Instruction-Level Parallelism: Concepts and Challenges

What is instruction-level Parallelism?

The potential overlap among instructions is called instruction-level parallelism (ILP).

There are two approaches to exploiting ILP:

1. an approach that relies on hardware to help discover and exploit the parallelism dynamically
2. an approach that relies on software technology to find parallelism statically at compile time.

the instructions are likely to depend upon one another, the amount of overlap within a basic block is likely to be less than the average basic block size.

Loop-level parallelism

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is called loop-level parallelism.

An example:

A loop adds two 1000-element arrays is considered. It is completely parallel.

```
for (i=0; i<=999; i=i+1)
x[i] = x[i] + y[i]
```

Techniques for converting such loop-level parallelism into instruction-level parallelism:

- Unrolling the loop either statically by the compiler or dynamically by the hardware
- Use of SIMD in both vector processors and Graphics Processing Units (GPUs)

Data Dependences and Hazards

While determining how much parallelism exists in a program, it is critical to determine how one instruction depends on another.

- If two instructions are parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming there is no structural hazard).
- If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped.
- In both cases, it is required to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences:

1. data dependences (also called true data dependences)
2. name dependences
3. control dependences.

An instruction j is data dependent on instruction i if either of the following holds:

- Instruction i produces a result that may be used by instruction j.
- Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i

A data value may be stored in registers or in memory locations. When the data is stored in a register, detecting the dependence is straightforward.

When the data is stored in memory locations, detecting the dependence is are more difficult since two addresses may refer to the same location but look different. In addition, the effective address of a load or store may change from one execution of the instruction to another.

Name dependence

- Two instructions use the same name but there is no flow of information.
- It is not a true data dependence, but is a problem when reordering instructions.

Antidependence

Instruction j writes a register or memory location that instruction i reads

Initial ordering (i before j) must be preserved

Output dependence

- Instruction i and instruction j write the same register or memory location
- Ordering must be preserved
- To resolve, renaming techniques can be used.

Control dependence

- Every instruction is control dependent on some set of branches.
- Control dependencies must be preserved to ensure program correctness.
- Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch.
- An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch
- Example

```
if C1 {
S1; };
 if C2 {
S2; };
```

S1 is control dependent on C1; S2 is control dependent on C2 but not on C1

• In general, two constraints are imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.

Example

- An instruction can not be taken from the then portion of an if statement and moved it before the if statement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

- Example
- A statement before the if statement can not be taken and moved it into the then portion

Compiler techniques for exposing ILP

1. Loop transformation technique can be used to optimize a program's execution speed:

- Instructions that control the loop can be reduced or eliminated
- e.g., pointer arithmetic and "end of loop" tests on each iteration

2. Latencies can be hidden.

- e.g., the delay in reading data from memory

3. Loops can be rewritten as a repeated sequence of similar independent statements

- Eg - space-time trade off.

4. Branch penalties can be reduced.

Step 1.Pipeline scheduling

Separate dependent instruction from the source instruction by the pipeline latency (or instruction latency) of the source instruction.

Example

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
Loop :  Load R0,0(R1)
        Add   R4, R0, R2
        Store  R4,0(R1)
        Add    R1,R1,#-8
        BNE    R1,R2,Loop
```

In the above example, data dependence is present. It has been shown by the arrows.

- The arrows show the order that must be preserved for correct execution.
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

Solution:

```
Loop :  Load R0,0(R1)
        Stall
        Add   R4, R0, R2
        Stall
        Stall
        Store  R4,0(R1)
        Add    R1, R1, #-8
        Stall
        BNE    R1, R2, Loop
```

But the above code requires nine clock cycles per iteration.

Step 2. Re-Scheduling

```
Loop :  Load R0,0(R1)
        Add    R1, R1, #-8
        Add   R4, R0, R2
        Stall
        Stall
        Store  R4,0(R1)
        BNE    R1, R2, Loop
```

After re-scheduling, 7 clock cycles per iteration are required.

Step 3.Loop unrolling

- It is a simple scheme for increasing the number of instructions relative to the branch and overhead instructions. It replicates the loop body multiple times, adjusting the loop termination code.
- For the above example, unrolling is done by a factor of 4. The resultant code is given below:

