Parallel Computer Architecture

Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 01
Lecture - 03

Lec 3: What is Parallel Architecture?

Hello everybody. We are doing the module introduction to parallel architectures. This lecture number 3 is on what is a parallel architecture. To understand that, first we will have a look at what a basic computer system looks like and how do we parallelize it to make it a big parallel system. The previous two lectures we have been studying the requirement of having parallel systems, right. So, we used to have a single processor and its performance increased with the increase in the number of transistors, the more logic, the improvement on the ISA, more memory and caches getting added and so on.

So, single processor performance increased, the transistor density increased, the speed that is the frequency increased, clock speeds increased and we also were able to exploit ILP that is instruction level parallelism. But beyond the point, we could not increase the clock frequency because the frequency increase led to increase in the power consumption and also heat dissipation. So, heat increase and we could not manage that much heat on the IC and so we had to put a cap on the frequency of the processor. So, as the chips became too hot to be reliable, we had to limit the heat dissipation and hence frequency stopped.

So, how do we exploit the increasing number of transistors without increasing the clock frequency and still get better performance? So, for this we have to move the concept from a single processor to a multiprocessor, we have to go parallel. So, this is what we have seen in the past two lectures. Today we will look at how a parallel system can be built about. So, generic parallel system is the theme of the next few slides. So, what are the basic components of a computer? We all know that there is a processor, there is a North Bridge chip, there is a PCI chip and so on.

So, we will quickly have a schematic drawn about how a basic computer looks like. So, I will start drawing it here. So, we normally have a processor, the basic processor which can be a single core or a multi core system. So, suppose I have two cores in this, core 0, core 1, so two cores and then they also have a shared cache in between. So, cache is normally popularly denoted by the dollar symbol.

So, I will say that this is the shared cache. This is all on the chip and both these cores can be talking with or using an off-chip cache. So, I have a off-chip cache connected to this system. So, this is the processor. Now this processor has to talk with the peripherals.

So, it goes via the North Bridge. So, I will use short forms here. So, there is a North Bridge and what is the significance of this? It helps for high speed connection across the peripherals. So, this is the North Bridge, which connects with the peripherals using the PCI bus. So, the peripheral component interconnect bus is helping you to connect the peripherals with the processor.

So, the PCI bus connects us with the network and also with the disk. right. So, PCI bus connects with the network and the disk. The North Bridge being a high bandwidth interconnect connects us with the GPU and the main memory. So, I will just say DRAM here. OkaySo, the basic computer consists of the core, the microprocessor core connected through the North Bridge with the GPU if you have one on your system, then the DRAM memory, then it connects through the PCI bus with the network interface and the hard disk.

We have few more peripherals which do not require such high speed interconnect and therefore, they are connected through another chip called the South Bridge chip which connects us to slow IO interfaces like keyboard, mouse, printers, etc. . This bus between the processor and the off-chip cache is called the backside bus. So, this is the back side bus and this is the front side bus. So, this is how a generic computer looks like. So I have, that was a crude diagram.

So, this is a better picture of the same thing. Your microprocessor followed by a front bus connects to the North Bridge to the GPU and the main memory, then to the PCI bus connects to the disk and the network interface. Then we have the South Bridge connecting to the slow IO devices. So with this as one node, we can call this, this is the one computing node which has a single core or a multi core device has now to be connected to multiple such to create a parallel system. So how do we do this? So let us give a try of drawing such a generic parallel architecture.

So I will start with a very high level usage of the previous diagram. right. So I have a processor, then the processor is connected to a cache and we also have a local memory. So these are all interconnected to each other. So I will call this as one node and I can have several such nodes here. So I will just draw 3 as a representative diagram.

So you have processor, cache and the memory connected to each other and I will say one more memory, cache and processor. This is the basic computer diagram which we saw in the previous slide. So these three I have taken here and now they have to communicate with each other or get connected through an interconnect so that we can build a bigger parallel system. right. So I need a interconnect on the top which I can call as a global interconnect. So this is the global interconnect, it could be bus based point to point and so on.

