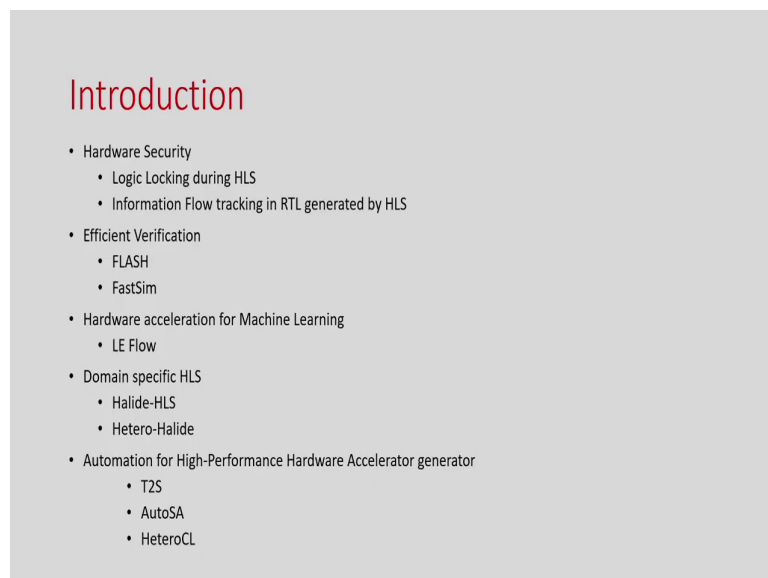


C-Based VLSI Design
Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 12
Recent Advances in C-Based VLSI Design
Lecture - 43
Recent Advances in C-Based VLSI Design

Welcome everyone. So, in today's class, we are going to discuss some Recent Advances in C Based VLSI Design. So, we are at the end of this course and we have already learned so many things in high level synthesis or in C based VLSI design. So, I am going to emphasize the things that have been happening in the last two to three years in the context of C based VLSI design.

(Refer Slide Time: 01:18)



So, there are five areas which is very active in terms of research in C based VLSI design; they are basically security, hardware security in context of high level synthesis or C based VLSI design, verification of this whole process, and then hardware acceleration for machine learning, and domain specific high level synthesis, and automation for high performance hardware accelerator generator.

So, these are the five areas in my opinion that are very active in the domain of research in the context of C based VLSI design. So, what I am going to do is, I am going to just take each topic at a time and I am going to talk about what are the recent works and what are the kind of things happening in that particular domain.

(Refer Slide Time: 01:58)

Hardware Security

- Logic Locking during HLS
 - TAO
 - C. Pilato, F. Regazzoni, R. Karri and S. Garg, "TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis," 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1-6
 - HOST
 - Chandan Karfa, T. M. Abdul Khader, Yom Nigam, Ramanuj Chouksey and Ramesh Karri, "HOST: HLS Obfuscations against SMT Attack," in 23rd Conference on Design, Automation and Test in Europe (DATE'21), pp. -, March 2021.
- RTL attack
 - SMT attack
 - Chandan Karfa, Ramanuj Chouksey, Christian Pilato, Siddharth Garg and Ramesh Karri, "Is Register Transfer Level Locking Secure Against SAT Attacks?," in 22nd Conference on Design, Automation and Test in Europe (DATE) 2020, pp. 550-555, March 2020.
- Information Flow tracking in RTL generated by HLS
 - TaintHLS
 - C. Pilato, K. Wu, S. Garg, R. Karri and F. Regazzoni, "TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 798-808, May 2019,

So, let us start with hardware security. So, we have already discussed in this course itself, that because of this fabulous production of the ICs your IP or intellectual property of the design host goes to a third party. And there may be some rogue employee there or some untrusted employee there, they might steal the IP; they can do some overproduction or they can do IP piracies.

And to stop that, logic locking is something a very popular technique and is a very novel technique evolved. And in this course also we have discussed that, this logic locking is basically nothing but you add some extra key in the circuit and you give that locked circuit to a third party, but do not give the key. When you ship that whole IC, you just store that particular key in a non volatile memory and you just give that IC functional IP to the customers.

Since that correct key is already there in the functional IP, it will give the correct output. But since that locked circuit, which the untrusted employee, untrusted component access;

he does not have the access to the key, he cannot use it basically. So, that is the idea of logic locking.

And we have seen that there is a plethora of work happening in the gate level logic locking; say how to lock the circuit at the gate level, how to break it and so on. And then there are recent works, specifically this TAO and HOST that is mentioned here is basically trying to do the logic locking at the time of high level synthesis.

Specifically TAO says that if I give a C code, I am going to give you RTL code, which is locked with some extra key, maybe 200, 500 whatever the number of keys. And HOST is basically take the RTL which is generated by high level synthesis and you try to lock that RTL circuit So, it is basically not during high level synthesis, but it is basically at the end of high level synthesis; but this HOST supports only RTL, which is kind of generated by high level synthesis tool.

So, these are some recent work on locking the circuit during higher level of abstraction or during high level synthesis. And also as I mentioned during discussion of the course that, whenever you come up with some kind of locking technique, there is always a scope of checking whether the locking is robust or not. So, basically there are some attack methods that have also evolved.

So, in the recent work is a SMT attack which is basically proposed in this paper, which basically can break TAO. So, and then basically this HOST evolved which basically says that, this SMT attack cannot break this HOST, . So, whatever the drawbacks that SMT attack identifies in the TAO, it basically resolves those issues.

So, this HOST is a technique where you cannot apply this SMT attack directly. So, this kind of work is happening in this domain; there is and I mean actually I have covered all this I mean TAO I have covered, SMT attack I have covered; HOST I have not covered, but that is something similar to in TAO line.

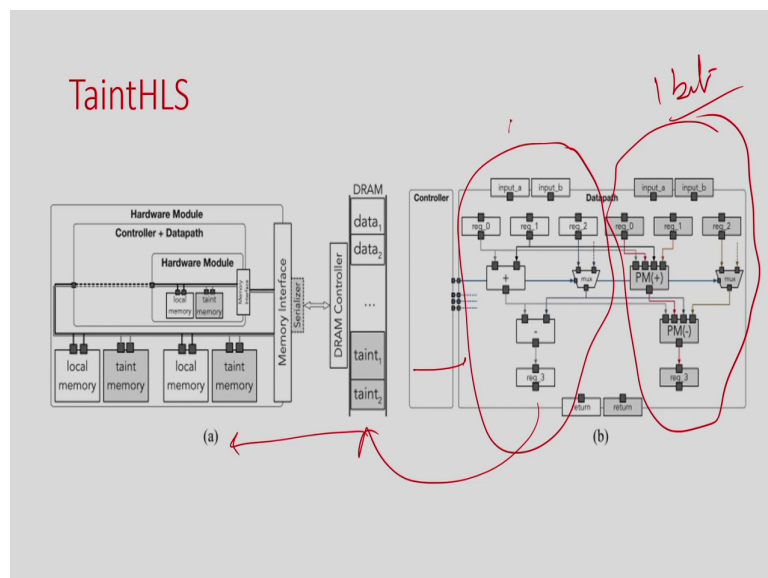
And another important aspect in this hardware security is information flow tracking. So, and is a very well researched area, al. So, basically you have a circuit which is

happening, you have a processor which is running and you try to see if any secure data is going to some unsecure component of the whole processor.

