Foundation of Modern Computer Architecture

Silvio Stanzani , Raphael Cóbe and Jefferson Fialho

UNESP – Center for Scientific Computing

silvio.stanzani@sprace.org.br, raphael.cobe@sprace.org.br, jefferson.fialho@sprace.org.br

Agenda

- Parallel Architectures
- Memory System
- Vectorization
- Optimizing Memory Access;
- Vectorization Process;
- Auto Vectorization;
- Guided Vectorization;

Parallel Processing

- A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem

- Parallel processing includes techniques and technologies that make it possible to compute in parallel

  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …

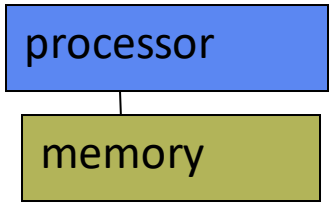
- Parallel computing is an evolution of serial computing
  - Parallelism is natural
  - Computing problems differ in level / type of parallelism

Classifying Parallel Systems Flynn's Taxonomy

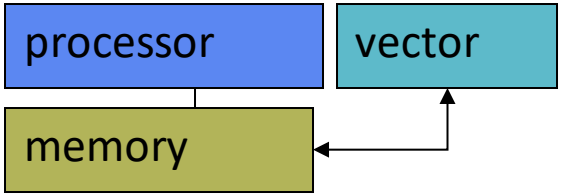
- Distinguishes multi-processor computer architectures along the two independent dimensions
  - Instruction and Data
  - Each dimension can have one state: Single or Multiple
- SISD: Single Instruction, Single Data
  - Serial (non-parallel) machine
- SIMD: Single Instruction, Multiple Data
  - Processor arrays and vector machines
- MISD: Multiple Instruction, Single Data (unusual)
- MIMD: Multiple Instruction, Multiple Data
  - Most common parallel computer systems

Parallel Architecture Types

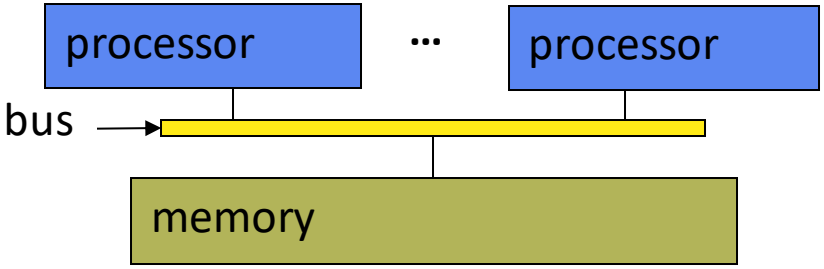
- Uniprocessor
  - Scalar processor



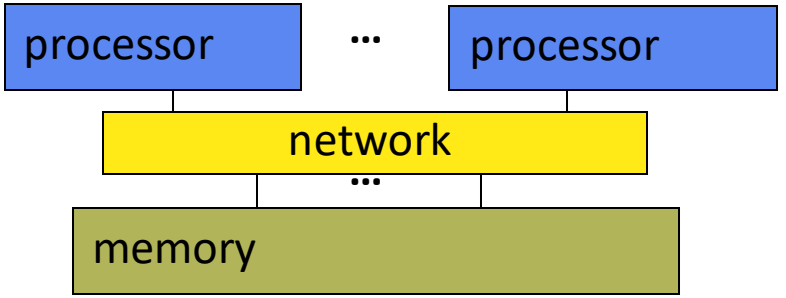
  - Vector processor



- Shared Memory Multiprocessor (SMP)
  - Shared memory address space
  - Bus-based memory system