Now every node has to connect with this global interconnect and I would say I will use interface to do this. So I use a network interface between every node and the interconnect. So there is a NI sitting in between here which helps us to connect a node to the global interconnect. And what about the disk? So if I have a hard disk, a larger storage device, even that needs to be connected through the same interconnect and hence I will extend this and connect it to the disk. Okay So this is how we can build a generic parallel architecture.

So a neater diagram is there on this slide. So we have processor, cache, memory, the network interface and each of this node is connected through a global interconnect to each other. Alright. So what are the main components? The processor node, then several such processor nodes are connected to the interconnect network. What is this network? It helps us to transmit data from one node to the other node. It can be a point to point interconnect or a bus-based interconnect and a node is connected to the interconnect via the network interface.

Other I/O devices, a disk or any other peripherals you want to add are to be connected to the interconnect. So what are the critical components of a parallel architecture? The processor, the memory hierarchy and the interconnect. Okay So these are the main components. We all have studied processors, but just a quick recap of what constitutes a processor. You all know that it has a central processing unit, arithmetic logic unit, fetch, decode logic etc.

So this is how a basic CPU is and you can have multiple such cores forming today's multi-core processor. If you have multiple cores on a single chip, it is called a chip multiprocessor. The more popular term is called a CMP. So CMP is nothing but multiprocessor with multiple cores attached on the same chip. And these cores will execute a common task coordinating with each other.

So this is how a multiprocessor system works. And every processor is these days mask manufactured. So you can, it is a more generic commodity device rather than specialized devices for most of the generic computing systems. They come with a single level cache or a multilevel cache hierarchy. All these cores these days are pipeline.

Earlier designs were bus-based, single instruction-based, but nowadays we have for risk based systems, most of these cores are pipeline. Once you have pipeline, we need to handle the data hazards, more complex ILP has to be exploited in this and so on. So the pipeline processor design is complex and this is the advantage we have got with the more transistors which Moore's law gave us. So we came up with better and better pipeline processors and further development was out of order execution where instructions which are not in the sequential order of the program can still be executed using dynamic execution algorithms. Okay

So that is how a processor is and it comes with pipeline and out of order features. Next is the memory. We all know that memory is consisting of starting from the registers to cache to main memory and disks. So that is the hierarchy of the memory. And why do we need memory? Because to do any computation the processor needs data and instructions.

These are going to come from first from the disk to the main memory and then of course we have the cache in which this content is copied before it is executed. So compared to processor memory is slow and why is it slow? Because of the physical design constant it has to face. So the first requirement from a memory is that it has to be non-volatile, because once I turn off the power the data which I have stored I want to retain it for future and it cannot be erased. So the memory has to be non-volatile and the more popular ones are the hard disk drives and the most recent solid state drives. So HDD and SSD are the non-volatile memory components.

Then these are the mainly disk-based system. Now from here I need to bring the data into the main memory. Now main memory size is also increasing. However we still say that it is much slower to access the main memory compared to the speed at which the processor runs. Why? The reason is because of the size of the memory the more addresses we are able to store in it and the data size.

So the wire delays are required to fetch the data from such larger memories is much higher due to the size and capacity. Okay So what did we see? We saw that there are physical constraints. We need non-volatile memory for permanent storage and the access time increases with the size. As I increase the size of the memory access time increases because of the wire delays. The larger the address I need more time for decoding.

We need more time to propagate the address to select the row, then to select the column the bit line propagation also takes more time. Okay So the main memory is slower because of the capacity. However, how do we manage this? We are able to cope up with this delay using caches. Right. So this can be solved to some extent by use of a cache. So with these physical constraints what is my next requirement? My next requirement is cost.

I need the memory to be as cheap as possible. Elaborating on the same argument, the speed gap between the DRAM and the processor is increasing. So DRAM is the main memory and processor accesses at gigahertz frequency. So if I take an example, if your processor is running at 1 gigahertz speed and the DRAM suppose it takes 100 nanoseconds to access one location and provide the data to the CPU. So with 1 gigahertz speed the processor is able to execute 1 instruction every 1 nanosecond potentially and the memory is going to provide 1 data item after 100 nanoseconds.

