I had flashed up on the screen a while ago this equation that captures all the factors that go into figuring out how expensive is it to actually send data from one processor to the other. Or this could even apply on a single machine where a processor is talking to the memory -- you know, loads and stores. The same cost model really applies there. If you look at how processors -- a uniprocessor tries to improve communication, and some of the things we've mentioned really early on in the course for improving communication costs, what we focused on is this overlap.

There are things you can do, for example, sending fewer messages, optimizing how you're packing data into your messages, reducing the cost of the network in terms of the latency, using architecture support, increasing the bandwidth and so on. But really the biggest impact that you can get is really just from overlap because you have direct control over that, especially in parallel programming.

So let's look at a small review -- you know, what did it mean to overlap. So we had some synchronization point or some point in the execution and then we get data. Then once the data's arrived, we compute on that data. So this could be a uniprocessor. A CPU issues a load, it goes out to memory, memory sends back the data, and then the CU can continue operating. But the uniprocessors can pipeline. They allow you to have multiple loads going out to memory. So you can get the effect of hiding over overlapping a lot of that communication latency.

But there are limits to the communication, to the pipelining effects. If the work that you're doing is really equal to the amount of data that you're fetching, then you have really good overlap. So we went over this in the recitation and we showed you an example of where pipelining doesn't have any performance effects and so you might not want to do it because it doesn't give you the performance bang for the complexity you invest in it. So, if things are really nicely matched you get good overlap, here you only get god overlap -- sorry, these, for some reason, should be shifted over one. So where else do you look for performance?

So there are two kinds of communication. There's inherent communication and your algorithm, and this is a result of how you actually partition your data and how you partitioned your computation. Then there's artifacts that come up because of the way you actually do the implementation and how you map it to the architecture. So, if you have poor distribution of data across memory, then you might unnecessarily end up fetching data that you don't need. So, you might also have redundant data fetchers. So let's talk about that in more detail.

The way I'm going to do this is to draw from wisdom in uniprocessors. So in uniprocessors, CPUs communicate with memory, and really conceptually, I think that's no different than multiple processors talking to multiple processors. It's really all about where the data is flowing and how the memories are structured. So, loads and stores are the uniprocessor as what and what are to distributed memory. So if you think of Cell, what would go in those two blanks? Can you get this? I heard the answer there. You get input. So, DMA get and DMA put. That's really just the load and a store. It's just doing, instead of loading one particular data element, you're loading a whole chunk of memory.

So, on a uniprocessor, how do you overlap communication? Well, architecture, the memory system is designed in a way to exploit two properties that have been observed in computation. Spacial locality and temporal locality, and I'll look at each one separately.

So in spacial locality, CPU asks for a data address of 1,000. What the memory does, it'll send data address of

1,000, plus a whole bunch of other data that's neighboring to it, so 1,000 to 1,064. Really, how much data you actually send, you know, what is the granularity of communication depends on architectural parameters. So in common architecture it's really the block side. So if you have a cache where the organization says you have a block side to 32 words, 32 bytes, then this is how much you transfer from main memory to the caches. So this works well when the CPU actually uses that data. If I send you 64 data bytes and I only use one of them, then what have I done? I've wasted bandwidth. Plus, I need to store all that extra data in the cache so I've wasted my cache capacity. So that's bad and you want to avoid it.

Temporal locality is a clustering of references in time. So if you access some particular data element, then what the memory assumes is you're going reuse that data over and over and over again, so it stores it in the cache. So your memory hierarchy has the main memory at the top level, and that's your slowest memory but the biggest capacity. Then as you get closer and closer to the processor, you end up with smaller caches -- these are local, smaller storage, but they're faster. So, if you reuse a data element then it gets cached at the lowest data level, and so the assumption there is that you're gonna reuse it over and over and over again. If you do that, then what you've done is you've amortized the cost of bringing in that data over many, many references.

So that works out really well. But if you don't reuse that particular data elements over and over again, then you've wasted cache capacity. You still need to fetch the data because the CPU asks for it, but you might not have had the cache, so that would have created more space in your cache to have something else in there that might have been more useful.

So in the multiprocessor case, how do you reduce these artifactual costs in communication. So, DCMA gets inputs on the cell, or just in just message passing, you're exchanging messages. Typically, you're communicating over a course or large blocks of data. What you're usually getting is a continuous chunk of memory, although you could do some things in software or in hardware to gather data from different memory locations and pack them into contiguous locations. The reason you pack them into contiguous locations again to export spatial locality when you store the data locally.

So to exploit the spatial locality characteristics, what you want to make sure is that you actually are going to have good spatial locality in your actual computation. So you want things that are iterating over loops with well-defined indices with indices that go over very short ranges, or they're very sequential or have fixed striped patterns where you're not wasting a lot of the data that you have brought in. Otherwise, you have to essentially just increase your communication because every fetch is getting you only a small fraction of what you actually need. So, intuitively this should make sense.

Temporal locality just says I brought in some data and so I want to maximize this utility. So if I have any computation in a parallel system, I might be able to reorder my tasks in a way that I have explicit control over the scheduling -- which stripe executes when. Then you want to make sure that all the computation that needs that particular data happens adjacent in time or in some short time window so that you can amortize the cost.