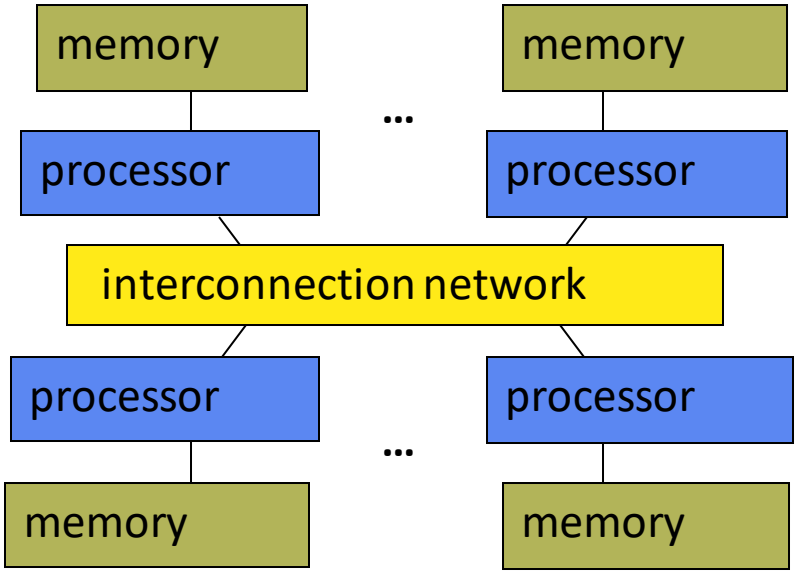


  - Interconnection network



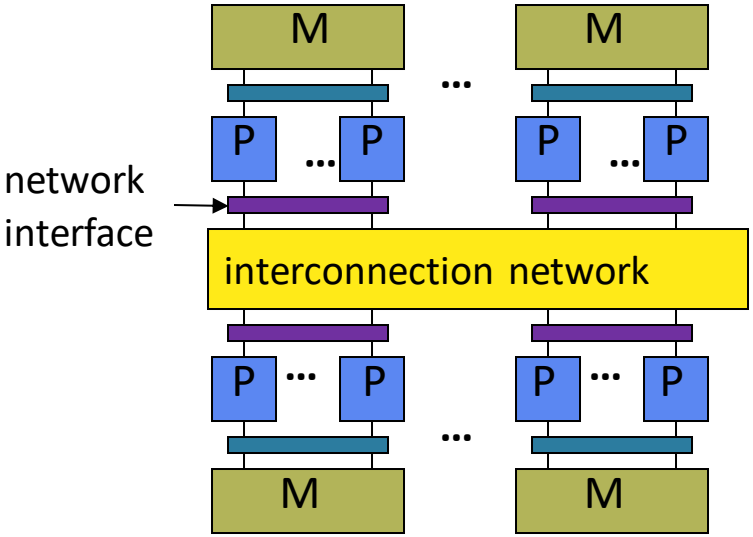
Parallel Architecture Types (2)

- Distributed Memory Multiprocessor
  - Message passing between nodes



  - Massively Parallel Processor (MPP)
    - Many, many processors

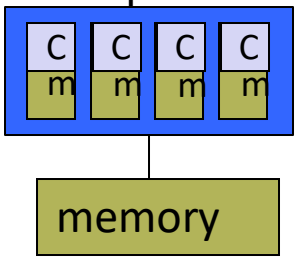
- Cluster of SMPs
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes



  - Can also be regarded as MPP if processor number is large

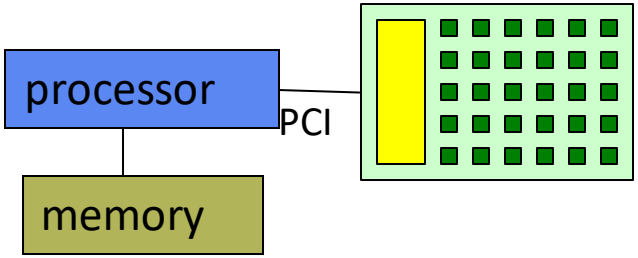
Parallel Architecture Types (3)

- Multicore
  - Multicore processor

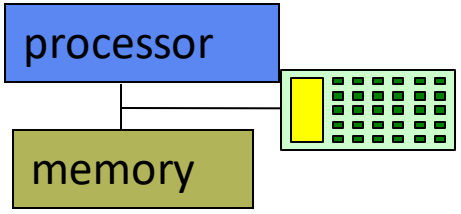


cores can be hardware multithreaded (hyperthread)

