# SHARED MEMORY PARALLEL COMPUTING

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

APRIL 2024

Australian National University



# Logistics

- Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.





# Simultaneous Multi-Threading (SMT or Hyperthreading) and Single-Instruction Multiple Data (SIMD)





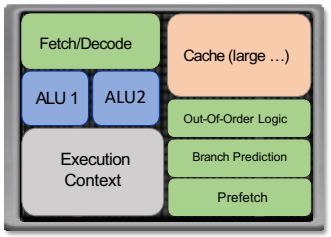
# Reference Material

- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein
- *Using OpenMP – The Next Step*, R. van der Pas, E. Stotzer, and C. Terboven, Chapter 4
- *Intel Intrinsics Guide*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- Chapter 4 from *Computer Systems A Programmer's Perspective*, Third Edition, Randal E. Bryant and David R. O'Hallaron, Pearson Education Heg USA, ISBN 9781292101767.

# Simple Single Core Superscalar CPU Architecture

- Multiple fetch/decode units
- Multiple ALUs, in general multiple Functional Units (FUs)
- One Execution Context (EC)
- Exploits Instruction Level Parallelism (ILP) through superscalarity and pipelining

- ILP requires sophisticated, additional logic to yield good performance
  - Out-of-Order (OoO) execution
  - Pre-fetching
  - Branch prediction
  - Big caches
- This extra logic is tightly coupled to the FUs and to the EC
- **Thus, this view of the CPU architecture is too simplistic**





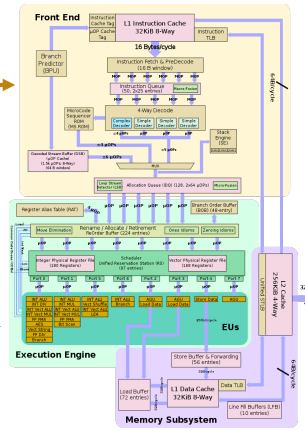


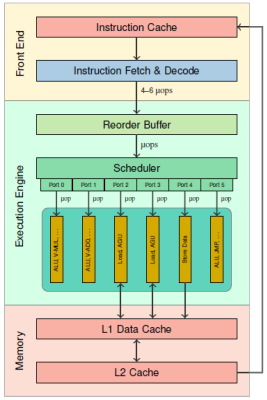

# A real single core architecture

**The front end** retrieves the instructions from memory and translates instructions into a format that can be understood by other components of the CPU.

Intel Skylake

**The execution unit** is the part of the CPU that actually carries out the instructions



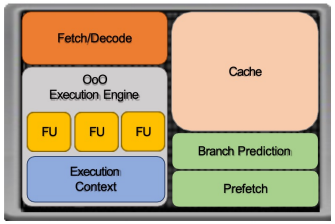



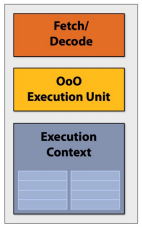





# More schematic single core architecture

- In the real core architecture Functional Units, Out-of-Order Execution Logic, Execution Context are tightly coupled at the hardware level
- Map Front End to Fetch/Decode
- Map Execution Engine to Functional Units plus Out-of-Order Execution Logic plus Execution Context
- Leave the rest of the logic, Branch Predictor, Caches, Prefetching out of our representation



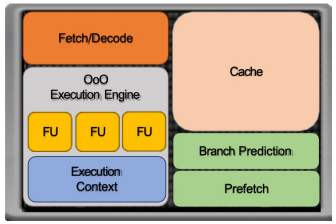



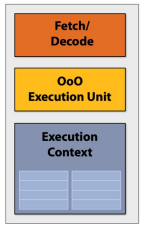





# Under utilisation of ILP

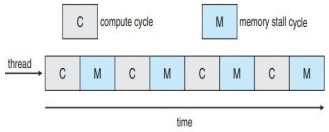
What are the main causes of under utilisation of ILP on this single-core architecture?

