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

Module - 09
Impact of Compiler Optimizations in Hardware
Lecture - 30
Optimizations in HLS: Case study 1

Welcome students, in today's class we are going to discuss about various optimizations and their impact in high level synthesis results with some example case studies. So, start with we will talk about this code optimizations or generally co optimizations.

(Refer Slide Time: 01:03)

Code Optimizations

- Why is it important to enhance the quality?
  - Economically: "1% performance improvement saves Google millions of dollars" —Google
  - Scientifically: scope and precision of scientific simulation, reliability for critical missions
  - **Cluster** was a constellation of four European Space Agency spacecraft which were launched on the maiden flight of the Ariane 5 rocket, Flight 501, and subsequently lost when that rocket failed to achieve orbit.
    - Dead code, with inadequate protection against integer overflow led to an exception handled inappropriately—halting the whole inertial navigation system that otherwise would have been unaffected.
    - Cost: $370 million

Google datacenter

They have a huge impact in the final outcome right so, in terms of performance area power and timing. So, I have taken two such examples. So, for example, I have it says that, "1 percent performance improvement saves Google million dollars" in data center. So, just having a 1 percent performance using some kind of optimizations can save a million of dollars right it is a good study.

On the other hand once you do this code optimizations you are changing the behavior of the behavior of the input specification right. So, if you are not doing it carefully it may you may end up having a incorrect functionality right. So, you have to do these things very carefully in the other hand also right.

So, we have an example here where I am saying that some European Space Agency spacecraft was the rocket failed to achieve orbit because of some problem in the dead code and specifically using integer overflow right, I am just taken two arbitrary example from internet just to emphasize this to impact right.

So, on the other hand this code optimization is very important in terms of performance improvement and this performance improvement can give you a huge benefit monetary benefit on the other hand it has to be done very carefully ok.

(Refer Slide Time: 02:23)

Key Optimizations

- Loop Unrolling
- Loop Pipelining
- Array Partitioning
- Task-level Pipelining/Dataflow

So, in context of high level synthesis that I have already discussed that this high level synthesis specifically for behavior where there is no array and loop high level synthesis can synthesize the things into hardware efficiently, but in presence of loop and when this arrays we found that this high level synthesis may not able to give you the best hardware always right.

So, you need to understand how to apply the loop related optimizations of the high level synthesis tool as well as the optimization related to the array in the high level synthesis tool efficiently for efficient generation of hardware ok. On the other hand many a time this synthesis tool may not able to apply or may apply the optimizations, but it is since it is automated tool it may not understand everything on the specification.

So, the user may need some time a manual modification of the behavior as well ok. So, and also we have seen that this task level pipelining or data flow which is basically to achieve the maximum performance in the hardware by doing the parallelization of the behavior in the hardware that is also very important in the making a efficient hardware ok.

So, what I am going to do in this particular class I am going to primarily take the example of matrix multiplication and I am going to see how this loop unrolling, pipelining, array partitioning and then this task level pipelining actually helps to achieve a better hardware.

(Refer Slide Time: 03:56)

Loop Unrolling

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
    res[i][j] += mat1[i][k] * mat2[k][j];
}}} }
```

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
#pragma HLS unroll
    res[i][j] += mat1[i][k] * mat2[k][j];
}}} }
```

- **Idea:** Unroll loops to execute multiple iterations simultaneously
- The inner loop of the multiply function could be unrolled for concurrent execution of iterations

To start with let us first take the matrix multiplication example right. So, I think it is all everybody knew that, but just for refreshing your memory I am just explaining, what is matrix multiplication. So, I have given two matrix A and say B or you can say this is mat 1 matrix 1 right and matrix 2 two matrix is given right and the size of the matrix the 1 or A is basically say M into N right and this is N into say some P.

If you do this matrix multiplication; that means, A has M number of rows N number of column and this B has N number of rows and P number of columns you will get a resultant matrix if you do a multiplication of these two matrix which will be of size M into P ok.

So, in this discussion for making the process simple I will just assume this all M, N and P all are a same size. So, I am going to consider arrays N into N into N right and the resultant matrix will be also N into N this is for simplicity, but the whole discussion actually valid for M and P as well ok.

(Refer Slide Time: 05:21)

So, what it does matrix multiplication just to recap is basically you to calculate one element of the resultant matrix C. So, this is A, C, this is B and this is A, what we do. We take an row of this array A we will take the column of the array B and we multiply element wise right so; that means, to obtain this element right. So, to obtain A 1 1 sorry say that AB 11 I am going to multiply A 11 with B 11 then I am going to multiply A 21 with B sorry A 12 with B 21 plus A 13 into B 31 and A 14 into B 41 right.

So, to obtain a single element of the resultant matrix I need to take a row of the A array one column and you if I do the element wise addition multiplication and then the addition that will be my resultant matrix right. So, this is how I can calculate. So, for example, to obtain this element what I have to do, I have to take this row and this column right that will give me this element right and so on.

So, this is what is matrix multiplication. So, you can understand to for a given N cross N and N crosses matrix you need kind of order of n cube number of multiplications right. Because for a single element I need N multiplication right so for this is 4 cross 4 so, 1, 2,

3, 4 right so four multiplication. And, so there are N square element here or 16 elements so 16 into 4 64.

So, it is kind of order of n cube multiplication and also order of N cube addition operations right. So, although this code which is actually calculating this matrix multiplication it looks as very 3 4 5 lines code, but it is actually is a very competitive in computation intensive.

Because for a N cross S matrix multiplication it is kind of order of n cube multiplication operations and n cube order of addition operations ok. So, let us see this code what is happening. So, what is happening this is the output locations right i and j right. So, for each output location i and j of the resultant matrix i and j I am doing N multiplications and this k is basically is the N multiplications happening in the this internal for loop right.

So, what is happening here is basically i k into k j right so; that means, I am taking the row here I am adding the k. So, i 1 1, 1 2, 1 3, 1 4 and this is I am varying the columns. So, it is basically taking the column. So, keeping j fix. So, basically it is 1 1 2 1 3 1 and so on right.

So, this particular loop actually doing this multiplication and addition for a given location for each ij location right. This is what is the multiplication code and as I mentioned this particular code also look very simple only 5 6 line it is kind of very computation intensive. So, you need to do take some proper clear for making a efficient hardware from this particular loop right.

So, one of the important optimization is loop unroll. So, you can unroll this loop. So, there are three loops are here you can unroll this innermost loop. So, if you try to unroll the innermost loop what is going to happen? So, you can see there are kind of n cube number of operations and if you unroll the innermost loop; that means, you are actually removing this loop right.