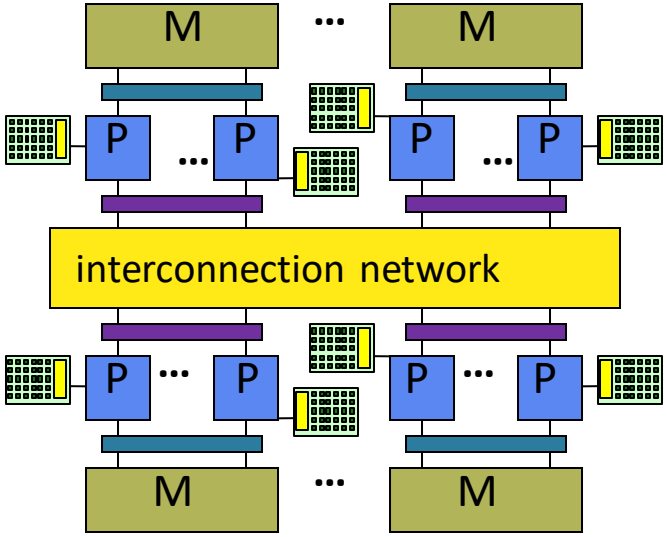
  - Coprocessor



  - "Fused" processor accelerator

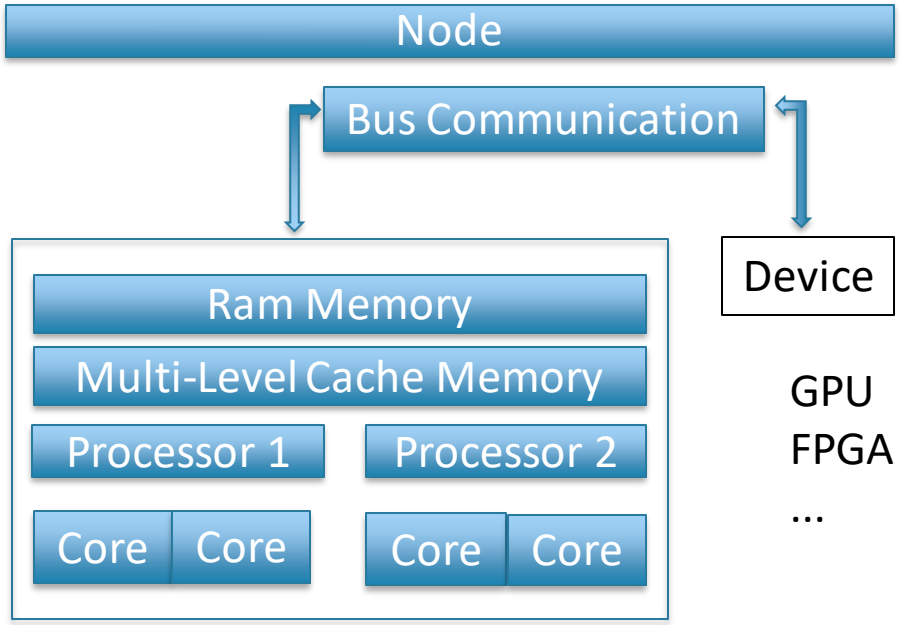


- Multicore SMP+coprocessor Cluster
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes
  - Coprocessor attached



Heterogeneous Computational Systems

- Heterogeneity
  - Processing resources
  - Memory resources
- Multi-level parallelism:
  - Computing cluster
    - Multiprocessing
  - Coprocessors and accelerators
    - Offload
  - Chip multiprocessor
    - Multithreading
  - Processing core
    - Vectorization



Memory System

Fast

CPU Register: internal Processor Memory. Stores data or instruction to be executed

Cache: stores segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations

Main memory: only program and data currently needed by the processor resides in main memory

Auxiliary memory: devices that provides backup storage

Larger in Size

Cache Memory

- Cache Memory is employed in computer systems to compensate for the difference in speed between main memory access time and processor.

