Advanced Computer Architecture
Dr. John Jose, Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati, Assam

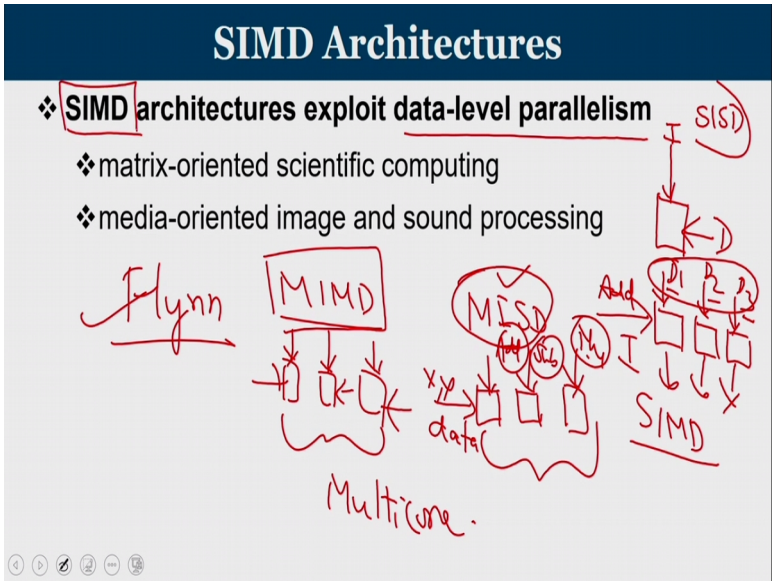
Lecture 12
Exploiting DLP: Vector and GPU Architectures

Welcome to the 12th lecture on exploring data level parallelism with the help of vector processes and graphics processing units. Over the last, close to 10 lectures, our focus was on instruction level parallelism and the concepts of instruction pipeline with a simple in-order scalar pipeline to out-of-order scalar pipeline out of order super scalar pipeline, we have seen all these aspects of this. So, in all of these scalar and super scalar processor, the instruction pipeline was designed to explore parallelism between instructions.

We were trying to overlap instruction execution. And there was techniques that were explored in order to fetch instruction parallelly decode instruction parallelly and execute instruction parallelly. Whenever there were dependency between instructions, the hazards that were caused, because of that were addressed with the help of sophisticated functional units. When we work on advanced processors, we can improve performance not only by running instructions parallely.

We can also exploit techniques where data level parallelism also can be exploited. In this lecture, we are giving a quick glance on how can we explored data level parallelism. So, parallelism is in different levels, out of which the most important and the commonly exploited 1 is instruction level parallelism. This is a variation from the total approach. We will see that in today's lecture.

(Refer Slide Time: 02:24)



Coming into SIMD architectures, SIMD stands for single instruction multiple data stream architectures, our focuses is on to exploit data level parallelism. Let me draw your attention to let us say this is a processor, and you are going to bring instruction, and you are going to get data. So this is the data this is the instruction. So we have single instruction and single data, we called in us SISD processor, single instruction, single data stream processor.

Yet another aspect is let us say we have different processing units. And they all are supplied with the different data. So this is data 1 data 2 and data 3. This is the value of AB, this is CD, this is EF like the different data is there. And that is I am going to give 1 instruction, let us add, so all of them will take their corresponding data, and are going to give you the result of that. So, it is basically a single instruction but we have multiple data they are called SIMD single instruction multiple data stream techniques.

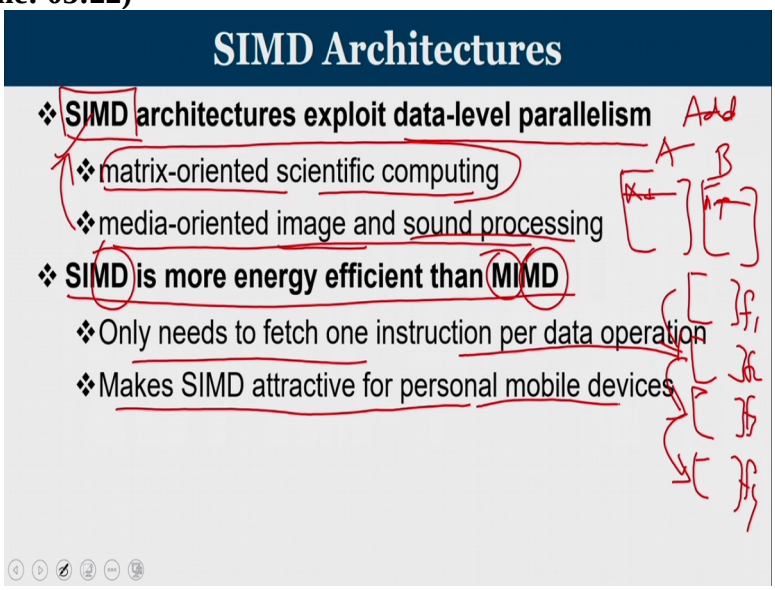
The third approach is when you have multiple instructions on a single data. So here you have the same data that is going to come. The data is same you just imagine, but I have different instruction that are been separated said this is an add instruction, this is subtract instruction, this is multiplying instruction to be done on data X and Y. This is the scenario of multiple instructions are to be operated on a single set of data.

But generally these kinds of setup is highly unlikely. Because, such kind of programs won't not make much difference adding subtracting or whatever be the operation to be done on the same set of data. And the last category is called MIMD multiple instruction multiple data stream. So here you have different functional unit. This is operated on its own data, this is

operating on its own data and instruction. Here is also every functional unit is been supplied with their own set of instruction, and they are going to operate with their own set of data.

So multiple instructions are working together on multiple data this is exactly the concept of multi core processors MIMD concept and this whole classification of processors into SISD single instruction single data stream. Single instruction multiple data stream, multiple instruction single data stream and multiple instruction multiple data stream is known as Flynns classification. Our today's topic is on what are the different types of popular architectures that work on single instruction multiple data streams.

(Refer Slide Time: 05:22)



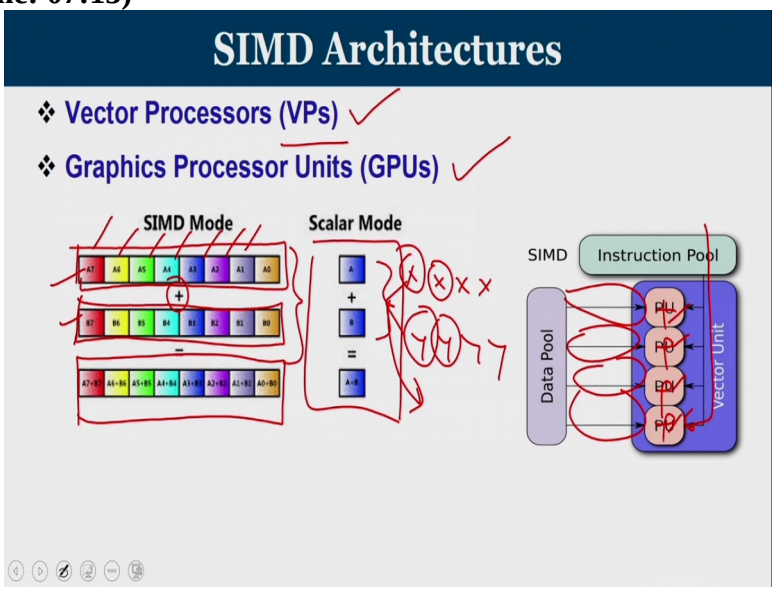
When we have matrix oriented scientific computing, let us say were we are going to have huge matrices and I wanted to add may be corresponding rows and columns of these matrices then basically it is add operation that is to be performed, but on different pair of data. So in matrix oriented scientific computing, then it is a kind of a single instruction multiple data stream operation. And when we are going to work on images and sound.

They are also the data is represented in terms of matrix across time periods across time samples, and we are going to operate on it. So from that said, this is the pixel value of a video at any given point of time. This is the pixel values of the next frame associated with the video. So like that, it is a set of pixel values for frame 1 frame 2 frame 3 and frame 4 the same video there may be having some changes, which gives us a feeling of it is a motion picture.

In all these cases also we are transforming this matrix into will apply another transformation going into the next set of pixels like that you apply a sequence of transformation in order to get the result. So in media oriented image and sound processing also, we require or it is off the pattern, single instruction multiple data stream. Single instruction multiple data stream is more efficient, energy efficient that multiple instruction multiple data stream, because I am just trying to feed in only one instruction.

And then you need to take care of only parallel aspects of data. But as in this case, we have to take care of parallel aspects of bringing multiple instructions as well as multiple data. We only need to fetch 1 instruction per data operation. And it makes single instruction multiple data stream attractive for personal mobile devices.

(Refer Slide Time: 07:13)

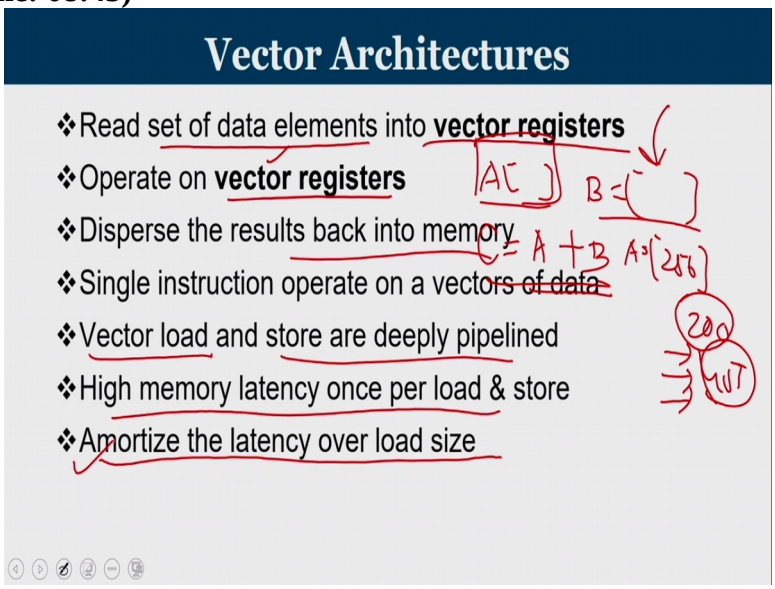


So vector processors and graphics processing units are the 2 most commonly used architectures in the SIMD category. So think of a case let us say I wanted to add 2 values A and B. And then this is called a scalar operation. I am going to add A + B or going to do the operation A+B here rather. Now, consider that a consists of many sub elements, let us say A is an array, and similarly B is also an array.