So; that means, for each output location whatever the N of multiplication is needed that is going to be flattened now right. So; that means, if you just unroll this innermost loop you need order of n square kind of time step if I assume that here I need kind of order of n cube time step because in every iteration one multiplication followed by addition is

happening, but here this since the inner most loop is unrolled you need kind of order of n square of timestamp right.

So, in similarly you can actually use this unroll in the middle loop right. So, instead of here you are putting here so; that means, you are actually unrolling both the loops effectively. So, if you have a innermost loop and if you put the pragma in the top loop; that means, both the loop will be unrolled. So, effectively you need kind of order of n timestamp because both the loop will be unrolled now, it would be both the unrolled and; that means, this is only for the I loop all the things will be unrolled right.

So, and you need kind of order of n time, I am assuming that all this operation is going to be executing constant number of time stamp right. Similarly, you can if you put this unroll in top most loop then every loop will be unrolled and is all there is no loop at all right.

So, then you have kind of constant time to execute this, but; obviously, this come performance improvement comes with cost right. So, here probably if you need a single multiplier, here you need order of N multiplier, here we need if you put this in here it is order of n square multiplier and if you put this in the top most level it will it need order of n cube multiplier right.

So, in the sense that you need to make a balance on between these two right, you can order of n cube multiplier if you just think about 10 n equal to 10 it is basically 1000 multiplier it is not available in many target device right. So, you need to make a proper balance of that. So, what I am going to do is, I am going to take this particular example and try to put this unroll in the innermost loop and try to see how it is going to impact the performance ok.

(Refer Slide Time: 11:18)

Loop Unrolling

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
    res[i][j] += mat1[i][k] * mat2[k][j];
}}} }
```

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
#pragma HLS unroll
    res[i][j] += mat1[i][k] * mat2[k][j];
}}} }
```

- **Idea:** Unroll loops to execute multiple iterations simultaneously
- The inner loop of the multiply function could be unrolled for concurrent execution of iterations

So, now you can understand that so, for one element there are N multiplications right. So, there are N square elements. So, to order of N cube multiplications are happening here. So, it is very computational intensive program although looks like a three line four line code, that is very and very complex computation intensive program and also there is a adder memory read there are two memory read memory.

So, three memory reads you are reading from a the resultant matrix and you are writing back right. So, obviously, if you run this program it will take the latency to be very high ok. So, the idea here is that if I try to apply unroll loop unrolling how it will improves it; obviously, you can understand that unroll means basically you flatten the loop right. So, there is no loop at all.

(Refer Slide Time: 12:12)

So, the good idea to unroll here is that you basically unroll the last loop right. So, that is what I did here. So, basic idea is that whatever the multiplication I just talked about I do not want to write the loop, but I will just write this direct equations right. So, if I just replace this unroll effectively it means it is basically for each element I am just writing this equations where this i.

And this 1 1 is basically i j right so i, j, k. So, if I do that ah; obviously, what I can that, I can now run these things parallel right because these all operations are there. So, I can schedule them differently I can run them parallelly. So, let us see how if I just do this how it will impact the performance.

(Refer Slide Time: 12:44)

Loop Unrolling

Base code: Performance (in clock cycles) and Resource Profile:

Clock Period: 10ns

Latency		Interval		
min	max	min	max	Type
215	215	215	215	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	-	6	186	498	-
Memory	0	-	128	10	0
Multiplexer	-	-	-	171	-
Register	-	-	[illegible]	-	-
Total	0	6	320	685	0
Available	270	240	82000	41000	0
Utilization (%)	0	2	~0	1	0

Optimized Code: Performance (in clock cycles) and Resource Profile:

Latency		Interval		
min	max	min	max	Type
89	89	89	89	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	-	12	264	626	-
Memory	0	-	128	10	0
Multiplexer	-	-	-	237	-
Register	-	-	0	-	-
Total	0	12	398	879	0
Available	270	240	82000	41000	0
Utilization (%)	0	5	~0	2	0

So, you take a so, I just take a 3 cross 3 matrix and try to run it for some target architecture and what I see if I just run without this unrolled pragma I get basically 215 cycle clock cycle is needed the latency is 215 cycle ok and if I just write this apply this pragma it is basically taking 89 cycle. So, it is kind of one - third right because this is a 3 cross 3 matrix.

So, it may not be exactly multiplication of that because there are some clocks needed to synchronize the pre and post operations and so on, right. On the resource side let us see is it something three times more, let us see. So, earlier it was since it is a small array it is not using any BRAM, it was using 6 DSP which is basically the multiplication operations happening there, 320 registers because this array mapped to registers and these are the combination of 685.

What is the change now? It change to 6 to 12. So, it is get double right. So, you might say that now I unrolled; that means all this operation is happening parallel why it is not 3 times because there are three multiplication is going to happen right. And the answer is the scheduler may not schedule everything in one cycle right based on the that combinational delay it might decide that I am going to execute in say two cycle or whatever it is right.

So, it is not that if you just unroll you will get that multiplier because multiply will be 3 times because I take a 3 time 3 into this 3 cross 3 multiplication right. It depends on the

how now a high level synthesis tool schedule; obviously, it will increase I can see the increase in the register also and also in combination unit; obviously, because of this handling this interconnections and all more more interconnections ok.

So, I can clearly see that from if I just apply unroll there my latency improves 3 times my number of time clock needed is reduced by 3 times and my area over rate is not 3 times. So, it is something is useful right because performance is greatly improved. So, one might ask me that if I do not unroll why it is taking 6 multiplier right, the answer is.

So, if I am here I taken integer right. So, my elements are integers right. So, it is 32 cross 32 and in the DSP board usually the multiplexers are not this multiplier are not 32 cross 32 it is usually say 16 cross 16 or some specific things. So, and then; that means, what so, achieve this you need to split it into smaller multiplier and you have to combine them and that combinations can be different right.

So, there are some many algorithm where there are many way you can actually break a bigger multiplier into smaller multiplier and combine the result to get the better results. So, that is why even if during rolled implementation I am although there is only single multiplication in each iteration, but still it is 6.

So, this might vary for the target architecture because if I take a different FPGA board the DSP configuration may be different right and based on the DSP configuration the number of multiplier is decided. So, here I probably taken a very smaller board, so that is why it shows more than 6 multipliers right, but I hope you understand the overall idea here.

(Refer Slide Time: 16:21)

Loop Unrolling

Idea: Loop Unrolling could be very costly in terms of hardware utilization. Unrolling could be constrained by Unroll factor for adjusting resource-performance trade-off.