So, basically as an in flow of information from higher security to lower security objects. And which is very very common and there is a plethora of work happening again here in the context of processors, hardware and all. So, and then basically in this particular work, this TaintHLS what basically is trying to do that.

I can actually generate RTL which is from a C code, which is the RTL plus the information flow tracking things already embedded into the RTL. So, I now have a RTL, which is generated by a high level synthesis tool from input C code, but in that particular C code or in that particular generator RTL can actually track the information flow.

(Refer Slide Time: 06:19)



So, the basic idea is like this. So, you have the data path and it actually adds a shadow circuit. So, this data path is say maybe 32 bit, 42 bit with data path, but this is a 1 bit kind of; it is basically you have to just check yes or no, whether a sensitive data flow to an unsecured component yes or no, it is kind of Boolean. So, this is actually the data path.

And then this is the overall architecture, then basically you have this common data path and it automatically adds the shadow data path, where we actually take the information flow. And the whole thing is basically have a controller which control both and then this data also come back to the the memory; you just store this ten value and whenever you kind of have some kind of taint, is basically the information flow, whenever you there is a you identify there is a violation of information flow from higher to lower security, probably you can actually give a kind of alarm signal.

So, that information flow tracking is not something new; but what is new is basically how we can actually generate a circuit, which is kind of auto, which can track the information flow within the circuit itself. So, that is again another interesting work. So, this is what is happening in the context of hardware security in high level synthesis and now let us move to the next one.

(Refer Slide Time: 07:44)

Verification of HLS

- Fast Simulation based Verification
 - FLASH
 - K. Choi, Y. Chi, J. Wang and J. Cong, "FLASH: Fast, Parallel, and Accurate Simulator for HLS," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4828-4841, Dec. 2020
 - FastSim
 - M. Abderrahman, J. Patidar, J. Oza, Y. Nigam, T. M. Abdulkhader and C. Karfa, "FastSim: A Fast Simulation Framework for High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, June 2021.
- Formal Verification
 - Phase-wise
 - R. Chouksey and C. Karfa, "Verification of Scheduling of Conditional Behaviors in High-level Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1-14, 2020.
 - End-to-end

Another very important, very active area of research is verification of high level synthesis. So, we have seen that whenever we generate RTL from a C code; because of the completely different semantics, different ways of execution of those behaviors. So, checking functionality whether the generator RTL is functionally equivalent to the input C code or not is extremely important . So, there may be some bugs introduced for many reasons.

Though there are two ways we usually verify, a simulation or formal verification. And in the context of formal verification of high level synthesis, till now there is no automated end to end formal verification tool, that does not exist basically, which basically take the C code and RTL without taking any information from the tool, you just check whether they are equivalent or not. So, that still is evolving.

So, we do not have any very mature tool there, there is some paper proposed; but they are not something robust enough to handle all kinds of optimizations that a high level synthesis tool offers. So, in the context of formal verification, the most popular approach is basically phase wise verification and I have actually covered both the topics in this course. And then we have seen that basically among all these verification steps, scheduling verifications is basically the most important and the most kind of decision made during scheduling.

So, verifying scheduling is very important and a lot of work is happening. I just cited here one of the recent work from our group; but if you just go into this, you will find many such papers are coming on this phase wise verification of high level synthesis.

In the context of simulation based verification, again basically we have to simulate the generator RTL with some input; I have to simulate the C code with the same input and you just compare whether they are giving the same output or not. So, that is the kind of simulation based verification.

And I have already explained in this course itself that simulating RTL is much much slower than simulating a C code. So, because this RTL is basically an RTL simulator that cycles accurate a lot of detailed information of the data path controller, so it has to go through all things, so it is basically slower. And in general it is basically 300 times slower than the C simulator; because C simulation is done with, say, any C compiler like GCC or LLVM whatever it is.

So, that something is a little bit of a bottleneck and so, this is what the recent work is trying to do, it is basically trying to do a fast simulation of high level synthesis. So, the basic idea that all this recent work is happening is basically you try to extract a C like

behavior from this high level synthesis output and then you try to simulate using the C compiler. So, then it will be 300 times faster.

But you have to make sure that your generated C code is not the input C code, rather it is a cycle accurate, it is a cycle accurate C code and it actually can give a performance estimation. You can give all the things that this RTL simulator gives. So, you have to make sure that those are the features available in the generated C code.

So, in this there are two recent works which are very important in this context; one is Flash and one is FastSim. In the flash what it does is basically, it generates a C code from the schedule after scheduling; because it just argues that, because once you do the scheduling everything is fixed. So, it basically takes the intermediate information from the tool after scheduling and from there you just generate a schedule C code and then it simulates.

So, that is what this flash does, but as I argued in this particular paper FastSim which is from our group is basically; if you just verify this, it only verifies the scheduling, it does not verify the allocation binding or the data path control generation phase. So, and also it has to take some intermediate information, scheduling information, that may not be available also in some cases.

So, what FastSim does is very important and very I would suggest that it does the very important job in the context of simulation based verification is basically it does not take any intermediate information in the tool, rather it takes the output RTL and from there it generates a C code.

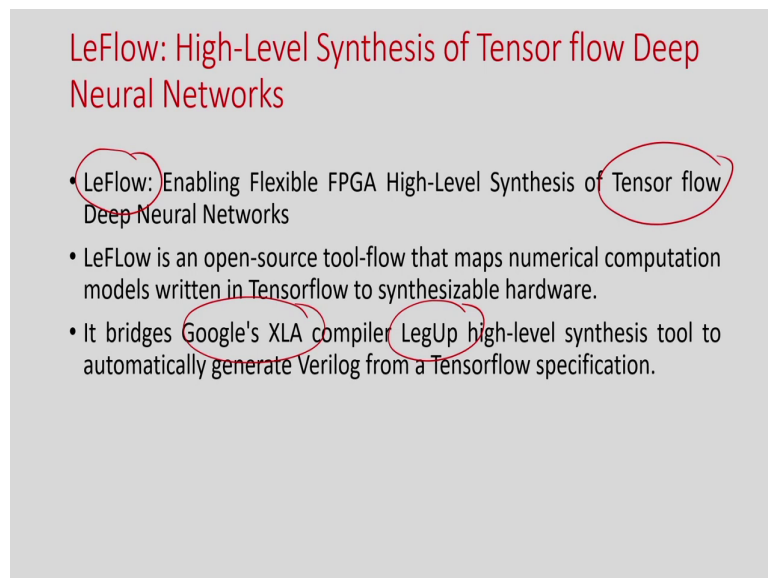
And that it is actually shown in this paper that, it is basically cycle accurate it can give accurate performance estimation. It can handle all the optimizations that the modern high level synthesis tool offers. And then you can use that particular C code for faster simulation using say C compiler.