Are those two concepts clear? Any questions on this?

So, you've done all of that. You've parallelized your code, you've taken care of your communication costs, you've tried to reduce it as much as possible. Where else can you look for performance -- things just don't look like they're performing as well as they could? So, the last frontier is perhaps single thread performance, so I'm going to talk about that.

So what is really a single thread. So if you think of what you're doing with parallel programming, you're taking a bunch of tasks -- this is the work that you have to do -- and you group them together into threads or the equivalent of threads, and some threads will run on individual cores. So essentially you have one thread running on a core, and if that performance goes fast, then your overall execution can also benefit from that. So, that's single thread performance.

So if you look at a timeline, here you have sequential code going on, then we hit some parallel part of the computation. We have multiple executions going on. Each one of these is a thread of execution. Really, my finish line depends on who's the longest thread, who's the slowest one to complete, and that's going to essentially control my speed up. So I can improve this by doing better load balancing. If I distribute the work [? so that ?] everybody's doing equivalent amount of work, then I can shift that finish line earlier in time. That can work reasonably well. So we talked about load balancing before. We can also make execution on each processor faster. If each one of these threads finishes faster or I've done the load balancing and now I can even squeeze out more performance by shrinking each one of those lines, then I can get performance improvement there as well. So that's improving single thread performance.

But how do we actually understand what's going on? How do I know where to optimize? How do I know how long each thread is taking? How do I know how long my program is taking? Where are the problems? So, there are performance monitoring tools that hard designed to help you do that. So what's the most coarse-grained way of figuring out how long your program took? You have some sample piece of code shown over here, you might compile it, and then you might just use time -- standard units command say run this program and tell me how much time it took to run.

So you get some alpha back from time that said you took about two seconds of user time, this is actual code, you took some small amount of time in system code, and this is your overall execution, this is how much of the processor you actually use. So, a 95% utilization. Then you might apply some optimization. So here we'll use the compiler, we'll change the optimization level, compile the same code, run it, and we'll see wow, performance improved. So we increased 99% utilization, my running time went down by a small chunk.

But did we really learn anything about what's going on here? There's some code going on, there's a loop here, there's a loop here, there's some functions with more loops. So where is the actual computation time going? So how would I actually go about understanding this? So what are some tricks you might have used in trying to figure out how long something took in your computation?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: Right. So you might have a timer, you record the time here, you compute and then you stop the timer and then you might printout or record how long that particular block of code took. Then you might have a histogram of them and then you might analyze the histogram to find out the distribution. You might repeat this over and over again for many different loops or many different parts that are code. If you have a preconceived notion of where the problem is then you instrument that and see if your hypothesis is correct. That can help you identify the problems. But increasingly you can actually get more accurate measurements. So, in the previous routine, you're using the time, you were looking at how much time has elapsed in seconds or in small increments. But you can actually use hardware counters today to actually measure clock cycles, clock ticks. That might be more useful. Actually, it's more useful because you can measure a lot more events than just clock ticks.

The counters in modern architectures are really specialized registers that count up events and then you can go in there and probe and ask what is the value in this register, and you can use that as part of your performance tuning. You use them much in the same way as you would have done to start a regular timer or stop a regular timer. There are specialized libraries that run. Unfortunately, these are very architecture-specific at this point. There's not really a common standard that says grab a timer at each different architecture in a uniform way, although that's getting better with some standards coming out from--. I'll talk about that in just a few slides. You can use this to, for example, measure your communication to computation cost. So you can wrap your DMA get and DMA put by timer calls, and you can measure your actual work by timer calls, and figuring out how much overlap can you get from overlap in communication and communication computation and is that really worthwhile to do pipelining.

But this really requires manual changes to code. You have to go in there and start the timers. You have to have maybe an idea of where the problem is, and you have the Heisenberg effect. If you have a loop and you want to measure code within the loop because you have a nested loop inside of that, then now you're effecting the performance of the outer loop. That can be problematic. So it has a better effect because you can't really make an accurate measurement on the thing you're inspecting.

So there's a slightly better approach, dynamic profiling. Dynamic profiling is really there's an event-based profiling and time-based profiling. Conceptually they do the same thing. What's going on here is your program is running and you're going to say I'm interested in events such as cache misses. Whenever n number of cache misses happen, let's say 1,000, let me know. So you get an interrupt whenever 1,000 cache misses happen. Then you can update a counter or use that to trigger some optimizations or analysis. This works really nicely because you don't have to touch your code.

You essentially run your program as you normally do with just one modification that includes running the dynamic profiler, as well as your actual computation. As far as multiple languages because all it does is just takes your binary so you can program in any language, any programming model. It's quite efficient to actually use these dynamic profiling tools. The sampling frequencies are reasonably small, you can make them reasonably small and still have it be efficient.

