PROFESSOR: So let's get started with the second lecture for today. So I guess one thing multicores did, is really shatter this nice view of writing in your programs and hardwares to take care of, giving you performance. So hardware just kind of completely gave that up. But so what you're doing in this class, is you're trying to do it by yourself. Give all the responsibility back to the program. And you realize as you go, it's a much harder job. I mean, this is not simple programming. So you need to have, you don't have MIT class students on every company to do this, so we need to have some kind of middle ground. And so some of the stuff we have been doing is trying to figure out are there any middle ground. Can you actually take some of that load away from the user into things like languages and compilers. So we will talk about some of those. So right now we are kind of switching from directly doing what's necessary, to do the Cell project into going breadth So this lecture, and then we will sit back and do a little bit of debugging and performance work and that will be directly helpful. And then next week we'll have lots of guest lectures to kind of give you breadth in there. So you'll understand no just Cell programming but parallel programming and parallel processing, what the world is like beyond that. So today we're going to have Bill talk about streams.

BILL THIES: OK very good. So my name is Bill Thies. I'm a graduate student working with Saman and Roderick, and others here. And I'll talk about the StreamIt language. So why do we need a new programming language? Well we think that languages haven't kept up with the architectures. So one way to look at this is that if you look back at previous languages, look at C with von-Neumann machine. Now I grew up in rural Pennsylvania not too far from Amish Country. And so to me these go together just like a horse and buggy. OK they're perfectly made for each other. They basically go at the same rate. Everything is fine. But the problem is, in comes the modern architecture. OK this is an F-16. you have a lot more that you can do with it then, then with the horse and buggy.

So how do you program these new architectures? Well architecture makers these days are basically faced with a really hard choice. On the one hand, you could get a really cool architecture and develop an ad hoc programming technique where you're really just leaving it to the programmer to do something complicated to get performance. And unfortunately I think that's the route that they took. I mean fortunately for the industry, but unfortunately for you, I think that's the route they took with cell which means all of you are going to become basically fighter pilots. You have to learn how to fly the plane. You have to become an expert. You're going to become the best people at programming these architectures. And unfortunately the only other option is to really bend over backwards to support the previous era of languages like C and C . And you can see what's coming here, it's just hard to get off

the runway. So you don't want this situation. Out of consideration for whoever is in the buggy, hopefully you'll never take off.

So looking from a more academic perspective, why do we need a new language right now? So if you look back over the past 30 years, I know you've seen this graph before. We were dealing with just one core in the machine for all this time. And now we have this plethora of multicores coming across the board. So how did we program these old machines? Well we had languages like C and FORTRAN that really have a lot of nice properties across these architectures. So it was portable, high-performance, composable-- you could have really good software development-- malleable, maintainable, all the nice things you'd like to see from a software engineering perspective. And really if you wrote a program back in 1970, you could keep it in C and have it continue to leverage all the new properties of machines over the past 30 years. So just one fine out of the box. And looking forward, that's not going to be true. So for example, we could say that C was the common machine language. That's what we say for the past 30 years, was common across all the machines. But now looking forward, that's not going to be true anymore. Because you have to program every core separately.

So what's the common machine language for multicores? We really think you need something where you can write a program once today, and have it scale for the next 30 years without having to modify the program. So what kind of language do you need to get that kind of performance? Well let's look a little deeper into this notion of a common machine language. So why did it work so well for the past 30 years? Well on uniprocessors, things like C and FORTRAN fran really encapsulated the common properties. So things like a single flow of control in the machine, a single memory image, are both properties of the language. But they also hid certain properties from the programmer. So they hid the things that were different between one machine and another. So for example, the register file, the ISA, the functional units and so on. These things could change from one architecture to another. And you didn't have to change your program because those aspects weren't in the programming language. So that's why these languages were succeeding.

And what do we need to succeed in the multicore era from a language perspective? Well you need to encapsulate the common properties again. And this time it's multiple flows of control that you have for all the different cores, and multiple local memories. There's no more monolithic memory anymore, that everyone can read and write to. Also you need to hide some of the differences between the machines. So some cores have different capabilities. On cell there's a heterogeneous system between the STEs and the PPE. Different communication models on different architectures, different synchronization models. So whatever common machine language we come up with, we'll have to keep these things hidden from the programmer.

Now a lot of different researchers are taking different tacts for how you want to invent the next common machine language. And the thrust that we're really excited about is this notion of streaming. So what is a stream program?

Well if you look at a lot of the high-performance systems today-- including Powerpoint which is running this awesome animation-- you can basically see that they're based around some stream of data. So audio, video, like HDTV, video editing, graphic stuff. I think actually, a lot of the projects in this class that I looked at, would fit into the streaming mold. Things like the software radio, array tracing, I probably don't remember them all. But when I looked at them, they all looked like they had a streaming component somewhere in there.