So, I have already covered the basic intuition of FastSim. So, basically you have a RTL to C converter. So, that I have already discussed in this class. And so, with that we have

we can show that we can actually do this simulation based verification of high level synthesis very fast.

So, this all about this verification of high level synthesis, I mean there are lot of research to be done in this front I would say and also in this places also there are so many complexities are there, which is not get handled by the current the synthesis tool; like this data flow, systolic array, implementation. So, many things are happening internally which are not supported by this particular existing works. So, we have a lot of scope to work in this formal verification context of high level synthesis.

(Refer Slide Time: 13:06)



LeFlow: High-Level Synthesis of Tensor flow Deep Neural Networks

- LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensor flow Deep Neural Networks
- LeFlow is an open-source tool-flow that maps numerical computation models written in Tensorflow to synthesizable hardware.
- It bridges Google's XLA compiler LegUp high-level synthesis tool to automatically generate Verilog from a Tensorflow specification.

So, let us move on, the next one is basically to do a kind of hardware generation for machine learning algorithms, which is done by the tool Le flow. So, basically if you write how do you write machine learning code? So, we usually write in tensor flow.

So, we have to write our code using python in tensor flow and then we have that code and we use it for training and other purposes. So, now, our objective is to try to generate a RTL for that particular tensor flow code.

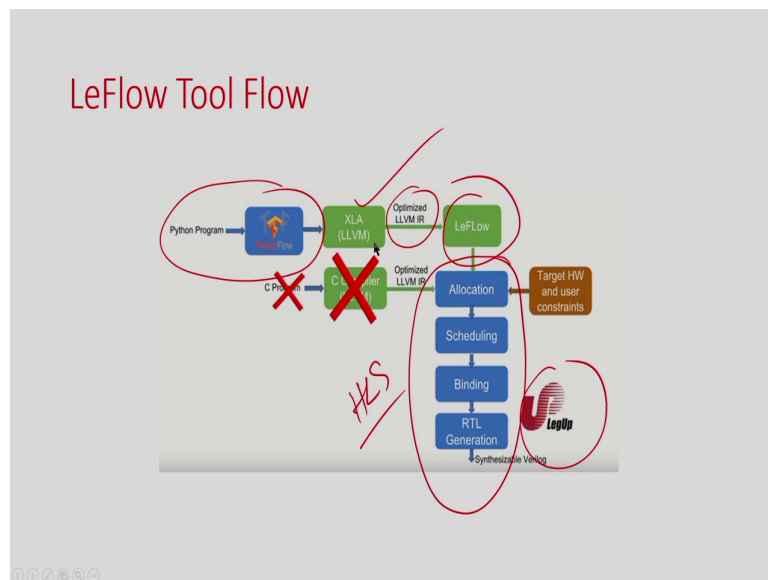
So, what should we do now? We have to manually write the corresponding C code. So, because the tensor flow code is very high level code it has some APIs, you just use it and it you just use an API convolution and so other operations. But here you have to

manually rewrite the corresponding code in C, then only this high level synthesis code tool can take it on.

So, what this Le flow does is very interesting. It basically does not evolve any new high level synthesis tool; it just uses an existing high level synthesis tool called LegUp, . What it does is basically take the tensor flow code, use Google XLA compiler and convert that tensor flow code into some IR, LLVM IR or some intermediate representation; and which this high level synthesis tool can understand.

So, basically if you understand the high level synthesis tool what it has; it basically has a c compiler at the start. So, which basically takes that C code and converts into some IR and from that IR the convention process goes on. This scheduling, allocation binding and all the physics has gone on. So, now the point here is that, if you have some way; you can just represent the tensor flow code into this IR form, then a high level synthesis tool will understand that and that is what this tool does.

(Refer Slide Time: 14:58)



So, basically the flow is like this, you have this Python program and then it uses the XLA compiler of Google and it basically generates some IR which is LLVM. LLVM will be another C compiler. So, it can be GCC for example, if it is vivado HLS can basically

understand the IR which is generated by GCC, but there are other tools like LegUp which basically understand the LLVM IR.

And this is nothing but this HLS tool; this is nothing but HLS. So, what it does is basically convert this into IR; but what this LeFlow does is very important. So, this IR, what it generated from the tensor flow code; so high level synthesis tool does not support many of this IR.

(Refer Slide Time: 15:45)

Le-Flow

- Google's XLA compiler which emits LLVM code directly from a Tensorflow specification
- Although the XLA compiler outputs LLVM IR, and the LegUp tool generate hardware from LLVM IR, there are several transformations to the IR that need to be made in order to ensure a seamless interface between the two tools

D. H. Noronha, B. Salehpour and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of TensorFlow Deep Neural Networks," *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, 2018, pp. 1-8.

<https://github.com/danielholanda/LeFlow>

So, basically XLA will give you IR. So, then you might ask what are the other things to be done. So, you generate IR and then use LegUp; but the problem here is that there are several transformations to be made in the IR to make it a seamless interface between these tools.

So, basically it does not mean that whatever the IR did generate that a LegUp understood. So, you have to understand certain IR, which does not is synthesizable or say it does not, LegUp does not understand those syntax. So, you have to convert those IR into some equivalent representation.

So, that is what this LeFlow does; it identifies those gaps between this IR versus the IR it understands and basically bridges the gap. So, with this is basically an open source tool and basically you can from any tensor flow code, you can generate the hardware what you want. So, you can actually follow this paper and the tool is given in this particular link.

So, this is about creating a flow from this tensor flow code to RTL. So, we will discuss by the end of this class that we need to do some more to generate efficient hardware; this actually gives you a flow from this tensor flow to RTL, but it does not always give you the efficient hardware. So, we need to do something more in this context and that is what is basically addressed in other works, I am going to talk about that.

(Refer Slide Time: 17:17)

Halide

- Domain specific language and synthesis
- It help programmers write high-performance image processing and tensors code easily on modern machines.
- One of the biggest advantages of Halide is that it decouples the algorithm description of the program from the scheduling — its execution strategy.
- When trying to optimize Halide code, programmers can simply modify the code of the scheduling part without changing the algorithm part to change how the program is executed

So, before going to that particular topic, I will just cover another aspect which is basically domain specific language and synthesis. So, this is something very important; because if you consider specifically the image processing or video processing domain.