So we could have potentially executed 100 instructions in the time taken to access the memory. So you can see the gap between the speed of the processor and the speed at which the DRAM is giving me the data. So this gap is increasing day by day and it has led to the memory wall problem. So the statistics say that on average the processor speed has increased by 50 percent every year whereas the DRAM speed has managed to increase up to 7 percent per year and as you can understand the gap has been increasing year after year. We are able to offset this gap a little by the presence of caches.

So the better way in which we can access or manage these caches we can bridge the latency gap between the processor and the main memory. Okay So we concluded that memory capacity is high it is slow to access compared to the processor speed but we want the size to be more and also at the same time we want the memory to be cheap that is the cost should be lesser. So this table shows some example figures of a particular year of the prices of different types of memories. So if you see the L2 cache which is the on chip L2 cache if it is a 1 MB cache it costs you 20 dollars and the access time is very tiny 5 nanoseconds. So it is if I have a 1 GHz processor you will need 5 clock cycles to access this L2 cache.

Going to the main memory the size increases from megabytes to gigabytes but the access time also increases from 5 nanoseconds to 200 nanoseconds. Moving further to the disk from 1 GB disk to 500 GB disk the time scale shifts from nanoseconds to milliseconds. Okay So as we increase the size the access time increases and if I just try to draw a trend if the size increases the access time is the latency, the latency increases but the good news is your cost decreases. Okay See the cost reduced from 20 dollars to 0.02 cents from main memory to from sorry from the L2 cache to the disk.

Okay So what is the goal of your memory hierarchy? So memory hierarchy is register cache main memory to disk. What is the goal of this hierarchy? To give an illusion to the processor that you have you as a processor has a monolithic memory, a large amount of memory, 1 unit of it available for you and at the same time what is the access? The access time is similar to the processor cycle time. So we want to produce this illusion that I have a big memory able to access at the processor speed. It has a size almost equal to the disk space and also the cost equal to the disk cost. So that is our aim of giving a nice memory hierarchy to the processor.

Alright, so the third component was interconnect. So on chip interconnects, so interconnects lie in different levels. So the first is on chip interconnects. These are mainly those wires or interconnections between different pipeline stages in a processor. They are also interconnects between cores and caches and cores across cores. If I have multi-processor system I need to connect the two cores or I want to connect the core to the cache.

So all the interconnects which are on the chip are coming under this category. Then we have the system interconnect which connects the processor to the memory and further to the I/O devices. Then we have the I/O interconnect which is helping us to connect different I/O devices. Then we have inter-system interconnects which connects different systems. So we have the short area network, the local area network, the wide area network and beyond this we have the most popular global interconnect network.

Okay So in any multi processor system when I want to connect one node to the other, they need to be able to communicate with each other through this interconnect and the communication can be established with an interface which is called the network interface. And what is the role of this network interface? Essentially it copies the data from the interconnect to the local memory of that particular node. So are we going to study all of these interconnects? They are important but our focus in this course is going to be only these two on chip interconnects and system interconnects which help us to connect cores to cores, cores to caches and inter on chip communication. So this was a glimpse of how parallel architecture can be built. Now where is the parallelism to be exploited? So we will take a look at that.

Okay So we had scalar processors in the initial years. Scalar processor means it is able to execute one instruction at a time. So you can see an instruction written there on the slide. So this instruction add R1, R2, R3, so it says I have to do the operation R1 should be equal to the contents of R1 will be contents of R2 plus contents of R3. This is the meaning of this assembly language instruction add 2 and 3 and put the answer in register 1. And how is this done? We first have to fetch the instruction, decode the instruction, then execute and then store the result. Okay

This would take some amount of time if I say 1 cycle at a time you would need 3 units of time to execute this and depending on the processor this time might vary. Okay So in a single scalar processor you will take some number of cycles but I can make this faster using a pipeline because a pipeline helps you to do multiple tasks in tandem that is you can be executing a different instruction in a different unit. So I can have instruction 1 here. So while instruction 1 is in this stage of the pipeline I can have instruction 2 in this stage and instruction 3 in this stage. So in a pipeline system you might have an initial latency but the throughput is very high so you can get one instruction executed potentially every clock cycle.