So what's special about a stream program compared to just a normal program? Well they have a lot of attractive properties. If you look at their structure, you can usually see that the computation pattern remains relatively constant across the lifetime of the program. So they have some well-defined units that are communicating with each other. And they continue that pattern of communication throughout. And this really exposes a lot of opportunities for the compiler to do some optimizations that it couldn't do on just an arbitrary general purpose program. And if you saw before, we have basically all the types of parallelism are really exposed in a stream program. There's the pipeline parallelism between different producers and consumers. There's the task parallelism basically going from left to right. And also data parallelism which means that a single one of these stages can sometimes be split to apply to multiple elements in the data stream.

So when you're thinking about stream programming, there's a lot of different ways you can actually represent the program. So whenever you have a programming model, you have to answer these kinds of questions. For example do the senders and the receivers block when they try to communicate? How much buffering is allowed? Is the computation deterministic? What kind of model do you have in there? Can you avoid deadlock? Questions like these, and we could spend a whole lecture answering these questions, putting them in different categories. But what I want to just to do, just to give you a feel is touch on kind of three of the major models that you might see come up in different kinds of programming models. And I'll just touch Kahn process networks, synchronous dataflow, and communicating sequential processes, or CSP.

So just one slide on these models. So let's compare them a little bit. First there's the Kahn process networks. So this is kind of the simplest model. It's very intuitive. You just have different processes that are communicating over FIFOs. And the FIFO size is conceptually unbounded. So to a first approximation, it's kind of like a Unix pipe. These processes can just read from the input, and they can push onto their outputs without blocking. But if they try to read from an input they do block until an input is available. And the interesting thing is that the communication pattern can actually be dependent on the data. So for example I could pop an index off of one channel, and then use that index to determine which other channel I'll read from on the next time time step. But at the same time it is deterministic. So for a given series of input values on the stream, I'll always have the same communication pattern that I'm trying from the other input. So if it's a deterministic model, that's a nice property. Let's see, what else to say here? There's actually a few recent ventures that are using Kahn process networks. So

there's commercial interest. For example Ambric is a startup that I think will be based on a Kahn process network for the programming model.

Looking at another model called synchronous dataflow, this is actually what we use in the StreamIt system. And compared to Kahn process networks, it's kind of a subset. It's a little bit more restrictive. So if you look at the space of all possible program behaviors, Kahn process networks are a pretty big piece of the space. And then synchronous dataflow is kind of a subset of that space where you know more about the communication pattern at compile time. So for example, in synchronous dataflow, the programmer actually declares how many items it will consume from each of its in put channels on a given execution step. So there's no more data dependence regarding the communication pattern. It'll always input some items from some of the channels and produce some number of items to other channels. And this is a really nice properties because it lets the compiler do to scheduling for you. So the compiling can see who's communicating to who and exactly what pattern. And it can statically interleave the filters to guarantee that everyone has enough data to complete their computation. So there's a lot of interesting optimizations you can do here. That's why it's very attractive for StreamIt. And you can statically guarantee freedom from deadlock, which is a nice property to have.

The last one I want to touch on is communicating sequential processes or CSP. And in the space of program behaviors, it's kind of an overlapping that from Kahn processing networks, and adds a few new semantic behaviors. So the buffering model is basically rendezvous communication now. So there's no bothering in the system. Basically anytime you send a value to another process, you have to block and wait until that process will actually receive that value from you. So everyone is rendevouzing at every communication step. In addition to that, they have some sophisticated synchronization primitives. So you can for example, discuss alternative behaviors that you have. You can either one thing or another which will introduce the nondeterminism in the model. Which could be a good or a bad thing depending on the program you're trying to express. And pretty much the most well-known encapsulation of CSP is this occam programming language invented quite a while ago. And some people are still using that today. Any questions on the model computations? OK.

So now let me get into what StreamIt is. So StreamIt is a great language. It's a high-level architecture-independent language. Oh question the back.

AUDIENCE: With the CSP I'm trying to understand exactly what that means or how it's different. Is basically what's it's saying is you have a bunch processes and they can send messages to each other.

BILL THIES: So all these models have that property.

AUDIENCE: They all fit into that. But it seems like from your explanation of CSP, that that was just sort of the essence of CSP is it more specific?

BILL THIES: So CSP is usually associated with rendezvous communication. That's the side of programs that fit inside Kahn process networks. It's any communicating model where you basically have no buffering between the processes. Now the piece that sits outside is usually lumped with CSP, or especially with occam They have a set a primitives that are richer in terms of synchronization. So for example, you can have guards on your communication. Don't execute this consumption from this channel until I see a certain value. So there's some more rich semantics there. And so that's the things that are usually outside. They're outside the other models. Does that make sense? Other questions?

