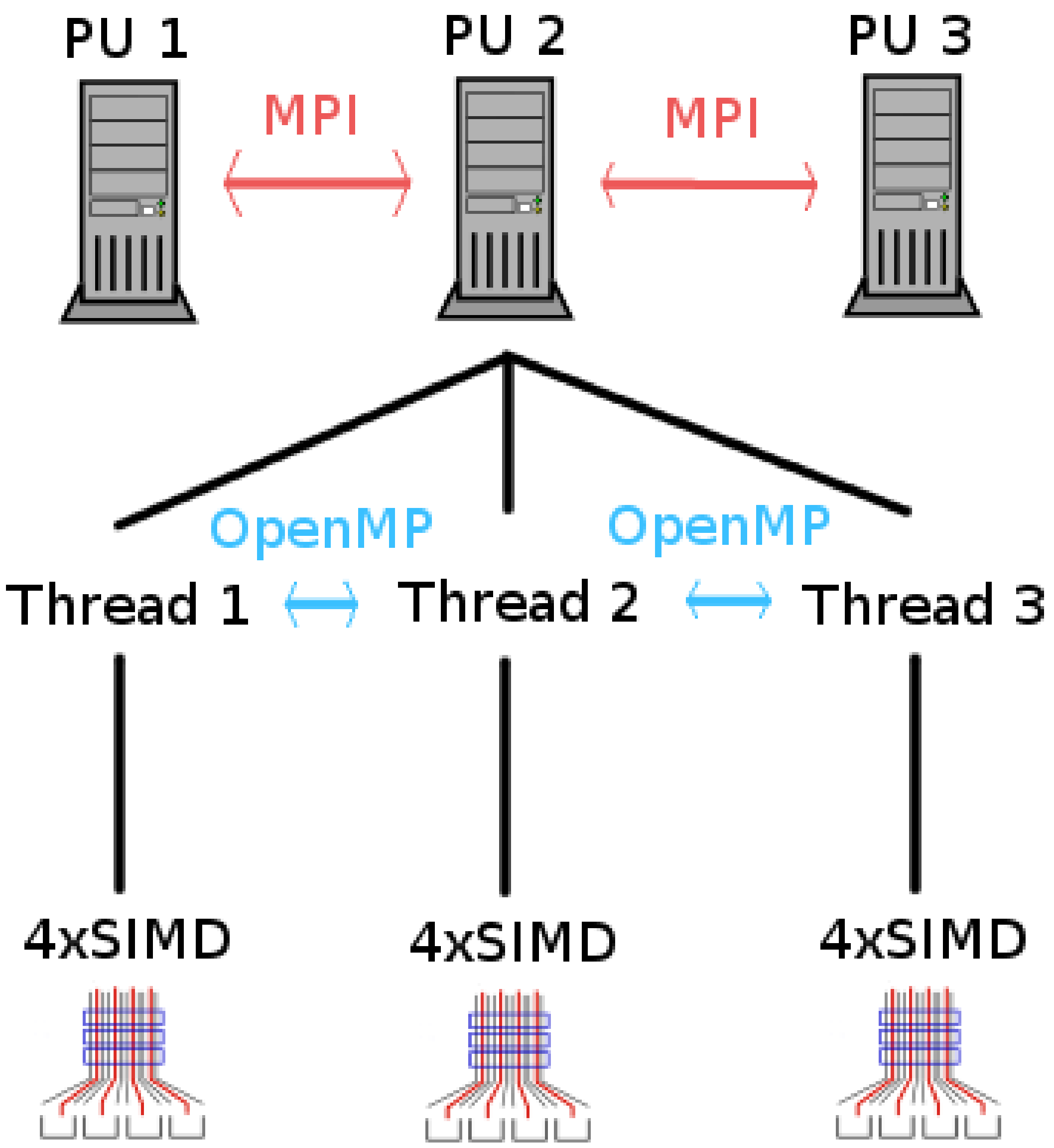
Multiprocessing, multithreading and vectorization

Sparsh Mittal

IIT Hyderabad, India

Courtesy for some slides: S. R. Sarangi and others

A parallel program running on three computation units with three threads each with width 4 SIMD unit in each thread



Background

Strong and weak scaling

- **Strong scaling:** how solution time varies with the number of cores for a fixed **total** problem size.
  - Use 2X machines for a task => solve it in half the time
- **Weak scaling:** how solution time varies with the number of cores for a fixed problem size per cores.
  - if the dataset is twice as big, use 2X machines to solve the task in constant time.

