OpenMP:
Part II

OpenMP Reference Material

- **Using OpenMP – The Next Step**, R. van der Pas, E. Stotzer, and C. Terboven, Chapter 1
- http://www.openmp.org/
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

The `for` Work-Sharing Directives

- The order in which threads execute is not predictable (OS scheduled).
- The way to map iterations to threads can be specified by the programmer (see later `schedule` clause).
- If the programmer does not specify the mapping between threads and iterations, the compiler decides which strategy to use.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

The `schedule` clause

- The `schedule` clause of the `for` directive specifies how iterations are mapped to threads

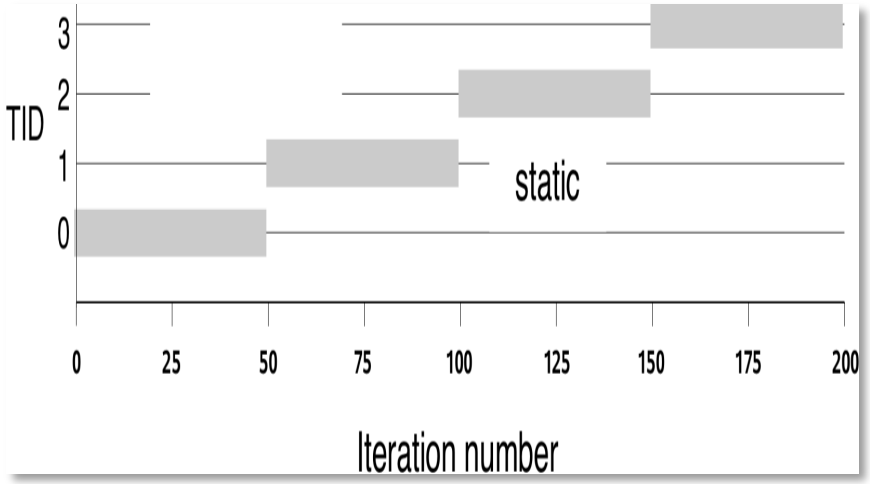
```
schedule(scheduling_clause[,chunk_size])
```

```
#pragma omp parallel for schedule ( scheduling_clause [, chunk_size ])
```

- The granularity of the workload distribution is a *chunk*, a continuous non-empty subset of the iteration space.

The `schedule` clause

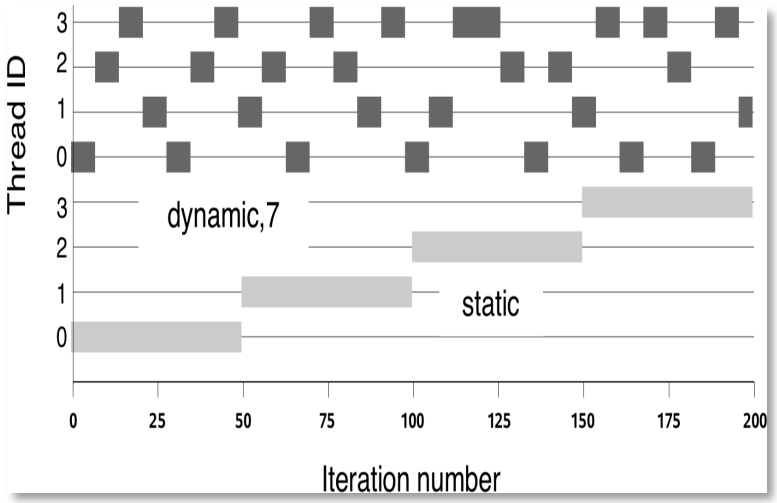
The most straightforward schedule is
`schedule(static[,chunk_size])`



- Splits the iteration space into chunks of size `chunk_size` and allocates to threads statically in a round-robin fashion
- No specification implies the number of chunks equals the number of threads Iterations are assigned in the order of the thread number and the last chunk may have a smaller number of iterations.
- It has the lowest overhead and is the default setting for many OpenMP compilers.
- **Potential load imbalance** - if iterations have different workloads, consider dynamic scheduling (schedule(dynamic)).