Consider the same multiplication function before for 8x8 matrices

Unroll factor could be adjusted using the pragma: **#pragma HLS unroll factor=k**

Model	Hardware Utilization				Performance (clock cycles)
	#LUTs	#FFs	#DSPs	#BRAMS	
Baseline	2070	1522	9	0	4389
K=1	3004	2419	36	0	933
K=2	2956	2039	24	0	1573
K=4	2368	1807	18	0	3109

4.5X

So, in unrolling you can understand that if you just completely unroll the number of operation grows very high right. So, the usually it is getting controlled by a unroll factor right. So, you can unroll the loop completely when you make it K equal to 1 unroll factor equal to 1 or you can unroll by 2 or you can unroll by 4 and so on right. So, this is unroll factor you can use and you can understand that it is basically partially unrolled right.

Here I just put some results I just take a you know 8 cross 8 matrix and for rolled implementation this is the resource utilization and this is a number of clock cycle right, you can see here there are 9 DSP the multipliers and these are the other units. When I completely unroll my multiplier is not 8 times right, it is only 4 times because it is 8 quadratic matrix multiplication if I unroll there will be 8 multiplications within the inner loop right, but that is not 8 times.

So, the again the reason is that I have explained. So, again if I just do a unroll factor this you can see here it is kind of 2.5 times right, and you can see here the reduction in the clock time is kind of almost 4.5 x right. So, you can understand that how much improvement you can get in the number of clocks and the number of resource and based on the available resource and the available budget you can decide which will be your best factor you should choose whether you are going to completely unroll or partially unroll right.

So, that is something this unrolling and it is basically useful for bigger very complex loops where there you cannot if you just completely unroll probably you just cannot fit the everything into the FPGA board. Because it is just might be resource might overflow right. So, this unroll factor can be key things to manage this blow up in the area ok.

(Refer Slide Time: 18:12)

Instruction Level Pipelining

Idea: Code segments within function blocks and Loop iterations can be pipelined for concurrent execution. This could be implemented using the pipeline pragma

Consider the same multiply function as before for 3x3 matrices

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
    res[i][j] += mat1[i][k] * mat2[k][j];
 }}} }
```

The inner loop of the multiply function could be pipelined for concurrent execution of iterations

```
void multiply(int mat1[N][N], int mat2[N][N],
              int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
   for (k = 0; k < N; k++) {
#pragma HLS pipeline
    res[i][j] += mat1[i][k] * mat2[k][j];
 }}} }
```

So, the next one is the instruction level pipelining or loop pipelining. So, you know that this high synthesis tool support pipeline. So, I if I just apply this pipeline in the innermost loop what will happen, now this multiple iterations of the loops will run in parallel right.

So, I do not I am not going into detail how the pipeline works because that is already explained, but it is basically you can understand that; obviously, the number of clock size is going to reduce because now this different iterations of the loop are actually overlapped right.

(Refer Slide Time: 18:45)

Instruction Level Pipelining

Base code: Performance (in clock cycles) and Resource Profile:

Clock Period: 10ns

Latency		Interval		
min	max	min	max	Type
215	215	215	215	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	-	6	186	498	-
Memory	0	-	128	10	0
Multiplexer	-	-	-	171	-
Register	-	-	6	-	-
Total	0	6	320	685	0
Available	270	240	82000	41000	0
Utilization (%)	0	2	~0	1	0

Optimized Code: Performance (in clock cycles) and Resource Profile:

Latency		Interval		
min	max	min	max	Type
167	167	167	167	none

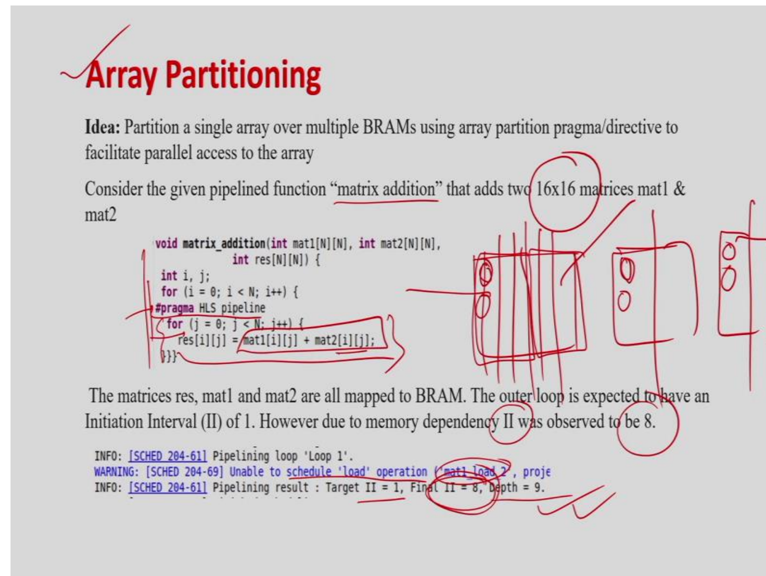
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	-	6	142	632	-
Memory	0	-	128	10	0
Multiplexer	-	-	-	186	-
Register	-	-	6	-	-
Total	0	6	276	834	0
Available	270	240	82000	41000	0
Utilization (%)	0	2	~0	2	0

So, let us see in this particular example again I just take a 3 cross 3 matrix and I just try to run and this is the result I have already explained right even for basic baseline case where we do not apply any optimizations, you take 215 cycles and this is the resource and now I just apply this pipeline I can see 167 number of clock cycle.

So, it is basically because it is a 3 cross 3 so the number the advantage or the benefits you are getting less, because the number of time you run the loop your benefit will be more right. So, since the loop is the innermost loop is only 3 times the benefits is not that significant, but if you just run the same thing for say 10 cross 10 multiplier this will be a significant improvement in the number of clock signal.

But most importantly here you see the number of resource does not increase much right. It is the number multiplier remain the same flip flop reduces and there is a some in small increment in the number of LUTs So, this suggests that the pipelining is something one of the most powerful optimization technique in high level synthesis is to get a good performance of a loops because it does not increase the resource by much, but if the number of iterations of the loop is more the benefits of the number of cycle or the latency will be greatly reduced by pipelining ok.

(Refer Slide Time: 20:10)



So, let us move on, the next most important optimizations is the array partitioning and it is kind of inevitable you have to use it and because of that the problem that of the block RAM. The block RAM has one or two ports so, you can have only one or two access to a memory right and when you have a array if a more than one access more than two accesses it is create a bottleneck in the loop in the loop pipelining also right.

Because if you just put it in there is a array access more access in the loop you try to pipeline you may not have achieved this pipeline initiation interval of 1 because of this access limitations ok. So, array partitioning this is something important and this high level synthesis tools does support that right.

So, for this I just first taken a matrix addition example first and then I will move on to the matrix multiplication and how this array partition will help there ok. Let us take a 16 cross 6 bigger multiplications and try to add it right, you can understand here this addition is very simple that if you have two matrix and if you try to add it basically it is a element wise add right.

You just add these two element and store here. I will add these two element you store here right, this is basically element wise addition right that is what is happening here and there is a 2 nested loop. And if I try to pipeline it you can see here even I am not able to get the initiation interval 1 because is become 8 here if I just run it in high level synthesis you can get it.

It will say that unable to schedule the load operations because there are so many memory read the matrix load 1 and so on. So, because of that target pipe initiation interval is 1, but I am able to achieve a initiation interval 8 and the my pipeline depth is 9 ok, how it is coming here.

So, you have to go into deep understand because these things will be splitted into three address operations and it can be split in more operation that is why we have a depth of 9 there. So, you can actually run and see the schedule and you can understand them much better there. But the problem is we can understand that because of this so many memory read we are actually it is creating a problem right.

(Refer Slide Time: 22:24)

Array Partitioning

In this case, Array partitioning might facilitate parallel access to the partitioned segments mapped different BRAM tiles. This might improve the hardware performance

The modified code shown below divides all the arrays into two blocks mapped.