Shared Memory vs Message Passing

- Shared Memory
  - All the threadds share the virtual address space.
  - They can communicate with each other by reading and writing values from/to shared memory.
  - Application ensures no data corruption (Lock/Unlock)
  - Example language: OpenMP, CUDA
- Message Passing
  - Programs communicate between each other by sending and receiving messages (e.g., sending emails
  - They do not share memory addresses.
  - Example language: MPI

Types of Parallelism

- Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow
- Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors
- Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Flynn's Taxonomy

Flynn's Classification

- Instruction stream → Set of instructions that are executed
- Data stream → Data values that the instructions process
- Four types of multiprocessors : SISD, SIMD, MISD, MIMD

SISD and SIMD

- SISD → Standard uniprocessor
- SIMD → One instruction, operates on multiple pieces of data. Vector processors have one instruction that operates on many pieces of data in parallel. For example, one instruction can compute the $\sin^{-1}$ of 4 values in parallel.

MISD

- MISD $\rightarrow$ Multiple Instruction Single Data
  - Very rare in practice
  - Consider an aircraft that has a MIPS, an ARM, and an X86 processor operating on the same data (multiple instruction streams)
  - We have different instructions operating on the same data
  - The final outcome is decided on the basis of a majority vote.

MIMD

- MIMD → Multiple instruction, multiple data (two types, SPMD, MPMD)
- SPMD → Single program, multiple data. Examples: OpenMP or MPI programs. We typically have multiple processes or threads executing the same program with different inputs.
- MPMD → A master program, delegates work to multiple slave programs. The programs are different.

Summary

- SISD: Single instruction operates on single data element
- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Multiprocessing

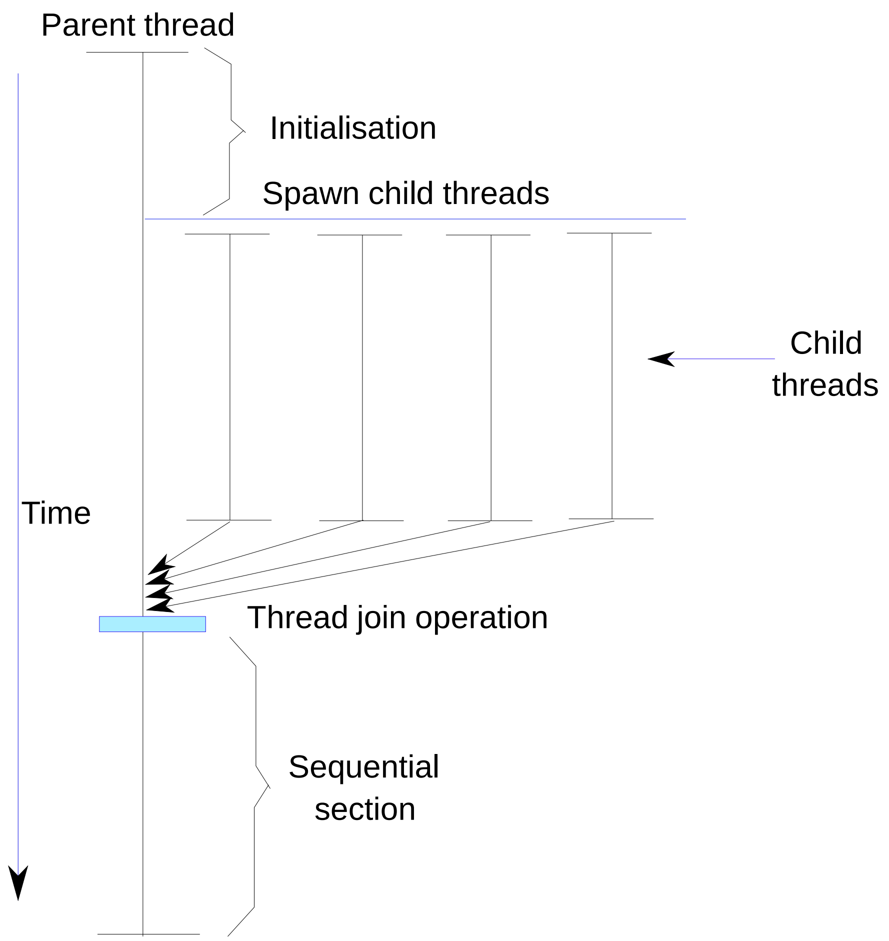
- The term multiprocessing refers to multiple processors working in parallel.
- This is a generic definition, and it can refer to multiple processors in the same chip, or processors across different chips.
- A multicore processor is a specific type of multiprocessor that contains all of its constituent processors in the same chip. Each such processor is known as a **core**.