OK so StreamIt. OK so StreamIt is architecture-independent. It's basically a really nice syntactic model for interfacing with these lower level models of computation for streaming. And really we have two goals in the StreamIt project. And the first is from the programmer's side. So we want to improve the programmer's life when you're writing a parallel program. We want to make it easier for you to write a parallel program then you would have to do in C or a language like Java, or any other language that you know. And at the same time, we want scalable and portable performance across the multicores. So an interesting thing these days is you'll find, is it's often very hard a tempt the programmer to switch to your favorite language based solely on performance. Or at least this has been the story in the past. It may change, looking forward. Because it's a lot harder to get performance these days. But usually you have to offer them some other carrot to get them on board. And you know the carrot here is that it's really nice to program in. It's fun to program in. It's beautiful. It's a lot easier to program and stream it then it would be in something like C or Java for a certain class of programs. So that's how we get them on board, and then we also provide the performance. We're mostly based on the synchronous in dataflow model. In that when there are static communication patterns, we leverage that from the compiler side. So I'll also tell you about some dynamic extensions that we have, that is the much richer model of communication.

So what have we been doing in the Streamit Project? We have kind of a dual thrust within our group building on this language. So the first thrust is from the programmability side, looking at applications and programmability What can we fit into the streaming model? And we're also really pushing the optimizations. So what can you do from both a domain specific optimization standpoint, as kind of emulating a DSP engineer or a signal processing expert in the design flow. And also architecture specific optimizations. So we've been compiling for a lot of parallel machines. And we were hoping we could have a full system for you guys, this IEP, so you could write it then stream it and then hit the button. And it would work the whole way down to cell. Unfortunately we're not quite there yet. But we do have a pretty robust compiler infrastructure. And you can download this off the web and play with it if you want to.

One of our backends that we've released so far actually does go to a cluster of workstations. So it's kind of an MPI-like version of C. It uses Pthreads for the parallelism model. And I mean, depending on what kind of a hacker

you are, you actually might be able to lower that down onto cell. So some of the stuff you might be able to use if you're have some initiative in there. And of course we'd be willing to work with you on this as well. so we have lots optimizations in the tool flow. And actually Saman will spend another lecture focusing on the StreamIt compiler, and how we get performance out of the model.

OK, so let's just jump right in and do the analog of Hello World in StreamIt. I'm going to kind of walk you through the language and show you the interesting pieces from an intellectual standpoint. What's interesting about a streaming model that you can take away. So instead of Hello World, we have a counter. Since we're dealing with stream programs here, you're not usually doing text processing. So how do you write counter? Well there are two pieces to the program. The first is kind of the interconnect between the different components. That's what we have up here. We're saying the program is a pipeline with two stages, it has a source, and it has a printer. And then we can write the source and the printer as filters. We call those basic building blocks filters in Streamit. So the source will just have a variable x that it initializes is zero. And then we have a work function which is automatically called by our runtime system every time through the steady state. So this work function well push one item on to the output channel, and it'll increment the value afterward. Whereas the intPrinter at the bottom here, will input one value. And its work function here just pops that value off the input tape, and prints it to the output.

Now how do we run this thing? Well there's no main function here like you see in Hello World. Is there comment? Oh, sorry.

AUDIENCE: The two meanings of push and two meanings of pop.

BILL THIES: Two meanings?

AUDIENCE: Push 1, 2, 3.

BILL THIES: Yeah, yeah. So the first push here is just declaring that this work function will push one item to the output tape. So this is the synchronous dataflow aspect. Were associating an output rate with this work function. So that's a declaration here. Whereas this push is just actually executing the push onto the output.

So how do we run this thing? Well we compile it with a StreamIt compiler, store C into a binary. And then when we run, we run for a given number of iterations. So you don't just call it once. Our model is that this is a continuous stream of data going through the program. And so when you run it, you run it for some number of iterations, or basically input or output items, is what you're running it for. So if you run this for four iterations, it would print in the 033. So we can leverage this steady flow of data. Yeah Amir?

AUDIENCE: 1, 2, 3, 4, pushing X plus plus .

BILL THIES: I think the plus, plus is executed after the push.

AUDIENCE: Push plus plus?

BILL THIES: So it starts at zero. So I think a PostFix expression executes after the actual obsession. Yeah. Yeah. Other questions? OK so let's step up a level and look at what we have in StreamIt. So the first question is, how do you represent this connectivity between different building blocks? How do you represent streams? And if you look at traditional programming models, kind of the conventional wisdom is that a stream program is a graph. You have different nodes that are communicating to each other. And graphs are actually kind of hard to analyze. They're hard to represent. They're a little but confusing. So the approach we decided to take in StreamIt is one of a structured computation graph. So instead of having arbitrary inner connections between the stages, we have a higher hierarchical description in which every individual stage has a single input and a single output. And you can compose these together into higher level stages. Of course there's some pages that do split and join with multiple inputs. We'll get to that.