So some counter examples. Clock cycles, so you can measure clock ticks. Pipeline stalls. This might be interesting if you want to optimize your instruction schedule -- you'll actually see this in the recitation next week. Cache hits, cache misses -- you can get an idea of how bad your cache performance is and how much time you're spending in the memory system. Number of instructions, loads, stores, floating point ops and so on. Then you can derive some useful measures from that. So I can get an idea of processor utilization -- divide cycles by time and that gives me utilization. I can derive some other things and maybe some of the more interesting things like memory of traffic. So how much data am I actually sending between a CPU and a processor, or how much data am I communicating from one processor to the other. So I can just grab the counters for number of loads and number of stores, figure out what the cache line size is -- usually those are documented or there are calibration tools you can run to get that value, and you figure out memory of traffic.

Another one would be bandwidth consumed. So bandwidth is memory of traffic per second. So how would you measure that? It's just the traffic divided by the wall clock time. There's some others that you can calculate. So these can be really useful in helping you figure out where are the things you should go focus in on. I'm going to show you some examples.

The way these tools work is you have your application source code, you compile it down to a binary. You take your binary and you run it, and then that can generate some profile that stores locally on your disk. Then you can take that profile and analyze it by some sort of interpreter, and some cases you can actually analyze the binary as well, and reannotate your source code. That can actually be very useful because it'll tell you this particular line of your code is the one where you're spending most of your time computing.

So some tools -- have any of you used these tools? Anybody use Gprof, for example? Good. So, you might have an idea of how these could be used. There are others. HPCToolkit, which I commonly use from Rice. Pappy is very common because it has a very nice interface for grabbing all kinds of counters. VTune from Intel, and there's others that work in different ways and so there are binary instrumenters that do the same things and do it slightly more efficiently and actually give you the ability to compile your code at run time and optimize it, taking advantage of the profiling information you've collected.

So here's a sample of running Gprof. Gprof should be available on any Linux system, it's even available on Cygwin, if you see Cygwin. I've compiled some code -- this is MPEG 2D code, a reference implementation. I specify some parameters to run it. Here I add this dash Rflag which says use a particular kind of inverse DCDT that's a floating point precise that uses double precision for the floating point computations in DCT -- inverse DCT rather. So you can see actually where most of the time is being spent in the computation. So here's a time per function, so each row represents a function. So this is the percent of the time, this is the actual time in seconds, how many times I actually called this function, and some other useful things. So the second function that's used here that happens is MPEG intrablock decoding and here you're doing some spatial decomposition, restoring spatial pictures by 5%. So if you were optimize this particular code, where would you go look? You would look in the reference DCT.

So, MPEG has two versions of DCT -- one that uses floating point, another that just uses some numerical tricks to operate over integers for a loss of precision, but they find that acceptable as part of this application. So you omit the Rflag, it actually uses a different function for doing the DCT. Now you see the distribution of where the time is spent in your computation changes. Now there's a new function that's become the bottleneck and it's called Form Component Prediction. Then IDCT column, which actually is the main replacement of the previous code, this one, is now about 1/3 of the actual computation.

So, this could be useful because you can Gprof your application, figure out where the bottlenecks are in terms of performance, and you might go in there and tweak the algorithm completely. You might go in there and sort of look at some problems that might be implementation bugs or performance bugs and be able to fix those.

Any questions on that?

You can do sort of more accurate things. So, Gprof largely uses one mechanism, HPCToolkit uses the performance counters to actually give you more finer grade measurements if you want them. So, in the HPCToolkit, you run your program in the same way. You have MPEG 2D code, dash dash just says this is where the parameters are to impact 2D code following that dash dash, and you can add some parameters in there that say these are counters I'm interested in measuring. So the first one is total cycles. The second one is the L1, so primary cache load misses. Then you might want to count the floating point instructions and the total instructions.

As you run your program you actually get a profiling output, and then you can process that file and it'll spit out some summaries for you. So it'll tell you this is the total number of cycles, 698 samples with this frequency. So if you multiply the two together, you an idea of how many cycles your computation took. How many load misses? So it's 27 samples at this frequency.

So remember what's going on here is the counter is measuring events, and when the event reaches a particular threshold it let's you know. So here the sampling threshold is 32,000. So whenever 32,000 floating point instructions occur you get a sample. So you're just counting how many interrupts you're getting or how many samples. So you multiply the two together you can get the final counts.

It can do things like Gprof, it'll tell you where your time is and where you spent most of your time. Actually breaks it down into your module. So, MPEG calls some standard libraries, libsi. I could break it down by functions, break it down by line number. You can even annotate your source code. So here's just a simple example that I used earlier, and each one of these columns represent one of the metrics that we measured, and you can see most of my time is spent here, 36% at this particular statement. So that can be very useful. You can go in there and say, I want to do some [? dization ?], I can maybe reduce this overhead in some way to get better performance.

Any questions on that? Yup.

AUDIENCE: [INAUDIBLE PHRASE]?

PROFESSOR: I don't know. Unfortunately, I don't know the answer to that.

There's some nice gooies for some of these tools. So VTune has a nice interface. I use HPCViewer. I use HPCToolkit, which provides HPCViewer, so I just grab the screenshot from one of the tutorials on this. You have your source code. It shows you some of the same information I had on a previous slide, but in a nicer graphical format.