Multithreading

The Notion of Threads

- We spawn a set of separate threads
- Properties of threads
  - A thread shares its address space with other threads
  - It has its own program counter, set of registers, and stack
  - A process contains multiple threads
  - Threads communicate with each other by writing values to memory or via synchronization operations

Operation of the Program



Multithreading

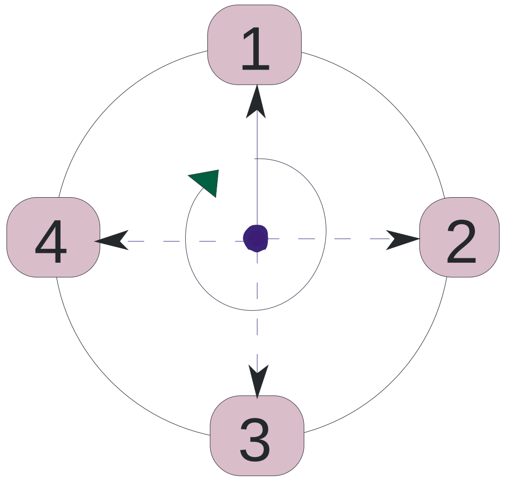
* Multithreading → A design paradigm that proposes to run multiple threads on the same pipeline.
* Three types
  * Coarse grained
  * Fine grained
  * Simultaneous

Analogy

- Consider a car which can be shared by 4 people (A, B, C, D) in following way:
- Option1: Four people ride the car simultaneously almost all the time to go to their destinations.
- Option2: A uses the car for 15 days, B for 14 days, C for 18 days and D for 16 days (and repeat).
- Option3: A uses the car for 8 hours, B for 9 hours, C for 5 hours and D for 7 hours (and repeat).
- Match the above three options to three types of multithreading

Coarse Grained Multithreading

- Assume that we want to run 4 threads on a pipeline
- Run thread 1 for n cycles, run thread 2 for n cycles, ….



Implémentation

- Steps to minimize the context switch overhead
- For a 4-way coarse grained MT machine
  - 4 program counters
  - 4 register files
  - 4 flags registers
  - A context register that contains a thread id.
  - Zero overhead context switching $\rightarrow$ Change the thread id in the context register

Advantages

* Assume that thread 1 has an L2 miss
    * **Wait** for 200 cycles
    * Schedule thread 2
    * Now let us say that thread 2 has an L2 miss
    * Schedule thread 3
* We can have a **sophisticated algorithm** that switches every n cycles, or when there is a **long latency event** such as an L2 miss.
    * Minimises idle cycles for the entire system

Fine Grained Multithreading

* The switching granularity is very small
  * 1-2 cycles
* Advantage :
  * Can take advantage of low latency events such as division, or L1 cache misses
  * Minimise idle cycles to an even greater extent
* Correctness Issues
  * We can have instructions of 2 threads simultaneously in the pipeline.
  * We never forward/interlock for instructions across threads