- Operating System controls the load of Data to Cache;
  - such load can be guided by the developer

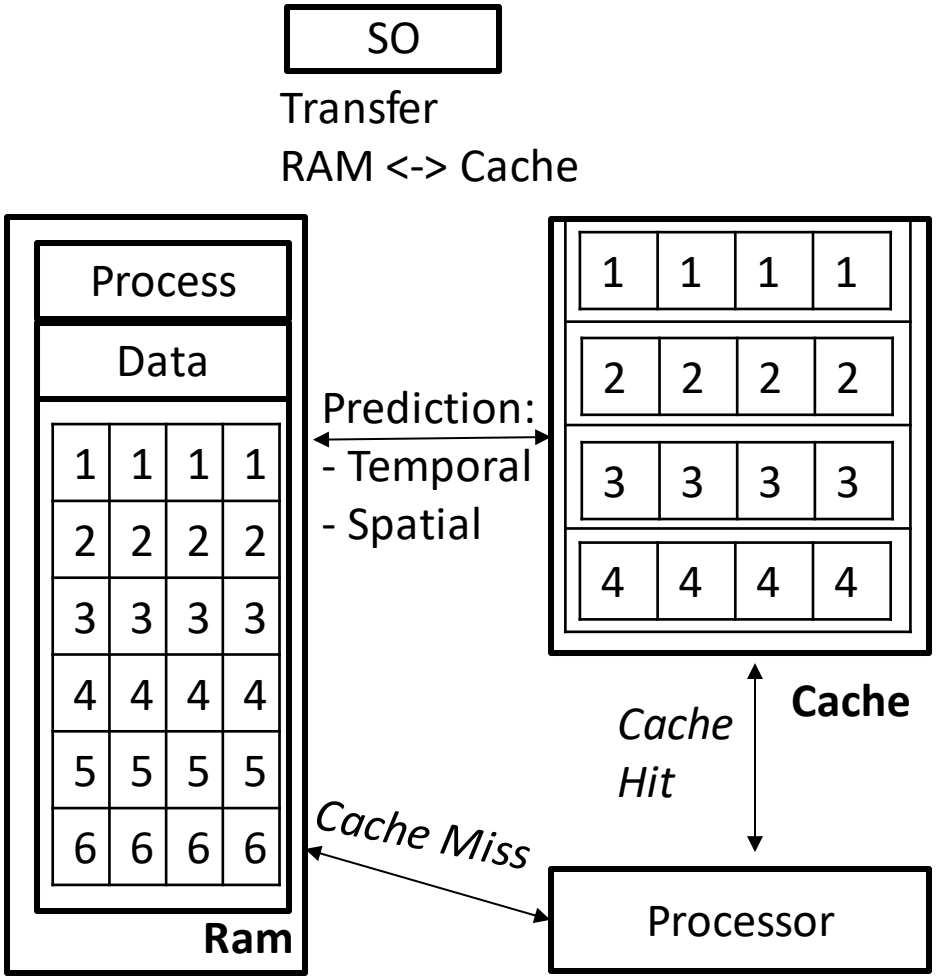
- The performance of cache memory is frequently measured in terms of hit ratio.

  - When the CPU refers to memory and finds the word in cache, it is said to produce a **hit**.

  - If the word is not found in cache, it is in main memory and it counts as a **miss**

Locality

- Temporal locality: if an item was referenced, it will be referenced again soon (e.g. cyclical execution in loops);
- Spatial locality: if an item was referenced, items close to it will be referenced too (the very nature of every program – serial stream of instructions)