```
Load R0,0(R1)
Add   R4, R0, R2
Store R4,0(R1)          ;drop Add & BNE
Load R6, -8(R1)
Add   R8, R6, R2
Store R8, -8 (R1)       ;drop Add & BNE
Load R10, -16(R1)
Add   R12, R10, R2
Store R12, -16 (R1)     ;drop Add & BNE
Load R14, -24(R1)
Add   R16, R14, R2
Store R16, -24(R1)
Add    R1, R1, #-32
BNE R1, R2, Loop
```

The loop is unrolled so that there are four copies of the loop body. The size of the array is assumed to be a multiple of 32 initially. Any redundant computation can be eliminated. Any register is not reused.

Three branches and three decrements of R1 are eliminated. The addresses on the loads and stores have been compensated to allow the DADDUI instructions on R1 to be merged.

Limitation

This optimization requires symbolic substitution and simplification.

4. Re-Schedule the Unrolled loop

The unrolled loop is rescheduled as given below:

```
Load R0,0(R1)
Load R6, -8(R1)
Load R10, -16(R1)
Load R14, -24(R1)
Add   R4, R0, R2
Add   R8, R6, R2
Add   R12, R10, R2
Add   R16, R14, R2
Store R4,0(R1)          ;drop Add & BNE
Store R8, -8 (R1)       ;drop Add & BNE
Store R12, -16 (R1)     ;drop Add & BNE
Store R16, -24(R1)
Add R1, R1, #-32
BNE R1, R2, Loop
```

- The execution time of the unrolled loop is total of 14 clock cycles, or 3.5 clock cycles per element, compared with 9 cycles per element before any unrolling or scheduling and 7 cycles when scheduled but not unrolled.

Overcoming Data Hazards with Dynamic Scheduling

- It is assumed that the pipeline is filled with instructions. A simple statically scheduled pipeline fetches an instruction and issues it. If there is a data dependence between the instruction already in the pipeline and the fetched instruction, then the hazard detection hardware stalls the pipeline starting with the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared.
- Solution: One solution is dynamic scheduling

Dynamic scheduling

The hardware rearranges the instruction execution. The stalls are reduced and data flow and exception behavior are maintained.

Advantages

1. Code that was compiled with one pipeline can be run efficiently on a different pipeline. It eliminates the need to have multiple binaries and recompile for a different microarchitecture.
2. There are some cases for which when dependences are unknown at compile time. It can be handled by dynamic scheduling.

- Example,
  - The instructions may involve a memory reference or a data-dependent branch.
  - They may result from a modern programming environment that uses dynamic linking or dispatching.

3. It allows the processor to tolerate unpredictable delays.
   - Example
   - When there are cache misses, other code can be executed while waiting for the miss to resolve.

Disadvantage

- Significant increase in hardware complexity can be noted when dynamic scheduling is used.

Limitation of simple pipelining techniques

- They use in-order instruction issue and execution. Instructions are issued in program order. If an instruction is stalled in the pipeline no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result.
- If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute.
- Example

```
Add R0,R0,R1
Subtract R4,R0,R5
Add R7,R8,R9
```

There is a dependence between the first two instructions. It causes the pipeline to stall. Though third instruction does not depend on the previous instructions, it has to wait. This hazard creates a performance limitation in the pipeline. It can be eliminated by changing the execution of program order.

- out-of-order execution

During issue process, the instruction is checked for structural hazard or if it is waiting for the completion of the previous instructions. In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID). but we want an instruction to begin As soon as data operands are available for any instruction, execution of the instruction starts. It is called out-of-order execution, which implies out-of-order completion.

Example