So you can have more instructions getting executed if I have a pipeline processor compared to a simple scalar processor. Okay So more advantages would come if we are able to execute multiple instructions not only one instruction happening faster but I want multiple instructions to be able to execute together in parallel. So where are the avenues for this? So we have two avenues the ILP and the TLP. ILP is instruction level parallelism and TLP is thread level parallelism and we are going to take a look at vector and array processors. So these three points will give you an idea how to go from sequential to a parallel setup.

So what is ILP? ILP is instruction level parallelism and it is essentially the parallelism available within the instructions of the same program. Okay If the instructions are independent I can exploit this parallelism, if they are dependent on each other then we cannot exploit this parallelism because result of one would be required by the result of the other and who helps us identify this? The compiler and the micro architecture. So we will take a quick example. Suppose I have a small I quickly write a small sequential program where we have four assignments.

So we had a C program which was doing X was doing A plus B. Y of course was using the previous value of X and then doing something with it and M was doing E plus F. So these are my four instructions I1, I2, so, sorry I0, I1, I2 and I3. So if we try to dig out parallelism in these four instructions what we can notice is we have I0 and until I finish I0 I cannot start I1. Right. So I1 cannot start without I0 finishing. But I2 instruction is independent because it uses E and F as the input operands and writing to M so there is no conflict with the previous instructions.

So I2 is independent of I0 and similarly I3 is also independent of I0. Hence what I can say is these three instructions I can start in parallel. This is an example of an instruction level parallelism where I have a sequential program but within this program I try to identify those instructions which I can do in parallel. So if you see these 3 instructions can potentially execute in parallel to each other and we can thus obtain a speedup. Okay

We will take a more bigger example. Here I have a loop for i equal to 0 to 1024 so we have 1024 elements and we have to add them. So I have an array B of 1024 elements, array C of 1024, add those two contents and write the answer in array A. Okay So we need to see if I can exploit ILP or I can identify where is the instruction level parallelism in this code segment. If you are interested you can pause the video and think for yourself before continuing.

Okay So let me give it a try here. So we have two arrays A, B and C. So B is my input array and C is the second input array. We need to add the contents, the corresponding elements between them. So there is no dependency. If I am adding B0 to C0, I am not affecting B1 and C1.

So once we have this, so I can add this to 0 and 0 can be added. B1 and C1 can be added and all of these can be added independently, because there is no data dependence between each of them. And so potentially what we can say all the 1024 elements can be done in parallel. I can do all of these in parallel because they do not depend on each other. Okay So there is complete exploitation of ILP in this example. So potentially every instruction or every iteration of this loop can be done in parallel very quickly.

So what is the parallelism I get? 1024 times parallel I can do it instead of doing this sequentially. Every loop iteration is independent and runs concurrently. Okay This example if I am running on a machine which is able to do only one instruction at a time, a simple machine with no significant improvements, what will you do? Even if there is a potential of doing it in parallel I have no resources to do it in parallel. So we will do it one by one. But if my hardware supports pipelining or I have more multiple cores, we can exploit this ILP and delegate the task to multiple processing engines to get this 1024 times parallel output.

Okay So that previous example was the vector addition example. So processors would need help from the compiler to do this because as a programmer, we have written that for loop and we do not try to understand where is the ILP or help the processor exploit ILP. So whose job is this? It is the job of the compiler and the microarchitecture. So processor will be able to fetch multiple such instructions. So in this one,

$A[0] = B[0] + C[0]$

and

$A[1] = B[1] + C[1]$

So two loops if I unroll, if I unroll this loop twice, I will have two instructions and both of these can potentially be done in parallel.

So processor takes help from the compiler and is able to fetch and decode multiple instructions, multiple because every instruction is operating on a different pair of data items. Okay So we have seen an example of vector addition. Here, the processor has to take help from the compiler to fetch and decode a large number of instructions because we had 1024 possible instructions which can be done in parallel. So the compiler has to help the processor to unroll the loop and give this work to do. If you have a scalar processor, then several instructions can be scheduled statically, but the compiler has to do this.