So the analog here is kind of ah analogous to structured control flow, in your favorite imperative programming language. Of course there was a day when everyone used goto statements instead of having structure control flow. We've got a fan of goto statements in the audience? OK, I'll get to you later. But the problem was, it's really hard to understand the program that's jumping all over the place because there's no local reasoning you can have. You know you're jumping to this location, you're coming back a different way. It's hard to reason about program components. So when people went to structured control flow, there's just if else, four loop statements. Those are the two basic constructs. You can basically express all kinds of computation in those simple primitives. And things got a lot simpler. And you know people objected at one point even. You know what about a finite-state machine? Don't you need goto statements for a finite-state machine, going from one state to another another. And now everyone writes in FSM with a really simple idiom. You usually have a while loop around a case statement. Right, you have a dispatch loop. So and now whenever you see that pattern you can recognize, oh there's a finite-state machine. It's not just at set of gotos. It's a finite-state machine.

So we think there are similar idioms in the streaming domain. And that's kind of the direction we're pushing from a design standpoint. So what are our structures that we have? Well here are our structured streams. At the base we have a filter. That's just the programmable unit like I showed you. We have a pipeline, which just connects one stream to another in a sequence. So this gives you pipeline parallelism. There's a splitjoin where you have explicit parallelism in the stream. So I'll talk about what these splitters and joiners can do. It's basically a predefined pattern of scattering data to some child streams, and then gathering that data back into a single stream. So the whole construct still remains single input and single output.

Likewise a feedback loop is just a simple way to put a loop in your stream. And of course these are hierarchical. So all of these green boxes can be any of the three constructs. So that's how you can have these hierarchical graphs. And again, since everything is single-input single-output, you can really mix and match. You know choose your favorite components, and they'll always fit together. You don't need to stitch multiple connections.

So let's dive inside one of these filters now. And I gave you a feel for how they look before, but here's a little more detail. So how do we program the filter? Well a filter just transforms one stream into another stream. And here it's transforming a stream of floating-point numbers into another floating-point number stream. I can take some parameters at the top. These actually fixed at compile time in our model, which allows the compiler to really specialize the filters code depending on the context in which it's being used. So for example here, we're inputting N in a frequency. And then we have two stages of execution. At initialization time-- this runs one at the beginning of the whole program-- we can calculate some weights for example, from the frequency. And we can store those weights as a local variable. So you can think of this kind of like a Java class. You can have some member variables. You can retains state from one execution to the next.

The work function is the closest thing to the main function. This is called repeatedly in the steady state. And here are the IO rates of the work function. This filter actually peaks at some data items. That means that it inspects more items on the input channel than it actually consumes on every iteration. So we'll look at N input items, and we'll push one new item onto the output and pop one item from the input tape every time we execute. So here we have a sliding window computation. It means the next time through, we'll just slide this window up by one and inspect the next N items on the input tape. And inside the work function you can have pretty much general purpose code. Right now we just allow pointers and a few other things to keep things simple. But the idea is this is general purpose imperative code inside the work function.

Now what's nice about this representations of a filter is for one thing is this peak function. So what we really have is a nice representation of the data pattern, that you're reading the data on the input channel. And if you look at this for example in a language like C, it's a lot messier. So usually when you're doing buffer management, you have to do some modulo operations. You have to keep a circular buffer of your live data. And increase you know, a head or tail pointer and mod around the side with some kind of modulo operation. And for a compiler, this is a real nightmare. Because modulo operations are kind of the the worst thing to analyze. You can't see what it's actually trying to read. And if you want to map this buffer to a network or to a combined communication with some other actor or filter in the graph, it's pretty much impossible. So here is we're exposing that all to the compiler. And you'll see how that can make a difference. And also it's just a lot easier to program. I mean, I don't like looking at the code. So I'm going to go to the next slide.

OK, so how do we piece things together? Let's just build some higher level components. So here's a pipeline of

two components. And we already saw a pipeline. You can just add one component after another. And add just basically has the effect of making a queue, and just queueing up all of the components that you added, and connecting them one after another. So here we have a BandPastFilter by connecting a LowPassFilter and feeding it's output into a HighPassFilter. You end up with a BandPassFilter.

OK what about a splitjoin? How do we make those? So a splitjoin has an add statement as well. And here we're adding components in a loop. So what this means is now when we say add, we're actually adding from left to right. So instead of going top to down, we're adding from left to right across the splitjoin. And we can actually do that in a loop. So here we're intPrinting a parameter N. And depending on that value, we'll add N BandPassFilters to this splitjoin. So it's kind of cool, right? Because you can input a parameter, and that parameter actually affects the structure of your graph. So this graph is unrolled at compiled time by our compiler, constructing a big sequence of computations. And it can resolve the structure and communication pattern in that graph, and then map it to the underlying substrate when we compile.

Also to notice here are the splitter and the joiner. So we have a predefined set of splitter and joiners. I'll go into more detail later. But here we're just duplicating the data to every one of these parallel components, and then doing a round-robin join pattern where we bring them back together into a single output stream. And if you want to do an equalizer, you basically need an adder at the bottom to add the different components together. And another thing you can notice here is that we have some inlining going on. So we actually embedded this splitjoin inside a higher level pipeline.