```
Add R0,R2,R4
Add R6,R0,R8
Subtract R8,R10,R14
Add R6,R10,R8
```

- There is an antidependence between second and third instructions, Add R6,R0,R8 and Subtract R8,R10,R14 instructions respectively.
- Second instruction waits for first instruction for Register R0.
- Let out-of-order execution is applied here. Now, third instruction is to be executed second. When it tries to write R8, Add R6, R0, R8 will read a wrong value from R8. This leads to Write After Read (WAR hazard).
- **Imprecise Exceptions**

✓ Out-of-order completion also creates major complications in handling exceptions

✓ It preserves exception behaviour by delaying the notification of an associated exception

✓ Dynamically scheduled processors could generate imprecise exceptions.

✓ The processor state during exception raise may change due to Out-of-order execution. The exception is called imprecise exception. It can occur because of two possibilities:

1. Instructions that are later in program order than the instruction causing the exception may have already completed.
2. Some instructions that are earlier in program order than the instruction causing the exception may have not yet completed.

✓ Imprecise exceptions make it difficult to restart execution after an exception.

- **Solutions for out-of-order execution complexities**
- Five-stage pipeline is split into two stages as following:
  1. Issue - This stage decode instructions and check for structural hazards
  2. Read Operands – This stage waits until no data hazards and then reads operands.

  ✓ Fetch stage precedes issue stage. It may fetch either into an instruction register or into a queue of pending instructions.

  ✓ Instructions are then issued from the register or queue. All instructions pass through the issue stage in order (in-order issue)

  ✓ Instructions can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order

  ✓ The execution stage follows the read operands stage.

  ✓ Execution may take multiple cycles.

  ✓ Scoreboarding

- It is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences.

Dynamic Scheduling Using Tomasulo's Approach

The IBM 360/91 floating-point unit used a sophisticated scheme to allow out-of-order execution. This scheme was invented by Robert Tomasulo. It tracks when operands for instructions are available to minimize RAW hazards. It introduces register renaming in hardware to minimize WAW and WAR hazards.

Register renaming

- It eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction. The out-of-order write does not affect any instructions that depend on an earlier value of an operand.
- Example

  Add R0,R2,R4
  Add R6,R0,R8
  Store R6, LOC
  Sub R8,R10,R14
  Add R6,R10,R8
- There are two antidependences:
  1. between second and fourth instructions (Add R6, R0, R8 and Sub R8, R10, R14)
  2. between third and fifth instructions (Store R6, LOC and Add R6, R10, R8)
- There is also an output dependence between second and fifth instructions (Add R6,R0,R8 and Add R6,R10,R8
- These lead to three possible hazards:

  WAR hazards on the use of
  1. R8 by second instruction
  2. R6 by fourth instruction

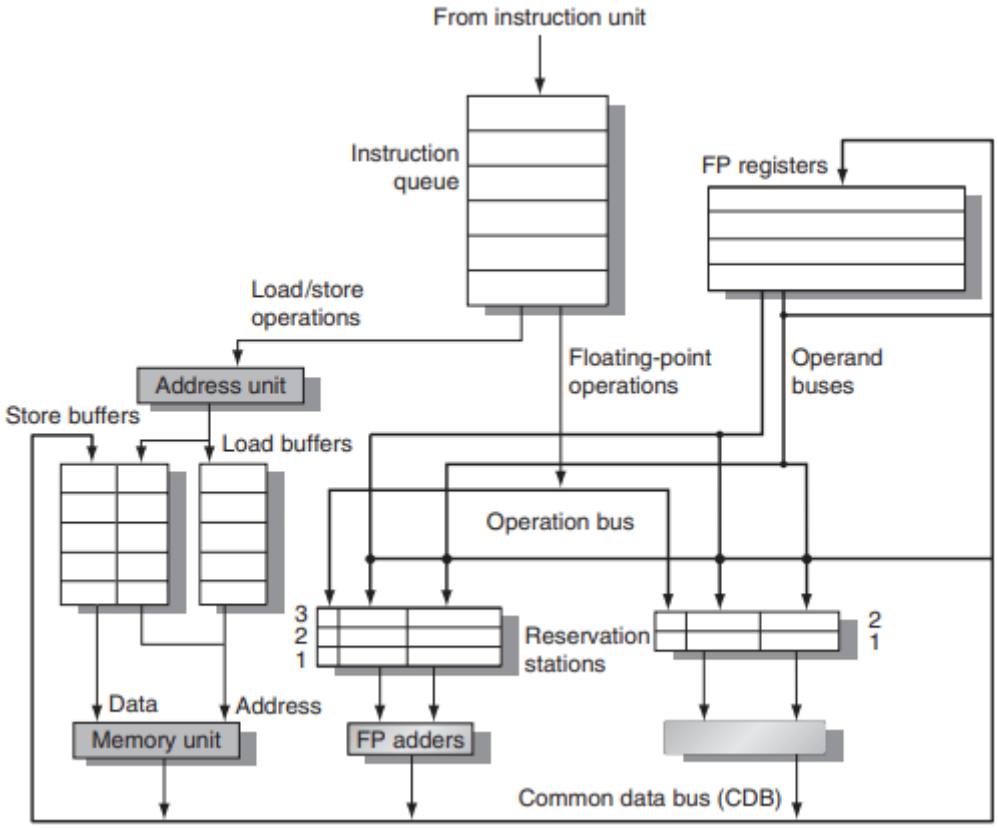
  WAW hazard
  3. Second instruction may finish later than fifth instruction.
- There are also three true data dependences:
  1. between first and second instructions
  2. between fourth and fifth instructions
  3. between second and third instructions
- **Solution**
  - The above name dependences can all be eliminated by register renaming.