But if I have a dynamically scheduled processor, then the microarchitecture can help us to schedule these instructions out of order, fetch several instructions and do it independently. So either the compiler or the microarchitecture helps to exploit ILP. As we understand, ILP is transparent to the programmer who writes only single threaded programs. Okay, moving on, we saw this example in the blue font about vector addition. The second example is a similar one, but here we have a single array $A[i]$ and the results get cumulatively added to the variable S.

Okay. So do we have ILP in this program? Again, you can pause the video and think whether there is ILP in the second example. Okay. So second example has got a dependence with the previous instruction. So the S in the second iteration depends on the value of S in the previous iteration, because it is S is equal to the previous S plus the new value of A. So there is a dependence between the instructions and hence we cannot exploit ILP in this particular example.

And hence this pink code will execute serially given any processor. Okay, moving on to the next type of parallelism, the thread level parallelism. Here the code has to be split into threads which will then execute independently on multiple cores. Okay, so there is a big sequential program, but it becomes a job of the programmer to divide this code into small task or small work segments which are potentially going to execute in parallel. But when we do this most of the time, there will be a requirement that these threads which we are creating will want to talk with each other, communicate with each other, exchange data values, synchronize with each other. Right.

So we need such support when we have a thread level parallelism. So what is the task of a programmer in a TLP? The programmer has to design new algorithms. So for TLP, we need new algorithms. We need to restructure the code. The code has to be rewritten to expose and manage the threads. And also we need to have program statements which help us to communicate and synchronize among the threads. Okay.

So ILP was straightforward, exploited by the compiler and the micro architecture. But for thread level parallelism, there is more onus on the programmer to write better algorithms, restructure the code and orchestrate for communication and synchronization. So let us continue with the same example, the same array,

$A[i] = B[i] + C[i]$

Again, I request you to think how can we write a multi-threaded program to do this task. Okay. So ILP was there because every iteration was parallelizable. In a thread level parallelism, how do we generate threads to do this? Definitely it is not auto-translated.

As a programmer, I have to think how to divide the task. Okay. So the most, an example which uses maximum number of threads will, I will say that thread 0 to thread 1024. Right. So I have these many threads and every thread is going to do addition of the corresponding elements. So thread T0 does

$A[0] = B[0] + C[0]$

and so on. That is similarly, say thread 100 does A100 adds to, sorry, in thread 100, B100 plus C100 gets stored in A100 and all of these threads can be running in parallel.

So T0 runs in parallel with T1 and so on up to thread 1023. So this way 1024 threads can be created to do it in parallel. However, because we do not have much work to do in every thread, the overhead of spawning the thread and then merging it would be more so we can decide to have lesser threads and give more work to every thread. Okay. So in this, I have shown an example where I have created four threads. The first thread is given one-fourth of the data, so from element 0 to 255.

The second thread from 256 to 511 and then so on. So I have four threads created. Each thread works on one-fourth of the data item. End of all these threads, our results are ready and there is no synchronization required essentially between these threads because they work on independent data items. Okay. The next example where I am having, where I am having a cumulative sum. So

$S = S + A[i]$

Again, pause the video and check whether you can write a multi-threaded program to do this. All right, so if I decide to make four threads or divide the task into 4 big segments.

So let us see instead of having 1024 threads, I am just making 4 threads dividing the data into 4 chunks, 255 size each. So we divided the data and then if we do a sigma over this first partition, we will get an answer. I will call it S0 because this is partition 0.

I partition the data into four, D0, D1, D2 and D3. So these are 4 partitions of the data. Every thread works on one partition and after doing the sigma of these elements, we get the answer in S0, S1, S2 and S3. So what is S0? S0 loop will run like S0 is S0 plus A of i where this i in this particular thread is from 0 to 255 or 254. Okay. So we get S0. Similarly in D1, we will not be able to add in S0 but what you will get is the sum will come in S1, here the sum will come in S2 and here the sum will come in S3.

We have got 4 partial sums here. However, the target is to get a one single answer. So end of these four threads, so we have created four threads, each thread has completed its addition and we have got four partial answers S0 to S3. Here comes the coordination and synchronization aspect that now as a programmer, we have to add these partial sums to create the final answer. Okay. So my final answer S will be equal to sum of all these partial sums. That is what we need to synchronize the threads that is when they all finish, collect the answers and add them. Okay. Right.