So what this does is it prevents you from having to name every component of your stream. You can have a single stream definition with lots of nested components. And the natural extension is, you can basically scale-up to basically the natural size of a procedure, just like you would in an imperative language. And here is for example, an FM radio where we have a few inline components. And the interesting thing here is that there's a pretty good correspondence between the lines of the text and the actual structure of the graph. And that's something that's hard to find in an imperative language. I mean if you want to for example, stitch nodes together with edges, it's often very hard to visualize the resulting structure of the graph that you have. But here if you just go through the program, you can see that the AtoD component goes right over to the AtoD, the demodulator to the demodulator, and so on. And even for the parallel components, you can kind of piece them together.

so that's kind of how we think of building programs. Any questions so far? OK so this is kind of how you go about programming in StreamIt. But programming is kind of a chug n' plug activity right? Nobody wants to be a code monkey. The reason we're all here is to see what's beautiful about this programming model. Right? Don Knuth said this, a famous computer scientist from Stanford during his Turing Award Lecture you know,

"Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write *grand* programs, *noble* programs, truly *magnificent* ones!"

Right. We want the best programs possible. It's not just about making it work. We want really beautiful programs. So what's beautiful about the streaming domain? What can you go away with and say wow, that was a really beautiful expression of the computation. Well for me I think one of the interesting things here is the splitjoin contruct. Splitjoins can really be beautiful. You know some mornings I just wake up and I'm like, oh I'm so glad I live in a world with splitjoins. You know? And and now splitjoins will be part of your world. You can say this tomorrow. This is just wonderful.

So OK, what do we having in the splitjoin constructs? You can duplicate data. You can do a round-robin communication pattern from one to another, or round-robin join. Now the duplicate is pretty simple. You just take the input stream and duplicate it to all the children. No problem. What do you do for a round-robin? Well you path N items from the input to a given child. So for example, if N is 1, we'll just distribute one at a time to the child streams. And you get a pattern like this, a round-robin just going across. And you can do the same thing on the joiner side. Let's say you're joining with a factor of one. You're just reading from the children and putting them into a single stream. OK so the pretty colorful, but nothing too fancy yet.

Let's consider a different round-robin factor. So a round-robin of 2 means that we peel off 2 items from the input, and pass those items to the first output. OK there actually is going to be a quiz on this. So ask questions of this doesn't make sense. OK pass the next 2 items, and 2 items round-robin like that. And you can actually have nonuniform weights if you want to. So on the right let's say we're doing round-robin 1, 2, 3. That means we pass 1 item from the first stream, 2 items from the next stream, and then 3 items from the third stream. OK, pretty simple. We're just doing 1, 2, and 3, and so on. Does that make sense? I'm going to build on this so any questions? OK this was colorful but this totally beautiful yet. So what's beautiful about this?

well let's see how you might write a matrix transpose. OK, something you guys have probably all written at one point in your life is transposing a matrix. Let's say this matrix has M rows, and it has N columns going across. And we're starting with a representation in which the stream is basically, I think this is row major order. The first thing that you're doing is going across the rows before you're going to the next the next row. I'm sorry you're going across the columns, and then down to the next row. So it's zigzagging like this. And you want to pass through a transpose, so you zigzag the other way. You do the first column, up, and then the next column, and so on. And this comes up a lot in a stream program.

So it turns out you can represent this as a splitjoin. Oh that's not good. Just in my moment in glory as well. OK, yeah slides. You can download slides. Actually I can do this on the board. This is the thinking part anyway. Yeah,

could you just bring up GMail? I have a backup on GMail. Can we focus on the board? So this is going to be a little exercise anyway. OK, here's what we had. We had M, M rows, N columns. We started with an interleaving like this. Right, and we want to go into a splitjoin. And this will be a round-robin construct. And what I want you to do is fill in the round-robin weight, and also the number of the streams. And you can have a round-robin at the bottom. And when it comes out, you want the opposite interleaving. This is M and N.

OK so there are 3 unknowns here, what you're doing the round-robin by-- can you guys see that over there-- how many parallel streams there are, and what you're joining the round-robin by. Ok so I'm going to give you a minute to think about this. Try to think about this. See if you can figure out what these constants are. You just basically want to read from this data in a row major pattern, M rows and N columns, and end up with something that's column major. What are the values for the constant? Does it makes sense? Ask a question if it doesn't make sense. Yeah?

AUDIENCE: So we assume that values are going to, based on the line that you drew, across it, tests like a stream line?

BILL THIES: Right, right, right. So those values are coming down the stream. You have a 1-dimensional stream. It's interleaved like this. So you'll be reading them like this. And then you the output a 1-dimensional stream that is threading the columns. Does that make sense? Somebody ask another question. Yeah

AUDIENCE: So the actual matrix transpose codes, it's my understand that nobody actually does it sequentially like that because of locality issues. Instead it's broken up into blocks.

BILL THIES: So there are ways to optimize this.

AUDIENCE: And after you've sort of serialized it, can you then capture..