  - Assuming the availability of two temporary registers, S and T, the sequence can be rewritten without any dependences as following:

Add R0,R2,R4
Add S,R0,R8
Store S, LOC
Sub T,R10,R14
Add R6,R10,R8

- **Reservation station**
  - It fetches and buffers an operand as soon as it is available.
  - It eliminates the need to get the operand from a register.
  - In addition, pending instructions designate the reservation station that will provide their input.
  - When successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.
  - **Components**
    - ✓ Op: The operation to perform on source operands S1 and S2 in the unit (e.g., + or –)
    - ✓ Vj, Vk: Value of Source operands are given.Oonly one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
    - ✓ Qj, Qk: Reservation stations producing source registers (value to be written); a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary
    - ✓ A – It is used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
    - ✓ Busy – It indicates that this reservation station and its accompanying functional unit are occupied.
    - ✓ The register file has a field, Qi:
      - ✓ The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.
    - ✓ Note: Qj,Qk=0 => ready
    - ✓ Store buffers only have Qi for RS producing result

- ✓ Busy: Indicates reservation station or FU is busy
- ✓ Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

➢ The basic structure of a MIPS floating-point unit using Tomasulo's algorithm



Tomasulo Organization

➢ The above picture shows the MIPS floating-point unit which uses Tomasulo's algorithm. It includes both the floating-point unit and the load/store unit. Each reservation station holds an instruction that has been issued. It is awaiting execution at a functional unit and either the operand values for that instruction, if they have already been computed, or else the names of the reservation stations that will provide the operand values.

➢ The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All reservation stations have tag fields, employed by the pipeline control.

The steps an instruction goes through (Three Stages of Tomasulo Algorithm)

1. Issue - Processer gets instruction from FP Op Queue. If reservation station is free, there is no structural hazard. The control issues instruction and sends operands (renames registers).
2. Execute - Processor operates on operands. When both operands ready then execution takes place. If both operands are not ready, Common Data Bus is watched for result.
3. Write result - Execution is finished. Result is written on Common Data Bus to all awaiting units. It is marked that reservation station is available.

Dynamic Scheduling: Examples and the Algorithm

Let us illustrate how the algorithm works with an example.

Load R6, 32(R20)
Load R2,44(R3)
Add R0, R2, R4
Subtract R8,R2,R6
Add R10,R0,R6
Add R6, R8, R2

Instruction Status

	Instruction Status		
Instruction	Issue	Execute	Write Result
Load R6, 32(R20)	√	√	√
Load R2,44(R3)	√	√	
Add R0, R2, R4	√		
Subtract R8, R2, R6	√		
Add R10, R0, R6	√		
Add R6, R8, R2	√		

The above table shows the instruction status. It shows that all of the instructions have been issued, but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit.

Reservation Stations

	Reservation Stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs(R3)
Add1	Yes	Subtract		Mem[32+Regs [R2]]	Load2		
Add2	Yes	Add			Add1	Load2	

Add3	No						
Add4	Yes	Add		Regs[R4]	Load2		