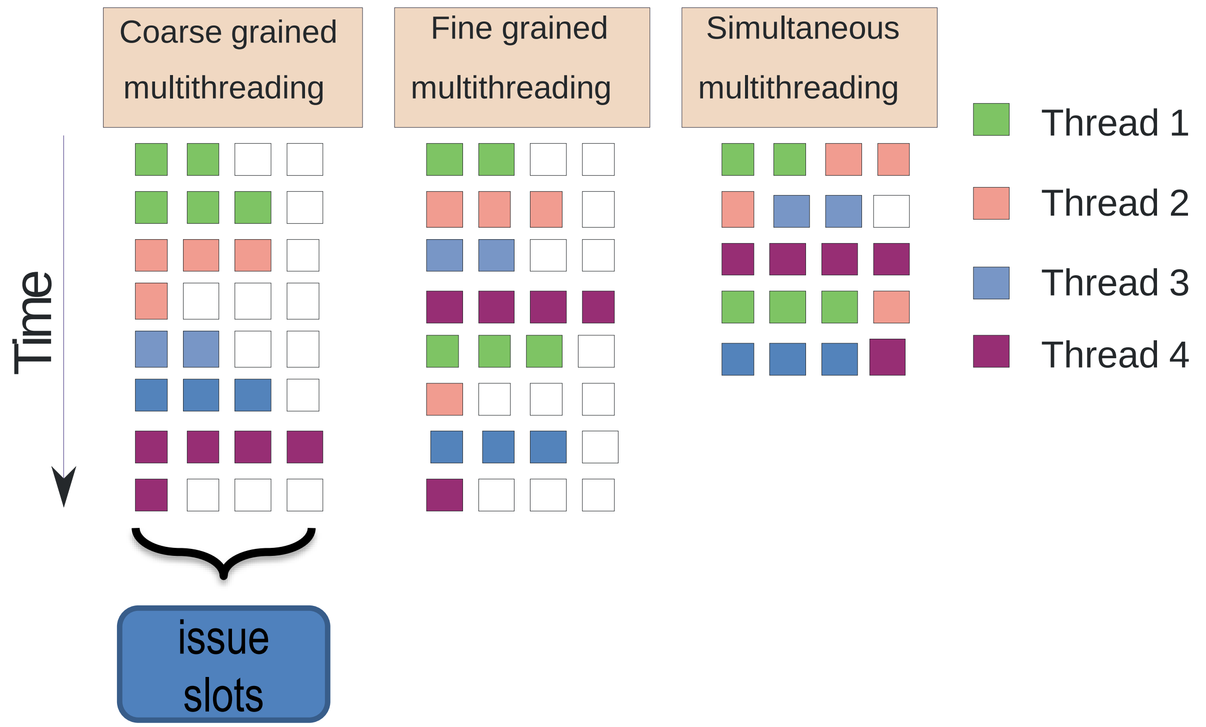
Simultaneous Multithreading

* Most modern processors have multiple issue slots
  * Can issue multiple instructions to the functional units
  * For example, a 3 issue processor can fetch, decode, and execute 3 instructions per cycle
  * If a benchmark has low ILP (instruction level parallelism), then fine and coarse grained multithreading cannot **really help**.