So, we have basically a lot of data coming, a lot of images, a lot of video coming; you do some kind of processing , you identify age or you do some other kind of processing and then you basically and that processing is nothing, but a nested loop and doing some

operation. So, there is a computation which is basically embedded in a nested loop ; that is all, that is what this particular application demands.

But here the problem is once you know; we have already discussed in this course as well that, once you try to synthesize this particular application for efficient hardware, the problem happens that you have to manually modify this code a little bit. So, not a little bit or maybe you have to try a different way .

So, sometime you said do some, loop reordering, you can do some loop splitting, you can do some loop interchange and so many loop kind of optimization you can do and you kind of generate different different version of the code and then you synthesize and see which is basically goes well in in hardware.

So, sometime you said do some, loop reordering, you can do some loop splitting, you can do some loop interchange and so many loop kind of optimization you can do and you kind of generate different different version of the code and then you synthesize and see which is basically goes well in in hardware.

(Refer Slide Time: 18:56)

Halide

```
void box_filter_3x3(const Image &in, Image &blur) {
  Image blurx(in.width(), in.height()); // allocate blurx array
  for (int x = ; x < in.width(); x++)
  for (int y = ; y < in.height(); y++)
    blurx(x, y) = (in(x, y) + in(x, y) + in(x, y)) / 3;
  for (int x = ; x < in.width(); x++)
  for (int y = ; y < in.height(); y++)
    blurry(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1)) / 3;
}
```

```
void box_filter_3x3(const Image &in, Image &blur) {
  Image blurx(in.width(), in.height()); // allocate blurx array
  for (int y = ; y < in.height(); y++)
  for (int x = ; x < in.width(); x++)
    blurx(x, y) = (in(x, y) + in(x, y) + in(x, y)) / 3;
  for (int y = ; y < in.height(); y++)
  for (int x = ; x < in.width(); x++)
    blurry(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1)) / 3;
}
```

Same algorithm, different organization One of them is 15x faster

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 1 \\ 0 & 2 \end{pmatrix}$$
$$\vdots$$

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \end{pmatrix}$$
$$\vdots$$

For example, I will just give an example here. So, this is a blurring example where you have this x and y and this is the actual compression is happening, So, you have some

array, you are just taking some value and then you calculate something and then you do it again this way.

So, the point here is that, suppose you understand that if you just do the interchange of this loop. So, basically you bring y first and then x; here x is first and then y here, it is y first and x in both cases. So, here by doing this transformation, the locality of reference actually improves and you actually can generate 5x to 15x faster hardware in terms of latency.

So, this is something, if you have to do, you have to take this code, you have to rewrite it. So, this is a very simple example. So, but if you understand that internally this computation remains the same. So, I do not change anything for this computation part; what I just do, I just do the change the schedule.

So, how in where in, basically this if you just interchange this for loop; there are say n square x y operation is happening. So, the order of this operation gets changed that is all; that means, the schedule of this execution on this operation basically gets changed.

So, this particular domain specific language basically does a very important job; what it does basically, it just decouple this algorithm from the schedule. So, whatever the way you do; whether you the way you write this version of the code or this version of the code, computation remains the same, I am not changing anything here, the only thing I am doing is just changing the schedule.

So, if you just separate this schedule and computations; then what will happen? I do not have to rewrite this whole thing. So, what the basic idea here is that, you basically specify your computation first and then you have some option, where you specify which kind of scheduling strategy you want to use.

(Refer Slide Time: 20:55)

Halide

- Decouple algorithm from schedule
- Algorithm: what is computed
- Schedule: where and when it's computed

So, basically it separates out this decouple from schedule and the algorithm is what exactly gets computed that what I just mentioned, schedule when and where it is computed. So, for example here, if you just interchange y and x; the way it is basically this value will be computed it will be different. So, for example, here if it is x and y; so basically you can say that 0, 0 location will be computed, then 0, 1, then 0, 2. This is the order of blur x y ; then 0 ,3 and so on.

Here what is going to happen; since it is y, so it is basically 0, 0; then it will be 1, 0; then 2, 0, it is the first row, this is the first column. So, this is the, so order that is getting changed. So, this is what the impact is. So, this is basically Halide is a kind of a language which basically allows you to write your code, where you can separate out this algorithm and schedule.

(Refer Slide Time: 21:54)



So, it has language, I am not going to detail the language details. So, what you can do; you can just say in blur, these are the two computations. So, if you go back to this blur example, there are two computations; this is the computation 1, this is the computation 2.

So, in this Halide language, instead of writing this for loop or this for loop; you just mention that actual computation is this two. And then here you specify whether you want to execute x first, x loop first or y loop first, what is the range of x, y, whether you want to do a tiling, whether you want to do a vectorization.

So, for example, here it is basically you do this computations for say 256, 0 256, but you do a tiling. So, you just do a tiling, where you basically break this loop into 32, a tile of 32, where you use the x i and y i as the tile and then also within that x i, you do the vectorize y of 8, you try to do a vectorized execution of this x loop and you just parallelize this y.

So, you just specify all those things using this comment . So, this is the language, where you can just specify instead of writing that C code; this C code I am going to just write this. So, I just specify what is my computation and then what is the optimization I want to use. If you are a very experienced user, you know which kind of optimization I should use, so that this particular loop will be useful if you have efficient hardware.

So, basically once you do this, what Halide does is basically is a compiler, it actually generates this actual code. So, this is the C code you want, where you actually have a tiling you see; you have this is the y tile, this is the x tile and then you have this loop which is basically divided into tiles, you just do the vectorizations. So, all this whatever the things you want is getting implemented.

Similarly, this is the next one where you apply this optimization and because of that this is happening. So, the Halide is basically a very important and powerful concept in my opinion; because it basically reduces your effort, because you do not have to anyway you have to write this version of the code.

Writing this version of the code needs a lot of effort, rather you just specify what is the computation; you specify what is the optimization you want and then you give it a Halide and then it will automatically convert this into the code that you want after applying these optimizations.

So, this is what halide, it give you a language to represent certain your computations and the optimizations or the schedule you want and it has a compiler inside, which is basically apply those optimization on the computations and gives you the C code you want, this is very interesting and very useful tool.

(Refer Slide Time: 24:44)

Halide

- <https://github.com/jingpu/Halide-HLS>
- There is currently no way to easily map the vast number of Halide programs to efficient FPGA accelerators
- Option 1: Rewrite Halide programs to hardware oriented DSLs, such as Darkroom, HIPAcc, PolyMage, SODA, HeteroCL
 - Created image processing DSLs and provided compiler tools using a line buffered pipeline microarchitecture to guide their hardware generation.
- Option 2: Halide to RTL
 - Halide-HLS
 - Hetero-HLS

So, the tool is available here and you can actually try, so with your examples. So, now, the question is that, once you generate this code; how do you synthesize it? So, here you can see here, you can probably apply this, you have to do this the basics. There are two options, one is basically you can actually rewrite this whole Halide program into domain oriented domain specific language for hardware domain hardware oriented DSLs domain specific language, such as Darkroom, SODA and all.