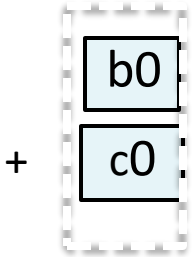


Scalar and Vector Instructions

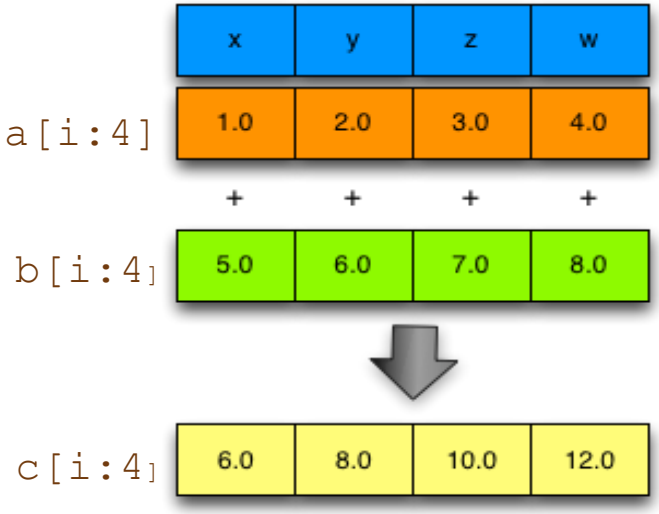
- **Scalar** Code computes this one-element at a time.
- **Vector (or SIMD)** Code computes more than one element at a time.
  - SIMD stands for **S**ingle **I**nstruction **M**ultiple **D**ata.
- Vectorization
  - Loading data into cache accordingly;
  - Store elements on SIMD registers or vectors;
  - Iterations need to be independent;
  - Usually on inner loops.

```
float *A, *B, *C;
for(i=0;i<n;i++){
  A[i] = B[i] + C[i];
}
```

- Scalar



- SIMD



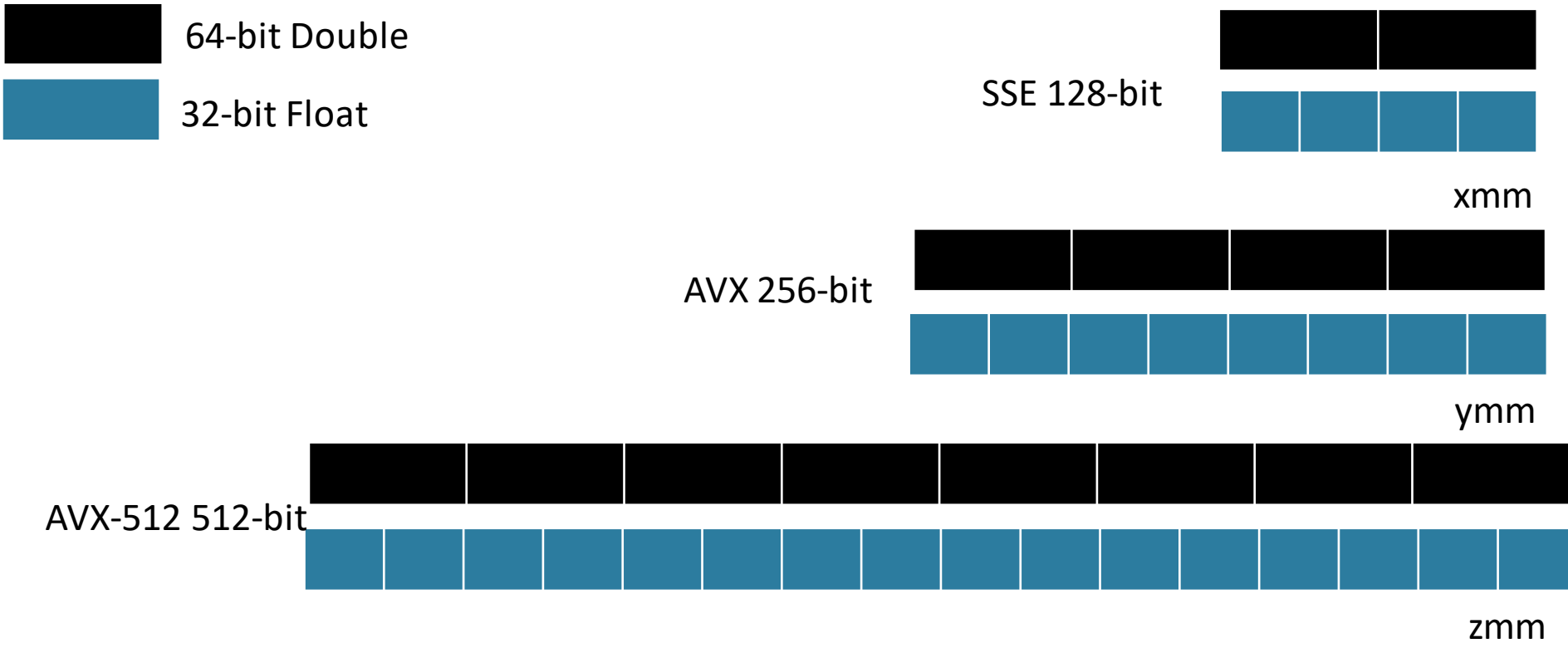
Vectorization

- **Hardware Perspective**: Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.