Generally what we do is we take the corresponding element, it is been brought to the ALU and then you are going to add the and store the result. Take the next pair of elements do the operation. That is called a scalar mode of operation. What if we can represent A in a bigger data structure where many elements of A can literally coexist and that of B and can we have an operation that is been defined on these huge structures.

So, that I will get the result in together. This is the concept of single instruction, I am going to tell only single instruction A on multiple data even A2 A3 A4. So, this has been done with the help of this vector processors. So we have different processing units, and then you are going to supply the same instruction and all of this processing units are being supplied with. So, this basically the same adder all are add operation. So they are being triggered by the add operation with different set of data.

(Refer Slide Time: 08:45)



Coming to vector architectures, what they basically do is they read a set of data elements into huge registers or registers that can accommodate multiple values or multiple operands called the vector registers. And then you operate on vector registers, that is A can be a vector register, which can accommodate 1000 elements, B is another vector register, which can accommodate another 1000 elements and can I defined C another vector register, which will store the result of vector addition of A and B. So A and B are not your symbol 8 bit register or 16 bit registers, they may be register of capacity more than 1000 and 10000 bits.

where I can store multiple words more than 100 or 200 of words. So you operate on vector registers. And then once you get the result the results are pushed into memory. And only a single instruction is operating on the vectors of data. But prior to that, we need to load the vector, vector load on vector store, they have to be done and they are basically pipelined operation. So it takes some time for the first data element to reach the register.

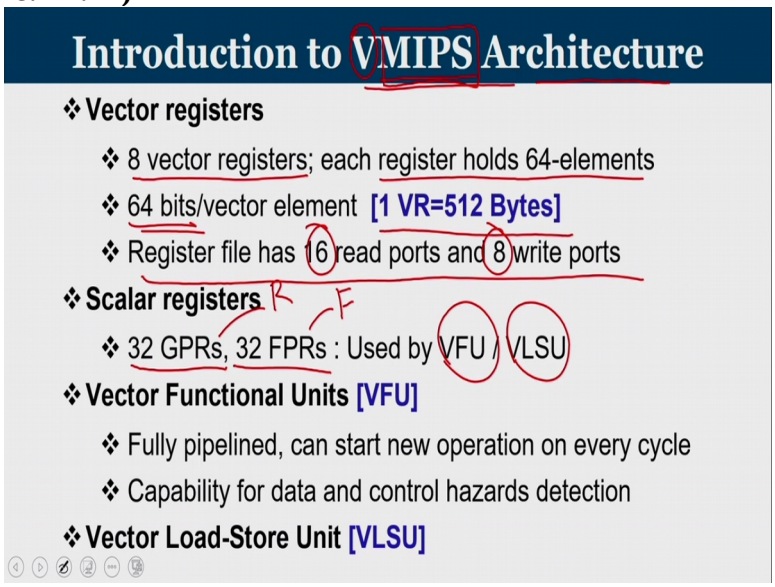
But parallelly you have already initiated the load or store operation of the next element in the vector. And generally this vector loading because you are the load, lot of numbers into this

vector register, it is a high memory latency operation, but when you amortize the cost, then you will get it simple. So, let us say we wanted to load into a vector register A, which has 256 locations. The first one will be loaded only at the 200 clock cycle.

Thereafter in everyone 1 clock cycle, you are going to get 1 more data. So, roughly around 455 cycles, the entire loading is over by virtue of it makes close to 2 cycles per element. But even though we encounter an initial latency, then the subsequent loading will be very easy. So, we have to find out amortize the cost of the latency over the total load size that we are going to work on.

Their access to special category of architects are known as we VMIPS architecture, it is called the vector MIPS architecture. When we studied about instruction pipeline, we were generally taking VMIPS as the case study MIPS processor. So, this is the vector version of it vector MIPS architecture.

(Refer Slide Time: 11:14)



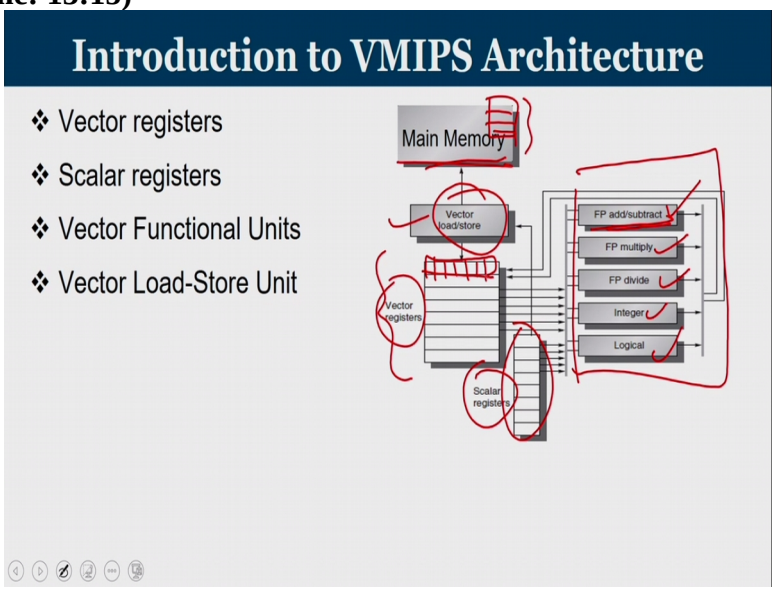
What are the peculiarity, they have vector registers, there are 8 vector registers, each register can hold 64 elements and each vector element is 64 bits. So, 1 vector register is now going to be 512 bytes of data. And then each of these registers or generally the register file has 16 read ports and 8 write ports. And along with the vector registers, we have the scalar registers as well 32 general purpose registers, they are called integer registers and another 32 floating point registers.

So, integer registers are represented by R and floating point registers are represented by F and they are used by the vector functional unit and the vector load store unit. And we how fully pipelined vector functional units that can start a new operation on every cycle that means, the initiation interval of all the vector functional unit is 1, they have a capability for data and the controllers are detection. So, once you detect the hazard appropriate operations and operand forwarding can be done.

And then we have a vector load store unit also which has an initial latency and once the initial latency period is over, thereafter, in every subsequent cycle, 1 element is getting loaded. So, coming into the VMIPS architecture that is being discussed, it is an architecture where you just special 8 vector registers and these vector registers can store multiple elements. So, altogether 1 vector register is 512 bytes.

And then we have the scalar registers also, which is used along with the vector registers in order to load values and in order to carry out operations, 32 general purpose registers called integer registers as well as floating point registers we have and then we have fully pipelined vector functional units and vector load store unit, this is the general architecture of the VMIPS system.

(Refer Slide Time: 13:15)

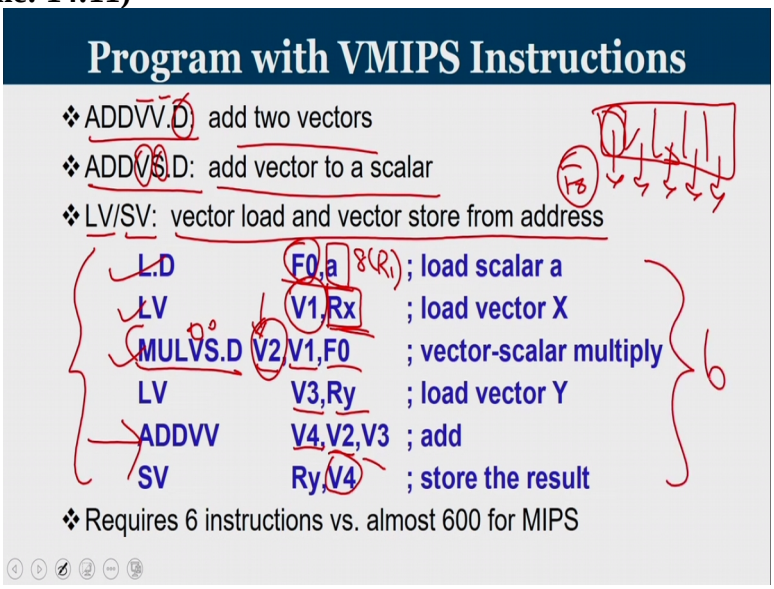


Now, Introduction to the general structure here coming into the architectural diagram, we have a main memory where stores your data and then from there we have this vector load store unit is there and then we have vector registers. And these are the scalar registers and

both vector and scalar registers are to be used by the integer unit then you are floating point add or subtract unit, multiply unit, division unit and the logical operation.

So, these are your vector functional units, which interact with the registers the vector registers as well as scalar registers they can be brought in and brought out and we use a special vector load store unit in order to load the contents into the vector registers. So, from memory multiple elements are being loaded into these vector registers. So, these are all huge registers, like I can store 1 full matrix inside a vector register.

