**Vector Processing CLC**

Alec Utzerath and Angel Velazquez

CST-307 Introduction to Computer Architecture

Professor Ricardo Citro

December 12, 2023

*Project Description:*

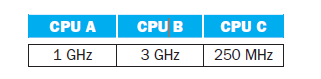
*In future systems, we expect to see heterogeneous computing platforms constructed out of heterogeneous CPUs. We have begun to see some appear in the embedded processing market in systems that contain both floating point DSPs and a microcontroller CPUs in a multichip module package. Assume that you have three classes of CPU:*

*CPU A: A moderate speed multi-core CPU (with a floating point unit) that can execute multiple instructions per cycle.*

*CPU B: A fast single-core integer CPU (i.e., no floating point unit) that can execute a single instruction per cycle.*

*CPU C: A slow vector CPU (with floating point capability) that can execute multiple copies of the same instruction per cycle.*

*Assume that our processors run at the following frequencies:*

**

*CPU A can execute 2 instructions per cycle, CPU B can execute 1 instruction per cycle, and CPU C can execute 8 instructions (though the same instruction) per cycle. Assume all operations can complete execution in a single cycle of latency without any hazards.*

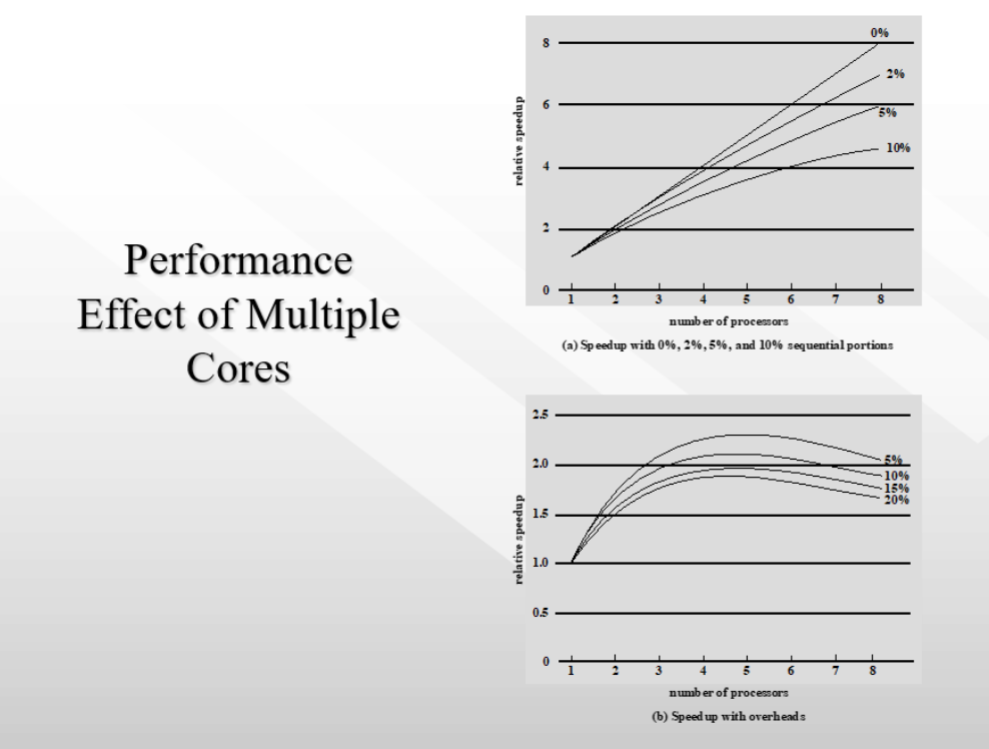
*All three CPUs can perform integer arithmetic, though CPU B cannot perform floating-point arithmetic. CPU A and B have an instruction set similar to a MIPS processor. CPU C can only perform floating point add and subtract operations, as well as memory loads and stores. Assume all CPUs have access to shared memory and that synchronization has zero cost.*

*The task at hand is to compare two matrices X and Y that each contain 1024 × 1024 floating point elements. The output should be a count of the number indices where the value in X was larger or equal to the value in Y.*