So these are the four threads. If you look at the top thread, it is generating the answer into my_S0. So my local sum 0, my local sum S1 for the next thread. So T0, T1, T2 and T3. So I have four threads, each computing its answer in its local variable my_S of something.

And then end of this, our program is not done. We do not have the final answer in S. To get the final answer in S, we need to write another small loop which joins these answers. So final S is going to be sum of all them, my_S[j] where j is 0 to 4. Okay. So we were not able to exploit ILP in this example, but we are able to exploit TLP in this example. However, it is not automatic. The programmer has to write a program to do it and the program is not as straightforward as the one shown in this corner.

So this is the original program. Right. Original program was only two lines, whereas a multi-threaded program will have some number of threads created. Some T every thread Ti will work like this and end of all these threads. So I have these four threads created T0 to T3 and when these threads finish, we need to synchronize it or join these threads and then create the sigma on the S. We have to add all the local sums to get the final answer. Okay. So can we automate this? So what do you think? Can I write a parallelizing compiler who will take a sequential program and create a multi-threaded program for us? Yes, there have been attempts.

However, all these attempts had some drawbacks and limitations. There has been limited success in auto-translation of programs from sequential to thread level parallelism. And why were we not successful? Because the efficiency of the generated parallel code is rather poor. It is very good if I have an embarrassingly parallel code that is for example, in the previous case where it is very obvious that there is so much parallelism, but it might not be the case in every sequential program. The second reason is there are techniques which try to do this, but in presence of pointer variables they fail because at compile time, we do not know the address to the pointer we will dereference to. So the values of the pointers are not known and hence it is difficult to identify whether I can parallelize this code or not.

So in presence of pointers, it became difficult and also the generated parallel code was not always the best solution. So best solution could be a redesigned completely new algorithmic solution rather than an auto-translated one. So although these methods are there, auto-translation, we still prefer to write programs ourselves as programmers. Also the algorithms which you would write under TLP should be tailored keeping in mind what is the size of your multiprocessor. If I had four cores, then my previous example would be best, but if I had 1000 cores, I would rather make more threads than only four threads.

But if I had only two cores, there was no point in making four or ten threads. So depending on the size of the multiprocessor, your algorithm should be tailored accordingly to have the best TLP output. Next we will take a look at vector and array processors. So compared to scalar processors which do one instruction at a time and these scalar processors exploit ILP in the best way. But to exploit TLP, we need more processing elements.

So apart from multi-cores, are there designs which help me to do execution of multiple instructions at a time. So vector processors and array processors give us this feature. Vector processors use pipelining and array processors use parallelism. Essentially they have multiple processing elements. So we will again take example to understand both these designs.

Same example,

$A[i] = B[i] + C[i]$

If I want to translate this to a instruction in a vector processor, it is very simple instruction. We have V add that is vector add, sorry, V O1 that is V O1 is a vector. So this is vector 2 plus vector 3 gets stored in vector 1. So that is the semantics of this V add instruction.

A single instruction which replaces this loop here in a vector processor. Okay. So what is stored in V O1? The array A from 0 to 1023. So the complete array of 1024 elements is represented by the vector 1. Vector 2 is array B and vector 3 is array C. The vector processors use pipelining to do this.

Once you have pipeline, that is one addition will happen at a time one by one. So you will send one addition, when that is happening the second addition is scheduled. So they will go one by one inside the pipeline and definitely this is not going to happen in a single cycle. It is going to take multiple cycles to complete the whole operation. If I have 1024 additions to do, they will take multiple cycles.

But the goodness of a pipeline vector is that you will get some output every clock cycle. So every clock cycle I am going to get one addition output. So how do these processors look like? For this particular example I am illustrating, a vector processor would have registers or so called vectors which look like this.

So vector 2 and 3. So vector 2 was 1024 elements. So we have one element in each of these. Okay. So every entry is one element. So you can see here, every entry or every row in this box is one element. Okay.