(Refer Slide Time: 14:11)



Let us see some simple programming instructions that has been used, which will help us to understand how can write a vector MIPS code. When we wanted to add 2 vectors, we generally use this statement

ADDVV v4,v2,v3

on a double data, adding of 2 vectors. I can add a vector with a scalar value. So, every element in the vector has been added of the scalar value add vector to a scalar I can load vectors I can store vectors vector load and vector store from the corresponding addresses.

Even if you do not know the syntax try to understand what is the program that has been given. First, I am going to load a scalar value into F0, so loading scalar. So you know, this a has to be mentioned as 8 of R 1 the conventional way in which VMIPS instruction is been given. Then we have loading of a vector so, we specify from where the loading has to

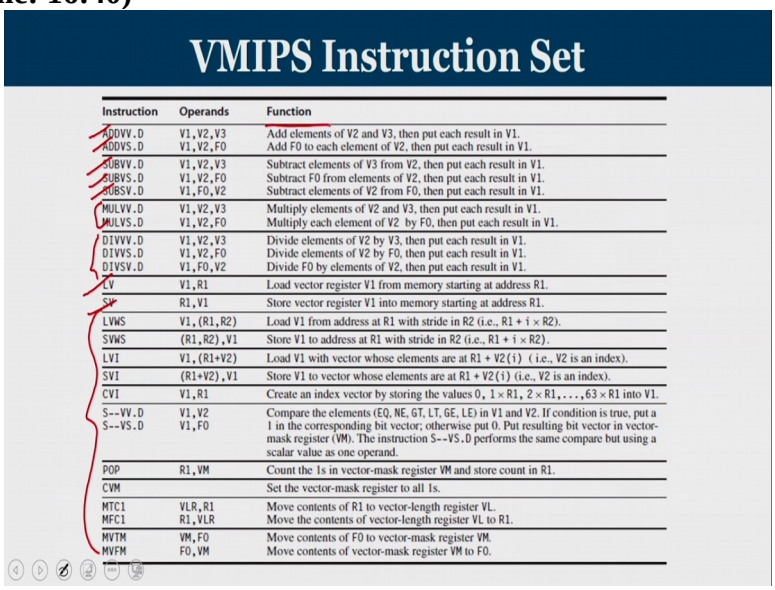
happen, and what is the destination register. So, into vector register V1 we are going to load a huge vector located from RX onwards.

Now, it is a multiplication offer vector as well as a scalar. So, V2 is the resultant V1 into V 0. So V1 is a vector register, which has multiple elements in it, take each element multiply with F0. So, that is the operation that has been done. So, the operation is multiplication with 1 of the operand is F0 and another operand is 1-1 element from the vector. So, in a conventional scalar process are you required to take the first element individually multiply it and then store into the next array.

So, here this 1 instruction will take care of a vector scalar multiplication operation. And then we are going to load another vector into a vector register V3 and then V4 is whatever is the result that you have in V2 that we are going to add with another vector. So here it is a pure vector addition, whereas in this case, it was a vector scalar multiplication, and then at the end, we store the result into Ry.

So, overall, the instruction looks similar to that of MIPS, only thing is somewhere we have to understand, when a register is considered as vector and what is the operation. So, even though you see it is just 6 instructions, but in a normal MIPS it takes roughly 600 instructions to complete.

(Refer Slide Time: 16:40)



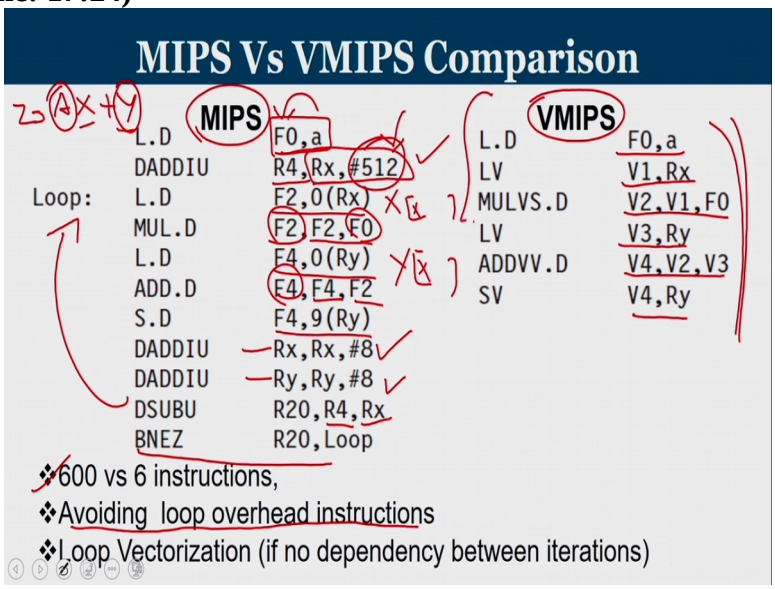
Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to vector-length register VL.
MFC1	R1,VLR	Move the contents of vector-length register VL to R1.
MVTM	VM,F0	Move contents of F0 to vector-mask register VM.
MVFM	F0,VM	Move contents of vector-mask register VM to F0.

And there are a lot of vector instructions, this chart will give you some input regarding what are the possible instruction combinations. I can add vectors together I can add a vector and scalar then subtraction of 2 vectors, subtraction of a vector from a scalar and the subtraction

of a scalar from a vector multiplication of 2 vectors, multiplication of vector and scalar, then division involved in vectors and scalar loading and storing of vectors.

And then configuring different special control registers associated with vector quantities. We are not going into the details of this syntax, you can just refer and find it out in their which context these instructions are being used, the functionality of each of instruction is also been given here.

(Refer Slide Time: 17:24)



Let us know compare same activity in what way a vector MIPS processor is better in terms of number of lines of code as far as an activity is concerned. So, in the previous case, we were trying to understand an instruction where you are going to perform a multiplication operation on a vector and then you are going to add another vector that say x and y are the 2 vectors that we have discussed initially.

And then we are going to perform a scalar multiplication on all elements of x, whatever you get, you are going to perform a vector addition and that is going to be your result. First let us look into how are we going to do we are going to load a value A that is the scalar value. So A is getting loaded into F0. And then we are making

R4 Rx 512

So, this is going to be used as the loop termination condition later.

So, you load the first value into F2 from 0 of Rx, multiply F2 with F0. F0 is a scalar quantity that we are talking about multiply F2 with F0 stored the result in F2 load the next element, so this is your element x, and this is relevant y. So after loading x multiply with F0, and the load

y and that the value of y in order to get this quantity x+y and then store the result. Now, there are many such elements, x is a huge array y is another huge array, now we have only taken care of the first element.

So now you move to the next element, you are going to increment the value of Rx and Ry. And then you are going to subtract the value of Rx from R4. So initially, we have made R4 value and Rx value appropriately with an index of 512. So every time you subtract, we are just taking whether upon subtraction, whether they are going to become equal or not. So initially how made R4 to be 512 values above RX.

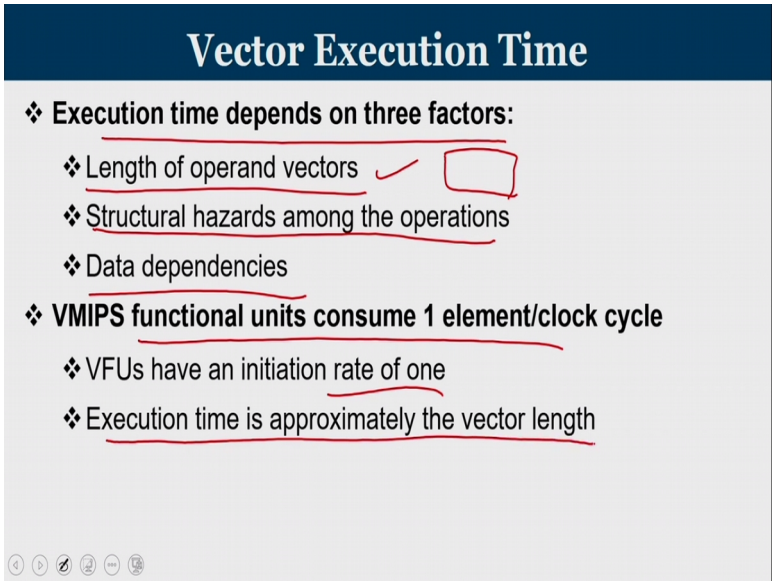
So every time when we subtract, we are going to get the new updated value so content of R4 has been subtracted by 512. That is your vector value. And then you are going to repeat the same thing. So you can see that you access each element and then perform the operation of a scalar multiplication. And then you are going to add another element appropriately change the indexes such that in the next iteration, I can go for it.

See the whole set of operation I am doing in a vector MIPS, we load scalar value, load the first vector which is a vector to scalar multiplication, load the next vector, it is a vector to vector addition and then store it back. So essentially, there is no loop with the single instruction is taking care of all of the operations. So, rather than 600 times or the 600 instructions, we have in a conventional MIPS versus 6 instruction in the case of VMIPS.

So, we are avoiding loop overhead instructions, there is no loop instruction. So, loop vectorization is also being done with the help of this VMIPS. So, here if you can see that the same operation of loading and element performing some scalar or a vector operation and then storing it back rather than picking one by one and then repeatedly doing it in a loop which involves loop variables and loop instructions as well those things are being get rid off.

And we are using the vector operations supported by a VMIPS processor. In this way, a same operation that is to be carried out on a huge data set that is called single instruction, let us say multiplication to be done not all the elements, adding to be done not all the elements. So rather than individually accessing them and doing, can we access them as bulk and perform operations, where in many quantities can be simultaneously been operated on.