1. *Describe how you would partition the problem on the 3 different CPUs to obtain the best performance.*
2. *What kind of instruction would you add to the vector CPU C to obtain better performance?*

Question 1

To attain optimal performance in comparing matrices X and Y, a judicious partitioning strategy must be applied, considering the strengths of each CPU. CPU A, with its multi-core architecture and floating-point unit, is well-suited for parallel tasks. Rows of the matrices can be divided among the CPU A cores, allowing simultaneous comparisons of different subsets. This exploits the multicore nature, accelerating the overall process. CPU B, being a fast single-core integer CPU, can efficiently handle tasks that don't involve floating-point arithmetic. It can be employed for control operations and indexing, maximizing its processing power. CPU C, the slow vector CPU with floating-point capability, is ideal for element-wise comparisons. Columns of the matrices can be assigned to vector lanes, enabling simultaneous comparison of multiple elements. This strategy optimally allocates tasks, harnessing the unique capabilities of each CPU and ensuring efficient parallelization.



*Image 1: Effective Speed up of Multiple Core Systems*

Image 1 shows the effective speed-up of multiple core systems. CPU A is a moderate-speed multicore system and can still execute 2 instructions per cycle compared to CPU B which is a fast single-core system but can only execute 1 instruction per cycle.

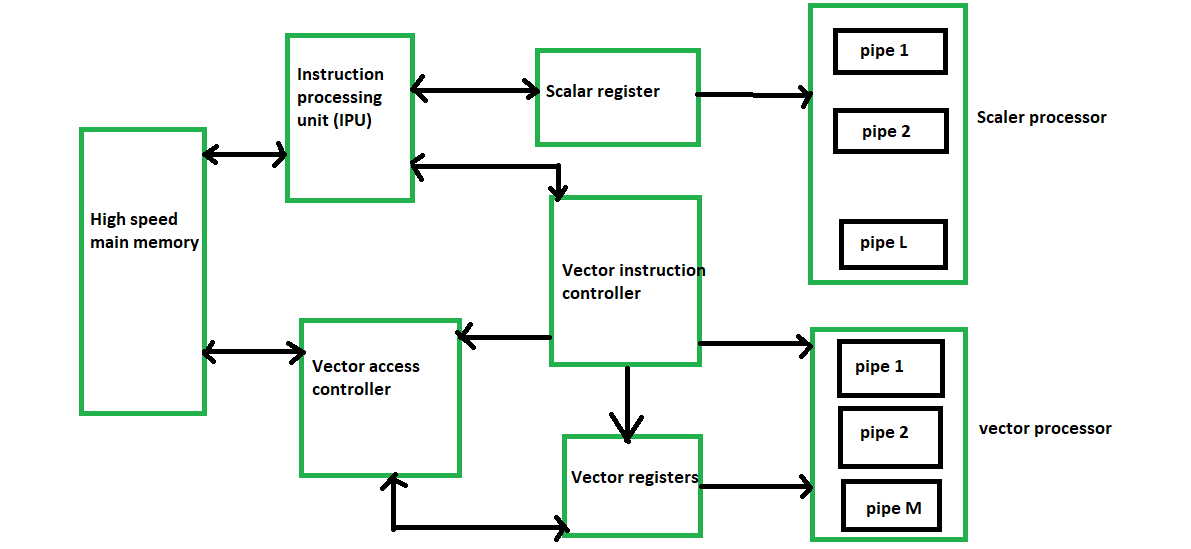
In optimizing the performance of our heterogeneous computing platform comprising CPUs A, B, and C for the task of comparing matrices X and Y, we strategically plan the partitioning strategy to leverage each CPU's unique capabilities. Considering the specific attributes of each CPU—CPU A being a faster multi-core processor capable of executing 2 instructions per cycle, CPU B as a swift single-core integer processor with no floating-point unit, and CPU C as a slower vector processor with floating-point capabilities capable of executing 8 instructions per cycle—we aim to distribute the workload effectively.