So, now I have all this information, how do I actually improve the performance? Well, if you look at what is the performance time on a uniprocessor, it's time spent computing plus the time spent waiting for data or waiting for some other things to complete. You have instructional level parallels, which is really critical for uniprocessors, and architect that sort of spent massive amounts of effort in providing multiple functional units, deeply pipeline the instruction pipeline. Doing things like speculation, prediction to keep that instructional level of parallelism number high so you can get really good performance.

You can do things like looking at the assembly code and re-ordering instructions to avoid instruction hazards in the pipeline. You might look at a register allocation. But that's really very low-hanging fruit. You have to reach really high to grab that kind of fruit. You'll actually, unfortunately, get the experience that is part of the next recitation. So apologies in advance. But you'll see that -- well I'm not going to talk about that. Instead I'm going to focus about some things that are perhaps lower-hanging fruit.

So data level parallelism. So we've used SIMD in some of recitations. I'm giving you a short example of that. Here, I'm going to talk about how you actually get data level parallelism or how do you actually find the SIMD in your computation so you can get that added advantage. Some nice things about data level parallelism in the form of short vector instructions is that the harder really becomes simpler. You issue one instruction and that same instruction operates over multiple data elements and you get better instruction bandwidth. I just have to fetch one instruction and if my vector lens is 10, than that effectively does 10 instructions for me. The architecture can get simpler, reduces the complexity. So it has some nice advantages.

The thing to go after is the memory hierarchy. This is because of that speed gap that we showed earlier on in the course between memory speed and processor speed, and if you optimize a performance usually it's like 1% performance and your cache registry gives you some significant performance improvement in your overall execution. So you want to go after that because that's the biggest beast in the room.

A brief overview of SIMD and then some detailed examples as to how you actually go about extracting short vector instructions.

So, here we have an example of scaleacode. We're iterating in a loop from zero to n, and we're just adding some array elements a to b and storing results in c. So in the scaler mode, we just have one add, each value of a and b is one register. We add those together, we write the value to a separate register. In the vector mode, we can pack multiple data elements, so here let's assume our vector lens is four, I can pack four of these data values into one vector register. You can pack four of these data elements into another vector register, and now my single vector instruction has the effect of doing four ads at th same time, and it can store results into four elements of c.

Any questions on that?

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: No. We'll get to that.

So, let's look at those to sort of give you a more lower level feel for this. Same code, I've just shown data dependence graph. I've omitted things like the increment of the loop and the branch, just focusing on the main computation. So I have two loads, one brings a sub i, the other brings b sub i. I do the add and I get c sub i and then I can store that. So that might be sort of a generic op code sequence that you have. If you're scheduling that, then in the first slot I can do those two loads in parallel, second cycle I can do the add, third cycle I can do the store. I can further improve this performance. If you took 6.035 you might see software pipelining, you can actually overlap some of these operations. Not really that important here.

So, what would the cycle or the schedule look like on a cycle-by-cycle basis if this was defector output? In the scaler case, you have n iterations, right? Each iteration's taking three cycles so that's your overall execution on time -- n times 3 cycles. In the vector case, each load is bringing you four data elements, so a sub i to a sub i plus 3. Similarly for b. Then you add those together. So the schedule would look essentially the same. The op codes are different, and here what your overall execution time be?

Well, what I've done is each iteration is now doing four additions for me. So if you notice, the loop bounds have changed. Instead of going from i to n by increments of 1, now I'm going by increments of 4. So, overall, instead of having n iterations, I can get by with n over 4 iterations. That make sense? So, what would my speed up be in this case? 4. So you can get more and more speed up if your vector lens is longer, because then I can cut down on the number of iterations that I need.

Depending on the length of my vector register and the data types that I have, that effectively gives me different kinds of vector lens for different data types. So you saw on Cell you have 128 bit registers and you can pack those with bytes, characters, bytes, shorts, integers, floats or doubles. So each one of those gives you different kinds of a different vector lens.

SIMD is really now, SIMD extensions are increasingly popular. They're available on a lot of ISAs. Alt of x, MMX, SSE are available on a lot x86 machines. And, of course, in Cell, in fact, on the SPU, all your instructions are SIMD instruction, and when you're doing a scaler instruction, you're actually using just one chunk of your vector register and your vector pipeline.

So how do you actually use these SIMD instructions? Unfortunately, it's library calls or using inline assembly or using intrinsics. You'll get hands-on experience with this with Cell, so you might complain about that when you actually do it. Compile technology is actually getting better, and you'll see that one of the reasons we're using an XLC compiler is because it has these vector data types, which also latest versions of GCC have that allow you to express data types as vector data types, and the compiler can more easily or more naturally get the parallelism for you, SIMD parallelism with you having to go in there and do it by hand.

But if you were to do it by hand, or, in fact, what the compilers are trying to automate are different techniques for looking for where the SIMD parallelism is. There was some work done here about six years ago by Sam Larson, who is now graduated, on super word level parallelism. so I'm going to focus the rest of this talk on this concept of SIMDization because I think it's probably the one that's most useful for extracting parallelism in some of the codes that you're doing.