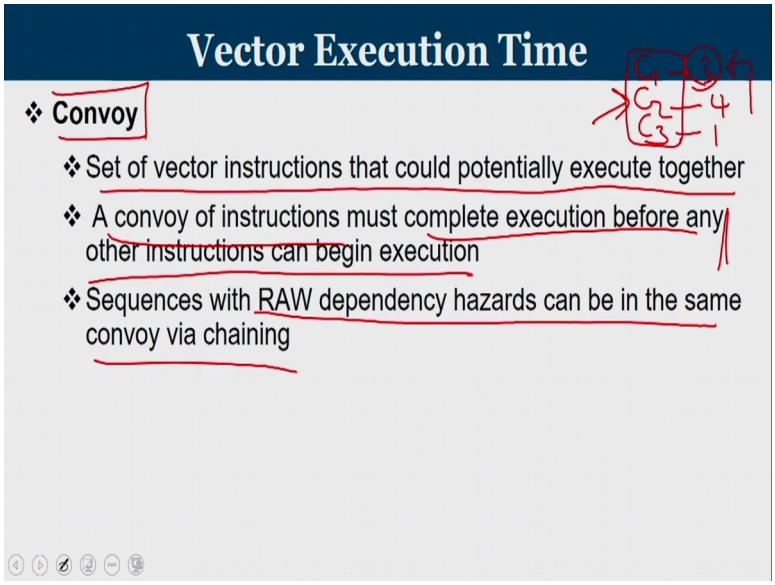
(Refer Slide Time: 20:49)



Let us know try to understand what are the parameters that will affect the execution time of a vector instruction code. Execution time depends on 3 factors first is the length of the operand values whether my vector register hold 10 elements 20 elements for 50 elements. Structural hazard among the operation, when we do operations, sometimes the functional unit that is performing the operation may be busy in executing the previous elements.

So that will create a structural hazard factor, structural hazard will delay the operation. And then there can be data dependencies between 1 element and the next element. In that case, we cannot execute them together. Generally, we assume that apart from the initial loading, operation or initial latency called startup latency, all VMIPS functional units consume 1 element for clock cycle. So vector functional units have an initiation rate of 1 means in every cycle. I can start 1 pair of operation. Execution time is approximately is equal to the vector length.

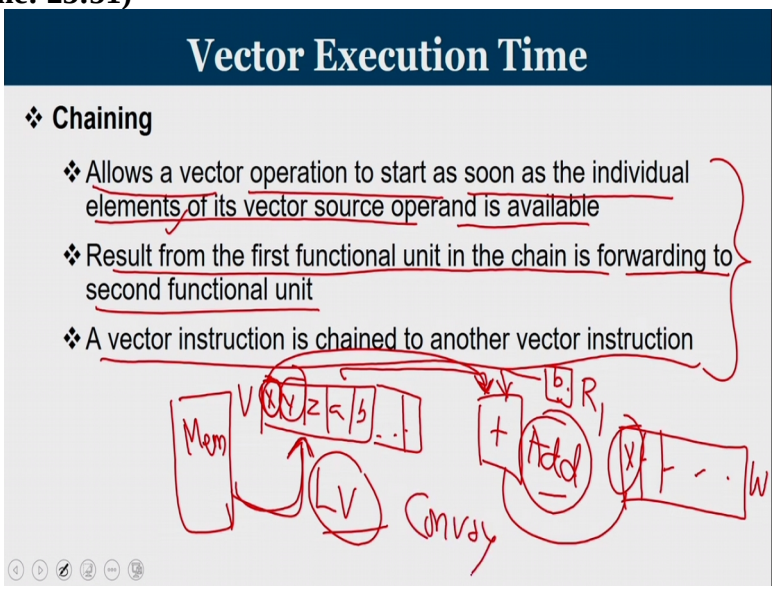
(Refer Slide Time: 22:51)



One important term that we have to understand while working with vectors for vector architecture is a term called Convoy. Convoy is nothing but it is a set of vector instruction that could potentially execute together a Convoy of instruction must complete execution before any other instruction can begin. Let us say in my program, I can broadly divided into 3 convoys C1, C1, C3, its C1 consists of 3 instructions, C2 consists of 4 and C3 consists of 1. All instructions in convoy 1 will execute, either paralelly or with a slight delay.

Instructions of C2 cannot start until instructions of C1 is over. So a convoy of instruction must complete execution, before any other instruction can begin execution. Any sequences with RAW dependency hazards can be still there in the same convoy via a technique called Chaining.

(Refer Slide Time: 23:51)



So chaining is a technique that allows a vector operation to start, as soon as the individual elements of vector source operation is available. Let us imagine that you are going to load into a vector. So from the memory, we are going to load into the vector one by one, X Y Z a b, there are all elements like that it goes. So huge vector. Now my task is to take each of these elements and going to perform an operation, let us say an add operation.

With 1 of the element coming from here, and the second element coming from a register called R 1. So assume that the functionality is the first we have to load a vector register that is a V from the memory, so take lot of time to fill up the entire vector register V. And then we how to add all these elements one by one, and I am going to store in another register W. So this W the first element is nothing but

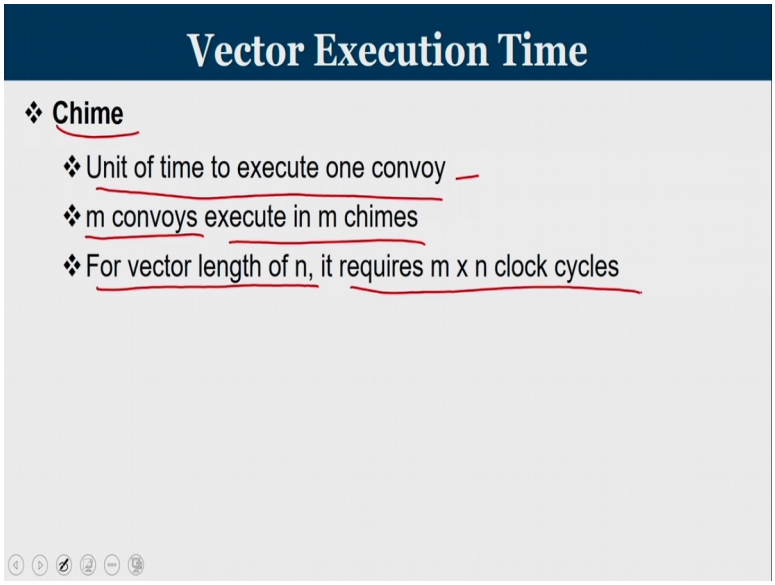
X + the content of (R1)

Let us say the content of R 1 is small letter B. So, that is going to be added here and this is going to be the result. Now you take the next element perform the same addition. So here there is a dependency first I have to make sure that this is a load operation. And here is an add operation. The add operation is dependent on this load. So in this case, by a technical chain, even though you feel that because of the data dependency my add gets delayed.

But add won't wait for the entire vector register to be populated with the elements. As on when X is available, I can pull up the X into this adder in the meantime, Y, Z and other these copying from memory into the vector register is happening in the background. At the same time, since the value is available, it is already been taken and population of this happens. So loading as well as adding happens parallely.

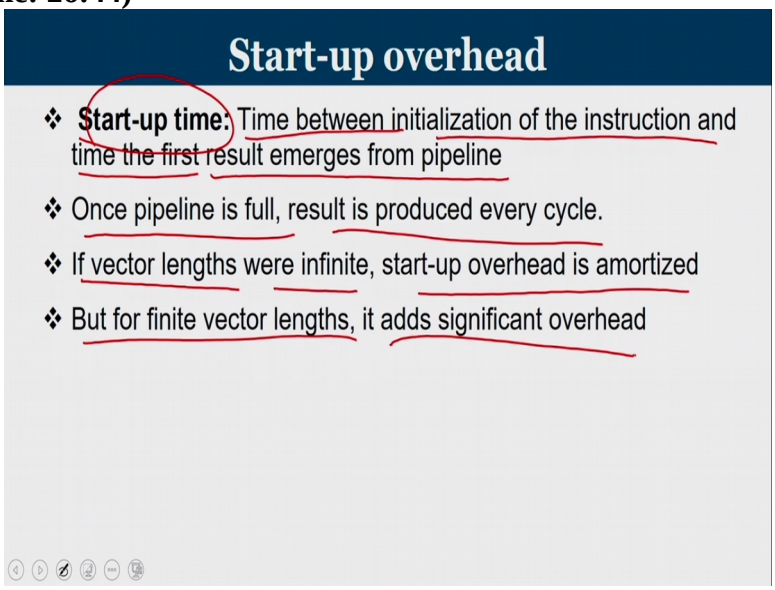
So this technique of allowing a vector operation to start as soon as the individual element of the vector source operand is available. The result from the first functional unit in the chain is forwarded to the second functional unit. A vector instruction is chained to another vector instruction. So is by virtue of a technical chaining, I can club multiple instruction to be part of 1 convoy. So in this case, all of them are part of 1 convoy.

(Refer Slide Time: 26:23)



Now we draw your attention to another term called Chime. Unit of time to execute 1 convoy we tell that we will take 1 chime to execute 1 convoy, so there are m convoys in their program, they will take m chmines to complete. So, for vector of length n it requires m into n clock cycles to complete the operation.

