The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation, or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: So this hour we're going to talk about SIMD programming with cell. First we'll talk a little bit about what SIMD is and then about the facilities that cell and the compiler provide for programming with SIMD. And then some design considerations that you have to keep in mind when you're doing things.

All right so the situation these days is that most compute bound applications are running through a large piece of data and running the same computations on it over and over again, or rather running the same computations across all the pieces of different data. And very frequently there'll be no dependence between iterations when you're going through this data. So that means there's opportunities for you to data parallelize.

So as an example, if we have for example, say we're multiplying a0 and b0 to get c0. And suppose we want to actually perform this operation across all the elements of arrays a, b and c. So instead of multiplying two elements or two integers together, we're actually going to be taking two arrays an element-wise multiplying each of the-- multiplying each of the pairs element-wise and writing the results to a third array.

So the picture's going to look something like this. And you would of course represent this using for example, four loop. Now what we're going to do is instead of-- let's see, so you can think of this as kind of an operation that's abstractly operating on these entire arrays.

And we're not going to go quite that far, but what we're going to do is we're going to think of these operations as acting on these kind of bundles of elements.

So we're going to bundle our arrayed elements into groups of four. And then each time we're going to take a group and multiply with another group using this element-

wise multiplication and write the result to this third bundle. OK does that make sense?

Now the thing about this kind of model is that cell is going to provide very good hardware support for something that looks kind of like this.

AUDIENCE: Is that actual cell [INAUDIBLE]

PROFESSOR: Yes, I'll get into this. In fact, this is what we'll be talking about the syntax and meaning of this kind of thing. All right?

So for this kind of thing to happen we need the compiler to support two different things. First is we need to be able to address these kind of bundles of elements and these are going to be called vectors.

And second we need to be able to perform operations on these vectors. So cell and the XLC compiler give us support for this. And what they're going to do is provide first, registers which are capable of holding vectors. Now normally you think of a register as holding on a 32-bit machine, one machine word will hold a 32-bit int, for example.

What we're going to have on the cell is these 128-bit registers which are going to be able to hold for example four ints right next to each other. So we're going to be able to take this bundle of ints and operate it-- operate on it as a unit.

And the second part is we're going to have operations that act on these vector registers. So the cell is going to support special assembly instructions and it's going to be able to interpret those as acting on particular vectors.

But also we're going to have C++ language extensions which are called intrinsics. And those are going to give us access to these special assembly instructions, but not require us to be poking around in the assembly.

All right now the big draw of this is that these operations are going to be pretty much as fast as single operations, which means that if we take advantage of them we can make our code run say four times as fast.

OK so how do we refer to these vectors when we're coding? XLC is going to provide us with these intrinsics. And we have these vector data types and each one is just going to specify how to interpret a consecutive group of 128-bits as some sort of vector.

And you can have vectors of varying element sizes and varying number of elements. So when you're programming on the PPU or the SPU you get these four different kinds of vector data types. You can declare things as for example, vector signed int, which is what I mentioned in the example. Which is where you have four ints next to each other each 32-bits.

You could also have vectors which contains 16-bit integers or 8-bit integers. And you could also have vectors of floating point-- floating point numbers.

I should mention that all of these signed integer types also have unsigned equivalence. Anyway, so you can just declare these anywhere in your code and use them as if they were a C++ data type. All right any questions?

On the SPU you also get some additional vector data types. One is vector signed long-long, which is 64-bit ints and you can fit two of those in 128. And you can also fit two double precision floating point numbers in 128-bits.

Now the compiler's actually support these types pretty nicely. So not only can you declare variables of these types pretty much anywhere in your code, you can also declare pointers to these types and arrays of these types. All right. And so they look pretty much like natural C++ types, except that they translate directly into these particular types that the hardware supports.

Now in order to manipulate these vector data, we're going to have the-- we're going to have compiler extensions called intrinsics, which are going to provide access to the assembly level features that we want. Remember we're going to have specific assembly instructions that correspond to for example, multiplying two vectors which contain each four, 32-bit integers.

And instead of writing out-- instead of going into the assembly and actually inserting that instruction ourselves, we just use a compiler intrinsic inside our C++ code. And what it does is it provides this notation that looks a lot like a function call, but the compiler automatically translates it into the correct assembly instruction.

And again you don't have to worry about going into the assembly and messing around with this instruction that's supposed to apply to these registers. You don't have to worry about register allocation at all. The compiler just figures out the right thing for you. And to use these in your SPU program you're going to want to include SPU_intrinsics.h.