So this is really ideal for SIMD where you have really short vector lens, 2 to 8. What you're looking for is SIMDization that exists within a basic block. So within a code block, within a body of a loop or within some control flow even. You can uncover this with simple analysis, and this really has pushed the boundary on what automatic compilers can do. Some of work that's gone on at IMB, what they call the octipiler, has eventually been transferred to the XLC compiler do a lot of defecniques that build on SLP and expand in various ways to broaden the scope of what you can automatically parallize. So here's an example of how you might actually derive SIMDization or opportunities for SIMDization.

So you have some code, let's say you're doing RGB computations where you're just adding the r elements, that's a red, green and blue. So this might be in a loop, and what you might notice it well I can pack the RGB elements into one register. I can pack these into another register, and I can pack these literals into a third register. So that gives me a way to pack data together into SIMD registers, and now I can replace this scaler code with instructions that pack the vector register. I can do the computations in parallel and I can unpack them. We'll talk about that with a little bit more illustration in a second.

Any questions on this?

Perhaps the biggest improvement that you can get from SIMDization is by looking at adjacent memory references. Rather than doing one load you can do a vector load, which really gives you a bigger bandwidth to memory. So in this case, I have a load from I1, I2, and since these memory locations are continuous, I can replace them by one vector load that brings in all these data elements in one shot. That essentially eliminates three load instructions, which are potentially most heavy weight for one ligher weight instruction because it amortizes bandwidth and exploits things like spatial locality.

Another one, vectorizable loops. So this is probably one of the most advanced ways of exploiting SIMDization, especially in really long vector codes, so traditional supercomputers like the Cray, and you'll probably hear Simmon talk about this in the next lecture. So I have some loop and I hvae this particular statement here. So how can I get SIMD code out of this? Anybody have any ideas? Anybody know about loop unrolling? So if I unroll this loop, I essentially -- that same trick that I had shown earlier, although I didn't quite do it this way. I change a loop bound from n to -- the increment from -- rather than stepping through one at a time, stepping through four at a time.

Now the loop body, rather than doing one addition at a time, I'm doing four additions at a time. So now this is very natural for a vectorization, right? Vector load, vector load, vector store plus the vector add in the middle. Is that intuitive? So this gives you another way of extracting parallelism. Looking at traditional loops, seeing whether you can actually unroll it in different ways, be able to get that SIMD parallelization.

The last one I'll talk about is about partial vectorization. Either it might be some things where you have a mix of statements. So here I have a loop where I have some load and then I'm doing some computation here. So what could I do here? It's not as symmetric as the other loop.

AUDIENCE: There's no vector and [INAUDIBLE PHRASE].

PROFESSOR: Right. So you might omit that. But could you do anything about the subtraction?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: If I can unroll this again, right? Now there's no dependencies between this instruction and this instruction, so I can really move these together, and once I've moved these together then these loads become contiguous. These loads are contiguous so I can replace these by vector codes, vector equivalents. So now the vector load bring in L0, L1, I have the addition, that brings in those two elements in a vector, and then I can do my scaler additions. But what do I do about the value getting out of this vector register into this scaler register that I need for the absolute values.

So this is where the benefits versus cost of SIMDization come in. So the benefits are great because you can replace multiple instructions by one instruction, or you can just cut down the number of instructions by specific factor, your vector lens. Low stores can be replaced by one wide memory operation, and this is probably the biggest opportunity for performance improvements. But the cost is that you have to pack data into the data registers an you have to unpack it out so that you can have those kinds of communications between this vector register here and the value here, this value here and this value here. Often you can't simply access vector values without doing this packing and unpacking.

So how do you actually do the packing, unpacking? This is predominantly where a lot of the complexity goes. So the value of a here is initialized by some function and the value of b here is initialized by some function, and these might not be things that I can SIMdize very easily. So what I need to do is move that value into the first element of vector register, and move the second value into the second element of the vector register. So if I have a four-way vector register, then I have to do four of these moves, and that essentially is the packing. Then I could do my vector computation, which is really these two statements here. Then eventually I have to do my unpacking because I have to get the values out to do this operation and this operation.

So there's an extraction that has to happen out of my SIMD register. But you can amortize the cost of the packing and unpacking by just reusing your vector registers. So these are like register allocation techniques. So if I pack things into a vector register, I find all cases where I can actually reuse that vector register and I try to find opportunities for extra SIMDization. So in the other case then, I pack one then I can reuse that same vector register.

So what are some ways I can look for to amortize the cost? The interesting thing about memory operations is while there are many different ways you can pack scaler values into a vector register, there's really only one way you can pack loads coming in from memory into a vector register is because you want the loads to be sequential, you want to exploit the spatial locality. So one vector load really gives you specific ordering. So, that really constrains you in various ways. So you might bend over backwards in some cases to actually get your code to be able to you reuse the wide-word load without having to do too much packing or unpacking because that'll start eating into your benefits.