```
void matrix_addition(int mat1[N][N], int mat2[N][N],
                int res[N][N]) {
 int i, j;
 for (i = 0; i < N; i++) {
#pragma HLS pipeline
  for (j = 0; j < N; j++) {
#pragma HLS array_partition variable=res block factor=2 dim=2
#pragma HLS array_partition variable=mat1 block factor=2 dim=2
#pragma HLS array_partition variable=mat2 block factor=2 dim=2
    res[i][j] = mat1[i][j] + mat2[i][j];
 }}}
```

```
INFO: [SCHED 204-61] Pipelining loop 'Loop 1'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('mat1_0_load_2', pr
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 4, Depth = 5.
```

After partitioning the array by factor of 2, the observed II doubled!

So, what we can do is basically we just unroll it. So, basically we can split the array right. So, what I did it here is basically I just split the array into two dimensions. So, basically I just split the array by 2 kind of right and now I can run this operations in parallel and basically you can see here that my now I I equal to 4 right.

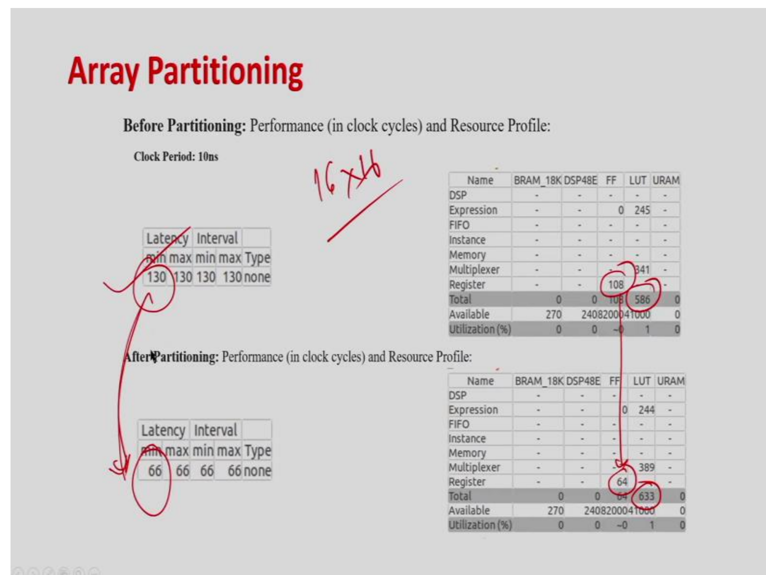
So, basically by just partitioning the array I am actually making enabling this loop pipelining because I am actually applying pipeline in the outer this outer loop not here. So, if we apply here it what will happen, it will basically unroll this loop that is why this is happening.

So, just to explain again so you can see here I am not applying this unroll a pipeline at the inner most loop I am applying in the outermost loop right. As a result what is happen this is going to unroll right and if you unroll you can understand that there will be so many memory read and because of that you are not getting this initiation 4.

But because this operations are element wise multiplication is a symmetric axis. So, what I can do, I can just split the array by like this right. So, then what I can do, I can whenever you try to run this so, if even if you unroll this 1 to 0 to N. So, basically what will happen what will happen that you just add this element right. So, this is basically i j. So, and if it is 0 to N; that means, it will this columns right. So, it will just do this columns 0 to N columns and what is going to happen.

So, this part of the things array access will be from one array and this part of the array will be access from the another memory as a result you can actually do this access parallel and that is what exactly happening here and that is why my pi initiation interval become from 8 to 4 because I just split it 4. And if you split it more probably you will get better this initiation interval right. So, we can understand that array partitioning is something is important just to reduce the memory accesses.

(Refer Slide Time: 24:27)



Let us now discuss the results here. So, if I do not partition the array and I if I synthesize into FPGA with some target device this is the latency I am getting right. So, it is a 130 cycle right. So, that is what I am going to get and if you look into the resource since it is

a matrix addition operation there is no DSP, no multiplication, the array it is a small array. So, map into flip flops no block RAM, but it is all mapped to flip flops and you need some combinationally you need to execute the other operations right.

So, now, you think about this after array partitioning manual array partitioning since I partition the things into two arrays. So, my latency is almost half right. So, latency improves right double right. So, your performance get doubled basically right. And if you look into the resource again there is no block RAM no DSP, but I can see the flip flop reduced to almost half and LUT increase little bit because of managing this partition thing.

But I think one of the reason for this reduction in the flip flop probably it is the use of the same flip flop to store the different partition of the array ok. So, it clearly says that this array partitioning help improving the performance of the matrix addition without much increase of the resource.

(Refer Slide Time: 25:56)

Array Partitioning at Source

Problem: Sometimes directive based array partitioning doesn't improve pipeline performance as the synthesizer fails to detect a well defined and symmetric array access pattern

For instance, in matrix addition, the iteration equation is **out(i, j) = A(i, j) + B(i, j)**. Due to the symmetry array access to all matrices, partitioned segments could be utilized simultaneously.

Consider a function with asymmetric array access pattern as shown below.