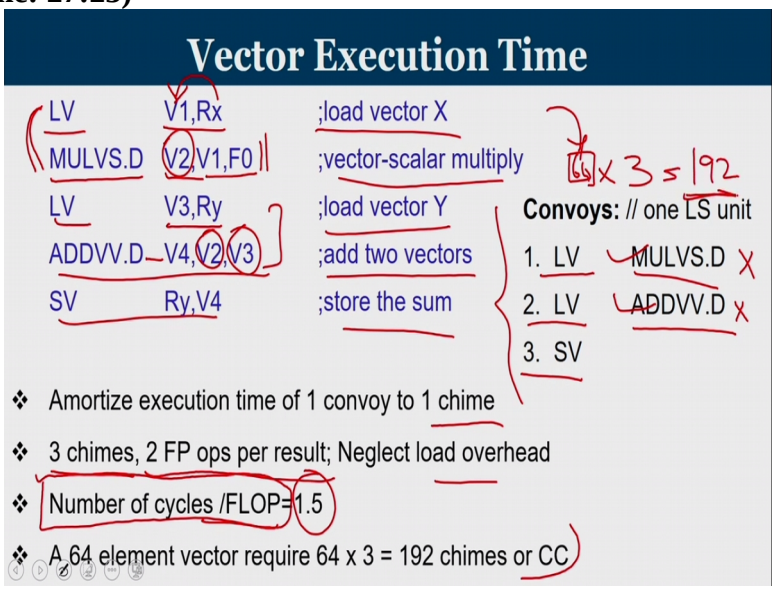
(Refer Slide Time: 26:44)



And always there is a startup overhead since you are going to deal with huge amount of data to be populated into these vector registers. Time between initialization of an instruction and the time the first result emerges from the pipeline that is called the start-up time. Once pipeline is full, the result is produced in every cycle. So if vector lengths were infinite, start-up overhead can be amortized. But for us, there is a finite vector lengths.

So, we have some substantial overhead as far as vector operation or vector loading is being concerned.

(Refer Slide Time: 27:23)



Now, let us take an example to find out with the terminologies that we learned convoy chaining and chime can we find out what is execution time of this given set of vector instruction. So, we have our first vector load vector, we are going a vector value into vector register V1 and then we are multiplying V1 with F0 some other register value. It is a scalar multiplication vector scalar multiplication to store the result in V2 and then we load the next vector.

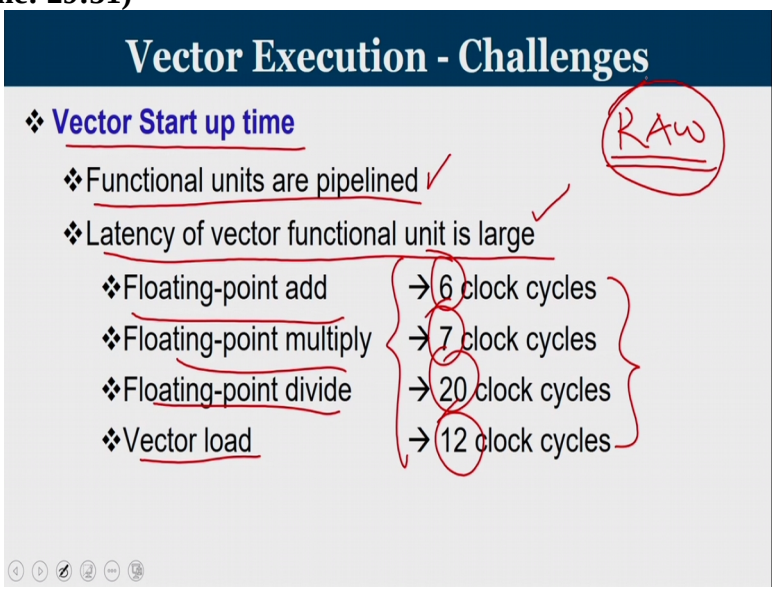
So, we are adding 2 vectors V2 and V3 to store the result in V4 and then store. So, this is loading of a vector, vecctor-scalar multiplication, loading of the second vector Y add 2 vectors and store the sum. This load and the first multiplication that can be part of 1 convoy because they are connected by chaining. So as and when the load happens a multiplication can begin with individual elements, it should not wait for the entire load to happen.

And then we have the next set of load and the add that can be also connected together by chaining so they will be part of the same convoy and then the store. So, this will start only after all instructions of the first one is getting over. So, you can amortize execution time of 1 convoy, I am defining it as 1 chime since we have 3 chimes of there and during this 3 convoys, so, it will take total of 3 chimes.

We are doing 2 floating point operations during 2 chimes, So 2 floating point operations are there per result, all other load overhead we can neglect for the time period. So, the number of cycles required per floating point operations per flop is 1.5. So, you require a total of 3 cycles to complete to floating point operations. So, on an average you require 1.5 cycles to complete a floating point operation.

So cycles per flop is an important quantity that is been defined to understand the efficiency or operational capacity of vector processors. A 64 element vector require 64 into 3, 192 chimes or clock cycles whatever the case may be. So, in this case, for example, we assume it as 1 that means it is requiring into 3. So, if each of the vector element is 64 then that takes 192 chimes or clock cycle as the case may be.

(Refer Slide Time: 29:51)



Now, what are the challenges when you execute these kind of vector instructions, first is the vector startup time and then we have to design functional units that have to be fully pipelined, but some of the latency of certain vector functional unit is large, for example, if it is a floating point in add, 6 cycles, floating point multiplication 7 cycles, floating point division 20 cycles, vector load into the vector registers 12 cycle.

So, these are all causing significant number of clock cycles. So, because of this significant number of clock cycle is there in some operations, when you have raw dependency between 2 operations. Since the delays is this much the stalls required in the raw dependency is also high. So, these are the challenges that we generally face. We have seen vector processors. So, some features of vector MIPS architecture were discussed.

And the quick introduction was given regarding what do you mean by vector processing, certain specific syntax instructions are there which will help us to conditionally process certain areas of the vector registers and how can get it get rid of loops, which were used initially to carry out operation on long arrays. So, vector architecture a special category of architecture, which operates on the fundamental principle of single instruction multiple data stream.

Now, we will see another category of data level parallelism exploitation that is called GPUs or graphics processing unit. Before going into the details, let me give a quick introduction of this very commonly used a term called GPU. You may be very familiar with the term GPU when it comes to specification of laptops or workstations where in more graphics operation or gaming operations or been there, we demand for quality GPU also to be integrated into the system.

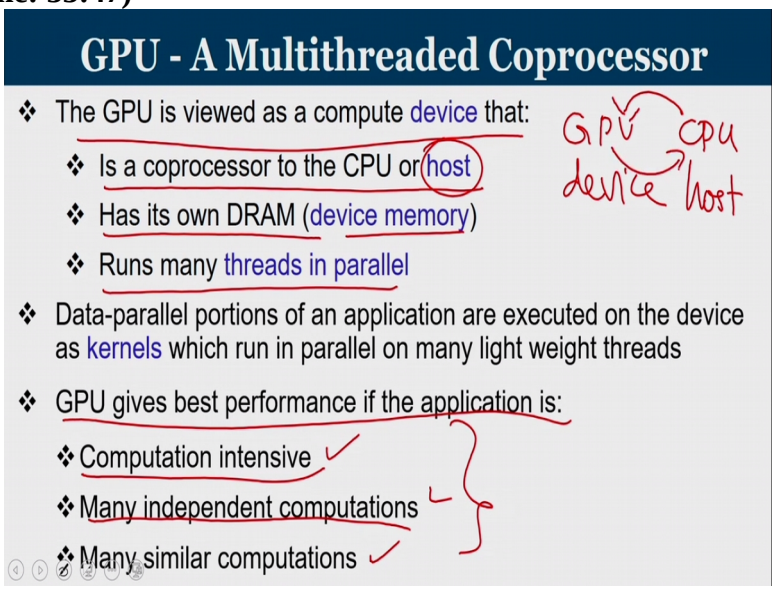
So GPU is a device which basically performs an operation, but this operation is to be performed on so many data then multiple such units are there and all these units are capable of performing the same operation. So, it is as good as like you have 2000 adders. Let us say if adding is operation that is to be done on a relatively large data, 1 adder may not help so you have to load an element add it take the next element add it Why can I load all these elements together into various adders.

Perform the adding operation and then put it back. So, when it comes to gaming applications, wherever a lot of rendering is required, we are changing the background scene very frequently. So from 1 set of pixel coordinates that represent 1 frame in a video based upon the actions we do, we will be getting the next frame. So, there is little deviation from 1 frame to another or whatever activity we do that is going to transform the current frame into the next frame.

So, in this context, we require so many functional units, some functional unit will take up the first 20 pixels, the next functional unit will take up the next 20 pixels like that the entire pixels is being truncated and fragmented and the pixel values are given to these functional units. So rather than having 1 functional unit, which will take care of processing of all these pixels, the load is been divided across many smaller units.

GPU is nothing but a collection of such smaller units, where the same operation needs to be carried out. It has now, see some of the points pertaining to the characteristics of GPUs.

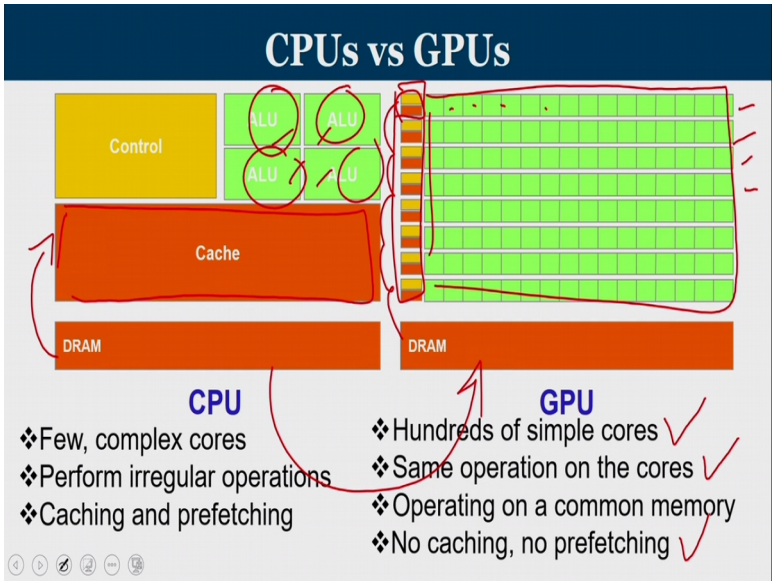
(Refer Slide Time: 33:47)



So GPU is a multithreaded coprocessor GPU is viewed as a compute device that is a coprocessor to CPU. So CPU we call it as host. And GPU has its own memory, which is known as device memory and it runs many threads. So GPU and CPU works together GPU do not have existence by itself and CPU is known as the host and GPU is known as the device. Data parallel portions of applications are executed on the device as kernels, which run in parallel on many lightweight threads.