Partitioning Approach:

* CPU A (Multi-Core): Distribute rows of matrices X and Y among the multiple cores of CPU A to harness its parallel processing capability. Given its higher speed and the ability to execute 2 instructions per cycle, assigning a substantial workload to CPU A ensures efficient and simultaneous execution across its cores.
* CPU B (Single-Core Integer): Task CPU B with control operations and indexing, focusing on aspects of the comparison that don't involve floating-point arithmetic. Despite lacking floating-point capabilities, CPU B's efficiency in executing 1 instruction per cycle makes it suitable for specific computational tasks.
* CPU C (Slow Vector CPU): Allocate the element-wise comparison task to CPU C by distributing columns of matrices X and Y among its vector lanes. Leveraging the slow but vectorized nature of CPU C, which can execute 8 instructions per cycle, allows for efficient parallelization of the matrix comparison.

**Question 2**

To enhance the performance of CPU C in the matrix comparison, a specialized "Vector Compare" instruction can be introduced. This instruction performs parallel element-wise comparisons between corresponding elements in two vectors and generates a vector mask indicating the results, such as greater than or equal. By tailoring an instruction specifically for the matrix comparison task, the vector CPU's ability to execute multiple copies of the same instruction per cycle can be leveraged. This enhancement aligns with the CPU's vector capabilities and the nature of the matrix comparison, resulting in a substantial boost in performance. The added instruction efficiently utilizes the CPU's architecture, ensuring that it excels in its role and contributes significantly to the overall speedup of the matrix comparison process.

**

*Image 2: Vector Float CPU Diagram*

To enhance the efficiency of CPU C in comparing matrices X and Y, we propose incorporating a specialized vectorized instruction designed for the specific task of element-wise matrix comparison. Given CPU C's capability to execute 8 instructions per cycle in a vectorized manner, introducing an instruction tailored for parallelized comparison operations can significantly improve its performance.

Vectorized Comparison Instruction:

The newly introduced instruction should enable the simultaneous comparison of multiple elements within the vector lanes. This instruction would leverage CPU C's parallel processing capacity, allowing it to perform element-wise comparisons on multiple pairs of matrix elements in a single cycle. The instruction may include features like conditional execution logic to determine the outcome of each individual comparison.

Explanation:

* Parallel Processing: By exploiting the vector lanes, the instruction empowers CPU C to concurrently compare multiple pairs of elements from matrices X and Y. This aligns with the nature of the matrix comparison task, enhancing the efficiency of the vector processor.
* Reduced Latency: Conducting multiple comparisons in a single cycle minimizes latency, resulting in faster decision-making for each element. This is particularly advantageous for large matrices where swift and efficient element-wise comparisons are critical.
* Optimized Resource Utilization: The vectorized comparison instruction optimizes the utilization of CPU C's vector lanes, maximizing throughput for the specific task of matrix comparison. This specialization contributes to overall system efficiency in a heterogeneous computing environment.

By introducing a dedicated vectorized instruction for matrix comparison, CPU C can leverage its strengths in parallel processing, thereby enhancing the overall performance and throughput of the heterogeneous computing platform.

Example of Vector Register Code:

VEC\_CMP Instruction:

* VEC\_CMP Vd, Va, Vb

Vd: Destination vector register

Va, Vb: Source vector registers for matrices X and Y

In this example, the resulting vector Vd indicates, element-wise, whether the corresponding elements in Va are greater than or equal to the elements in Vb. This vectorized approach enables concurrent comparison operations, optimizing the processing of large matrices in a heterogeneous computing environment.

**Conclusion**

In conclusion, the analysis of the heterogeneous computing scenario involving CPUs A, B, and C highlights the strategic partitioning of tasks to exploit the strengths of each processor. The effective speedup of multicore systems, with CPU A's ability to execute multiple instructions per cycle, contributes significantly to overall performance. Additionally, the introduction of a specialized vectorized instruction, such as the hypothetical "VEC\_CMP," empowers CPU C to efficiently perform element-wise comparisons, enhancing its role in parallelized tasks. This approach optimally utilizes the diverse capabilities of each CPU, presenting a compelling strategy for addressing complex computations and leveraging the advantages of heterogeneous computing platforms.