Add5	Yes	Add		Mem[32+Regs[R2]]	Add4		

Load R6, 32(R20)
Load R2,44(R3)
Add R0, R2, R4
Subtract R8,R2,R6
Add R10,R0,R6
Add R6, R8, R2

Although the Subtract instruction has completed execution, it does not commit until the third instruction commits.

	Register Status								
Field	R0	R2	R4	R6	R8	R10	R12	…	R30
Qi	Add1	Load2		Add4	Subtract1	Add4			

- At the time third instruction is ready to commit, only the two Load instructions have committed, although several others have completed execution. The subtract and sixth instruction will not commit until the third instruction commits. The fifth instruction is in execution, but has not completed solely due to its longer latency.
- The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed but are shown for informational purposes.

Data-Level Parallelism: Introduction
Flynn's Taxonomy
Flynn classified the computers as following:

- SISD - Single instruction stream, single data stream
- SIMD - Single instruction stream, multiple data streams
    - New: SIMT – Single Instruction Multiple Threads (for GPUs)
- MISD - Multiple instruction streams, single data stream
    - No commercial implementation is available.
- MIMD - Multiple instruction streams, multiple data streams
    - There are two types of MIMD are available.
    1. Tightly-coupled MIMD
    2. Loosely-coupled MIMD

Advantages of SIMD architectures

- It allows programmers to continue thinking sequentially.

- It can exploit significant data-level parallelism for:
    - matrix-oriented scientific computing
    - media-oriented image and sound processors
- It is more energy efficient than MIMD.

SIMD parallelism

The following architectures are using SIMD:

- Vector architectures
- SIMD extensions for mobile systems and multimedia applications
- Graphics Processor Units (GPUs)

Vector Architecture

- Vector architecture grabs sets of data elements scattered about memory, place them into large, sequential register files, operate on data in those register files, and then disperse the results back into memory.
- A single instruction operates on vectors of data, which results in dozens of register–register operations on independent data elements.
- The primary components of the instruction set architecture of VMIPS are the following:

Vector registers

- Each vector register is a fixed-length bank holding a single vector. VMIPS has eight vector registers, and each vector register holds 64 elements, each 64 bits wide. The ports will allow a high degree of overlap among vector operations to different vector registers. The read and write ports, which total at least 16 read ports and 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbar switches.

Vector functional units

- Each unit is fully pipelined, and it can start a new operation on every clock cycle. A control unit is needed to detect hazards. VMIPS has five functional units FP add/subtract, FP Multiply, FP divide, Integer and Logical.

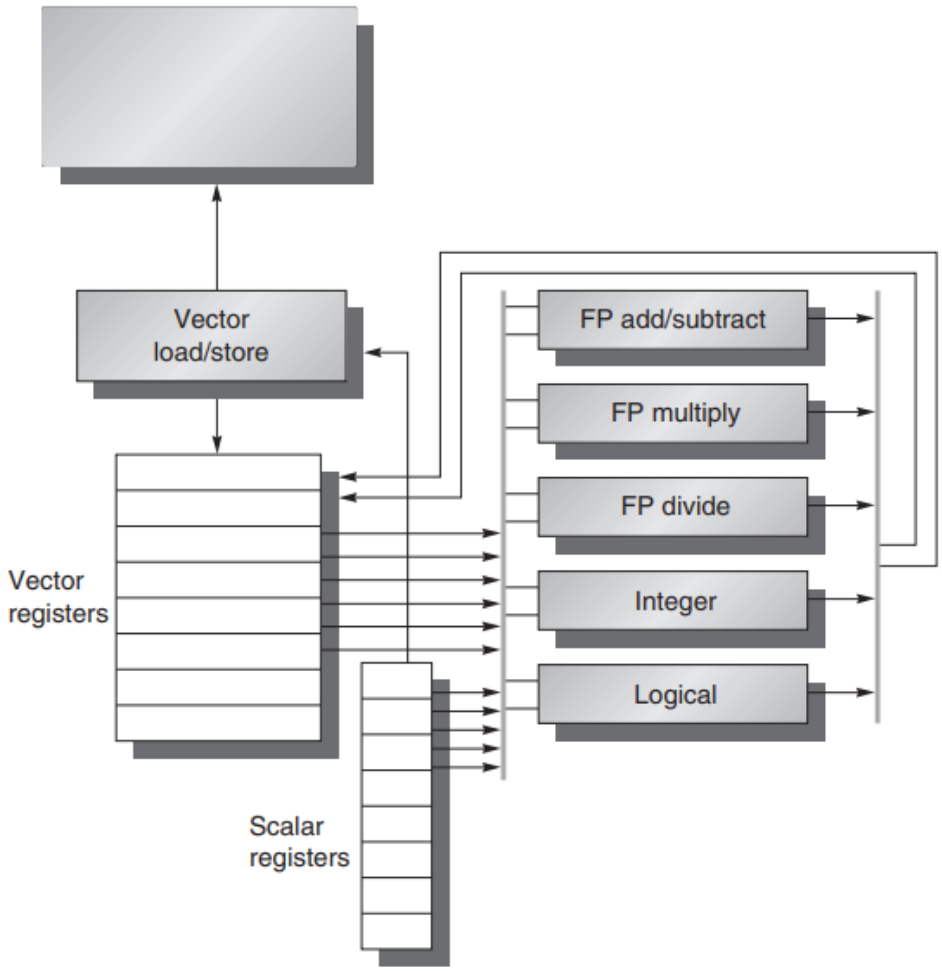
Vector load/store unit

- The vector memory unit loads or stores a vector to or from memory. The VMIPS vector loads and stores are fully pipelined. This unit would also normally handle scalar loads and stores.

A set of scalar registers

- Scalar registers can also provide data as input to the vector functional units. It computes addresses to pass to the vector load/store unit. These are the normal 32 general-purpose

registers and 32 floating-point registers of MIPS. One input of the vector functional units latches scalar values as they are read out of the scalar register file.