```
void multiply(int A[N][N], int B[N][N],
        int res[N][N]) {
 int i, j, k;
 for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
   res[i][j] = 0;
#pragma HLS pipeline
   for (k = 0; k < N; k++) {
    res[i][j] += A[i][k] * B[k][j];
 }}} }
```

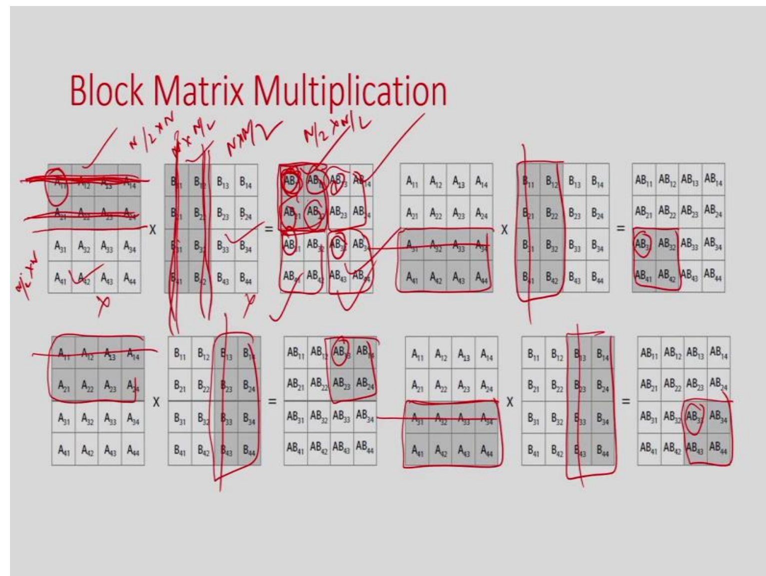
The iteration equation **out(i, j) = A(i, k) + B(k, j)** is not very symmetric in this case!

To improve pipeline performance in such scenarios, arrays have to be partitioned manually!

So, now for array partitioning is something HLS tool does right. You can just apply this array partitioning this pragmas and you can do that, but sometime you the tool have not able to decide right or you not able tool for tool it is may be difficult to basically partition and if the access is not so symmetric right and this is the classic example is the matrix multiplication right.

So, let us try to understand that how we can partition array to make this so that matrix multiplication get benefited right. So, again you can notice here that I am applying this HLS pragma at the second loop not the innermost loop so; that means, this loop will be unrolled.

(Refer Slide Time: 26:46)



So, let us I decide that I partition this array A by half in the row wise right. So, I just make it was N cross N I just do this, N by 2 into N and this is also N by 2 into N and the column matrix I divide by half by column wise. So, it is basically this array is basically N into N by 2 and this is also N into N by 2 and I decided to split the resultant array into 4 matrices everything was N by 2 into N by 2 right. So, this is one this is one part, this is one part, this is one part and this is one part right. So, this is how I split into 4.

So, now, let us see how this will help right. So, let us try to understand this important point here is that say I want to calculate these elements ok these 4 elements. So, to calculate this what I need, I need this say. So, basically it depends on how to consider. So, you can actually assume that this is B and this is A. So, I if I multiply this into this column I will get this right.

So, similarly if I multiply this row into this column I will get this. If I multiply this row into this column I will get this, if I multiply this row into this column I will get this. So, what I can understand that to calculate these two four element accessing this array and this is these array is sufficient, I do not need to access this part and this part right.

Similarly for other components so, if I try to access this if I just multiply this part with this part I can show that I will compute this part of the memory and I decided that I have a four different different part right. So, what I can understand here is that similarly if I just multiply this part with this part I will get this part of the resultant array and similarly if I just multiply this with this part I am going to get this part of the resultant array and you can verify it.

So, the here we can understand that now I can actually partition and this partition actually helps right because if I partition this way I can actually kind of parallelize this part right. Because, now I can run this four component this calculation of this component, this component, this component and this component this kind of in I mean independent right.

So, what and then I can actually kind of put this into two different memory this part and this part into two different memory and this four part into four different memories, then this I can actually kind of separate out the in dependencies and I can actually run these things in parallel right.

So, for example, I can just take this and this and I can calculate this at the same time I can just take this part, the other part, this part and this part and I can do multiplications and I am going to write this part of the array right so; that means, I can actually these two are kind of independent and this access is not interdependent right. So, this is what I can do and the tool may not understand this and may not able to partition this.

(Refer Slide Time: 29:59)

Array Partitioning at Source Code

Solution: Arrays involving asymmetric iteration equations can be manually partitioned and multiple independent set of equations involving the partitioned segments could be framed.

Consider the manually partitioned equivalent of the function shown previously.

```
void multiply(N_x_N_by_2 A, N_by_2_x_N B,
        N_by_2_x_N_by_2 &res) {
 int i, j, k;
 for (i = 0; i < N/2; i++) {
  for (j = 0; j < N/2; j++) {
   res.p1[i][j] = 0; res.p2[i][j] = 0;
   res.p3[i][j] = 0; res.p4[i][j] = 0;
#pragma HLS pipeline
   for (k = 0; k < N; k++) {
    res.p1[i][j] += A.p1[i][k] * B.p1[k][j];
    res.p2[i][j] += A.p1[i][k] * B.p2[k][j];
    res.p3[i][j] += A.p2[i][k] * B.p1[k][j];
    res.p4[i][j] += A.p2[i][k] * B.p2[k][j];
 }}} }
```

```
struct N_by_2_x_N_by_2 {
 int p1[N/2][N/2];
 int p2[N/2][N/2];
 int p3[N/2][N/2];
 int p4[N/2][N/2];
};