GPU gives best performance if the application is computation intensive, many independent computations and many similar computation. So when you have independent computations, when you have similar computations, CPU will offload the work into GPU and GPU do it and then give it back to the CPU again.

(Refer Slide Time: 33:48)

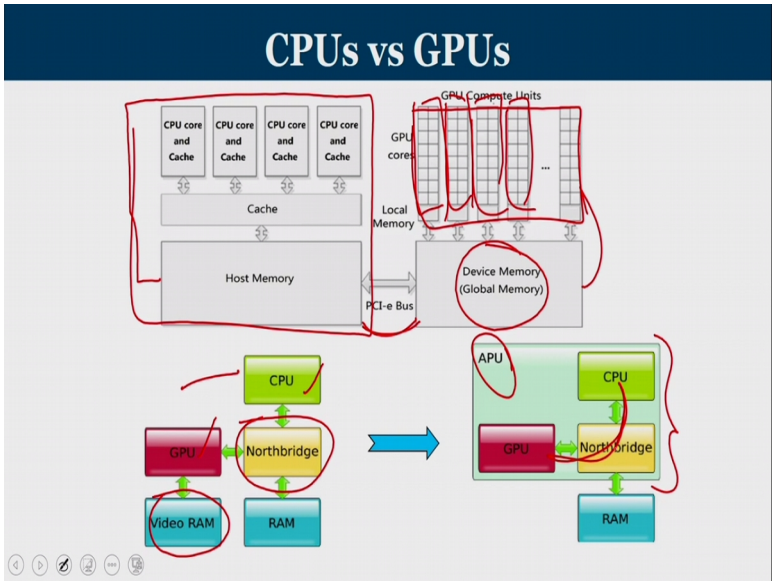


So this is the broader comparison of the architectural diagram of CPU and GPU. In the case of CPU, you have a control unit in you how many ALU that will do you have a reasonably big cache, and then from DRAM, you are going to populate into the cache. Whereas in the case of GPUs, we have many small functional units, it may not be highly sophisticated like ALUs that we have in CPU, many tiny units each capable of performing some simple activity.

And they have their own levels of cache. And GPU itself is divided into multiple grids and blocks. Each level of grid may have its own level of control and cache. And then we have a device memory, the DRAM so CPU is going to push up things into the DRAM, and based upon where activity happens, the appropriate caches have been populated. So in CPU, we have few complex cores, it can perform irregular operations and caching and prefetching has been done.

When it comes to GPUs, we have hundreds of simple cores, and the same operation has to be done in all the cores. And they are operating on a common memory generally not much of caching. And not prefetching is been done.

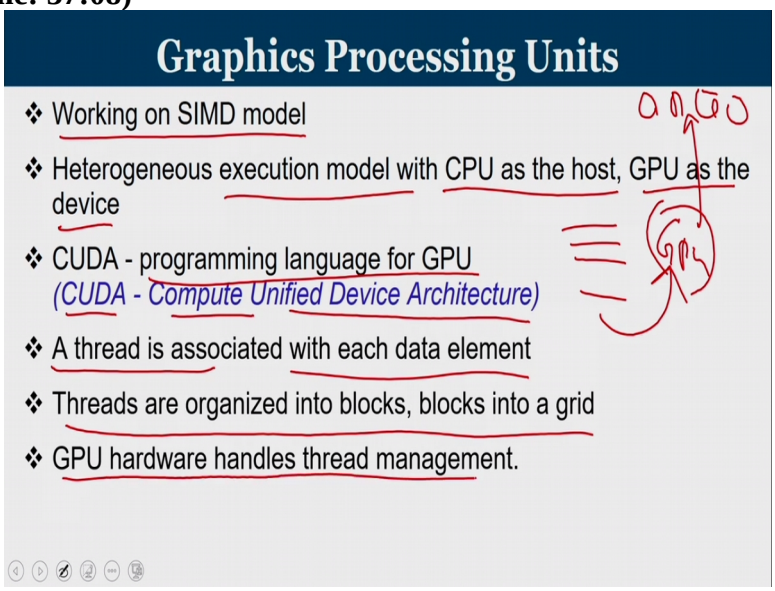
(Refer Slide Time: 36:00)



Few more comparison. So we have the host of the CPU cores that has been working together for multiple CPU cores, maybe you are multicore processor with its own cache and memory hierarchy. And then we have a bus that connects the CPU with the GPU and GPU has multiple simple cores, and the device memory is going to hold the data and then push it across this multiple racks of for multiple grids are blocks of GPU cores.

So, CPU is generally interacting with GPU using the North-bridge circuit to be kept on the motherboard. So, we have specific memory that is called video RAM or the device memory. But if you wanted to keep the CPU, GPU under the North-bridge together, then we call it as accelerated processing units. So, in this case, CPU and GPU separate chip here I can integrate them into single package. So this high speed communication highway can be put inside the chip. And they are known as APU use accelerated processing units.

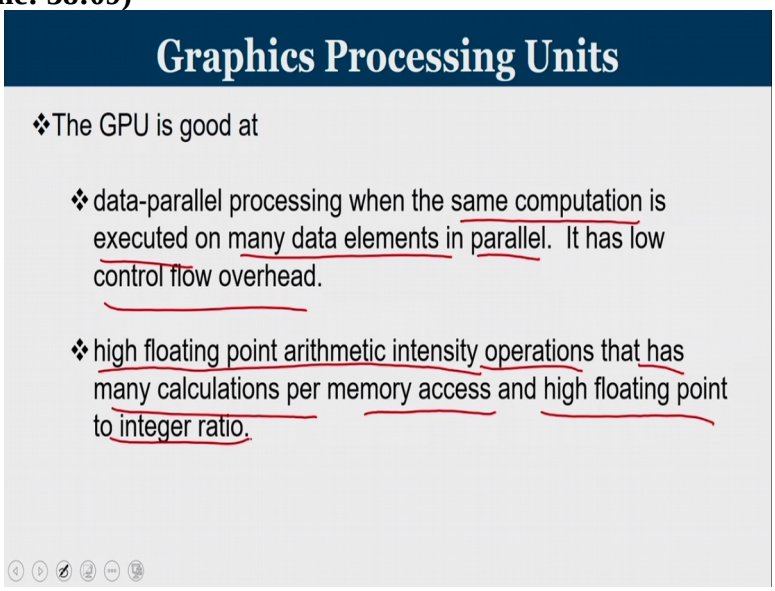
(Refer Slide Time: 37:08)



Coming into features of GPUs, GPU basically works on the SIMD model single instruction multiple data stream. So heterogeneous execution model with CPU as the host and GPU as the device and specific programs are to be given. And they are under the CUDA environment, it is a programming language for GPUs CUDA stands for Compute Unified Device Architecture. So, these are all special set of lines, which a GPU controller can understand these set of lines.

And they will push the values into smaller GPU cores, and will be able to do. So specific programming constructs are to be defined and this GPU controllers will initiate the rest of the actions of scattering the work across smaller GPU cores and gathering the work back once it is been done. And generally be operated a thread level. A thread is associated with each data element threads are organized into blocks and blocks are organized into grids and GPU hardware and in this thread management.

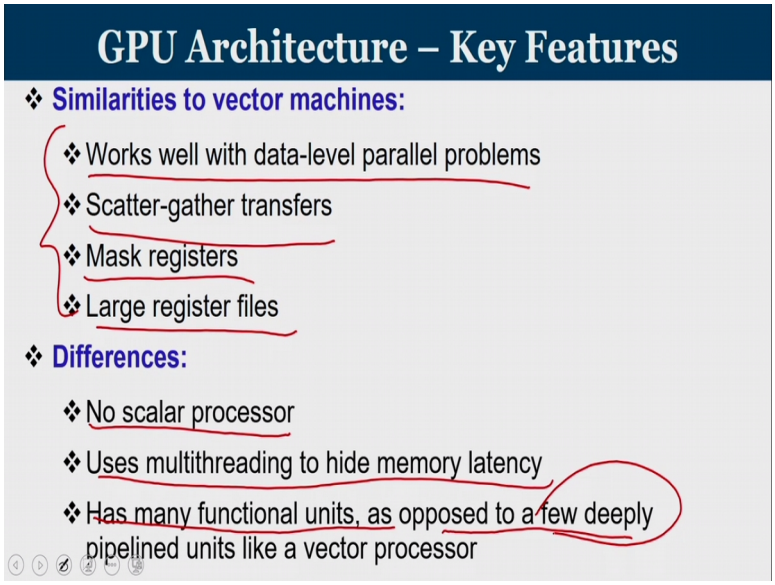
(Refer Slide Time: 38:09)



So, when is the GPU best suited? A GPU is good at data parallel processing, when same computation is executed on many data elements in parallel and there is low control flow overhead in this context or GPU is very good in context like when you have high floating point arithmetic intensity operations, that many calculations are to be done per memory access, and there is high floating point to integer ratio.