PROFESSOR: Guys, [UNINTELLIGIBLE PHRASE] has a blocking segment. So you can heirarchically do that. So, normally what happens is you do the blocks and inside the blocks, you can do it again. You can do it at two levels, basically.

BILL THIES: Any hypotheses? Anyone?

AUDIENCE: Is it N for first one, M for the second one?

BILL THIES: OK, what do we have

AUDIENCE: N for the first one, and M for the second?

BILL THIES: N,M and

AUDIENCE: Same for the M?

BILL THIES: Same, M?

AUDIENCE: Yeah.

BILL THIES: OK OK, this is a hypothesis. Other hypotheses?

AUDIENCE: N, M, 1.

BILL THIES: OK, anyone else? Any amendments?

AUDIENCE: How about 1, 1, 1?

BILL THIES: 1, 1, 1? OK lottery is closing. Yep?

AUDIENCE: !, M, M

BILL THIES: 1 M, M? 1, M, M, OK and last call? OK I believe two of the ones submitted are correct. Is this and, yeah? OK I think this, N, M, 1, works and 1, N, M, works. So let me explain the 1, N, M, this is how I like to think about it. One way to think about this is we just want to move the whole matrix. You doing OK, yeah? OK we just want to move the whole matrix into this splitjoin. So the way we can do that is have M columns of the splitjoin since we have N columns of the matrix. And what we'll do is we'll just do a round-robin one at a time, from the columns of the matrix into the columns of the splitjoin. So we'll take the first element, send it to the last, next element, next column, and so on. So we get the whole matrix here, Now I want to read it out column-wise. So we'll do a joiner of M. We'll read M from the left stream that'll read all M items from the columns, send it out, and then M items in the next column, and then send it out. Does that make sense? How many people understood that? All right,

So if you think about it, you can also do it in M, N, 1. That's basically, yeah, it's very similar. OK there we were. And yes. We can do 1, N, M. We basically read the matrix down, and then pull it down into column. And it's very easy to write this as a transpose. So we just have a transpose filter in which we're doing nothing. No competition in the actual rows or the actual contents of the splitjoin. And we just split the data by 1, have N different identity filters, and then join it back together by M. Any questions about this? An interesting way to write a transpose.

OK so there's one more opportunity to shine here. And that's a little more interesting permutation called a bit-reversed ordering. And so this comes up in an FFT and another algorithm. The permutation here, is that we're taking the data at the index n. And let's say n has binary digits b sub 0, b sub 1, up to b sub k. And we want to

rearrange that data, so this data goes to a different index with the reversed bits of its index. So if it was and index n before, it ends up at b sub k down to b sub 1, b sub 0. So for example, let's just look at 3-digit binary numbers. If we have 0, 0, 0, this is 1 input item. We're reversing those digits, you still get 0, 0, 0. Item at index 1 will be 0, 0, 1. We want to reorder that to index 4. OK, 1, 0, 0, 0, 1, 0, stays the same, 0 1, 1, goes to 1, 1, 0 shifts over. And from there on, it's symmetric.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

BILL THIES: Sorry?

AUDIENCE: [UNINTELLIGIBLE PHRASE].

BILL THIES: OK. So here I'm writing the indices. So I'm not writing the data. So index 0, 1, 2, 3 up through index 8., or index 7. OK and on the bottom you have indices 0 through 7. So the data will actually be moved, reordered like that. Does that make sense? It's a reordering of data. Does this transformation make sense? Other questions?

OK, it turns out you can write this as a splitjoin. And you just need 3 different weights for the round-robins. OK round-robin with 1 weight on the top here, and two different round-robin weights on the bottom. And here I'm assuming you have 3 binary digits in your index. So you're reordering in groups to 8. OK, so let me give you a second to think about this. What are the values for these weights? I'll give you a hint in just a second, or ask another question. Yes?

AUDIENCE: [UNINTELLIGIBLE PHRASE].

BILL THIES: So what we're doing here, is we're exposing the communication pattern. That's the thing. If you write this in an imperative way, you end up basically having your, you're conflating basically the data dependencies with the reordering pattern. So what I'm trying to convey here, is how you can use the streaming model to show how you're sending data around. Because when you're on an architecture-like cell, everything is about the data motion. You're taking data from one place and you're trying to efficiently get it to the producers or the consumers. And you really need to-- the compiler needs to understand the data motion. And also, it's just another way of writing it, which I think it's actually easier to understand once you see it. It's a way to think about the actual reordering from a theoretical standpoint

Any wagers on this? So what we to do, if you think about the bit-reverse ordering, what we want to do is distribute the data by the low-order bits. And then gather the data by the high-order bits. So you want a fine-grained parity when you're shuffling. You can also do it the other way around. It's totally symmetrical. But one way to think about it is, you want a fine-grained parity when you're distributing data, and then a course-grained when you're coming

together. Anyone see it? Give you ten more seconds. Come on? Yeah it is a little bit tricky.