So simple example of how you might find the SLP parallelism. So the first thing you want to do is start with are the instructions that give you the most benefit, so it's memory references. So here there are two memory references. They happen to be adjacent, so I'm accessing a contiguous memory chunks, so I can parallelized that. That would be my first step. I can do a vector load and that assignment can become a and b. I can look for opportunities where I can propagate this vector values within the vector register that's holding a and b. So the way I do that is I look for uses of a and b. In this case, there are these two statements. So I can look for opportunities to vectorize that. So in this case, both of these instructions are also vectorizable. Now I have a vector subtraction, and I have a vector register holding new values h and j. So I follow that chain again of where data's flowing. I find these operations and I can vectorize that as well.

So, sign up with a vectorizable loop where all my instructions, all my scale instructions are now in SIMD instructions. I can cut down on loop iterations of total number of instructions that I issue. But I've made some implicit assumption here. Anybody know what it is?

AUDIENCE: Do you actually need that many iterations of the loop?

PROFESSOR: Well, so you can factor down the cost. so here I've vectorized by 2, so I would cut down the number of iterations by 2.

AUDIENCE: You could have an odd number of iterations?

PROFESSOR: Right, so you could have an odd number of iterations. What do you do about the remaining iterations. You might have to do scaler code for that. What are some other assumptions? Maybe it will be clear here.

So in vectorizing this, what have I assumed about relationships between these statements? I've essentially reorganized all the statements so that assumes I have this liberty to move instructions around. Yup?

AUDIENCE: [UNINTELLIGIBLE] a and b don't change [INAUDIBLE PHRASE].

PROFESSOR: Right. So there's nothing in here that's changing the values. There's no dependencies between these statements -- no flow dependencies and no other kind of constraints that limit this kind of movement. So in real code it's not actually the case. You end up with patterns of computation where you can get really a nice case of classic cases you can vectorize those really nicely. In a lot of other codes you have a mix of vectorizable code and scaler code and there's a lot of communication between the two. So the cost is really something significant that you have to consider.

This was, as I mentioned, done in somebody's Masters thesis and eventually led to some additional work that was his PhD thesis. So in some of the early work, what he did was he looked at a bunch of benchmarks and looked at how much available parallelism you have in terms of this kind of short vector parallelism, or rather SLP where you're looking for vectorizable code within basic blocks, which really differed from a classic way of people looking for vectorization. [? And you ?] have well-structured loops and doing kinds of transformations you'll hear about next week.

So for different kinds of vector registers, so these are your vector lens. So going from 128 bits to 1,024 bits, you can actually reduce a whole lot of instructions. So what I'm showing here is the percent dynamic instruction reduction. So if I take my baseline application and just compile it in a normal way and I run it again an instruction count. I apply this SLP technique that find the SIMDization and then run my application again, use the performance counters to count the number of instructions and compare the two. I can get 60%, 50%, 40%. In some cases I can completely eliminate almost 90% or more of the instructions. So it's a lot of opportunity for performance improvements that might be apparent. One because I'm reducing the instruction bandwidth, I'm reducing the amount of space I need in my instruction cache, I have fewer instructions so I can fit more instructions into my instruction cache, you reduce the number of branches. You get better bandwidth to the memory, better use of the memory bandwidth.

Overall, you're running fewer iterations, so you're getting lots of potential for performance. So, I actually ran this on the AltiVec. This was one of the earliest generations of AltiVec, which SIMD instructions didn't have I believe double precision floating point, so not all the benchmarks you see on the previsou slide are here, only the ones that could run reasonably accurately with a single precision floating point. What they measure is the actual speed up. Doing this SIMDization versus not doing a SIMDization, how much performance you can get. The thing to take away is in some cases where you have nicely structured loops and some nice patterns, you can get up to 7x speed up on some benchmarks. What might be the maximum speed up that you can get depends on the vector lens, so 8, for example on some architectures depending on the data type.

Is there any questions on that?

So as part of the next recitation, you'll actually get an exercise of going through and SIMDizing for Cell, and whether that actually means SIMDize instructions for Cell might take statements and sort of replace them by intrinsic functions, which eventually map down to actually assembly op codes that you'll need. So you don't actually have to program at the assembly level, although in effect, you're probably doing the same thing.

Last thing we'll talk about today is optimizing for the memeory hierarchy. In addition to data level parallelism, looking for performance enhancements in the memory system gives you the best opportunities because of this big gap in performance between memory access latencies and what the CPU efficiency is. So exploiting locality in a memroy system is key. So these concepts of temporal and spatial locality. So let's look at an example.

Let's say I have a loop and in this loop I have some code that's embodied in some function a, some code embodied in some function b, and some code in some function c. The values produced by a are consumed by the function b, and similarly the values consumed by b are consumed by c. So this is a general data flow graph that you might have for this function. Let's say that all the data could go into a small array that then I can communicate between functions. So if I look at my actual cache size and how the working set of each of these functions is, so let's say this is my cache size -- this is how many instructions I can pack into the cache. Looking at the collective number of instructions in each one of these functions, I overflow that. I have more instructions I can fit into my cache any one time.

So what does that mean for my actual cash performance? So when I run a, what do I expect the cache hit and miss rate behavior to be like? So in the first iteration, I need the instructions for a. I've never seen a before so I have to fetch that data from memory and put in the cache. So, the attachments. So what about b? Then c? Same thing.