VMIPS instructions

- ADDVV.D: add two vectors.
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
    - Rx - the address of vector X
    - Ry - the address of vector Y

Execution time

- Vector execution time depends on:
- Length of operand vectors, Structural hazards, Data dependencies
- VMIPS functional units consume one element per clock cycle à Execution time is approximately the vector length.
- Convoy is a set of vector instructions that could potentially execute together.

Chaining and chimes

Chaining

It allows a vector operation to start as soon as the individual elements of its vector source operand become available

Chime

- It is a timing metric to estimate the time for a convoy. It is a unit of time to execute one convoy. m convoys are executed in m chimes.
- For vector length of n, m x n clock cycles are required.
- Sequences with read-after-write dependency hazards can be in the same convoy via chaining.

Challenges

The chime model ignores the vector start-up time determined by the pipelining latency of vector functional units.

Optimizations

Vector Optimizations will either improve the performance or increase the types of programs that can run well on vector architectures. They are listed below:

1. **Multiple Lanes**

   By using multiple lanes in the architecture, more than one element can be processed per clock cycle

2. **Vector Length Registers**

   These registers are able to handle non-64 wide vectors.

3. **Vector Mask Registers**

   Programs that contain IF statements in loops cannot be run in vector mode.

   Example

   ```
   for (i = 0; i < 64; i=i+1)  if (X[i] != 0)
   X[i] = X[i] – Y[i];
   ```

   The vector-mask control uses a Boolean vector to control the execution of a vector instruction. When the vector-mask register is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are one. The entries in the destination vector register that correspond to a zero in the mask register are unaffected by the vector operation.

4. **Memory Banks**

   The behaviour of the load/store vector unit is significantly more complicated than that of the arithmetic functional units. Ther will be more memory bank stalls which

reduce effective throughput. multiple independent memory banks optimize the memory system to support vector processors.

5. **Stride**

   The position in memory of adjacent elements in a vector may not be sequential. For vector processors without caches, it is required to fetch elements of a vector that are not adjacent in memory. This distance separating elements to be gathered into a single register is called the stride. They are useful in handling multi-dimensional arrays.

6. **Scatter-Gather**

   Sparse matrices are common. Scatter-Gather technique is used to allow programs with sparse matrices to execute in vector mode

   A gather operation

   It takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a dense vector in a vector register.

   A scatter operation

   After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a scatter store, using the same index vector. Hardware support for such operations is called gather-scatter

7. **Programming Vector Architectures**

In vector architectures, the compilers can tell programmers at compile time whether a section of code will vectorize or not. hints are often given as to why it did not vectorize the code. This straightforward execution model allows experts in other domains to learn how to improve performance by revising their code or by giving hints to the compiler.