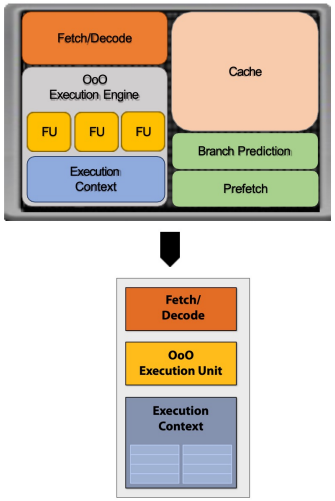


# Under utilisation of ILP

- The main causes of under utilisation of ILP on this single-core architecture: -
  - Branch misprediction
  - Bad instruction mix → cannot feed all replicates of the functional units
  - Thread stalls due to dependencies → thread cannot execute as it is waiting for operands to be fetched or dependencies to be resolved



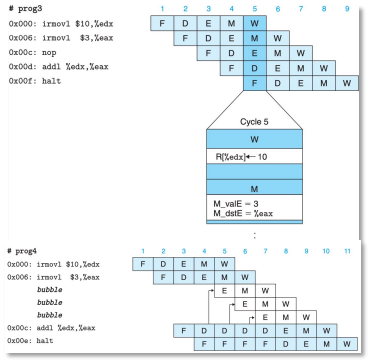








# Under utilisation of ILP

- ***Pipeline Hazards***: The next stage in the pipeline cannot execute in the following clock cycle
- Typically caused by dependencies between successive instructions
- ***Data dependencies***: the results computed by one instruction are used as the input data for a following instruction
- ***Control dependencies***: one instruction determines the location of the following instruction (*e.g*, jump, return)
- This can require the hardware to insert no-ops called *bubbles* in the pipeline, causing it to stall while the dependencies are resolved

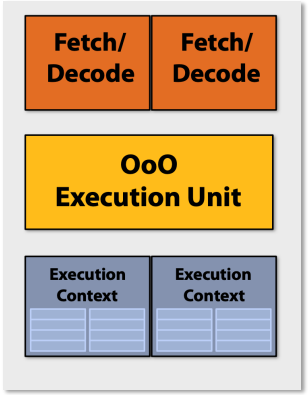







# Hyperthreading (Simultaneous Multi-Threading)

- Idea: *Interleave processing of multiple threads on the same core to hide stalls*
- This can hide the latency of one thread's stalls with another thread's execution
- What is a potential problem with this strategy?

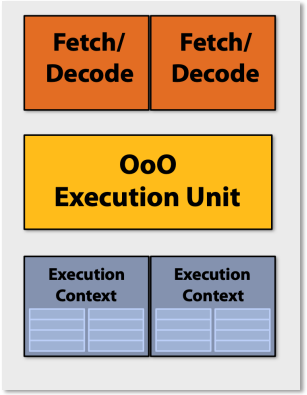






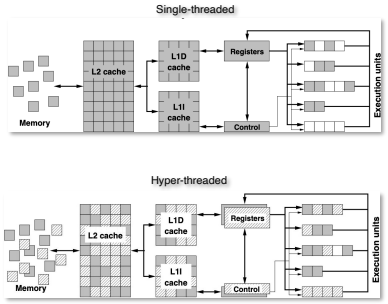

# Hyperthreading (Simultaneous Multi-Threading)

- Idea: *Interleave processing of multiple threads on the same core to hide stalls*
- This can hide the latency of one thread's stalls with another thread's execution
- What is a potential problem with this strategy? … *Context switch*



# Hyperthreading: A More Detailed View

- What is a potential problem with this strategy? Context switch -->
- Add dedicated separated execution contexts for these threads
- Since the contexts of these threads are in dedicated register space, context switch is either lightweight or for free
- → simultaneous multi-threading

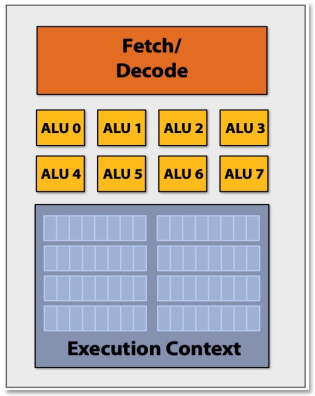







# Single Instruction Multiple Data (SIMD)

- Idea "2": Amortize cost/complexity of managing an instruction stream across many ALUs
- Single Instruction Multiple Data (SIMD) processing
- Fetch and decode one instruction
- Same instruction broadcast and executed in parallel on ALUs operating on different data elements
- Execution context must be larger, for example make registers larger (in bits)
- This requires different instructions