And then you use that basically that will allow you to basically represent things in hardware or you basically use high level synthesis to generate hardware, which is very easy ; because this is nothing, but a C code C or C plus plus code. So, from there I can use any high level synthesis tool to generate the hardware. So, this is what this Halide HLS and Hetero HLS does.

(Refer Slide Time: 25:45)

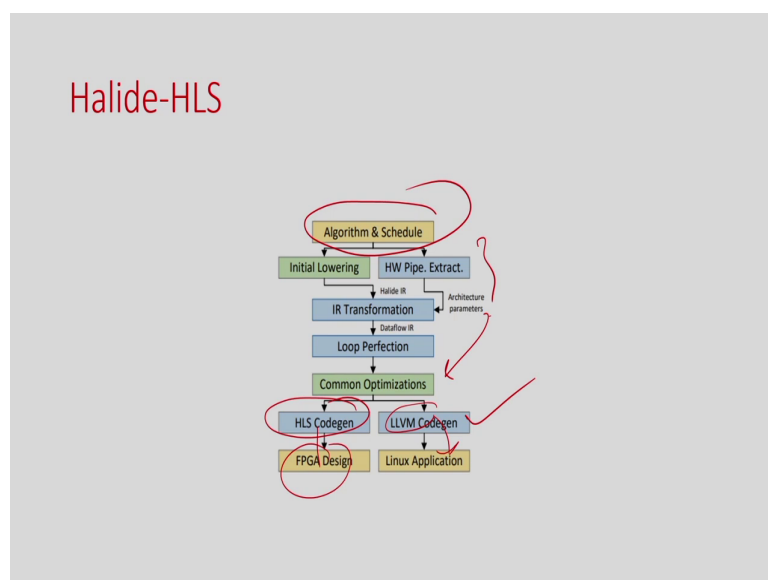
Halide HLS

- It generates synthesizable C and leverages existing HLS tools to create the final Verilog
- <https://arxiv.org/abs/1610.09405>
- <https://github.com/jingpu/Halide-HLS>

Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. ACM Trans. Archit. Code Optim. 14, 3, Article 26 (September 2017)

So, basically this Halide HLS is again it is taking this synthesizable C code and it takes some existing high level synthesis tool. So, you do not have to develop another high level synthesis tool and you generate the Verilog, the RTL. Again this Halide HLS available here, the details of these things you can write get in this paper.

(Refer Slide Time: 26:04)



And then the overall flow is basically you have this Halide code, you have this tool which basically generates this IR and then you basically can use the high level synthesis

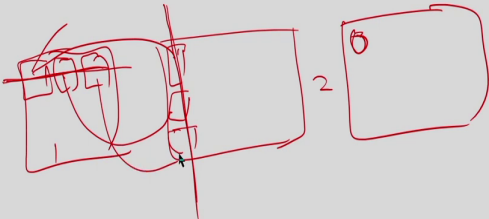
tool to generate the RTL or you can actually synthesize the same thing into the processor. You can run the same code in your processor using that C compiler or C plus plus compiler.

So, that is all about this it is a domain specific language is very important or powerful area and whenever you develop something in that context, you have always option to convert them into RTL using high level synthesis, So, the where this Halide HLS or high level synthesis will be useful in on top of domain specific language like halide.

(Refer Slide Time: 26:53)

Efficient Hardware Accelerator with HLS

- Temporal to spatial conversion is must
- Application-specific systolic arrays (SAs) implemented on modern FPGAs for efficient compute acceleration
 - AutoSA
 - T2S
 - HeteroCL
 - HeteroHalide



The diagram consists of two main rectangular components. The left component is a large rectangle containing several smaller, interconnected boxes and lines, representing a complex hardware structure. A red '1' is written near its bottom-left corner. The right component is a smaller, simpler rectangle with a red '2' written near its bottom-left corner. A red line connects the two components, and a circled 'b' is written inside the right component. The entire diagram is drawn in red ink on a light gray background.

So, let us move on. So, as I mention when I just talk about this LeFlow, although this LeFlow gives you an option to generate hardware from this tensor flow code, it does not have any kind of power to give a very efficient hardware. So, and this is already I have discussed in this course several times that, once you have a code; specifically the nested loops is something where you have to give your maximum attention, array access this is where you have to give a maximum attention, you have to make sure that you can have a efficient synthesis of those loops and arrays.

And the things that we have understood I mean in the class also with the matrix multiplication example that, once you have a big computation is happening; if you just do it sequentially, you are not getting the power of this hardware, specifically in the

FPGA if you think about there are lot of CLBs they are actually running in parallels, lot of DSPs running in parallel, lot of rams are running in parallel.

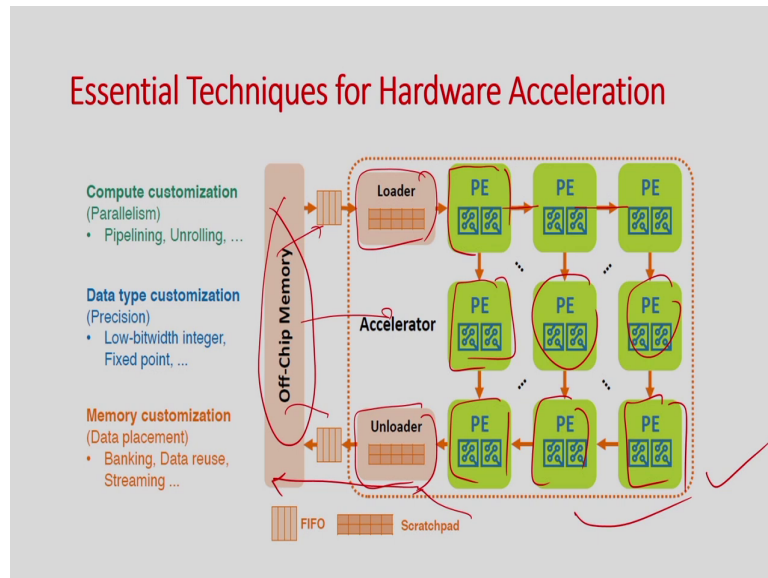
So, unless you kind of parallelize your input C behavior, you would not get the performance. So, for efficient hardware accelerator generation, that is a must. So, you have to somehow parallelize your input behavior, and that is what basically is known as either temporal to spatial conversion. So, spatial is basically parallelized or also it is some form of systolic array generation.

So, if you have an application, you try to parallelize; you basically find out some component, the basic unit component, and try to execute those components in parallel. So, for example, if I just recall my matrix multiplication example again, it is basically two matrices that get multiplied. So, one row multiplies the column and I get one element. So, basically for one element, there is no multiplication needed.