The number of floating point operations that you are doing is very high compared to the number of integer operation that is what it is called as high floating point integer ratio.

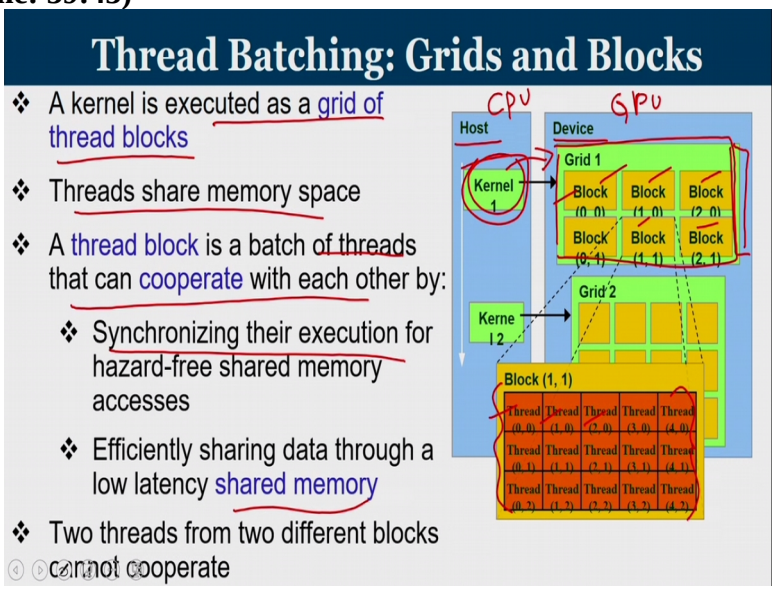
(Refer Slide Time: 38:52)



Now, you may be thinking, what are the similarities or dissimilarities of the vector machine that we have seen earlier with respect to the GPU. So coming to vector machines, they work well with data level problem that high level parallel problems and we use a scatter and gather approach we are using mask registers for conditional operations and then we have large register file, when it comes to the differences, so in this cases, both vector machines and GPUs are similar.

The differences there is no scalar processor it uses multithreading concept to hide memory latency. Has many functional units as opposed to do few deeply pipelined units. So, in vector case, you have few deeply pipelined units, whereas in the case of GPUs, we have many functional many simple functional units.

(Refer Slide Time: 39:43)

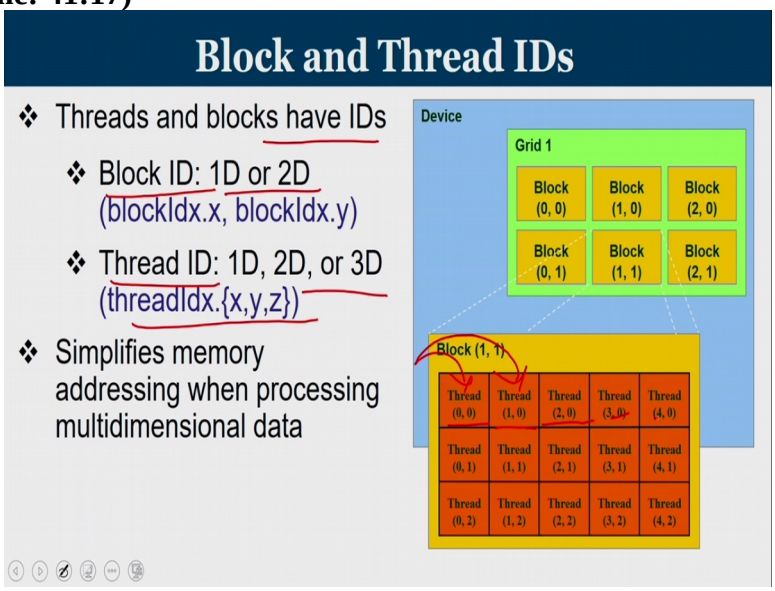


Let us try to understand so this is the device which is the GPU and this is your CPU, CPUs run as host. So host will run a kernel. And that is pushing it into the device. And kernel1 is going to operate on something called grid. So a kernel is executed as a grid of thread blocks. So the grid consists of multiple thread blocks. This thread share memory space, a thread block is nothing but it is a batch of threads that can cooperate.

So you can see that each of the thread block has many threads. So all these threads can share data between them. So we these are all minor threads, which have something in common between them. So they can cooperate each other. Synchronizing their execution for hazard-free shared memory. So basically, a thread block is a batch of threads that can cooperate with each other by synchronizing their execution for hazard-free shared memory access.

And they efficiently share data through the shared memory that the host or this particular grid may have some common memory that they have, and they are going to work. So what we have to understand is we have a kernel that will push things into a grid in the GPU. And this grid consists of multiple blocks organized as row and column and each of the blocks have multiple threads. Threads inside the same block can share the data. 2 threads from 2 different blocks cannot cooperate or cannot share data.

(Refer Slide Time: 41:17)

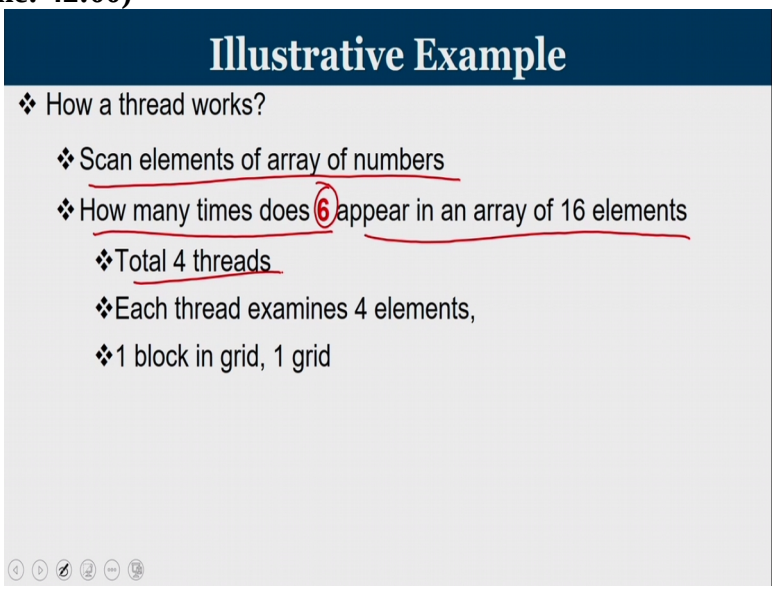


Now this threads and blocks that we have seen, they have their own IDs block ID can be 1 dimensional or 2 dimensional, thread ID can be even 3 dimensional as well, like whatever numbering that you have given. So this simplifies memory addressing when processing multi

dimensional data, because we organize the data, we push data into appropriate threads, let us say it is in a matrix, first 20 elements go here, next 20 elements go here, like that.

So I can divide my entire data into this various threads and each thread is going to independently operate let us say there is a dependency between certain iterations inside the matrix. Since they are all part of the same block, we can actually communicate and then get data associated with it.

(Refer Slide Time: 42:00)



Now how a thread works, I would like to show an illustrative example. Let us say I wanted to scan the elements of an array. And we have to find out how many times this number 6 appears in an array of 16 elements. Let us considered the case that I am going to use 4 threads. So the job that is being given here is we have 16 numbers and then we have to find out how many times the number 6 appears in these 16 elements.

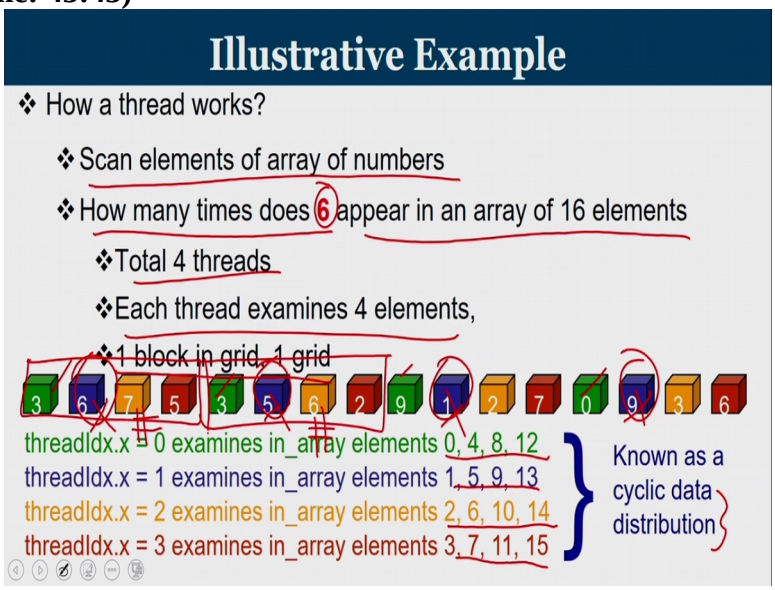
So if I am going to work with 1 thread, take the first element, check whether it is 6 or not take the second element check whether it is 6 or not like that whenever we encounter 6, you increment the counter, like that you scan it across. That is the way how a single thread does it. Now, can you divide this work into 4 threads, I am assigning 4 threads will do it, the first thread will take care of only 4 numbers.

And see number of times 6 occur there. So he will keep his own counter like that each thread is going to have their own counter, they will work only in a small range of data not entire data of 16 elements, everybody is restricted to just 4 elements, find out the number of times where

you get the match, you keep a separate count. That can be done parallelly. And once this operation is over, then there is a serial task, sum up the counter values.

That is exactly what has been done. So this could have been if it is a CPU that has been doing I will do the first one scan from one into another. If it is a GPU who does the task, then I am assigning to 4 GPU small units GPU functional unit, each functional unit will work on only on 4 data, and then find out the result and then you club the result at the end in CPU.