- **Compiler Perspective**: Determine how and when it is possible to express computations in terms of vector instructions

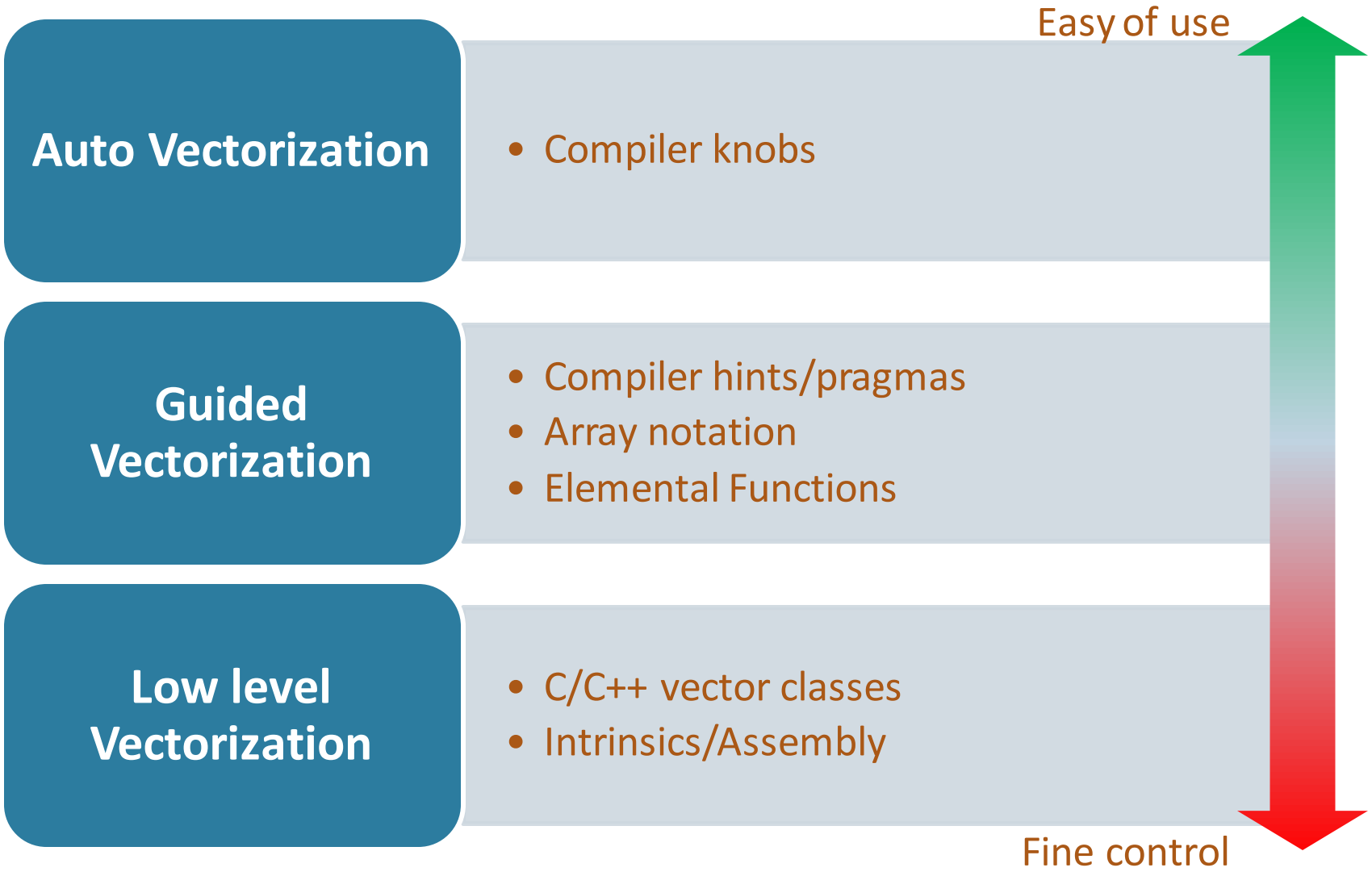
- **User Perspective**: Determine how to write code in a manner that allows the compiler to deduce that vectorization is possible