OK well let me explain how it works. So 1, 2, and 4. Ok so what these round-robin 1 splitters do, is these are basically the fine-grained parities. So OK, the first round-robin, that will send all the even bits to the left, and all the odd values to the right. Right, that's the lowest order bit. Because it's doing every other one, shuffling it left and right. So this round-robin is seeing only the even values. Now it's going to split them up based on who's divisible by 4. Now we'll go to the left or go to the right. This is basically shuffling in the order of the bits, from low-order bits to high-order bits. So these will be ordered in terms of their low-order bits. And now we just want to read them out from left to right. Just take the order that we made with those round-robins, and read them out from left to right. And since they have 8 values, you can do that just by chunking them up. We'll read 2, 2, and then we'll read these two, 2 and 2, and now put all 8. Does that make sense?

OK, so yes, it's a bit clever. So I think it's a nice way of thinking about what a bit-reversed ordering means. And you can write this in a very general way. You just have a recursive bit-reversed filter for N values. And base case, you only have 2. So there's no reordering to do when you when you think about it. So you're not doing any computation. Otherwise yo have a round-robin split in half, and then have a coarse-grain joiner. So, you get a structure like this. If you're, as you're building up, just distributing and then bringing back together in a course-grained way. OK. Let's see how do I don't do this? OK so one thing to notice, there's one more example of a splitjoin. Question?

AUDIENCE: [UNINTELLIGIBLE].

BILL THIES: OK so in general, at the base of this hierarchy, we could've added some other filter to do some competition. Identity just means we're doing no computation. It's a predefined filter that just does nothing.

PROFESSOR: On complex data.

BILL THIES: On complex data. Sorry this is a templated filter. So we're reordering complex values. And we're passing the input to the output. Amir?

AUDIENCE: [UNINTELLIGIBLE]

BILL THIES: In general the language does not have support for templates. We only do it for these base classes. That's more of an implementation detail. Yeah.

AUDIENCE: [UNINTELLIGIBLE].

BILL THIES: Right now there isn't, but nothing fundamental there. Yeah? Other questions? Yeah?

AUDIENCE: How did you know that there are two filters after that?

BILL THIES: Two filters, sorry here?

AUDIENCE: [UNINTELLIGIBLE].

BILL THIES: OK. OK. So we have two add statements between the split and the join. So that branches to two parallel streams. Is that your question?

AUDIENCE: Yeah. How do you that theorem, that there's not like three.

BILL THIES: So the compiler will analyze this, the compile time. And it'll know these values of N and propagate them down at compile time. So it'll basically symbolically evaluate this code, see there are two branches, and you can unroll this communication pattern.

AUDIENCE: I think another way of thinking about it is, each add statement essentially adds another branch in your splitjoin.

PROFESSOR: That is another box.

BILL THIES: It's about to get clear actually. Other questions?

AUDIENCE: [UNINTELLIGIBLE]

BILL THIES: That's one way to think about it, yeah. Wait say again, counting?

AUDIENCE: A rating sort.

BILL THIES: A rating sort, right. OK, well is it sorting? It's not really sorting.

AUDIENCE: Well it's not really sorting.

BILL THIES: It's a permutation.

AUDIENCE: Could you do a rating sort?

BILL THIES: You could do a rating sort. So actually, what I want to show next is how you can morph this program into a merge sort by changing only a few lines. OK look carefully. Don't blink. OK there's merge sort. So very similar pattern. This is one of those idioms. It's a recursive idiom with splitjoins. But now in the base case, we have a sort. So we would basically branch down. What we ended up with was a sort in the base case. We're just sorting a few values. And we call [? merge sort ?] twice again, and then we do a merge. So instead of identity at the base

case here, we now have a basic sorting routine. And we merge those results from both sides. And the only thing I changed in terms of the communication rate, is to be more efficient we just distributed data in chuncks instead of doing a fine-grained splitting. Actually you do it however you want in a merge sort. But this is chunked up. And let's just zoom in here. This is how a merger sort looks in StreamIt. So we split the data two ways, both directions, come together, do a sort on both sides, and then you merge.

And so by having the, you know you can interleave pipelines and splitjoins like this. So you have these hierarchical structures that are coming back together. Does this make sense? OK. I'm going to hold off on messaging actually. Let me see, what do I want to cover? OK let me actually skip to the end. Oh, I can show you this. Yeah, I'm going to cut short a little bit. So here's how other programs look written in StreamIt. OK, you can have a Bitonic sort. OK so you see a lot of these regular structures. And the compiler can unroll this and then match it to the substrate. This is how and FFT looks. It's quite an ah elegant implementation of an FFT. It'd be good to go into in more detail.

You can do things like block matrix multiply. You don't always have to have column or row-wise ordering. It's natural to split things up like this. We have a lot of DSP algorithms, the filter bank, FM radio with equalizer, radar array front end. Here's an MP3 decoder. And let's see, I'm going to skip this section and just give you a taste for the end here. I'm skipping a hundred slides. Yeah.