Simultaneous Multithreading

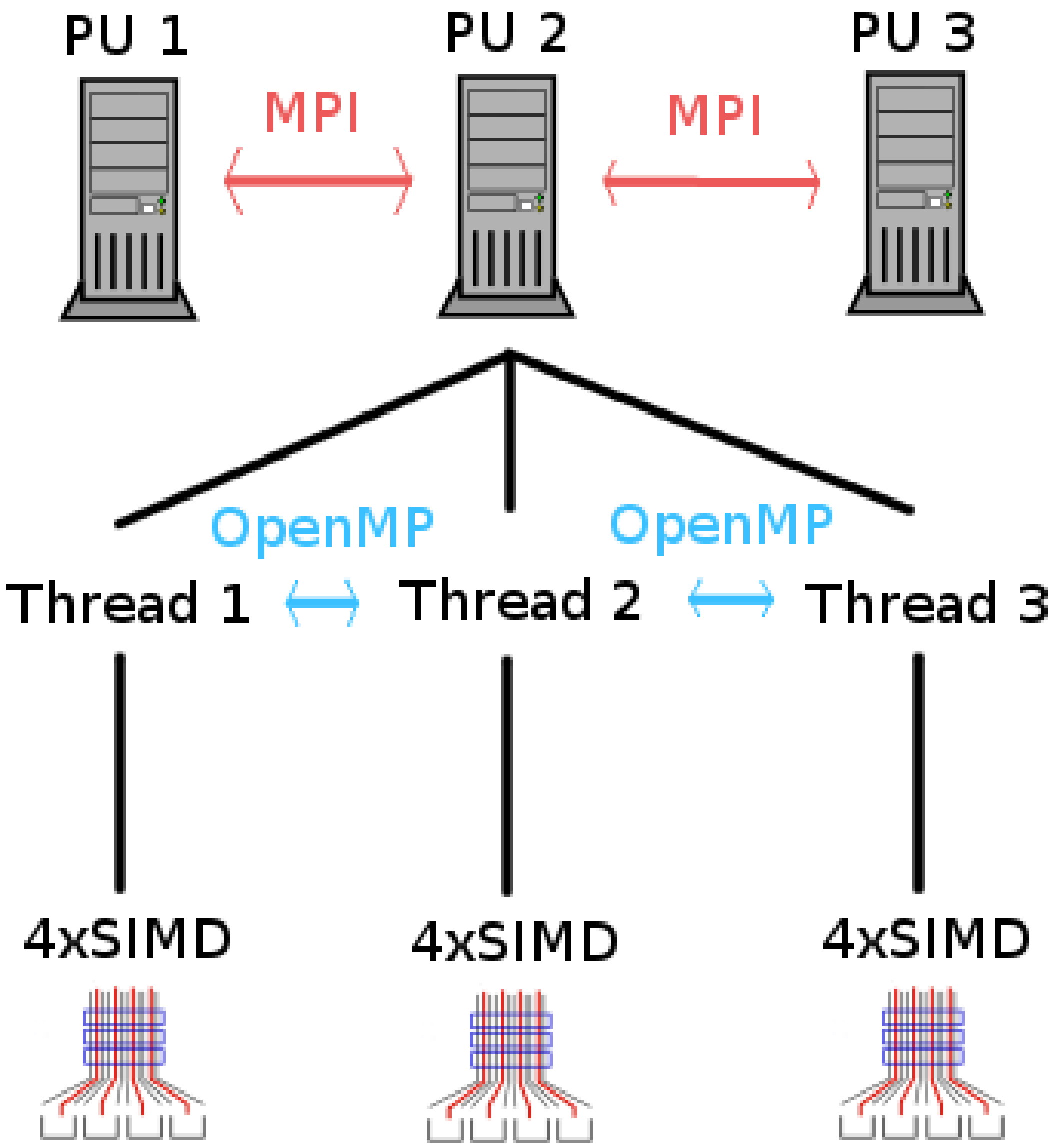
- Main Idea
  - Partition the issue slots across threads
  - **Scenario** : In the same cycle
    - Issue 2 instructions for thread 1
    - and, issue 1 instruction for thread 2
    - and, issue 1 instruction for thread 3
- Support required
  - Need smart instruction selection logic.
    - Balance fairness and throughput

Summary


Coarse grained multithreading
Fine grained multithreading
Simultaneous multithreading
Time
issue slots
Thread 1
Thread 2
Thread 3
Thread 4