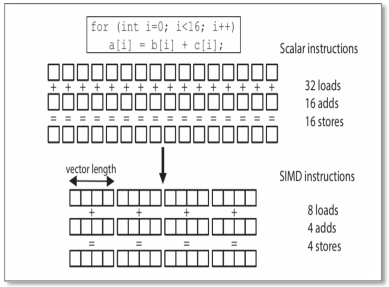





# Introduction to SIMD

**SIMD provides data-parallelism at the instruction level**

- A single instruction operates on multiple data elements in parallel SIMD instructions use special registers a larger width (*vector length*)
- SIMD instructions are as fast as their scalar counterpart, leading to potential speedup of up to the vector length
- In practice, the speedup achieved may depend heavily on memory operations needed to move the data








# Using SIMD Capabilities

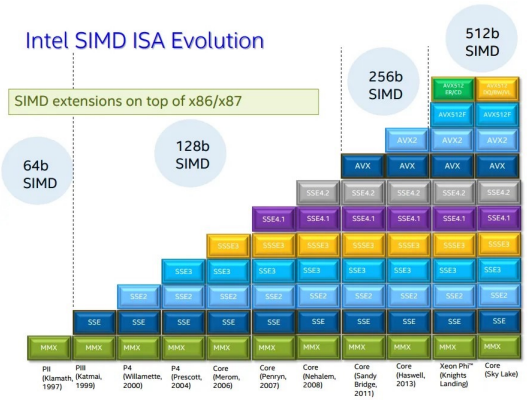
- **Vector Extensions or Intrinsics**
  - Vendor provided code extensions close to assembly level
  - Advantages: Provide performance and control
  - Disadvantages: Extremely verbose, generally not portable
- **Compiler Flags**
  - Advantages: No additional coding effort
  - Disadvantages: Vendor specific, performance and success compiler-dependent, problematic for complex code, almost no control on implementation
- **OpenMP SIMD**
  - Pragmas in OpenMP for implementing SIMD parallelism
  - Advantages: Portable, concise and easy to use
  - Disadvantages: Performance compiler-dependent, correctness left to programmer, no low-level control on implementation

# Vector Extensions (Intrinsics)

**SSE instructions**: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)

**AVX instructions**: 256-bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)

**AVX512**: 512-bit operations: 16x32 bits or 8x64 bits (16-wide float vectors)








# Vector Extensions (Intrinsics)

**SSE instructions**: 128-bit operations: 4x32 bits or 2x64 bits (4-wide float vectors)
**AVX instructions**: 256-bit operations: 8x32 bits or 4x64 bits (8-wide float vectors)
**AVX512**: 512-bit operations: 16x32 bits or 8x64 bits (16-wide float vectors)

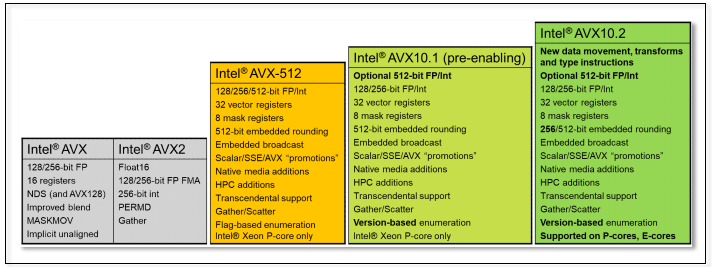



Figure 1-2. Intel® ISA Families and Features

The Converged Vector ISA: Intel® Advanced Vector Extensions 10
Technical Paper
July 2023 Revision 1.0





# Scalar program

```
void sinx(int N, int terms, float* x, float* result)
{
  for (int i=0; i<N; i++)
  {
    float value = x[i];
    float numer = x[i] * x[i] * x[i];
    int denom = 6; // 3!
    int sign = -1;
    for (int j=1; j<=terms; j++)
    {
      value += sign * numer / denom;
      numer *= x[i] * x[i];
      denom *= (2*j+2) * (2*j+3);
      sign *= -1;
    }
    result[i] = value;
  }
}
```