Now what's a little bit confusing is that you're going to have slightly different intrinsics available on the PPU and the SPU, because those are actually going to have different instruction sets. But anyway as an example, you can declare two variables of type vector signed int and then you can multiply them using this intrinsic called SPU add. All right and assign them to a third vector signed int. Questions? Yep.

AUDIENCE: In what way are they introduced if you're on the SPU or the PPU? Is it just-- not entirely the same set of operations available? Or are there actually semantic differences? Could make a little header file that masks over the differences mostly.

PROFESSOR: There are going to be some operations that are only available on one and not the other. But in general, if you look at the names, if the names correspond, and I'll go into that in a little bit, then they should perform essentially the same function.

AUDIENCE: [INAUDIBLE] mostly was the [INAUDIBLE] also some name differences where there really don't need to be. For instance, if you try to do a shift on the PPU I believe it's a [INAUDIBLE] SL, shift logical right, shift logical left or shift arithmetic right. Sort of things you would remember. On the SPU it's the acronym for rotate and mask for shift. So R-O-T-M-A-R or something like that. So yes there's some differences that don't need to be there.

PROFESSOR: OK so to actually create these vectors there's a couple of different notations you

can use. The first is, you can use this thing which looks like a cast to, for example, vector signed int. So you do vector signed int in parentheses and then a list of four integers you want to fill in. And that will create an integer vector and assign it to a.

You can also, I believe you can also use that notation with just one integer and it will fill in that integer in all four positions. There's also an SPU intrinsic called splats that you can use to basically copy the same integer to all four components.

AUDIENCE: How does it know you're not using a comma operator?

PROFESSOR: Yeah I don't-- is that right David, with the parentheses in the second part? OK.

AUDIENCE: Whatever.

AUDIENCE: Another caveat here from someone who's been in the trenches is that XLC likes this notation. GCC sometimes likes curly brace notations instead.

PROFESSOR: So I'd seen both of those and I do know which to do.

[INTERPOSING VOICES]

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK great thanks. All right.

And after you've assigned some of these variables in order to get back the pieces out, one way you can do it is to use this union trick where you assign or rather you allocate something of vector signed int and then you tell C++ that it can find an array of integers in the same place. And what that will do is pull out the components.

So if you define this union this way, then you get a type called intVec. And any time you have an intVec you can either do dot Vec to get at the vector signed int, the vector data type. Or you can use dot vals to-- with an array index get at the components of the vector. And you could also use this intrinsic call SPU extract to pick out the same components.

XLC provides a bunch of different vector operations that you can use. There's

integer operations, floating point operations, there are a permutation and formatting operations which you can use to shuffle data around inside vector. And there's also load and store instructions. And I believe we have a reference linked off the course website if you want to figure out more about these. I'm only going to touch on a few of them.

OK so the arithmetic and logical operations like I said, most of these are the same between the PPU and the SPU. There's some that are named slightly differently and some that are not available on the PPU. So these are all things you would expect, add, subtract.

Madd is multiply and then add with three-- with three arguments. Multiply, re is for reciprocal. You can also do bit-wise and, or xor and I believe there are other logical operations there too.

Now the thing is you have to worry about which PPU or SPU instruction you're using. But you usually don't have to worry about selecting the right vector type. The compiler should figure out which vector types you're using and substitute the appropriate assembly instruction that produces a result of the same vector type. So all these operations are what we call generic. And they stand in for all the specific instructions, which are-- which only apply to a single vector type. Does that make sense?

OK so one handy thing is a permutation operation. And this allows you to rearrange the bytes inside a vector or two vectors arbitrarily. And so the syntax is SPU shuffle a, b which are your source vectors and pattern which tells you how to shuffle them. And pattern is going to be interpreted as a vector of 16-bytes.

And each byte is going to tell you-- each byte is going to tell the compiler how to pick out one of these bytes in the result. And how the byte is interpreted is the low, the low four bits are going to specify which position the source is going to come from.

And the fourth byte is going to specify whether you're going to pull from a or b. So

as an example, here's the pattern VC and if you look at the second byte which is one, which is one, four in hex. Then that means the destination register is going to contain the fourth byte or the fourth byte of b, all right.

So four means select the element numbered four and one means select from b. Does that make sense? And this is very versatile by putting in the right, by putting in the right pattern vector you can arrange for all these bytes to be shuffled around however you want.

AUDIENCE: The pattern is a constant [INAUDIBLE].