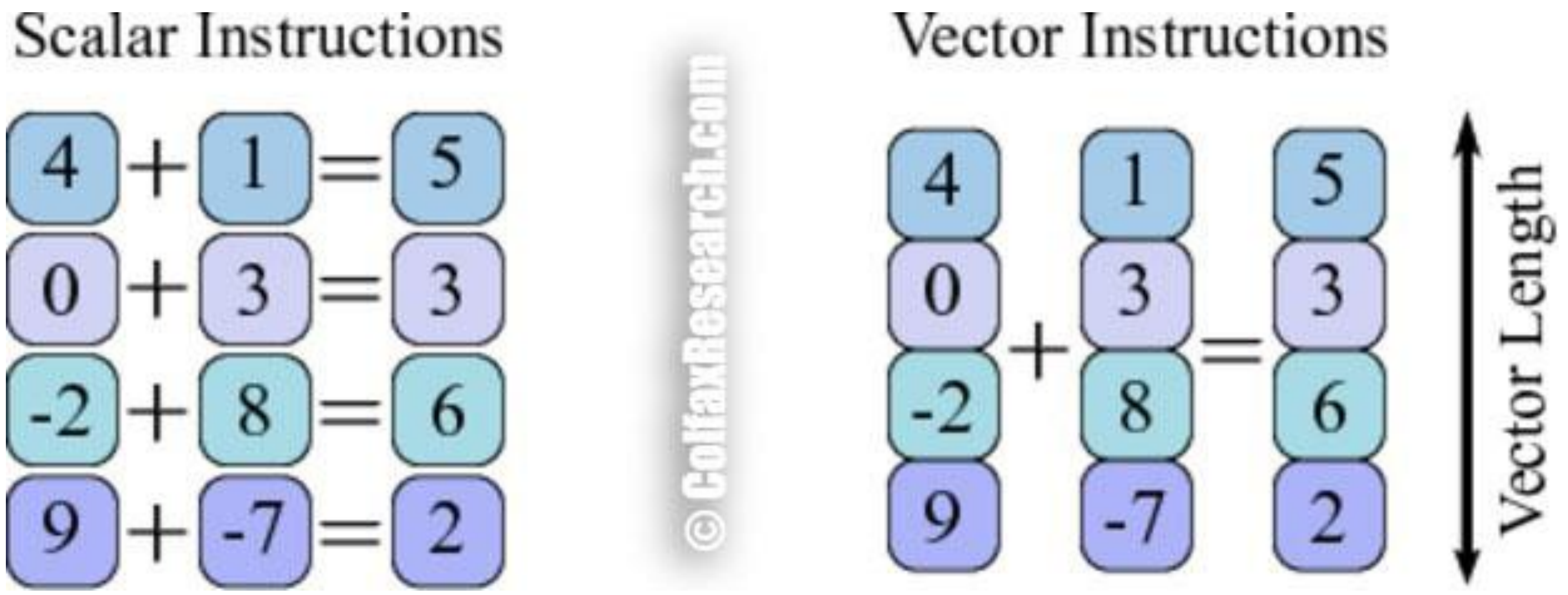
Vectorization (and vector processor)

A parallel program running on three computation units with three threads each with width 4 SIMD unit in each thread



BIG PICTURE

Vectorization



https://colfaxresearch.com/knl-avx512/

Some of the SIMD instruction sets used in industry

Register size	Instruction set
80 bits	MMX
128-bits	SSE1/SSE2 etc.
256-bits	AVX/AVX2
512-bits	AVX-512