So now I'm back at the top of my loop. So if everything fit in the cache then I would expect a to be a what? You'll be a hit. But since I've constrained this problem such that the working set doesn't really fit in the cache, what that means is that I have to fetch some new instructions for a. So let's say I have to fetch all the instructions for a again. That leads me to another miss. Now, bringing a again into my cache kicks out some extra instructions because I need to make room in a finite memory so I kick out b. Bring in b and I end up kicking out c. So you end up with a pattern where everything is a miss. This is a problem because the way the loop is structured, collectively I just can't pack all those instructions into the cache, so I end up taking a lot of cache misses and that's bad for performance.

But I can look at an alternative way of doing this loop. I can split up this loop into three where in one loop I do all the a instructions, in the second loop I do all the b's, and the third loop I do all the c's. Now my working set is really small. So the instructions for a fit in the cache, instructions for b fit in the cache, and instructions for c fit in the cache. So what do I expect for the first time I see a? Miss. Then second time? It'll be hit, because I've brought in a, I haven't run b or c yet, the number of instructions I need for a is smaller than what I can fit into the cache, so that's great. Nothing gets kicked out. So every one of those iterations for a becomes a hit. So that's good. I've improved performance.

For b I have the same pattern. First time I see b it's a miss, every time after that it's a hit. Similarly for c. So my cache miss rate goes from being one, everything's a miss, to decreasing to 1 over n where n is essentially how much I can run the loop. So we call that full scaling because we've taken the loop where we've distributed, and we've scaled every one of those smaller loops to the maximum that we could get.

Now what about the data? So we have the same example. Here we saw that the instruction working set is big, but what about the data? So let's say in this case I'm sending just a small amount of data. Then the behavior is really good. It's a small amount of data that I need to communicate from a to b. A small amount of data you need to communicate from b to c. So it's great. No problems with the data cache. What happens in full scaling case?

AUDIENCE: It's not correct to communicate from a to b.

PROFESSOR: What do you mean it's not correct?

AUDIENCE: Oh, it's not communicating at the same time.

PROFESSOR: Yeah, it's not at the same time. In fact, just assume this is sequential. So I run a, I store some data, and then when I run b I grab that data. This is in sequential.

AUDIENCE: How do you know that the transmission's valid then? We could use some global variable.

PROFESSOR: Simple case. There's no global variables. All the data that b needs comes from a. So if I run a I produce all the data and that's all that b needs. So in the full scaling case, what do I expect to happen for the data? Remember, in the full scaling case, all the working sets for the instructions are small so they all fit in the cache. But now I'm running a for a lot longer so I have to store a lot more data for b. Similarly, I'm running b for a lot longer so I have to store a lot more data for c. So what do I expect to happen with the working set here? Instructions are still good, but the data might be bad because I've run a for a lot more iterations at one shot. So now I have to buffer all this data for a to b. Similarly, I've run b for a long time so I have to buffer a whole lot data for b to c. Is that clear?

AUDIENCE: No.

PROFESSOR: So let's say every time a runs it produces one data element. So now in this case, every iteration produces one data element. That's fine. That's clear? Here I run a n times, so I produce n data elements. And b let's say produces one data element. So if my cache can only hold let's say n by 2 data elements, then there's an overflow. So what that means is not everything's in the cache, and that's bad because of the same reasons we saw for the instructions. When I need those data I have to go out to memory and get them again, so it's extra communication, extra redundancy.

AUDIENCE: In this case where you don't need to store the a variables [UNINTELLIGIBLE PHRASE].

PROFESSOR: But notice these were sequential simple case. I need all the data from a to run all the iterations for b. Then, yeah, this goes away. So let's say this goes away, but still b produces n elements and that overflows the cache.

So there's a third example where I don't fully distribute everything, I partially distribute some of the loops. I can fully scale a and b because I can fit those instructions in the cache. That gets me around this problem, because now a and b are just communicating one day data element. But c is still a problem because I still have to run b n times in the end before I can run c so there are n data elements in flight. So the data for b still becomes a problem in terms of its locality. Is that clear? So, any ideas on how I can improve this?

AUDIENCE: assuming you have the wrong cache line and you have to do one or two memory acceses to get the cache back.

PROFESSOR: So, programs typically have really good instruction locality just because the nature of the way we run them. We have small loops and they iterate over and over again. Data is actually where you spend most of your time in the memory system. It's fetching data. So I didn't actually understand why you think data is less expensive than instructions.

AUDIENCE: What I'm saying say you want to read an array, read the first, say, 8 elements to 8 words in the cache box. Well then you'd get 7 hits, so every 8 iterations you have to do a rewrite.

PROFESSOR: Right. So that assumes that you have really good spatial locality, because you've assumed that I've brought in 8 elements and I'm going to use every one of them. So if that's the case you have really good spatial locality and that's, in fact, what you want. It's the same kind of thing that I showed for the instruction cache. The first thing is a miss, the rest are hits. The reason data is more expensive, you simply have a lot more data reads than you have instructions. Typically you have small loops, hundreds of instructions and they might access really big arrays that are millions of data references. So that becomes a problem.