Graphical Processing Unit – GPU

- A GPU is available with hundreds of parallel floating-point units, which makes high-performance computing more accessible. Many scientific and multimedia applications today use GPUs.
- It is of heterogeneous execution model where CPU is the *host*, GPU is the device
- A C-like programming language for GPU, Compute Unified Device Architecture (CUDA) is used for programming GPUs. OpenCL for vendor-independent language
- The Programming model is “Single Instruction Multiple Thread” (SIMT).
- NVIDIA systems are representatives of GPU architectures. GPUs work well only with data-level parallel problems. They also have gather-scatter data transfers and mask

registers, and GPU processors have even more registers than vector processors. GPUs also rely on multithreading within a single multithreaded SIMD processor to hide memory latencies.

Threads, blocks, and grid

Thread

- A thread is associated with each data element. In CUDA, thousands of threads are utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP

Blocks

- Threads are organized into blocks. Groups of up to 512 elements are Thread Blocks. Blocks are executed independently and in any order. Different blocks cannot communicate directly but can coordinate using atomic memory operations in Global Memory
- Example

Multithreaded SIMD Processor: hardware that executes a whole thread block (32 elements executed per thread at a time)

Grid

Blocks are organized into a grid.

- Thread management is handled by GPU hardware and not by applications or OS.
- The GPU hardware contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (bodies of vectorized loop).
- It also has a Thread Block Scheduler
- A Thread Block Scheduler determines the number of thread blocks needed for the loop and keeps allocating them to different multithreaded SIMD Processors until the loop is completed.

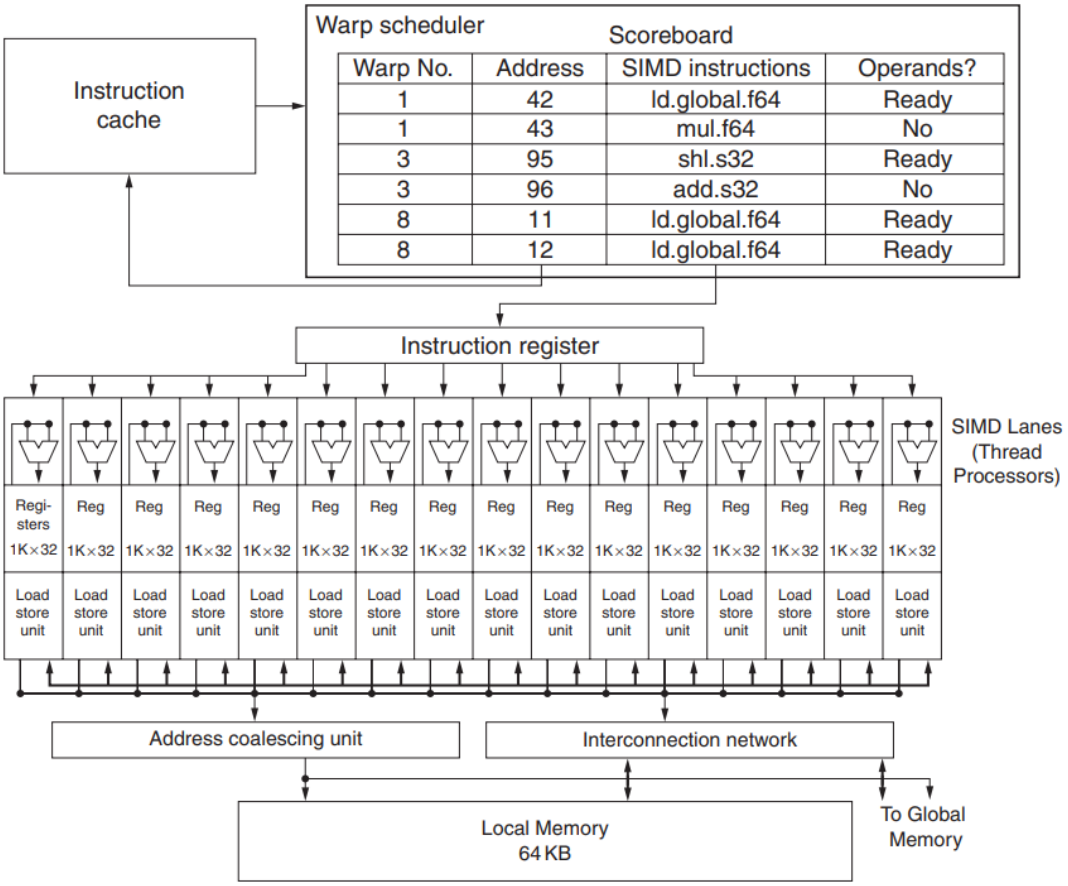
Simplified block diagram of a Multithreaded SIMD Processor is shown below:

- SIMD Processors are full processors with separate PCs and are programmed using threads.
- It has 16 SIMD lanes.
- The SIMD Thread Scheduler has, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.
- It has many parallel functional units which are called as called as SIMD Lanes.
- The machine object that the hardware creates, manages, schedules, and executes is

a thread of SIMD instructions. It is a traditional thread that contains exclusively SIMD instructions.

- Threads of SIMD instructions have their own PCs and they run on a multithreaded SIMD Processor.
- The SIMD Thread Scheduler includes a scoreboard that lets it know which threads of SIMD instructions are ready to run.
- The threads of SIMD instructions are sent to a dispatch unit.
- GPU hardware has two levels of hardware schedulers:

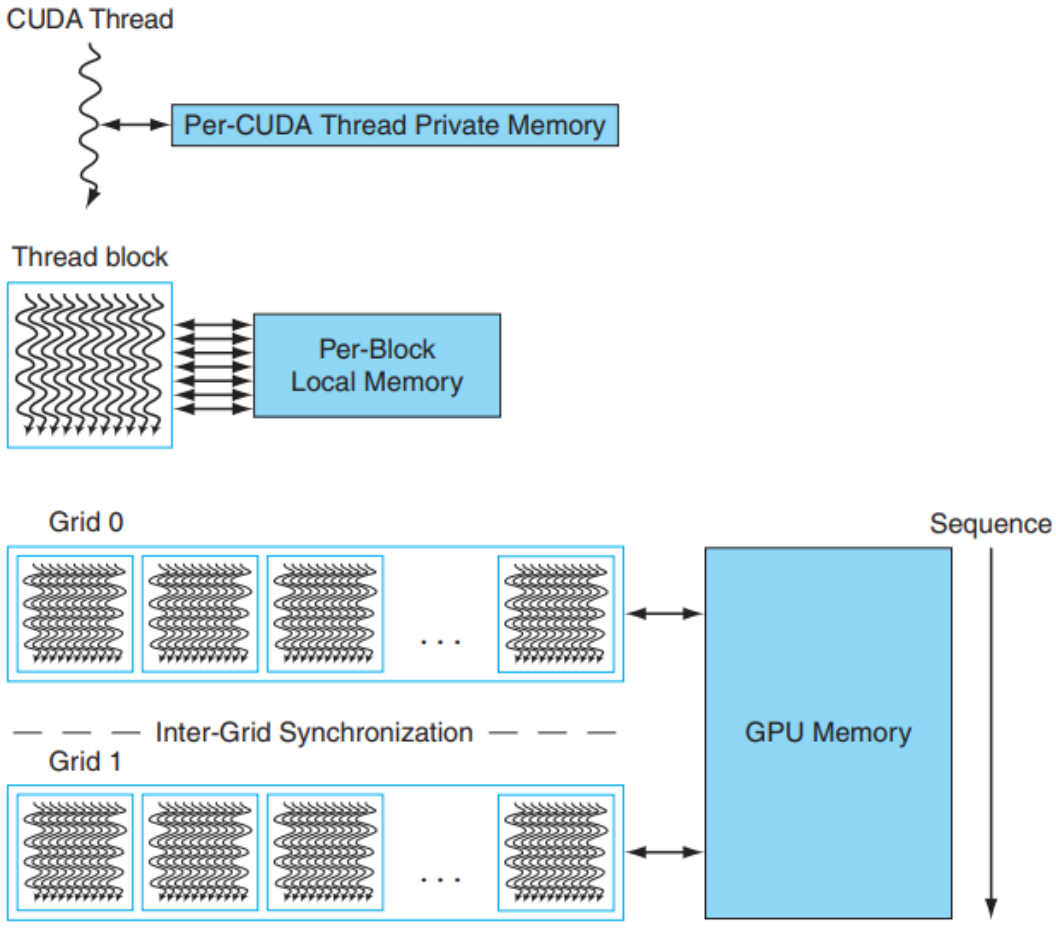
1. Thread Block Scheduler
    - It assigns Thread Blocks (bodies of vectorized loops) to multithreaded SIMD Processors, which ensures that thread blocks are assigned to the processors whose local memories have the corresponding data.
2. SIMD Thread Scheduler
    - It is present within a SIMD Processor. It schedules when threads of SIMD instructions should run. The SIMD instructions of these threads are 32 bits wide.



- The scheduler selects a ready thread of SIMD instructions. It issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