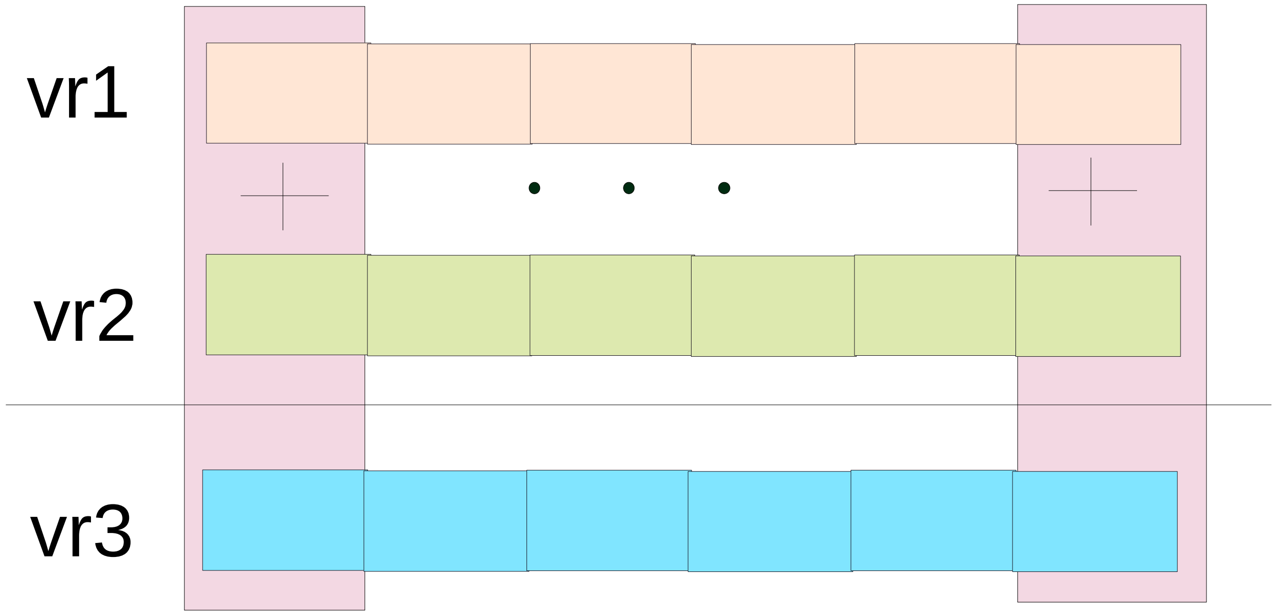
Vector Processors

- A vector instruction operates on arrays of data
  - Example : There are vector instructions to add or multiply two arrays of data, and produce an array as output
  - **Advantage** : Can be used to perform all kinds of array, matrix, and linear algebra operations. These operations form the core of many scientific programs, high intensity graphics, and data analytics applications.

Software Interface

- Let us define a vector register
  - Example : 128 bit registers in the MMX instruction set → XMM0 … XMM15
  - Can hold 4 floating point values, or 8 2-byte short integers
  - Addition of vector registers is equivalent to pairwise addition of each of the individual elements.
  - The result is saved in a vector register of the same size.

Example of Vector Addition



Let us define 8 128 bit vector registers in SimpleRisc. vr0 ... vr7

Loading Vector Registers

- There are two **options** :
  - Option 1 : We assume that the data elements are stored in contiguous locations
  - Let us define the v.ld instruction that uses this assumption.

Instruction	Semantics
v.ld vr1, 12[r1]	vr1 $\leftarrow$ ([r1+12], [r1+16],[r1+20], [r1+24])

  - Option 2: Assume that the elements are not saved in contiguous locations.
    - For this, there are scatter-gather instructions

Scatter Gather Operation

- The data is scattered in memory
  - The load operation needs to gather the data and save it in a vector register.
- Let us define a scatter gather version of the load instruction → v.sg.ld
  - It uses another vector register that contains the addresses of each of the elements.

Instruction	Semantics
v.sg.ld vr1, vr2	vr1 ← ([vr2[0]], [vr2[1]], [vr2[2]], [vr2[3]])

Vector Store Operation

* We can similarly define two vector store operations

Instruction	Semantics
v.sg.st vr1, vr2	[vr2[0]] ← vr1[0] [vr2[1]] ← vr1[1] [vr2[2]] ← vr1[2] [vr2[3]] ← vr1[3]

Instruction	Semantics
v.st vr1, 12[r1]	[r1+12] ← vr1[0] [r1+16] ← vr1[1] [r1+20] ← vr1[2] [r1+24] ← vr1[3]

Vector Operations

- We can now define custom operations on vector registers
  - v.add → Adds two vector registers
  - v.mul → Multiplies two vector registers
  - We can even have operations that have a vector operand and a scalar operand → Multiply a vector with a scalar.

Design of a Vector Processor

* Salient Points
    * We have a vector register file and a scalar register file
    * There are scalar and vector functional units
    * Unless we are converting a vector to a scalar or vice versa, we in general do not forward values between vector and scalar instructions
    * The memory unit needs support for regular operations, vector operations, and possibly scatter-gather operations.

References

- S. Mittal et al, “A Survey on Evaluating and Optimizing Performance of Intel Xeon Phi” 2019 (pdf)