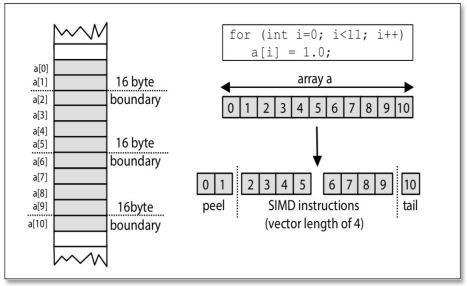
# Vector Program Using AVX2 Intrinsics

```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* result)
{
  float three_fact = 6;  // 3!
  for (int i=0; i<N; i+=8)
  {
    __m256 origx = _mm256_load_ps(&x[i]);
    __m256 value = origx;
    __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
    __m256 denom = _mm256_set1_ps(three_fact); float sign = -1;
    for (int j=1; j<=terms; j++)
    {
      // value += sign * numer / denom
      __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1_ps(sign), numer), denom);
      value = _mm256_add_ps(value, tmp);
      numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
      denom = _mm256_mul_ps(denom, _mm256_set1_ps((2*j+2) * (2*j+3)));
      sign *= -1;
    }
    _mm256_store_ps(&result[i], value);
  }
}
```

## Alignment

To get full benefit from SIMD the starting address of the vectors (arrays) needs to be aligned on a correct memory address boundary

- Starting array address in memory must be a multiple of the SIMD length
- No optimal alignment → compiler has to use *loop peeling*
- Loop peeling generates a loop peel and (usually) a loop tail that is treated separately
- The peel and tail section are typically not executed using SIMD instructions








# *Demo*

# Vector Program Using AVX2 Intrinsics





## Challenges for compiler vectorization

The compiler can identify operations suitable for vectorization (or you can give it a hint using OpenMP directives), but it must prove that it is a legal operation.

The following issues tend to prevent the compiler from vectorizing your code:

- Imprecise dependence information (potential pointer aliasing)
- Bad data layout and alignment
- Branching
- Calls to functions
- Loop bounds that are not multiples of the SIMD length





## The `simd` construct

- The OMP compiler may transform a loop marked with the `simd` construct into a SIMD loop.
- A *SIMD chunk* of iterations (equal to the SIMD length) is executed by a single thread
- Within a chunk each iteration is executed by a *SIMD lane*
- The following clauses are supported on the `simd` construct

**private** *(list)*
**lastprivate** *(list)*
**reduction** *(reduction-identifier : list)*
**collapse** *(n)*
**simdlen** *(length)*
**safelen** *(length)*
**linear** *(list[:linear-step])*
**aligned** *(list[:alignment])*

# The `simd` construct

- The OMP compiler may transform a loop marked with the `simd` construct into a SIMD loop.
- A *SIMD chunk* of iterations (equal to the SIMD length) is executed by a single thread
- Within a chunk each iteration is executed by a *SIMD lane*
- The following clauses are supported on the `simd` construct

```
# pragma omp simd [ clause_list ]
  for - loop
```

**private** *(list)*
**lastprivate** *(list)*
**reduction** *(reduction-identifier : list)*
**collapse** *(n)*
**simdlen** *(length)*
**safelen** *(length)*
**linear** *(list[:linear-step])*
**aligned** *(list[:alignment])*





# The `simd` construct

- If you use the `simd` construct, you are instructing the compiler to use SIMD instructions
- Where there is pointer aliasing your code will give incorrect results
- Where you do not provide the SIMD length through the `simdlen` clause, the compiler will select an appropriate vector length
- Similarly to the `for` construct the compiler will create a new instance of the loop variable *i* for each SIMD lane.

```
# pragma omp simd [ clause_list ]
  for - loop
```

```
1 void simd_loop(double *a, double *b, double *c, int n)
2 {
3   int i;
4
5   #pragma omp simd
6   for (i=0; i<n; i++)
7     a[i] = b[i] + c[i];
8 }
```





# The `simd` construct

- The `reduction` and `collapse` clauses works for SIMD loop as well!
- For each variable in the `reduction` list, a private instance is used during the execution of the SIMD loop, with all private instances being combined using the reduction operator
- When using `collapse` use the compiler commentary to check that vectorization was able to be performed

```
# pragma omp simd [ clause_list ]
  for - loop
```

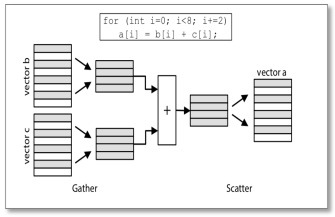
```
1 void simd_loop_collapse(double *r, double *b, double *c,
2                         int n, int m)
3 {
4   int i, j;
5   double t1;
6
7   t1 = 0.0;
8   #pragma omp simd reduction(+:t1) collapse(2)
9   for (i = 0; i<n; i++)
10      for (j = 0; j<m; j++)
11          t1 += func1(b[i], c[j]);
12  *r = t1;
13 }
```