struct N_x_N_by_2 {
 int p1[N][N/2];
 int p2[N][N/2];
};
struct N_by_2_x_N {
 int p1[N/2][N];
 int p2[N/2][N];
};
```

Since all the partitioned segments are automatically mapped to different BRAM tiles, the four iteration equations in inner loop could be executed in parallel!

This improves the pipeline performance

So, here I just do it manually right. So, what I did is basically I just split my resultant array into four N by 2 in arrays that A array into N into N by 2 and the other part is N by 2 into 2 arrays this I split into column wise this I split into row wise and then inside this loop I just do this four operations in parallel right. So, I can do this operations in parallel that is the biggest advantage right.

So, what I can do is I can multiply this into this I will get that at the same time I can multiply this with this I will get this element at the same time I can multiply this with this and I will get this same element and I can multiply this with this and I can get the same element right. So, this is how I will do that because now I can just read once. So, if I just read this element using that I can calculate this one element for each memories.

So, this is what is the advantage you will get and this is something what I did here right. So, this is how I can actually parallelize this operations in different memory access and sometimes doing by tool is bit difficult right. Nowadays there are tools which can actually do that, but usually we have to do this kind of partition manually to improve the performance and let us see how it impacted right.

(Refer Slide Time: 31:20)

Array Partitioning at Source Code

Unpartitioned Code: Performance (in clock cycles) and Resource Profile:

Clock Period: 10ns

Latency		Interval		
min	max	min	max	Type
2052	2052	2052	2052	none

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	6	0	688	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	289	-
Register	-	-	473	-	-
Total	0	6	473	977	0
Available	270	240	82000	41000	0
Utilization (%)	0	2	~0	2	0

Manually Partitioned Code: Performance (in clock cycles) and Resource Profile:

Latency		Interval		
min	max	min	max	Type
516	516	516	516	none

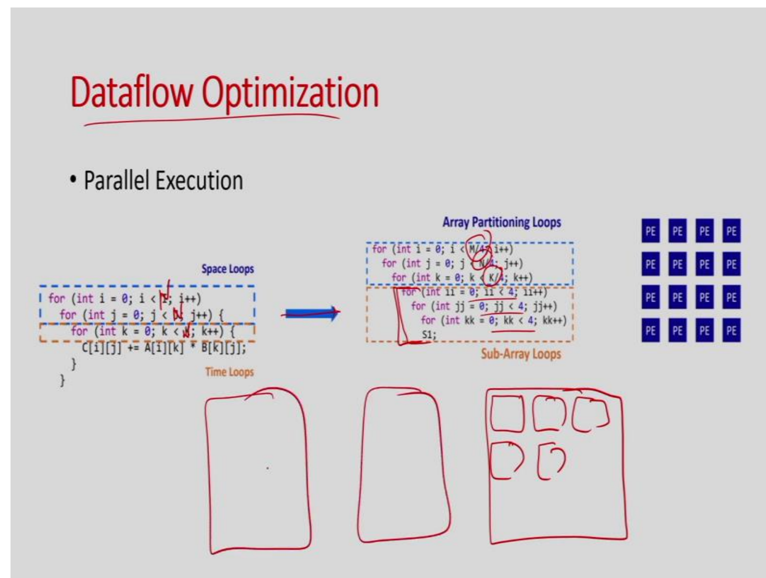
Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	24	0	1609	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	465	-
Register	-	-	1216	-	-
Total	0	24	1216	2074	0
Available	270	240	82000	41000	0
Utilization (%)	0	10	1	5	0

So, you see initially if you do not apply any optimization it takes 2052 clock cycle now it is taking 516 because I just kind of four time faster because I split my resultant array into 4 and I have actually calculating four different element of the resultant matrix at the same time. So, it is kind of four time improvements ok and the area wise also since I am actually performing this multiplication.

So, the number of multiplication is kind of four time more and the resource is also kind of has a significant improvement and as I expected because I am now doing this things in parallel. So, you have to need more resources, but I can see there is a huge improvement in the time right the number of time required to execute that.

So, this is what I just this how this matrix multiplication can be the array can be splitted and basically it can improves the performance of the generated hardware ok. So, so far what I discuss is about either apply loop unroll or you apply a pipeline or you partition array, but I did not talk about the parallelize this operations right. So, how do parallelize this calculations that is very important.

(Refer Slide Time: 32:41)



And that is where is basically the data flow optimization. So, where I mean, I am calculating the whole matrix multiplication can I parallelize certain operations right. So, and that actually give you huge benefit right. So, let us try to understand that. So, what I did here is I have this bigger array right I just split it into smaller array right of size 4 by 4 ok.

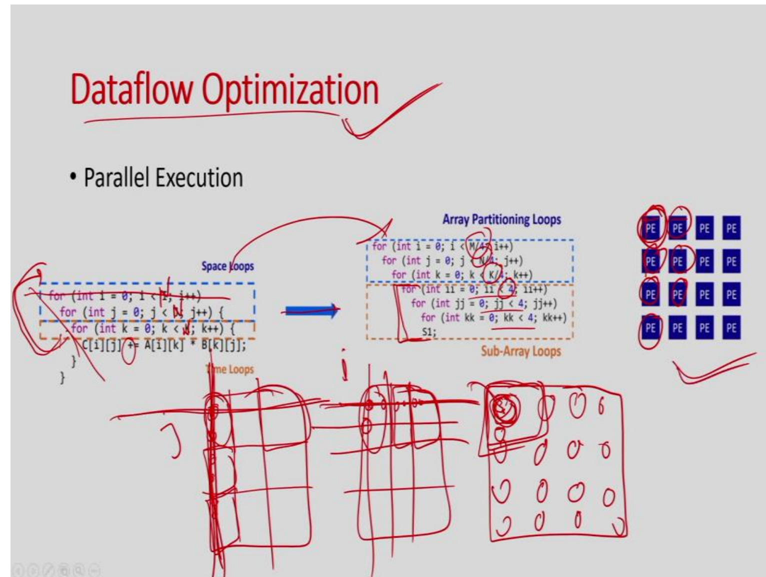
So, what is happening so, try to understand that this conceptually so I am going to explain this in conceptually that I just do this right. So, this is basically loop tiling you can understand that right. So, what I did it here is say basically earlier it is N cross N. So, this is 0 to N you can understand this is all N and now it is basically M by 4. So, basically whatever the size I just make it 4 and then internally I just run another loops basically this is 0 to 4 0 to 4 0 to 4 right.

What does it mean, you have this bigger array and this is applicable to all three arrays I am just taking about one example and I am actually try to split the whole thing into small matrix right that is what is happening and that is what is exactly this right. So, this is basically I split the whole thing. So, this calculation is for this and this is happening for each such PE right.

So, that is what this loop tiling means and I can actually split this way. And the advantage of doing this has huge right. So, what is going to happen you have to

understand that to calculate a element of the resultant matrix I will multiply one so, just to give the example so to calculate one element of the resultant matrix.

(Refer Slide Time: 34:27)



What I usually do is, basically I multiply one column into one row to get this element right, but in the basic idea is that intermediately I can just store some of them right. So, basically what I can do, I can just do these two element multiplication these two and result I store here, I will come back after some time and then I will multiply these two with corresponding these two and then that add I will just put it here again right.

So, on top of that it is a result plus plus right this is this is the plus right and then I will come back and then I will take the next two element I multiply the next two element and that multiplication result again I just sum up with this right and that is what I try to mean here that to calculate the resultant element one element I can just take this part and the and let us assume that the column matrix is that. So, I will just take that.

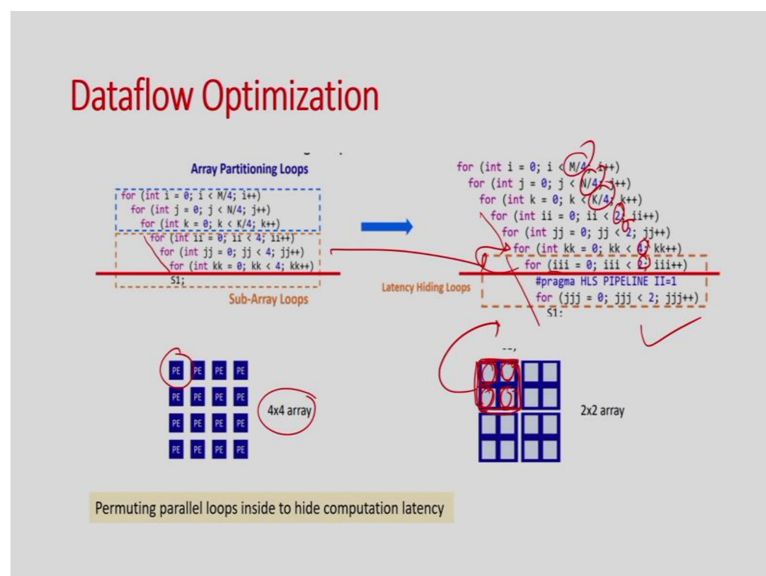
So, I will multiply these two I will store here, I will then I will take this block I will take this block and multiply the first row and this first column and then I will add that on top of that right. So, what I try to mean here is that if you just do this there is no nobody tell us that you have to multiply all this column all this element of this column wise I mean one column and one row at the same time.

What I can do I can actually split this whole calculations into small small group and then I can actually run this things in parallel right that is the advantage I try to talk about right. So, I can just read this. So, you have two such blocks right as I mentioned. So, this is a array a and this is array. So, what I can do I can just. So, I just split this array right. So, I just split the whole thing into say 3 cross 3 matrix.

So, I will just take this part and this part and I multiply and the resultant things will I will just update partially here then I will take this part and I will take this part and again multiply and again I update the element, just sum this element on top of whatever they already stored, then I will take this and this and this and we can multiply and I can store this. So, what I try to mean here is that by just doing this splitting I have a scope of parallelization of the calculation of the one element for one element there are N multiplication period.