So ideas on how to improve this?

AUDIENCE: That's a loop?

PROFESSOR: That's a loop. So what would you do in the smaller loop?

AUDIENCE: [INAUDIBLE PHRASE].

PROFESSOR: Something like that?

AUDIENCE: Yeah.

PROFESSOR: OK. So in a nested loop, you have a smaller loop that has a small number of iterations, so 64. So, 64 might be just as much as I can buffer for the data in the cache. Then I wrap that loop with one outer loop that completes the whole number of iterations. So if I had to do n, then I divide n by 64. So that can work out really well.

So there's different kinds of blocking techniques that you can use on getting your data to fit into your local store or into your cache to exploit these spatial and temporal properties. Question?

AUDIENCE: Would it not be better to use a small [UNINTELLIGIBLE] size so you could run a, b, c sequentially?
PROFESSOR: You could do that as well. But the problem with running a, b, c sequentially is that if they're in the same loop, you end up with instructions being bad. That would really, this case -- so even if you change this number you don't get around the instructions.

So you're going to see more optimizations that do more of these loop tricks. I talk about unrolling without really defining what unrolling is or going into a lot of details. Loop distribution, loop fision, some of the things, like loop tiling, loop blocking. I think Simmon's going to cover some of these next week.

So this was implemented, this was done by another Master student at MIT who graduated about two years ago, to show that if you factor in cache constraints versus ignoring cache constraints, how much performance you can get. This was done in the context of StreamIt. So, in fact, some of you might have recognized a to b to c as being interconnected as pipeline filters. We ran it on different processors, so the StrongARM processor's really small. InOrder processor has no L1 cache in this particular model that we used. But it had a really long latency -- sorry, it had no L2 cache. It had really long latency to memory. Pentium, an x86 processor. Reasonably fast. It had a complicated memory system and a lot, a lot of memory overlap in terms of references. Then the Itanium processor, which had a huge L2 cache at its disposal.

So what you can see is that lower bars indicate bigger speed ups. This is normalized run time. So on the processor where you don't actually have caches to save you, and the memory communication is really expensive, you can get a lot of benefit from doing the cache aware scaling, that loop nesting to take advantage of packing instructions instead of instruction cache, packing data into data cache and not having to go out to memory if you don't to. So you can reduce run time to about 1/3 of what it was with this kind of cache optimization.

On the Pentium3 where you have a cache to help you out, the benefits are there, but you don't get as big a benefit from ignoring the cache constraints versus being aware of the cache constraints. So here you're actually doing some of that middle column whereas here we're doing third columns, the cache aware fusion. In a Itanium you really get no benefits between the two. Yep?

AUDIENCE: Can you explain what the left columns are?

PROFESSOR: These?

AUDIENCE: Yeah.

PROFESSOR: So this is tricky. So the left columns are doing this.

AUDIENCE: OK, sort of assuming that icache is there.

PROFESSOR: Right, and the third column is doing this. So you want to do this because the icache locality is the best. So you always want to go to full or maximum scaling. I'm actually fudging a little just for sake of clarity. Here you're actually doing this nesting to improve both the instruction and the data locality. So you can get really good performance improvement. So what does that mean for your Cell projects or for Cell, we'll talk about that next week at the recitation. Yeah?

AUDIENCE: Is there some big reasons [UNINTELLIGIBLE PHRASE].

PROFESSOR: Well it just means that if you have caches to save you, and they're really big caches and they're really efficient, the law of diminishing returns. That's where profiling comes in. So you look at the profiling results, you look at your cache misses, how many cache misses are you taking. If it's really significant, then you look at ways to improve it. If your cache misses are really low, you missed rate is really low, then it doesn't make sense to spend time and energy focusing on that. Good question.

So, any other questions?

So summarizing the gamut of programming for performance. So you tune to parallelism first, because if you can't find the concurrency, your Amdahl's law, you're not going to a get a whole lot of speed up. But then once you figured out what the parallelism is, then what you want to do is really get the performance on each processor, the single track performance to be really good. You shouldn't ignore that.

The modern processors are complex. You need instructional level parallelism, you need data level parallelism, you need memory hierarchy optimizations, and so you should consider those optimizations. Here, profiling tools could really help you figure out where the biggest benefits to performance will come from.

You may have to, in fact, change everything. You may have to change your algorithm, your data structures, your program structure. So in the MPEG decoder case, for example, I showed you that if you change the flag that says don't use double precision inverse DCT, use a numerical hack, then you can get performance improvements but you're changing your algorithm really. You really want to focus on just the biggest nuggets -- where is most of the performance coming in, or where's the biggest performance bottleneck, and that's the thing you want optimize. So remember the law of diminishing returns. Don't spend your time on doing things that aren't going to get you anything significant in return.

That's it. Any questions? OK. How are you guys doing with the projects? So, one of the added benefits of the central CBS repository is I get notifications too when you submit things. So I know of only two projects that have been submitting things regularly. So, I hope that'll pick up soon. I guess a few minutes to finish the quiz and then we'll see you next week. Have a good weekend.