So, and, but to do this (Refer Time: 29:00) I have to access the complete row and complete column. So, what I can do, I can actually break this into small-small components and I can just do this part of this computation one time; next time I will just do this component with this time, then next time I am going to compute this with this time.

So, I can actually split my whole array into this small component and basically you have a multiply and accumulate operation happening. So, I can actually do this and this one time and then I can parallel do this in parallel, do this in parallel and so on.

(Refer Slide Time: 29:33)



So, the basic idea of this efficient hardware generation is the architecture you want in the hardware. So, you have to identify what is this, what is the actual computation is happening just like in matrix multiplications basically multiply and accumulate.

And you can actually break the whole computation into small small components. So, you just break into a smaller component and you can actually connect them in a systolic array format, so that you can actually propagate the data through one component to another component and so on.

So, you run all this PE in parallel and then only you will get the maximum benefit and this is what is called systolic architecture. And then basically your data is in some outside memory and you actually feed them through the FIFO channel to this in the hardware accelerator and where you store the small-small components in the memory in say local RAM.

So, this is the loader and also the output you can actually put in another RAM and you can actually serially go into outside. So, if you just try to map your application into this kind of architecture, then only you will get the maximum benefits. This is what I want at the hardware level which is basically efficient. So, just to, so this is what I want and then

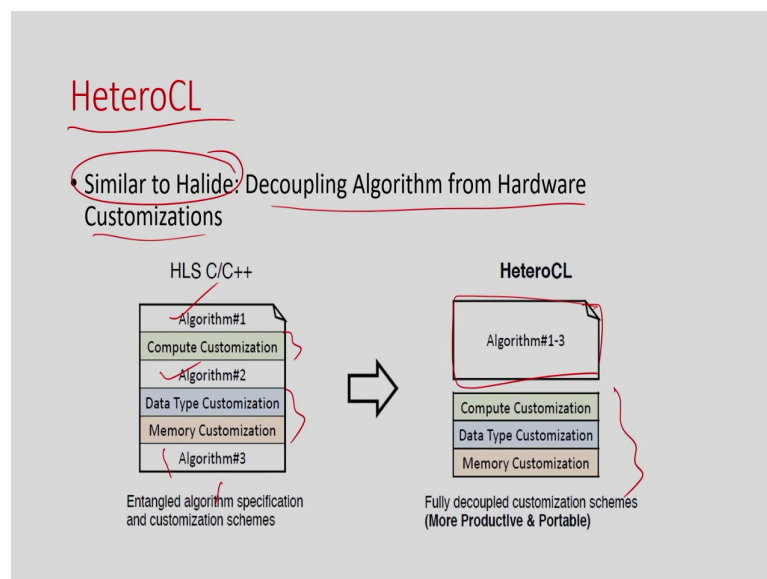
the point here is that; but I have given a secret sequential example. So, I have given a sequential behavior.

So, the question is, how can I achieve this, because this will result in a very efficient hardware. So, how can I generate this? So, for that four tools are available. So, there are tools called AutoSA, T2S, HeteroCL, HeteroHalide, I am going to talk about all of them briefly.

So, there are two avenues of approach; here the first approach is that, is basically just like halide, this tool allows you to write your computations and the schedule differently. So, it has the option to specify tiles just like Halide; how to vectorize, how to parallelize those comments are there and you have the computations and then this tool just like Halide compiler, you can generate this kind of architecture. So, that is one approach and these three tools are basically on that domain.

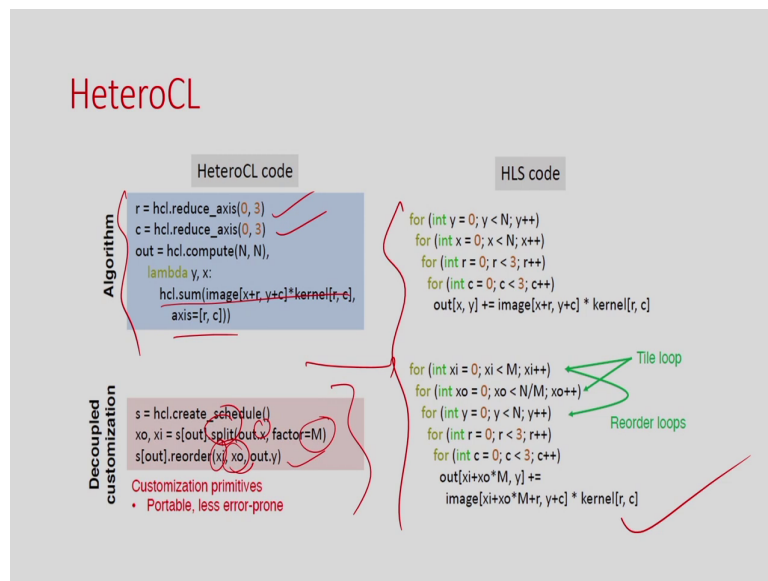
AutoSA basically tries to automate this. So, it does not ask you to rewrite your application in a specific language just like Halide; what it does is, basically you take your C code, you do some kind of this analysis and it generates this kind of systolic architecture. So, let me just go through them one by one.

(Refer Slide Time: 32:20)



So, in the HeteroCL what I just talked about it is basically similar to halide, it is basically decoupling this algorithm from hardware customizations. So, the example that I have given here is algorithm 1 you need this kind of customizations, algorithm 2 you need this kind of customizations and algorithm 3 you have something else. So, what you can do is specify the algorithm as a computation and then you specify all the customization you need together . So, this is similar to this Halide.

(Refer Slide Time: 32:51)

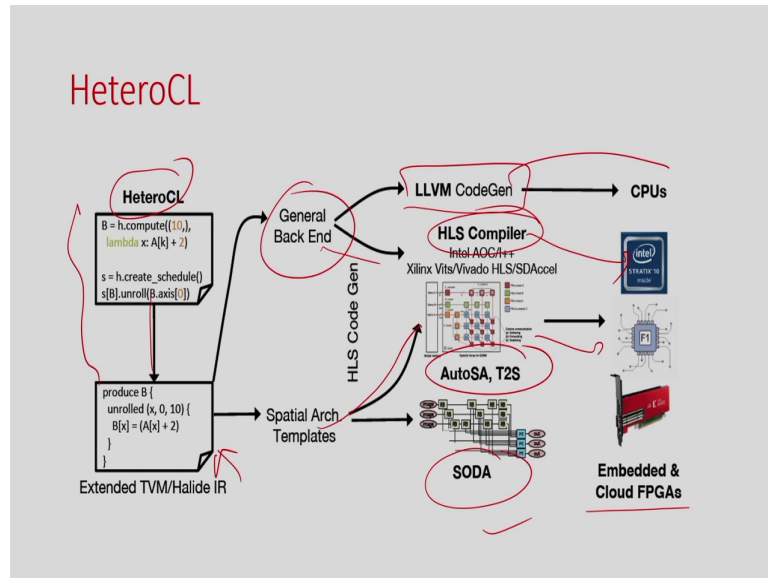


So, here is an example. So you just specify your algorithm here, you have this is the range and you try to compute this and then you specify here that you split this particular x with a factor of M, you reorder x i and x o and you generate this.