I can just do N by 4 at a time then store there after that I will again do the next N by 4 of multiplications and store the put on top of that and so on. And since these things are actually I mean not dependent I can actually now run these things in parallel right. So, this is what the parallelization means. So, that is what this data flow optimization does and this is what the classic example where I just do this I just created a option to do the things is in parallel mode right, this is what is called systolic array kind of representation.

(Refer Slide Time: 37:13)



The next thing I did very interesting that so you have this you remember then the this matrix multiplication the loops are basically i, j and k it means basically I will take one column. So, this is for row actually I will take this I mean to be honest this is the row and this is the column, but you can just swap that does not matter. So, you take this first kind of say one column one row. So, this is my j and this is my i and then I just multiply and then I will create a single element and this multiplication happening in this k right.

Now, if I just bring this k at top. So, what will happen now, now what is going to happen now I am actually for each k right k equal to 0. So, what is going to happen then k equal to 0 I am going to vary this i j so; that means, I am going to do 1 1 multiplication for each all locations right. So, i j if I just vary i j; that means, I am going to get all elements location right for each locations I am going to do one multiplications.

So, for example, say for this first location I multiply this with this right I store it here if I just put the k loop outer right. Then what I am going to calculate the next element what I do I just take this and the second row, I just do take the first element and first element and store it partial results here. So, that is going to happen. So, what is going to happen here is that. So, now, what I did the next thing is did that right. So, what I just did it here I just split it further.

So, I just split it further within the basic block also I try to parallelize further right. So, what I did it here, you see here earlier it was i j k now what I did. So, this is there I split it further and then I put k as the outermost loop then i and j so; that means, what I am going to do it now, I am going to calculate the resultant 111 multiplications for the resultant matrix in parallel right. So, that is what is going to happen here. So, that is the second optimization you can do and then you can do more parallelization here right.

So, the basic idea is that for one resultant location I need N multiplication right, what I am doing now I am actually splitting into N by N by k or N by 4 component that is what I did first right. So, I just do it by N by 4 here. Now I am going to calculate not all N multiplication at the same time rather I am going to do, N by 4 multiplications I store the result partial result there then I am going to do next N by 4 elements multiplication in parallel and you can store the result there and this is what this optimization means.

And then within this N by 4 also I have to do 4 multiplication say N by 4 multiplications, what I did I actually split it there and then there are. So, there are say there are 4 say if

this is 4 cross 4 arrays say each of them is 4 cross 4 arrays. So, what I did is basically I just put it into 2 cross 2 array and then this calculation also I just do in parallel right. So, this is this loops means.

So, now, if we just so from you can understand it earlier it was 3 nested loop and now it is basically 1 9 nested loop, but the number of iteration is same, earlier it was N and now it is you can see it is M by 4, N by 4, K by 4 and then this is 2, 2 and then this is basically 4 and this is 2 right. So, this is what I do. So, with this so, what I mean I can achieve here is the whole purpose of doing all these things is basically parallelize this calculations of this matrix multiplication for each element right.

For each element there are N matrix or multiplication sorry this multiplication operations how to parallelize them and execute them in parallel right. So, that is what is the things and this is how we can actually modify the code and then you can apply this data flow optimization there and you can actually get a great performance benefit ok. So, I do not go into the synthesis result, but you can always try this and see the benefits of doing all these things ok.

(Refer Slide Time: 41:34)

So, the next one is the smaller things or very important thing is the pre synthesis computation ok and this is also kind of manual thing. So, many a time we write code where we have very complex operations present ok. For example, this cos and sin and here this you can see here the cos k is basically we need say row is some N. So, N

number of cos operations. So, cos 1 to 1 to cos k cos N right and similarly I need sin 1 to sin N.

So, these are the things I need, but the problem here is that this loop running for N square time right. So, I am going to calculate this cos k operations. So, I need kind of say all are N. So, all are running so, N, N and N right. So, basically you can understand that kind of cos 1 you can calculate N square time, cos 2 is going to calculate at N square time and when and so on. So, cos N also going to be calculated N square time and similar this.

So, and this remember this calculation of cos in the hardware is very costly right you can actually understand that the quadratic algorithms are there and all. So, implementing cos sin this kind of functions in hardware is very high it is first of all the area is it needs a I mean it is a complex operations, it need time as well as area ok so, but here I can see that this I need only N cos value and sin value right that is all I mean it is not something data dependent.

They are data independent right they are basically input independent computations whatever the value of is there I mean I am going to just need the value cos 1 to N right. Similarly, sin 1 to N what I can do, I can just create a array and store this values right sin cos 1 to cos N and I will take another array where I just calculate the sin and I just stored the value sin 1 to sin N and then I do not need to calculate it here right.

So, basically this is something kind of very harmless when you run the C code, but this is actually is very computation intensive things in hardware and it actually degrade the performance like anything ok. So, what I just did it here I just calculate this and I just store in a array and the array name I just give pre compute p1 and p2. So, basically that is what I did and then I also partition the things ok.

So, basically I just stored this in four arrays now and I do a array partition here ok and just doing this that I instead of calculating this sin cos in the hardware rather I just I know what is sin, sin 1, what is sin 2, I just calculate pre computed and then I store those value in the hardware and I just split the array into 4 times to improve the performance and I have seen that I have a 195x faster code this right.

So, this will if this takes 195 times or say kind of 200 time right. So, if it is take 10 cycle 10 if this is taking 10 clock it was taking 2000 clock 2000 cycle because of this

calculation of sin and cos. So, you can understand here I just try to emphasize here just a small precomputation can actually give a huge benefit in terms of performance which is kind of kind of 200x performance benefit for this example ok.

(Refer Slide Time: 45:03)

Expression Redundancies - Manual

- Constant terms and 'y' dependent computations could be migrated to the outer loop