PROFESSOR: Pardon?

AUDIENCE: The pattern is a constant in intermediate parameter.

PROFESSOR: You can fill in the parameter at run time if that's what you're asking.

AUDIENCE: [INAUDIBLE]

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, also useful are these rotation operations which will let you shift your vector left or right by some amount.

Now one thing to be aware of is that on the SPU you only have these 128-bit registers. So on the PPU you have different registers which are suitable for holding different types. For example, there's word-sized registers for holding ints and PPU also has these 128-bit registers. But the SPU has nothing else.

So that means whenever you're using scalar types on the SPU they're all going to be using these large registers. No matter what the size of the scalar you're using. And depending on the size of the scalar you're using it's going to go in a particular position inside this wide register. It's called a quadword register, because it's 16-bytes.

Now the thing to watch out for is that whenever you load something from memory

into-- whenever you load a scalar from memory into one of these registers, there's going to have to be a little extra processing done in order to shift-- in order to possibly shift the scalar into the right place inside this register.

And furthermore, the hardware is always going to want to grab one of these quadwords all at a time. So loading a scalar is not going to be any cheaper than loading one of these quadword registers. So one possible you're going to want to load an entire quadword register at a time. And if you just need a part of it, then you can figure that out later. But you might as well get the whole thing.

Questions?

AUDIENCE: So when you-- just a scalar question. So when you load a scalar value that's not aligned-- it's not aligned with the preferred position is that-- is there overhead associated with that?

PROFESSOR: I'm not sure how much overhead is associated with that. Pardon? Oh do you know?

AUDIENCE: Well, unlike scalar it's [INAUDIBLE], it can only load on 16-byte boundaries. So it's going to load the-- load something that includes that and that's going to have to shift to the another position.

PROFESSOR: So do unaligned-- when it has to shift the scalar around, does that actually take longer than went in-- when it's natural?

AUDIENCE: I don't know if it's-- well what you can do, you can set some flags in XLC that say, align all of my scalars correctly. And we'll waste 4x overhead. It'll even say align my array, have my elements so that I can have the scalar array at the back-- I can load and it will waste the overhead's that everything in the array is [INAUDIBLE]. So you can have the compiler trade off space versus time for you off two switches.

PROFESSOR: I see.

OK so we're going to want to look at the sim application from recitation two. And we want to adapt that to make use of SIMD data types and intrinsics. So what we've done is, remember we had these x, y, z coordinates that we were manipulating.

What we're going to do is we're going to pad each one. It was three words before and we're going to pad each one so that it fills a quadword.

And so for each quadword of course the first three words are going to correspond to the x, y, z components. And we can grab those out using SPU extract or some other intrinsics. Now when we're doing manipulations with these components for example, we wanted to find the displacement between two locations. And that's just subtracting two of these coordinates.

So we can do that subtraction which before required three floating point subtractions. We can replace that with a single SIMD instruction. Does that make sense? OK so all this-- most of this has already been done and we're providing most of the implementation of this SIMD version of sim.

And what we want you to do is download this, download the tarball for this recitation and then go in there and what we want you to do is fill in one of the blanks. All right. So there's just one function here that's been left unimplemented. And to see if you know what's going on, see if you can fill in the implementation for this. Any questions?

So this question-- this function you want to implement is basically going to take a vector float and if that float contains a, b, c and d you want to return a-- you want to return a vector which each of whose elements is a plus b plus c plus d. Questions?

AUDIENCE: What directory under the--

AUDIENCE: [INAUDIBLE].

PROFESSOR: So we're going to go into sim a list.

AUDIENCE: But we can stay around afterwards and help you figure out what's going on.

PROFESSOR: OK so here's one implementation. Basically we're going to just declare another vector float and that vector float we're going to-- that's basically we're just going to do these swaps. So notice in this one we're swapping the first and second-- we're

swapping the first and second words.

So that means down here we're going to want to carry bits four, five, six, seven and then-- or bytes four, five, six, seven first and then bytes 0, 1, 2, 3. And then over here we want bytes 12, 13, 14, 15 and then 8, 9, 10, 11. Everyone see what's going on for the first shuffle? And then we're going to just add that to our original vector to get this.

And we can do that again, this time now we just want to swap these two halves. So the shuffle pattern is going to be 8, 9, 10, 11, 12, 13, 14, 15 followed by 0, 1, 2, 3, 4, 5, 6, 7.

Make sense?