So we have 1024 elements stored in these two vectors. Next we need to add them. So we will adder at the bottom. So this is an adder but it is a pipelined adder. Pipelined because we will be able to schedule one instruction every clock cycle. Of course this does not happen automatically. You need a control unit to do this.

So the control unit has to give an instruction to the adder to and then also instruct the vectors to send the data out. So we get one data item and instruction goes to the adder. The adder performs the addition. So index 0 plus 0 gets stored in the answer index 0. Index 1 plus 1 gets stored in answer index 1.

So to store these answers, I also need a similar vector at the output of the adder. Okay. So my adder answer vector V01 is also a vector of 1024 entries. Okay. So each of them will store the corresponding answer. This is how a vector processor will be executing that particular instruction. So it has a pipeline, a control unit which fetches the vector instruction and issues it to the pipeline arithmetic unit.

You get an output every clock cycle and the results are stored in the memory at the same rate. So the memory also should be efficient enough to gather the answer every clock cycle. The next example is of an array processor. This is not pipelined.

This is a parallel system which has got multiple processing elements. We do not have a single adder but we have multiple adders. And when you have a processing engine or a processing element, it needs a local memory. So once several processing elements are there, every processing element has its local memory. It's the job of the control unit to first allocate data across these memory elements in respective processing elements and then give an instruction to the processing elements to compute on the local data, and produce the answer and store it in its local memory. Okay. So we will again rerun the example on the slide. On the left you can see the vector processor which we have just seen and then on the right hand side let's see how the array processing works.

So every array processor has got an execution unit or a processing element. So this is the processing element or the processing engine. So I have several of these. There are multiple of these. All of them are connected to the control unit because control unit fetches the instruction from a global memory and gives the work or copies the instruction to each of these execution units. Okay.

Then how are they going to execute? They need memory from where to fetch the data. So each of them has got a local memory and the vectors which we were talking of V02 is my vector B and V03 is my vector C. Okay. If I want every processing element to add corresponding values then if B0, C0 is in this element I want B1, C1 to sit here and B 1023 and C 1023rd index to sit in the last memory element. So we are having an interleaving of the data items. So the data items have to be interleaved across all these memory blocks, each of them keeping the corresponding pair to operate upon. Okay. So the complete vector, we do not have vector 0 stored in the first green box and vector 2 in the second green box but we have every element of every vector available, the corresponding elements in each of these local memories.

So how does this work? The control unit sends an instruction to every execution unit. Each execution unit fetches the corresponding pairs from its local memory and writes the answer back to the local memory because with these two, we also have A0 allocated space in the same memory. Okay. So the corresponding all three indexes are in the local memory of the array processor. So vector processors use pipelining, array processors have parallelism and then there is compiler support to do this for us. Vectors and array processors have been there for quite some time and auto translation and auto execution of all this is possible and this is a mature technology.

So there is not much challenges in this, it is an established way of doing parallel processing. So the same vector architecture which we saw in the example of the adder pipeline, I am extending it to multiple such vector units. If we go back here we had this vector architecture where there were two registers that is V01 and V03 which were going to the adder pipeline and then the control unit sent the appropriate instructions. Two operands got added at a time and got saved in the orange V01 register. Okay. So I can extend the same architecture to duplicate the units called the vector lanes because we cannot be handling only one adder, we want more parallelism, more throughput.

So I will extend or copy one vector unit into multiple called vector lanes. So here lane 1, lane 2, lane 3 and so on. Okay. So there is a master controller which will distribute the workload across every lane. Every lane would suppose calculate 256 additions, the second one 256 and so on. So at the same time we can have lot of additions getting completed by the lanes of these vector units. Okay. So these vector processors are normally having lanes.

I showed you an example of adder but there could be different functional units to do various arithmetic and floating point operations. So these are called vector functional units and the controller has to decide to schedule the operation on a particular functional unit. This will improve the parallelism to a big extent. Similar to array processors which are doing things in parallel, the vector processors also do things in parallel on multiple data items using multiple vector functional units. So these have special hardware to accelerate the memory access because you will imagine that so much data is processed at a time so you need to fetch that much data from the memory.