```
for(y = sy; y < ey; y+=0.5) {
 for(x = sx; x < ex; x+=0.5) {
  destX = (float)(a*x+ b*y+c*x*y+d);
  destY = (float)(e*x+ f*y+i*x*y+j);
  Out[(int)(destY+0.5)][(int)destX]=
  Img[(int)y][(int)(x+0.05)];
}}
```

```
for(y = sy; y < ey; y+=0.5) {
#pragma HLS pipeline
  preX=b*y+d;
  preY=f*y+j;
  multX=c*y+a;
  multY=i*y+e;
  for(x = sx; x < ex; x+=0.5){
   destX = (multX*x);
   destY = (multY*x);
   Out[(int)(destY+preY+0.5)]
   [(int)(destX+preX)]=
   Img[(int)y][(int)(x+0.05)]; } }
```

The optimized routine with pipelined gives performance speed up of 23 times with smaller design area than the base code.
It also helps to mitigate loop carry dependencies during instruction pipelining.

And the last one is I talked about this expression redundancies many a time we just write x the code or you do not understand how they that will impact in the hardware ok. So, this is kind of say one segmentation code I mean in image processing algorithm, what I am doing here I am just calculating some x and y value using this equation and then I store certain data in that particular equation location right. So, I just store certain value in that particular location of output right.

So, that is what I am doing, but you can see here is that this b y. So, the s x this x and y is kind of independent right. So, what I can actually identify what is the things which is basically loop invariant which is not changing with the value of x right and that part I can actually pre compute right.

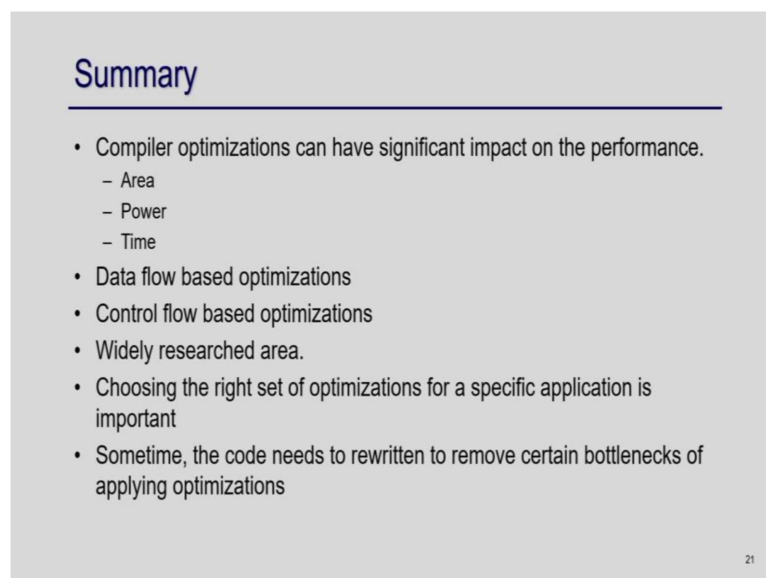
This is my outer loop and this is my inner loop and I can see here is that this part of the code this b y d and then this part these are the things which does not change inside this x loop. So, I just bring out I mean I just break this expressions and found out this sub expression and I put them into outer loop.

So; that means, earlier they are calculating kind of N square time, now they are going to calculate N time. So, that is the kind of advantage I do and for this high level synthesis tool identify search things and moving it sometime it is difficult. So, if we found this kind of repetition sorry kind of invariant code you should do it kind of manually right.

So, if you just do it manually and I just do this optimization and run high level synthesis and I got kind of 23 time faster harder and because this calculations I do not have to do I just do it N time N less times right. So, earlier it was N square time, now it is N time, it may be that there are some expression which is also independent of x or y right I can move it there also out of the loop then you will get the maximum benefit, but in this case it is not the case.

So, I just move some of the invariant operations one level up ok. So, that also gives ah. So, small small thing this pre computation of sin cos can give 200 time faster hardware, just doing this equations I mean redundant calculations you identify and put it earlier just doing this you can get a 23 times faster hardware right. So, this small thing, but important things ok.

(Refer Slide Time: 47:22)



So, in summary what I understood that this optimizations are very important specifically this unrolling, pipelining, data flow optimizations and array partitionings and they can have a huge impact on area power and time I usually do not bring the power into this discussion, but you can also see the impact on power as well ok. And many a time these

tools apply them, but you have to understand where to apply I mean how it will impact it, sometime probably you have to changes the code you have to rewrite the code.

So, that this optimization can be get applied right otherwise you may not able to apply for example, the matrix multiplication you have to make it that nested loop and then you have to apply data flow then only you will get a benefit right. And it is kind of a not a one time process it is kind of iterative process and you when you try to develop a high efficient hardware you have to do kind of design plus expression applying different even combination of the optimizations and see how it is get impacted.

And you have to understand the impact right and sometime you might see that because of the code my initial code I am not getting the impact. So, you have to manually change the code little bit. So, that we can enable certain optimization in that code and as a result you get a efficient hardware ok. So, with this I conclude today's class.

Thank you.