OK so if I give you a feel. Our biggest program written in StreamIt so far, is the complete MPEG-2 encoder and decoder. So here is MPEG-2 decoder. And I think you've seen block diagrams of this already in the class. And so it's a pretty natural expression. You can really get a feel for the high-level structure of the algorithm mapping down. And for example, here on the top we're doing the spatial decoding looking inside each frame. Whereas at the bottom we're doing the temporal decoding between two frames, the motion compensation. And one thing that I didn't have a chance to mention, is that we have a concept of teleport messaging. What this means is, I showed you how the steady state flow data goes between these actors in the stream. But sometimes you want to control the stream as well.

For example, this is a variable length decoder at the top. It's parsing the input data. It might want to change how the processing is happening downstream. For example, say that you have-- you know in this case you have different picture types coming in. And you want to tell other components to change their processing based on a non-local effect. And that's hard to do if you just want static data rates. But what we have is this limited notion of limited dynamism, where you're basically poking into somebody else's stream. And we let you do that very precisely. I don't have time to go into the details, but you can basically synchronize the arrival of these messages with the data that's also flowing through the stream, And so in this case, were sending through the picture type. And it really simplifies the program code. I didn't have time for details, but why don't we put in the slides anyway, if

you're interested.

And if you do a similar communication pattern in C, it's a little bit of a nightmare. You have all these different, basically memory spaces, different files. And the control information is basically going left and right all over. So this really helps both the compiler and the programmer as well in StreamIt.

So it's all implemented. It's about 2,000 lines of code in StreamIt. Which is about 2/3 of the size of the C code, taking into account similar functionality there. And it's a pretty big program. You can write 48 static streams. And then we expand that to more than 600 instantiated filters. So this gives you a lot of flexibility when you're trying to get parallelism. Question?

AUDIENCE: When a compiler downloads all bytes?

BILL THIES: Oh the object code, you mean. OK, so right now our current implementation, we duplicate a lot of code. So it end up being bigger than it needs to be. There's no reason for us to do that. That's kind of a-- we have a research compiler that make that easy.

AUDIENCE: So object-wise , its not data.

BILL THIES: Object-wise we still need to do that comparison. Yeah that's a good question. Yeah. Other questions? OK so let me cut to the end. OK, so we have the StreamIt language. And we think it really preserves the program structure. It's a new way of thinking about how you orchestrate the data reordering with the splitjoins, showing you who is communicating to who, and how you can stitch together different pieces in your program development. And again, really our goal is to get this scalable multicore performance. But you can't get people on board just on a performance stat. You need to show them a new programming model that actually makes their lives easier. So that's what we're working on. And thinks with listening.

[APPLAUSE]

BILL THIES: Any last questions? Yes?

AUDIENCE: So in the anti-decoder, you have a lot of computations size that were not sequential streams. Like for example, the output of the distinct cosine transform is not a stream of pixel, you are going to have coefficients and things like that. Which are a logical sort of chunk.

BILL THIES: Yes so depending on the granularity of the competition, you don't need to pass individual values over the stream. For example, you can have a stream that inputs the whole array at a time. And so we basically advocate that if you have something that course-grained, you should be passing an array or a macroblock, in the

case of MPEG, or a set of coefficients in a structure. So when you have coarse-grain parallelism, you write your program in a course-grained way. The fine-grained things I showed for the bit interleaving and so on, is more for the fine-grained programs. AUDIENCE: Can you do both? In the sense that can you stream over an array, so it's stream of stream, so to speak.

BILL THIES: So there's an interesting multidimensional problem there. Right now we've taken a 1-dimensional approach. So far it's basically the programmer has to set an iteration order, and end up with a 1-dimensional stream coming into and out of every filter. We're working on extending that. Yeah, but when you have basically streams of 2-dimensional data, you like the freedom to either iterate basically in time or in space, depending on what you're doing. And so I think that's more of a research problem. So far we're just been doing a 1-dimensional representation. Yeah good point. Other questions? Yeah?

AUDIENCE: Why did you decide on this synchronous dataflow model as opposed to something more general?

BILL THIES: So our philosophy has been that you want to start with the most kind of basic block of a stream program, and optimize it really well. And then you can stitch those together into higher level blocks. So we think of synchronous dataflow as being kind of the basic block of streaming. You know what's coming in, you know what's coming out. And even if you have a more general model, they'll be pieces that fit under their synchronous dataflow model. And so we saw a lot of optimizations opportunities in there. And really knowing those IO rates can let you do a lot of things that you can't do in a general model. So I wanted to get the simple case right first. And actually kind of our focus now is on expanding, and how do you look at the heterogeneous system, and how do you optimize a more dynamic system. Yep. Other questions? OK. Yes. You can check out our web page. Yeah if you Google for StreamIt, I'm sure you can find it. Yeah, we have a public release. Yes, send us any problems. It's actually it's a good test. We want to make sure it works for everyone. But I mean, we've had, you know, hundreds of downloads. There are a lot of people using StreamIt. It shouldn't break if you download it. Yeah. OK good. Thanks.