So, this is the I am not going to detail of the language, but this is similar to Halide; you can understand that you specify your computation, specify your optimization that you want or the customization you want and it basically generate this code for you , this is what, this is what this HeteroCL does, . So, and this is basically when you try to synthesize in hardware, you will see that this actually can generate this kind of architecture.

So, this is what HeteroCL and then from there you have the option to go into RTL. This is the overall flow here.

(Refer Slide Time: 33:47)



So, you have the option to specify this in the HeteroCL language and then it basically or internally converts into that IR, which is basically after applying these optimizations into this code. And then in the back end you can actually use the LLVM code generator and you can go to CPU, you can run in CPU or you can actually go into some high level synthesis tool like Vivado or Intel tool and you can run it into FPGA.

Or you can actually from this code you can apply a AutoSA, I will going to talk about that or T2S and you can generate things for FPGA and also you can write some domain specific; you can represent this in some domain specific language in hardware level and you can run it in some embedded or cloud FPGAs. So, this is the overall flow of HeteroCL. But the basic idea here again is, applying this kind of optimizations, creating a systolic kind of architecture from a C code is something that is kind of achieved here.

(Refer Slide Time: 34:44)

Spatial
T2S: Temporal to Spatial transformation

- Massive compute resources, plus memory, distributed over a 2-D plane
 - Exploit massive parallelism, and minimize data movement
 - Potential for ~~big boost to power efficiency~~ and thus performance

FPGA

PE PE PE PE PE PE PE PE
PE PE PE PE PE PE PE PE
PE PE PE PE PE PE PE PE
PE PE PE PE PE PE PE PE

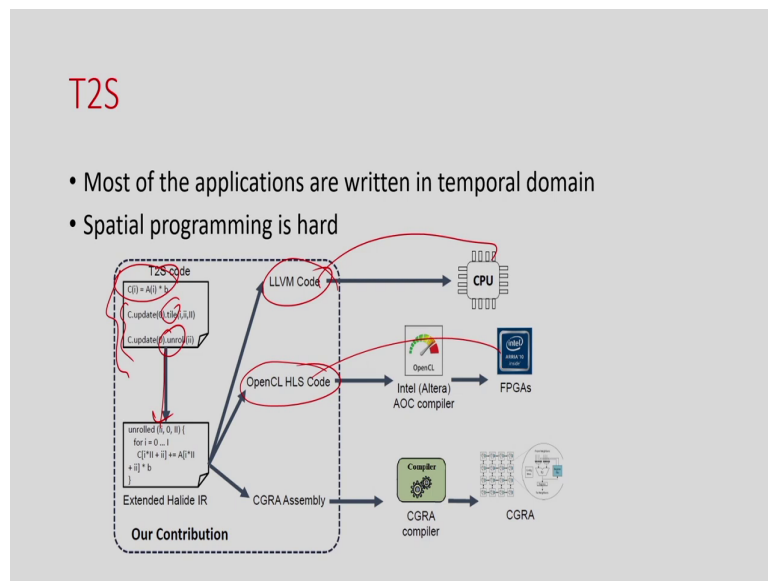
L1 Cache

L2 Cache Slice

T2S also has a similar thing, basically again is similar to HeteroCL in that, you have this in the FPGA board there are so many parallel blocks ; these are the CLBs and they are connected and some of them as DSP, some of them are ALUs and all. And so, basically if we can break my whole behavior into this kind of small small component that I have shown in the previous slide as well and then I can map them into parallel modules.

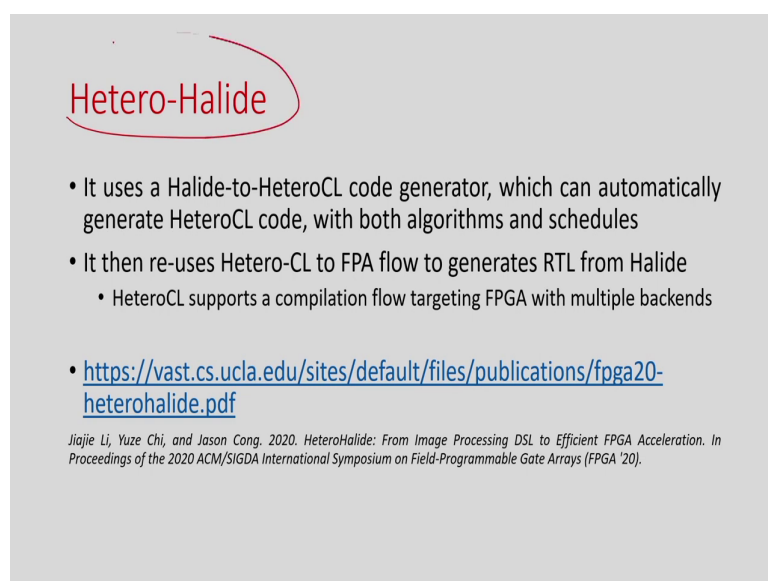
So, this is the idea. And then you have this L1 cache, L2 cache which is to handle this, manage this locality of reference of data. So, this is what basically we need. So, basically you have a sequential code or temporal code, you have to convert into spatial. So, that is what this tool does.

(Refer Slide Time: 35:36)



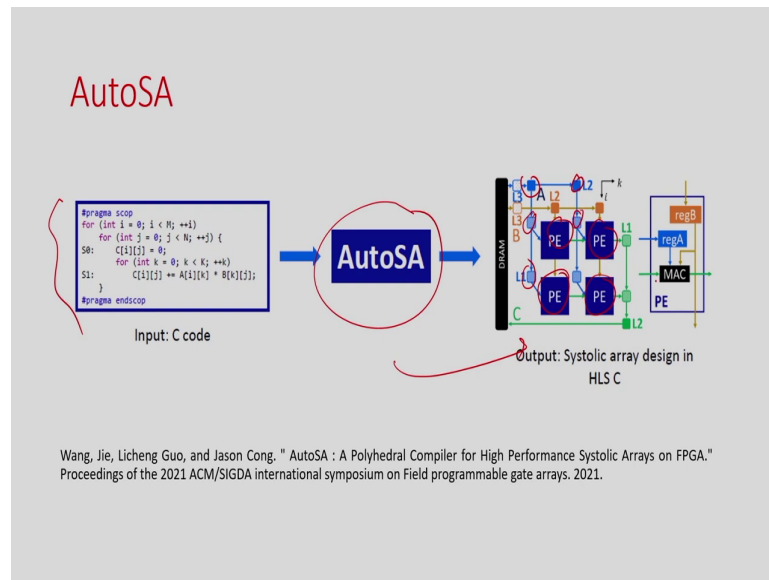
So, basically what is happening here is that, again it has an option like you write the computations and there is an option to specify the optimization that you want, like tile, unroll and all, it is similar to Halide again. And then this tool converts them into this parallel architecture and then using again the same thing that either open CL, HLS code or LLVM you can map it to a CPU or FPGA and so on. So, this is what this T2S does similar to HLS.