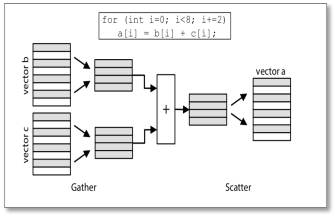
# Gather/scatter data in/out of vectors

- Scalar data elements are packed into vectors, operated on collectively as a vector by SIMD instructions, and then unpacked.
- When accessing the scalar data with stride 1 (assuming optimal alignment) a single SIMD load/store instruction can be used for packing/unpacking.
- When accessing the scalar data with stride > 1, the compiler will need to write code to perform gather and scatter operations for vector packing/unpacking




# Gather/scatter data in/out of vectors

- A gather operation reads scalar data elements from memory linearly but with a stride greater than one.
- A scatter operation writes the scalar data elements in a vector back to memory linearly with a stride greater than one.
- Some architectures support SIMD gather and scatter instruction.

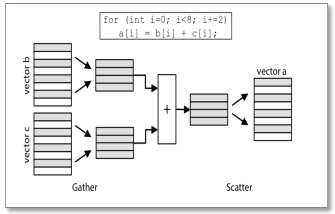







# Gather/scatter data in/out of vectors

- In general gather/scatter operations are much more expensive than single vector load/store instruction (best scenario)
- The next best scenario is when the access pattern is linear but with a stride that is greater than one and gather and scatter instructions may be used.
- The worst-case scenario is when no linear access pattern can be determined, and the scalar data elements must be individually packed into and unpacked from vectors.








# The `aligned` clause

```
#pragma omp simd aligned(list[:alignment])
  for-loop
```

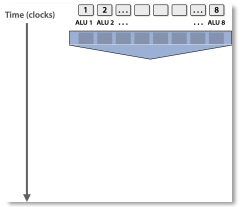
- For compiler optimizations if your arrays have been allocated at optimal alignment boundaries (e.g. `aligned alloc`, `memalign`, `posix memalign`)
- In C, a variable that appears in the clause must have an array or pointer type.
- In C++, a variable that appears in the clause must have array, pointer, reference to array, or reference to pointer type.





# Conditional Execution

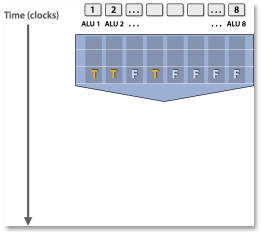
```
<unconditional code>
float x = A[i];
if (x > 0) {
  float tmp = exp(x,5.f);
  tmp *= kMyConst1;
  x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp; }
<resume unconditional code>
result[i] = x;
```




In SIMD, all processing elements execute the same instruction at the same time.

# Conditional Execution

```
<unconditional code>
float x = A[i];
if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp; }
<resume unconditional code>
result[i] = x;
```



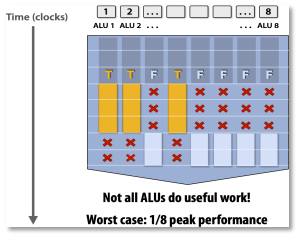

When a *conditional statement, e.g. an* `if` statement, causes the control flow to diverge (i.e., some processing elements need to execute one instruction while others need to execute a different instruction), this can lead to inefficiencies





# Conditional Execution

```
<unconditional code>
float x = A[i];
if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp; }
<resume unconditional code>
result[i] = x;
```




Many processing elements might have to remain idle, which can reduce the overall performance (throughput)





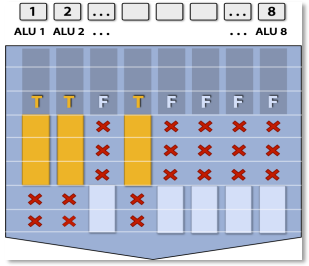
# Terminology

Instruction stream coherence ("**coheren"t** execution)

- Same instruction sequence applies to all elements operated upon simultaneously
- Coherent execution is necessary for efficient use of SIMD processing resources
- Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream

"**Divergent**" execution

- A lack of instruction stream coherence








# SIMD Execution on CPU and GPU

**Execution on CPU**

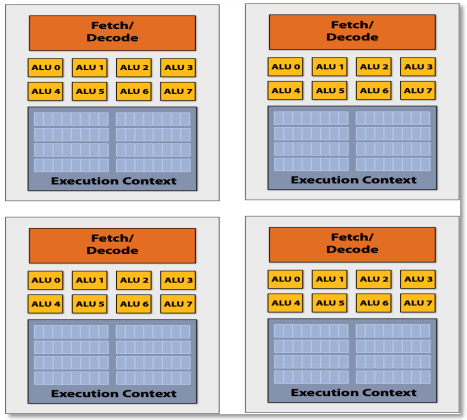
- **Explicit SIMD execution**: SIMD parallelization is performed at compile time. Instructions are generated by the *compiler* (e.g., AVX512 instructions)
  - Parallelism explicitly requested by programmer using intrinsics
  - Parallelism conveyed using parallel language semantics (e.g., `pragma omp simd`)
  - Parallelism inferred by dependency analysis of loops (hard problem, even best compilers are not great on arbitrary C/C++ code)

**Execution on GPU**

- **Implicit SIMD execution**: *Hardware* (not compiler) is responsible for simultaneously executing the same instruction from multiple instances on different data on SIMD ALUs
- Compiler generates a scalar binary (scalar instructions) but N instances of the program are always run together on the processor
- SIMD width of most modern GPUs is 32
- Divergence is a very big issue (divergent code might execute at 1/32 the peak capability of the machine!)

# SIMD Multi-Core

- 4 cores
- Each core has 8 SIMD ALUs
- = 32 operations



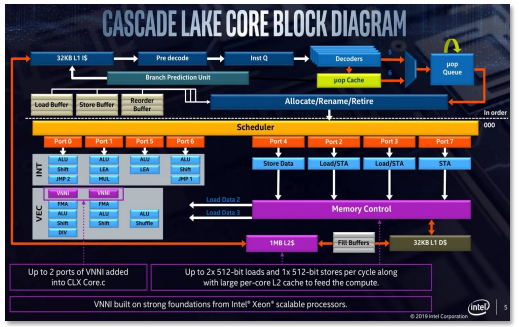





# SIMD Multi-Core

Cascade Lake CPUs on Gadi, Intel(R) Xeon(R) Platinum 8268 CPU @ 2.90GHz

- 24 cores, each core 2 × AVX512 units, equivalent to 16 × 64-bit ALUs
- Hyperthreading level 2 (enabled by default)



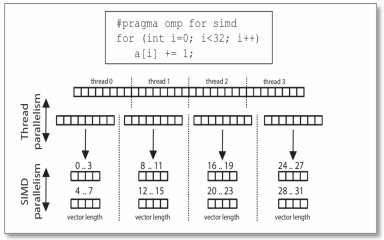





# The composite `for simd` construct

```
# pragma omp for simd [ clause_list ]
   for - loop
```

- Chunks of loop iterations are first distributed across the threads in a team according to the clauses on the `for` directive (e.g. `schedule`)
- Each chunk of loop iterations may be converted into SIMD loops in a way that is determined by any clauses that apply to the `simd` construct



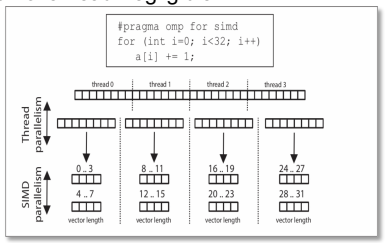





# The composite `for simd` construct

```
# pragma omp for simd [ clause_list ]
   for - loop
```

- There may be a significantly increased memory pressure due to the fact that at runtime a new copy of each private variable will be allocated per SIMD lane.
- Typically, the chunks performed with SIMD need to be large enough to make the loop peel/tail overhead negligible.



# The composite `for simd` construct

```
# pragma omp for simd [ clause_list ]
   for - loop
```

- Remember that you can use the `simd` schedule modifier.
- The modifier will adjust the chunk size according to the formula (*[chunk size/SIMD_length]* × *SIMD_width*)
- eg on AVX2 for *SIMD_length = 32bits, SIMD_width = 256 bits,* the chunk size will be 8
- This ensures that the size of chunk is at least as long as the SIMD length.

```
10 void func_2(float *a, float *b, int n)
11 {
12   #pragma omp for simd schedule(simd:static, 5)
13   for (int k=0; k<n; k++)
14   {
15     // do some work on a and b
16   }
17 }
```