OK so the way we translated the program we just used into SIMD was we used a struct of arrays. Basically each of these structs that we had from our previous implementation just carried over and we just put all those into an array. So the structs were right next to each other.

Alternatively we could have laid out the, laid out the data in memory in a different way and this is called an array of structs layout. Instead what we can do is put all the like fields next to each other so that we have, for example, an array of all the x components, then an array of all the y components, then an array of all the z components. And when you reorder the data this way you get different ways you can use to process it.

So for example, now each quadword instead of containing the data for a single point is going to contain the data for the same component of four consecutive points. Everyone see that? And actually we can implement the algorithm from before in this new layout.

But we have to be a little bit more clever in how we're putting together the elements. Before we were able to just subtract each-- subtract or multiply the quadwords with each other, because those would just correspond to for example, subtracting the coordinates of two points.

Now this time we have to do some additional computation in order to put all the pieces together. The trick behind this structure of arrays implementation which I'll just gloss over is, if we're storing state for eight objects then we're going to need-- eight objects hold the-- hold 24. Rather for each object we need the position and the velocity. And for each of those we have x, y and z. So that means to store state for each object, for eight objects we need 8 times 6 is 48 words.

And so we can put those in 12 quadwords if we pack them right. And when we do SIMD operations on these quadwords that we pull out, we can get four pair interactions. So suppose this is a quadword and it's contained-- and it contains data corresponding to elements a, b, c and d. And over here we have a quadword containing data corresponding to elements one, two, three, and four.

With some SIMD operations we can kind of figure out the pairwise interaction between objects a and one, between b and two, c and three and d and four. But of course we have to be able to find the interactions between any pair. It's not just these pairs that lineup. So what we have to do is rotate the quadword over by one word and then do the same thing again. We do that four times in all and then we add up the results.

So as you can see this implementation is a little bit more involved and less-- it maps to the original implementation less directly.

On the other hand, it does give us a really dramatic speedup. Because we're using more of the vector words. Notice that in the first packing we had. We had x, y and z and then the fourth blank was unused. Anyway, this time the structure of a race implementation is actually 7 1/2 times faster than the array of structures implementation.

So choosing this data layout correctly can actually be one of the really big determinants of how your program performs.

AUDIENCE: The scalar version was like what 480, something like that? Or is it not comparable?

PROFESSOR: I let's see, David, do you remember?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK so--

AUDIENCE: [INAUDIBLE].

AUDIENCE: No that was just on the PPU.

AUDIENCE: [INAUDIBLE]

[INTERPOSING VOICES]

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, so something like 400 for the double buffered one and 300 for array of structs.

OK one other thing to worry about is when you're dealing with these-- when you're dealing with these SIMD instructions, you want to make sure that all your data are aligned correctly in memory. And like I said before, when you're pulling things in from memory you want to make sure that whatever you're pulling in is going to be aligned on a quadword boundary.

And you can use the align compiler directive to tell the compiler, I want this piece of data aligned at a particular place. And if you do that on all your arrays for example, and make sure that your array-- the array elements are going to fit neatly into quadwords then you should be OK.

Again like I said before, you also want to transfer only multiples of 16-bytes on the load and store. And so when you're doing processing it may help, it may help you if you actually pad the end of your-- pad the end of your array's so that they fill out a multiple of 16-bytes. Because it's easier to just do that processing with the SIMD instruction rather than just have one or two elements hanging off and have to worry about those.

AUDIENCE: Question.

PROFESSOR: Yep.

AUDIENCE: Is it a good idea to pass parameters 2.2 [INAUDIBLE] I mean, which one is preferred? [INAUDIBLE].

AUDIENCE: So you should [INAUDIBLE] for figuring out whether something can scale easily or not. So you might make [INAUDIBLE]. So in cases where you can avoid using pointers, you should do that.

PROFESSOR: OK.

[SIDE CONVERSATION]

PROFESSOR: So one last thing that I should mention. I haven't really let on, but compilers can actually generate some of these SIMD instructions by themselves. If you declare your types to be vector and then use just regular operations apparently GCC and XLC yes, will substitute the correct intrinsics for you.

Of course that doesn't get you all the operations which are available with intrinsics, but anyway automatically simbianizing your code is something that's really worth looking into. As we saw it can give you a great performance improvement.

And the thing is that compilers are still not very good at automatically doing this transformation. So unlike instruction scheduling where if your passing 05 your compiler will do a much better job than you would have time to do. This is something that you should probably reserve some time for.

That's all. If you have any questions you can stick around and I'll try and help you.