(Refer Slide Time: 36:09)



And Hetero Halide is the same thing that, you basically convert this Halide code into HeteroCL code and from there you can actually use that HeteroCL HLS to generate it to RTL. So, again the code base is available here and you can go to this paper to understand more about Hetero Halide.

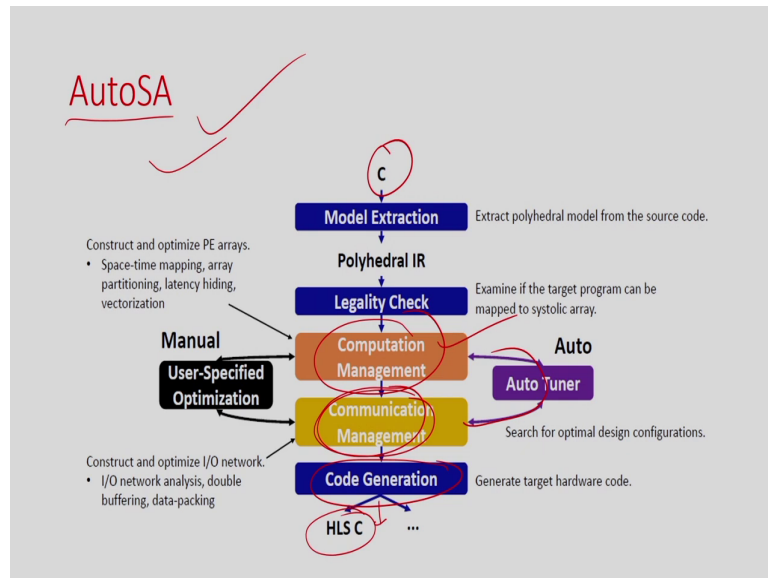
(Refer Slide Time: 36:28)



So, these three tools are of the same type . AutoSA is not something you have to specify your algorithm in a different language, just like halide. Here it is basically given the C code and it automatically generates this kind of parallel architecture.

So, this is some component processing unit, processing elements and then they are actually connected in a systolic array format and then you have all this memory support FIFOs and also locality of references and then this is happening. So, what it does is basically the detail is in this paper, you can go through it.

(Refer Slide Time: 37:04)



And it basically does this; basically it has to do a polyhedral analysis. So, it basically creates a polyhedral model from this and then it just tries to do computation management and communication management. So, basically if you go into this diagram, this is the computation happening here and this is the communication unit.

So, from this code, you have to understand the basic computations. So, that is the computations and this is where this computation is happening and then if you just break the loop, your communication gets changed. So, communication is the flow of data, so which data I need first. So, this is a matrix multiplication example.

So, it is basically manage this, try to break this whole computation into small small unit and then you try to make sure that you have the data flow, data is ready and the interconnections is according to the data data combination needed by this kind of architecture, this is what is systolic array design.

So, this is the overall tool and it just kind of do some kind of iterations and also I am not going to detail of that and finally, it is basically have this give you a C code which is basically a high level synthesis code, where you actually apply; it is actually generate a high level code where actually all the optimizations things are getting embedded. So, you just run that code using any RTL, HLS compiler and then you will get efficient hardware.

So, this tool's objective is the same, you have to parallelize your behavior; but the way this HeteroCL does and AutoSA does is different. So, but this is something that is more relevant if you try to do something from C; because for HeteroCL on T2S, you have to specify your algorithm in the language that they support. But this AutoSA supports C, but it is again generating parallel hardware.

(Refer Slide Time: 38:57)



So, I have last for these three, AutoSA, HeteroCL and T2S. I have just taken some pictures from this particular tutorial by these three professors. So, this is their work, I just took this diagram from their slides to explain this.

(Refer Slide Time: 39:23)



So, just to conclude this, we have discussed today the recent work that is happening in the context of high level synthesis and we have seen different areas like security, verification and then this creating a flow part from the tensor flow to RTL. And most importantly doing a temporal to spatial conversions, which is basically parallelizing your C code, which is one of the very active domains of research and where I have discussed many tools that are actually recently proposed.

So, just as a concluding remark for this course, we have understood that this background technique like allocation binding, this scheduling, those things are kind of matured. So, the tools are kind of matured. So, these are the two areas where actually the work is happening to make this high level synthesis a powerful tool, which can generate a hardware or a RTL, which will be equivalent to a code, which can if a RTL designer write it himself.

So, the two areas are basically converting your sequential behavior into some parallelized behavior, so that you can actually utilize the power of this high level synthesis tool. And internally this tool actually applies different kinds of optimizations, like unrolling, pipelining, data flow and there are many constraints. So, many times this

particular tool may not be able to apply this, because of certain dependencies or because of some constraint.

So, the internal to the tool, if you ask me to work hard, I mean try to remove some of these restrictions in the sense that they try to automate internally, so that these kinds of restrictions can be modified into some form which, so that your pipelining has an efficient implementation.

So, in the context of the tool, that is where the maximum effort is going on, specifically for the data flow I would suggest; because in the data flow sometimes if you have the code, if you just apply HLS pragma data flow, automatically it generates this kind of parallel architecture. So, that we have already discussed.

So, we have also seen many kinds of restrictions. So, how we can actually automatically analyze certain things and I can rewrite my code internally, the tool itself, so that I can apply this data flow optimizations. And in the front end, because as a user when you try to use it, I mean everybody it is well understood that for any generic C code, you would not get hardware.

So, you need to understand the way this high level synthesis works and then probably you have to rewrite some of the code and so that this gets mapped correctly in the hardware. And there we see some recent work that tries to automate that process. So, instead of rewriting yourself, can you automate that process.

So, that is what is the front end this auto SA and hetero HLS kind of tools are there. So, that something is a very interesting area, which actually reduces your effort of designing space exploration. So, otherwise you have to try option 1, option 2, option 3 and see which is happening, which is maps to a different hardware.

So, that is the design space explorations in context of parallel hardware is happening at the top end. So, these are the areas, which is something it is very important to make this high level synthesis tool useful for generating efficient hardware accelerators. So, with this comment I conclude this course.

Thank you.