NVIDIA GPU Memory Structures

- Each SIMD Lane in a multithreaded SIMD Processor is given a private section of off-chip DRAM. It is called as Private Memory.
- It is used for the stack frame, for spilling registers, and for private variables that don't fit in the registers. SIMD Lanes do not share Private Memories.
- The on-chip memory that is local to each multithreaded SIMD Processor is called as Local Memory. It is shared by the SIMD Lanes within a multithreaded SIMD Processor. This memory is not shared between multithreaded SIMD Processors.
-



- The multithreaded SIMD Processor dynamically allocates portions of the Local Memory to a thread block when it creates the thread block, and frees the memory when all the threads of the thread block exit.

- That portion of Local Memory is private to that thread block. The off-chip DRAM shared by the whole GPU and all thread blocks is called as GPU Memory.
- The picture shows the GPU Memory structures.
- GPU Memory is shared by all Grids (vectorized loops).
- Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop).
- Private Memory is private to a single CUDA Thread.
- **Similarities to vector machines:**
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- **Differences:**
    - No scalar processor is available.
    - It uses multithreading to hide memory latency.
    - It has many functional units.

Detecting and Enhancing Loop -Level Parallelism

Loop-level parallelism

It focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations.

Loop-carried dependence

It determines whether data accesses in later iterations are dependent on data values produced in earlier iterations.

Example 1:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

- In this loop, the two uses of x[i] are dependent, but this dependence is within a single iteration and is not loop carried.
- There is a loop-carried dependence between successive uses of i in different iterations, but this dependence involves an induction variable that can be easily recognized and eliminated.

Example 2

```
for (i=0; i<100; i=i+1) {
A[i+1] = A[i] + C[i]; /* S1 */  B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration
- So there is a loop-carried dependence between S2 and S1.
- Replacing the loop above with the following code sequence, the dependence between the two statements can be eliminated.

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {  B[i+1] = C[i] + D[i];
A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Example 3

```
for (i=0;i<100;i=i+1)
{  A[i] = B[i] + C[i];
   D[i] = A[i] * E[i];
}
```

- Ther is no loop-carried dependence in this example.

Example 4

```
for (i=1;i<100;i=i+1)
{  Y[i] = Y[i-1] + Y[i];
}
```

- There is a Loop-carried dependence in the form of recurrence

Finding dependences

- Let an array element with index value a × i + b be stored and be loaded from the same array with index value c × i + d, where i is the for-loop index variable that runs from m to n.
- A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, that are both within the limits of the for loop.

   i.e. $m \le j \le n$, $m \le k \le n$.
2. The loop stores into an array element indexed by a × j + b. It later fetches from that same array element when it is indexed by c × k + d. That is, a × j + b = c × k + d.

- Dependences cannot be determined at compile time.
- Test for absence of a dependence can be done.
- Presence of antidependencies and output dependencies can be checked.