Vector Processing Units (VPU)



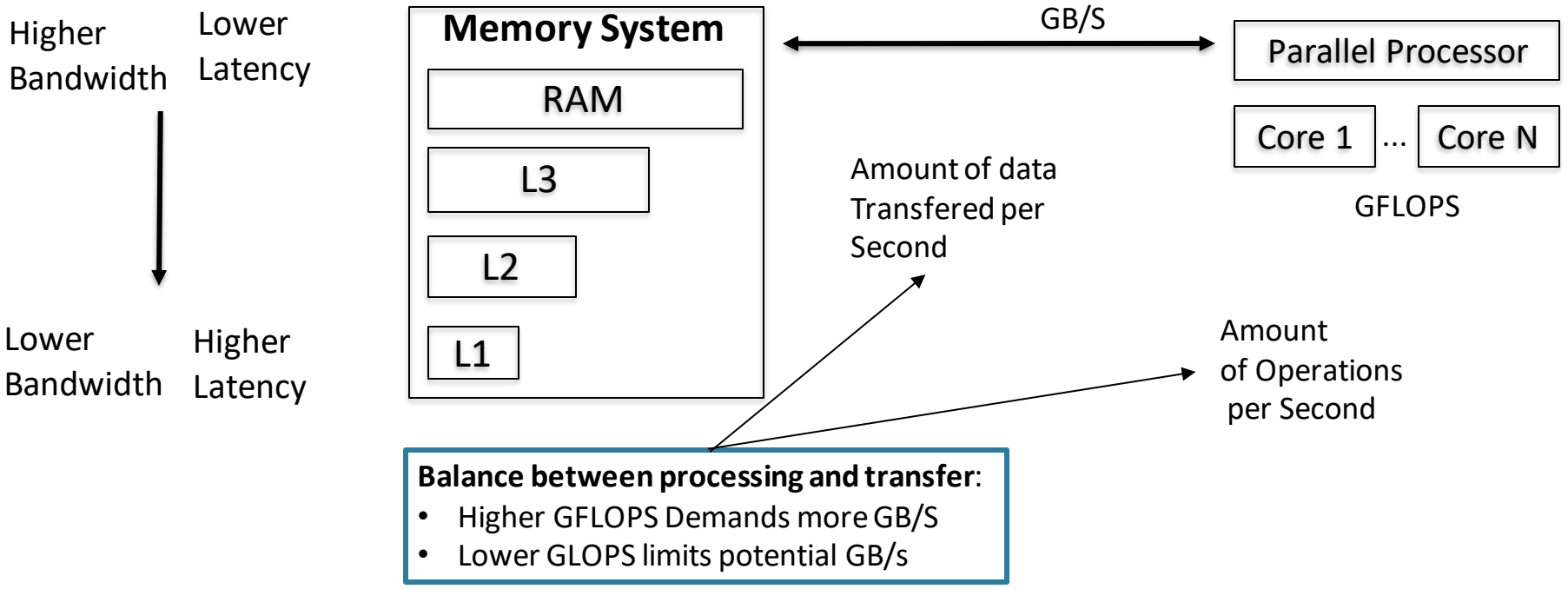
- Each Processors model offer a specific set of VPUs
- For each VPU a specific Vector Instruction set need to be used
- Compilers has means to define what instruction set to use

Vectorization



Performance Model

- Relation between memory system and vector processing

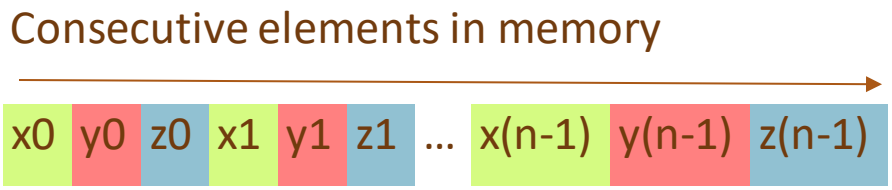


"Memory bandwidth and machine balance in high performance computers" John D. McCalpin. IEEE TCCA 1995.