So once you need lot of memory accesses, very high memory bandwidth is required to fetch all these items. So that is all about vector processors. The next example is of GPUs. You all are familiar about GPU. They are graphic processing unit and it is a hardware design for highly parallel applications, mainly used for graphics which requires fast and better rendering.

So we require quick rendering techniques and hence lots of arithmetic units are required. So ALUs in particular. Again similar to vector processors, these also need very high bandwidth. So we need a very high bandwidth here because lots of operations are to be performed. So equivalent amount of data should be brought, because I have several ALUs or several arithmetic units available, we are effectively doing a hardware based multi-threading.

So GPU is a very highly parallel hardware architecture. Just to give you a comparison, so this picture showing you a layout of a CPU which has one control unit say 4 to 8 ALUs at the top. So you have some number of ALUs, a cache and a DRAM. And how does the GPU look like? So the GPU, the color coding is matched here, the blue is the control. So we have a tiny control available along with a small cache and then all these pins are the lanes of the arithmetic unit.

So I can say that this is the ALU inside the GPU. So we have a grid of ALUs available and each grid is managed or controlled by its independent controller and a local cache. All of these also share a global memory and eventually also communicate with the DRAM to do the major read and write operations. Within this GPU, we can logically say that it is partitioned across something called a SM.

SM is streaming multiprocessor. Okay. So SM is streaming multiprocessor. So we have several of these. So a grid of streaming multiprocessors is laid out on the chip. Most of them share the common L2 cache connected to the global memory. Then of course, we have the thread scheduler and the memory controller and so on.

What is inside every streaming multiprocessor? It has the cores which are the processing units. So here is where the processing happens. Then the associated shared cache, the register file and a warp scheduler. So what is a warp? It is seen in this diagram here where every work is divided into variety of threads.

So there is a thread 1 to say thread number 3. So we have multiple threads and every thread is further divided into warps. So we have first threads. This thread is divided into a warp. Okay. And then these get scheduled on the streaming multiprocessor. So this is how we achieve parallelism in a GPU. It has multiple arithmetic units and lots of work has to be definitely given to this. Okay.

So generalizing the idea of a GPU or a vector processor or an array processor, this picture is showing you a grid of processing elements. They could be any functional unit or a generic functional unit which can be reconfigured to whatever task you want to do. There is a master control processor which delegates the task. There is a memory which is not shown in this picture, which provides the data to these processing elements. Alright. So here the operations are performed all in parallel, each one operating on its own set of data items and then storing the results in local memory, later forwarding it to a global memory.

The only thing is this is very efficient if you have a symmetric data structure like arrays, vectors or matrices. Okay. So all these examples of parallel architectures and the available parallelism which we have been trying to understand can be categorized under these four types. So this was categorized by Flynn, so called the popularly called the Flynn's taxonomy. It's devised the design in terms of the distinct instructions that are issued at a time and the number of elements it operates upon.

So there are four categories. S is for single and M is for multiple, I for instruction and D for data. So you have four alphabets S, I, D and M, so combinations of them. So SISD is single instruction single data. It takes a single instruction and works only on a given single data item. That's the old style uniprocessor system.

Then the other option is single instruction multiple data, SIMD, which takes one instruction but operates on different data items. For example, the GPU, vector and array processors, all of them were doing for example an addition. So the operation was add, but they were adding different pairs of numbers. So the data was different, but the instruction was same. Okay. Then the third one is MIMD, multiple instructions multiple data. Multiple instructions essentially says we have different programs to run or different threads as well because every thread could be operating on a different instruction, it could have a different control path executed from the same parent program.

So multiple instructions are either thread level or multi program workloads and because they are different programs, they naturally operate on different data items. So multiple data items and multiple instructions. And the last one, of course, it doesn't have any meaning, but just kept for completeness multiple instruction single data. So this has no real implementations, but just kept here for completeness. So that was Flynn's taxonomy. So to summarize this lecture, we have discussed instruction level parallelism, then thread level parallelism, then we move down to data parallelism where we could operate on different data items, but do the same type of an operation.

Other types of architectures that exist are systolic and dataflow, which we won't cover, but interested students can read up the literature to understand. So overall, this is the taxonomy of the parallel architectures. All right. Thank you.