(Refer Slide Time: 43:43)



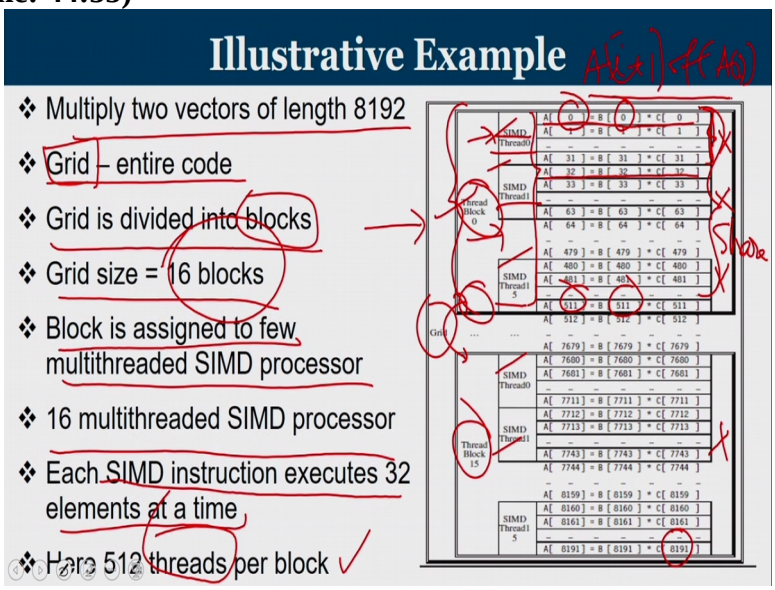
So in this case, we can find that each thread is going to examine 4 elements, 1 block in grid, and then 1 grid. So let us say these are the 16 numbers. So wherever you see green color, these are been the numbers that are checked by 1 of the thread, the blue is being checked, or the second thread let us called the blue thread will examine these numbers and see whether it is matching your requirement.

And then you have the yellow one. There is another category like that you have 4 threads. So, thread ID 0 examines array elements 0 4 8 and 12, thread ID 1 examines array elements 1 5 9 and 13 thread ID 2 examines array elements 2 6 10 and 14 and thread ID 3 examines array elements 3 7 11 and 15. So, this is 1 way of spreading it is also known as cyclic data distribution. I can always give these 4 elements to thread 1.

Next to 4 elements to thread 2 that is yet another distribution that is known as sequential data distribution, this is known as cyclic data distribution. So the purposes each one will

independently operate on this data get their local results and towards the end try to sum of the results.

(Refer Slide Time: 44:55)



Let us now consider multiplication of 2 vectors of length 8192. So, basically the operation that you do is

$$A = B * C$$

Now let us put the entire grid it is the matrix multiplication of relatively large size. Let me put the grid consists of your entire code the grid is now divided into blocks that say how 16 blocks so my entire data of 8192 has to be divided into multiple thread blocks. Now I am dividing them into 16 blocks.

Now block is assigned to few multithreaded as SIMD processor. Now I have special processes which will take a single instruction over multiple data. Now each of these block have multiple threads. So here I have 16 threads each. So single instruction multiple data stream threads 0 thread 1 like that thread 15 so each thread is going to operate only on 32 data. So 16 multithreaded SIMD processor are being connected each SIMD instruction execute 32 elements at a time.

So when this is in progress, this is also parallely done. So they are all parallely doing so when

$$A0 = B0 * C0,$$

that is under progress at the same time, A 32, so the very first line of each of this thread is being computed. So you get some elements. Now the second line of each thread is in

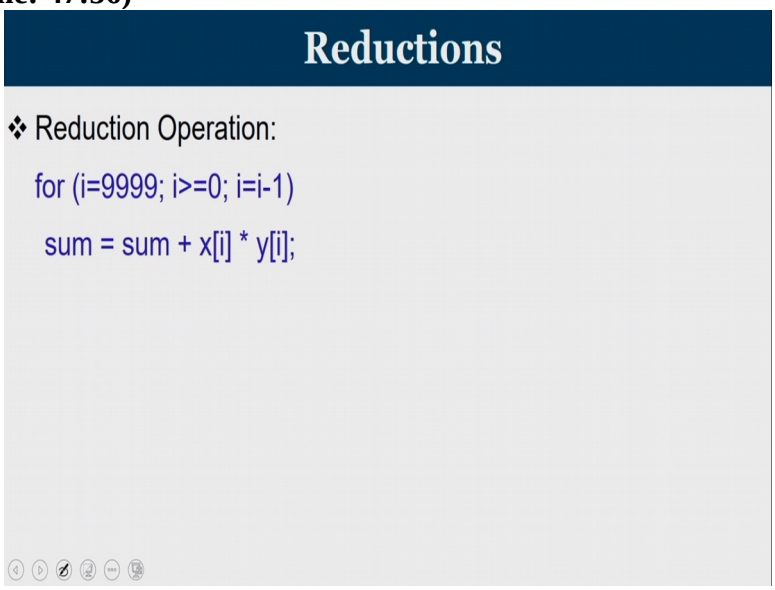
progress. So then the second line of this is in progress. So each SIMD instruction execute 32 elements at a time.

So here, there are a total of 512 threads per block. That is what the number that you get. So, this is the way in which a complex matrix multiplication process rather than doing it on a single CPU, you are taking care of the help of a GPU. So, we know that GPU has different level of hierarchy. You have grids, you have blocks, and then you have threads that are part of the same block. So the elements that are part of the same block, they can actually share data.

So, any dependency between A0 all the way upto A511 or B0 upto 512, this can share data between them. So, even though there is a dependency between them, that can be share and manipulated. Let us say if you have a case where A of I+1 is going to be dependent on A of I. So, I need to know what is the value of A of I, so if A of I and A of I+1 are part of different thread block there it is difficult. Of course, the problem is there in the border elements that has to be specially handled or else any kind of dependency.

Any value that you want from the previous iteration or previous index of the same matrix can also be taken.

(Refer Slide Time: 47:56)

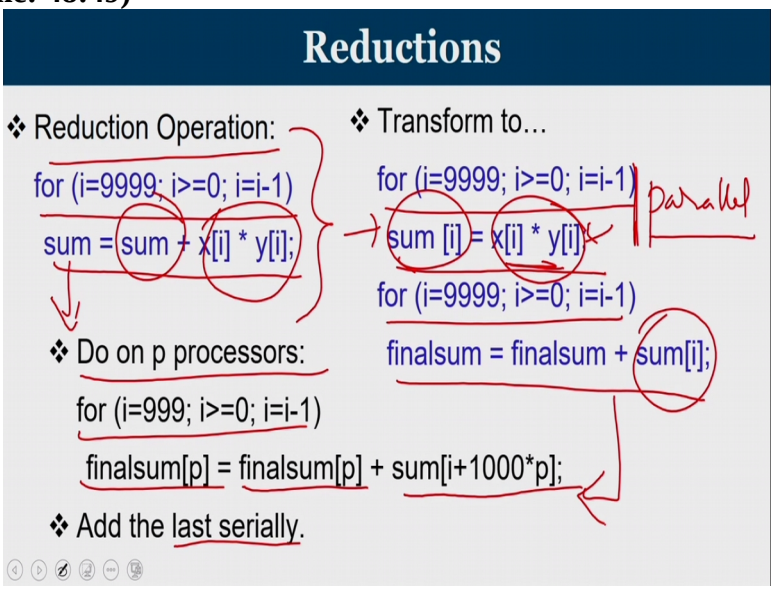


Now let us see sometimes when you give programs, these programs how certain problems which make them unfit for GPUs, so the conventional way of programs that we are writing, they are generally sequential step, step by step operation. Now, when these steps of

operations are being given to CPU, it is meant to be run on CPU. So, the program carries a sequence of steps, but when you go for a GPU, it is not sequential, you are going to think do things parallel.

So if inherent sequentiality is there in the program, it has to be transformed to a GPU friendly program. So, this CUDA compilers and all will help us to transfer or to fine tune our conventional programs into parallel programs. Let us now see, 1 important technique there has been used and that is known as Reduction.

(Refer Slide Time: 48:49)



So, we will see what is reduction operation consider the case that you are going to operate on a loop where

$$sum = sum + xi * yi$$

so, you have to perform a multiplication and then the product you have to add into sum. So sum is to cumulatively added. We know that this can be done only sequentially because at the end what we want is sum so the sum of the previous iteration is needed. So firstly I add

$$x1 * y1$$

you get the product that is added to sum.

Now that sum is required even though you have x 2 + y 2 , I need to get the previous sum so their access to dependency between them. So, what I can do is can I transform first. I am computing x i into y i the corresponding product I will do into another array called sum of i. So each individual product this section I can parallelize in GPU. So, each GPU unit functional unit may take care of an individual multiplication

$$x1 * y1$$

is carried out at the same time

$$x2 * y2$$

is carried out.

$$x3 * y3$$

is carried out so this complex multiplication process happens parallelly and the corresponding products are shown in sum, So sum of 1 is part of the first GPU unit, sum of 2 is part of second GPU unit and then you write you have something like final sum where the

individual sums are being added together. So, do on p processors. So, I will be able to do it on p processor

$$\text{final sum (p)} = \text{final sum (p)} + \text{sum of i + thousand} * p$$

and then add the final result serially.

So, this is the way how a program which was not parallel friendly is now being divided into 2 portion a parallel portion followed by your serial portion. So, even the serial portion I can still make it faster by dividing it into p processors. So, with this we come to the end of this lecture. So we have learned about vector architectures and then we learned about GPU architectures both are best suited whenever we have context when the same operation is to be repeatedly carried apart different set of data. Thank you