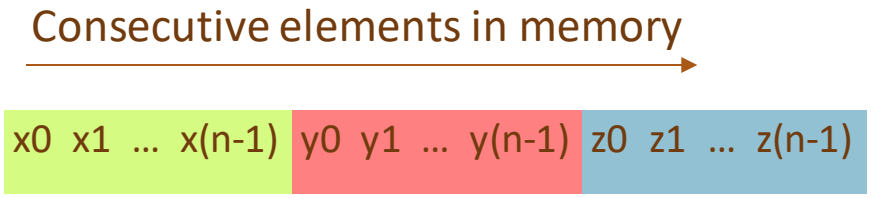
Data Layout

- AoS vs SoA (Array of Structures vs Structure of Arrays)
  - Layout your data as Structure of Arrays (SoA)

```
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
    …
for (int i = 0; i < N; i++)
    … = … f(crd[i].x, crd[i],y,
crd[i].z);
```




```
// Structure of Arrays (SoA)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
    …
for (int i = 0; i < N; i++)
    … = … f(crd.x[i], crd.y[i],
crd.z[i]);
```



Data Layout

- Stride:
  - Step size between consecutive access of array elements;

- Strided access with stride k means touching every kth memory element
  - Unit Stride :
    - Sequential access (0, 1, 2, 3, 4, 5, 6, ...)
  - Non-unit stride
    - Constant Stride =
      - 2 is (0, 2, 4, 6, 8, ...)
    - k is (0, k, 2k, 3k, 4k, ...)
    - Random Access;

- Strides > 1 commonly found in multidimensional data
  - Row accesses (stride=N) & diagonal accesses (stride=N+1)
  - Scientific computing (e.g., matrix multiplication)

Auto vectorization

- Relies on the compiler for vectorization
  - No source code changes
  - Enabled with **-ftree-vectorize** compiler knob (default in `-O2` and `-O3` modes)
- Compiler smart enough to apply loop transformations
  - It will allow to vectorize more loops

Option	Description
`-O0`	Disables all optimizations.
`-O1`	Enables optimizations for speed which are know to not cause code size increase.
`-O2/-O` (default)	Enables intra-file interprocedural optimizations for speed, including: • **Vectorization** • **Loop unrolling**
`-O3`	**Performs O2 optimizations and enables more aggressive loop transformations such as:** • **Loop fusion** • **Block unroll-and-jam** • **Collapsing IF statements** This option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. However, it might incur in slower code, numerical stability issues, and compilation time increase.

Vectorization: target architecture options

- march: Generate code for given CPU
  - march=native : uses the instruction set of compiling machine
  - Options:
    - SSE: **-march=ivybridge**
    - AVX: **-march=core-avx2**
    - AVX-512: **-march=skylake**
  - Show vectorized loops: -fopt-info-vec
  - Show non-vectorized loops: -fopt-info-vec-missed
  - **-S** generate assembly code, useful to evaluate is vector instruction set have been used by compiler

Auto vectorization: not all loops will vectorize

- Data dependencies between iterations
  - Proven Read-after-Write data (i.e., loop carried) dependencies
  - Assumed data dependencies
    - Aggressive optimizations

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

- Vectorization won't be efficient
  - Compiler estimates how better the vectorized version will be
  - Affected by data alignment, data layout, etc.

Inefficient vectorization

```
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

- Unsupported loop structure
  - While-loop, for-loop with unknown number of iterations
  - Complex loops, unsupported data types, etc.
  - (Some) function calls within loop bodies

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Guided vectorization:

- `#pragma gcc ivdep`
  - Force loop vectorization ignoring **all** dependencies
    - Additional clauses for specify reductions, etc.
    - **Bug responsibility relies on programmer!**

```
void v_add(float *c, float *a, float *b)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

```
void v_add(float *c, float *a, float *b)
{
#pragma gcc ivdep
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```