The `schedule` clause

The schedule `(dynamic[,chunk_size])` enables basic dynamic load balancing

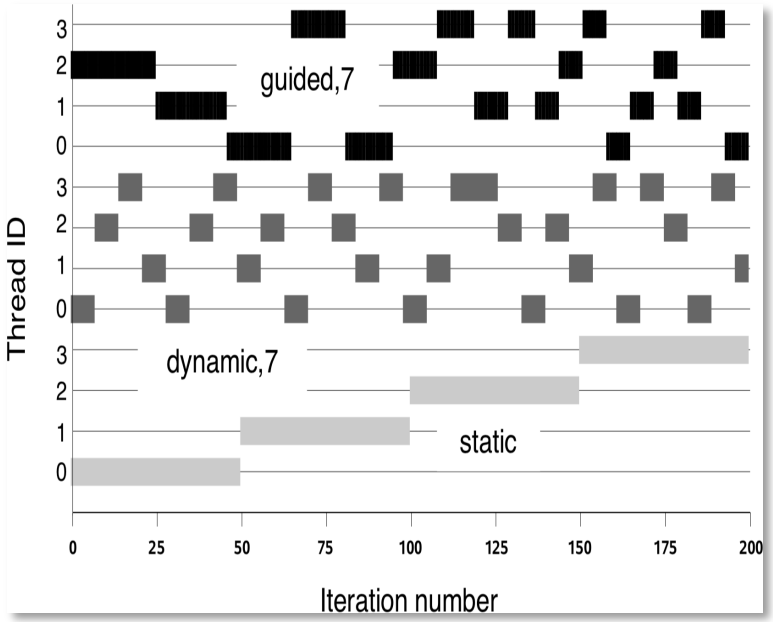


- iteration space split into `chunk_size` blocks that are scheduled dynamically as they complete the work on the current chunk, threads request additional chunks
- if not specified `chunk_size` is set by default to 1
- Do you think there is any disadvantage in this schedule?

The `schedule` clause

`schedule(runtime)`

- The choice of the optimal schedule may depend on the problem size
- The schedule and (optional) chunk size are set through the `OMP_SCHEDULE` environment variable



Loop Schedules

Example of loop schedules in OpenMP

- 20 iterations by three threads
- Default chunksize for `dynamic` and `guided` is one
- Note that only the `static` schedule guarantee that the distribution of chunks stays the same between runs

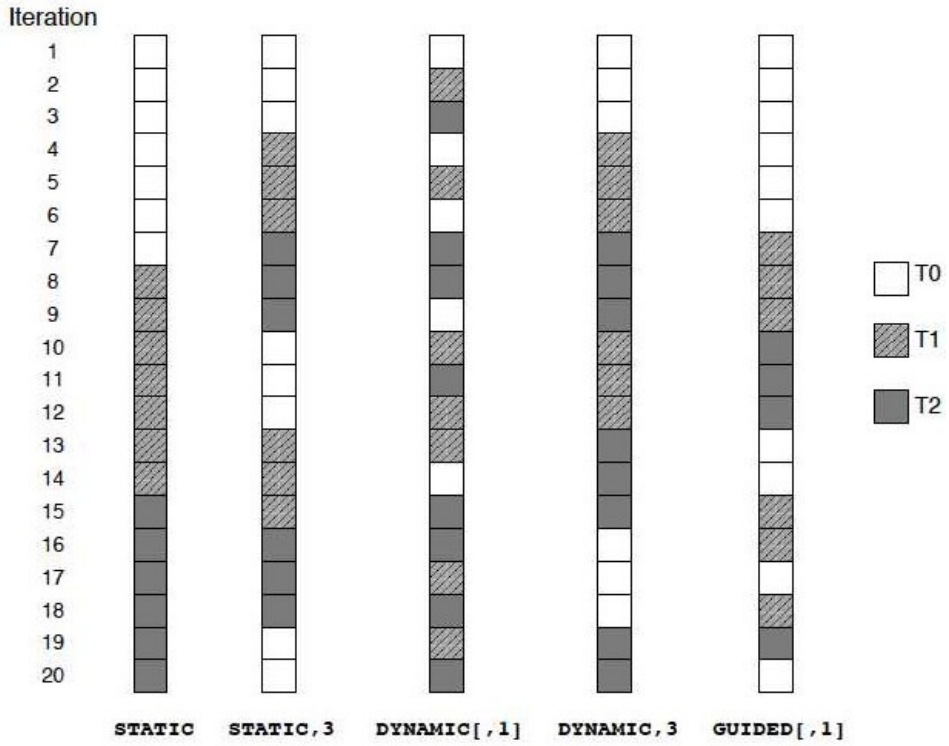


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for DYNAMIC and GUIDED is one. If a chunksize is specified, the last chunk may be shorter. Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

Introduction to High Performance Computing for Scientists and Engineers, Hager and Wellein, Figure 6.2

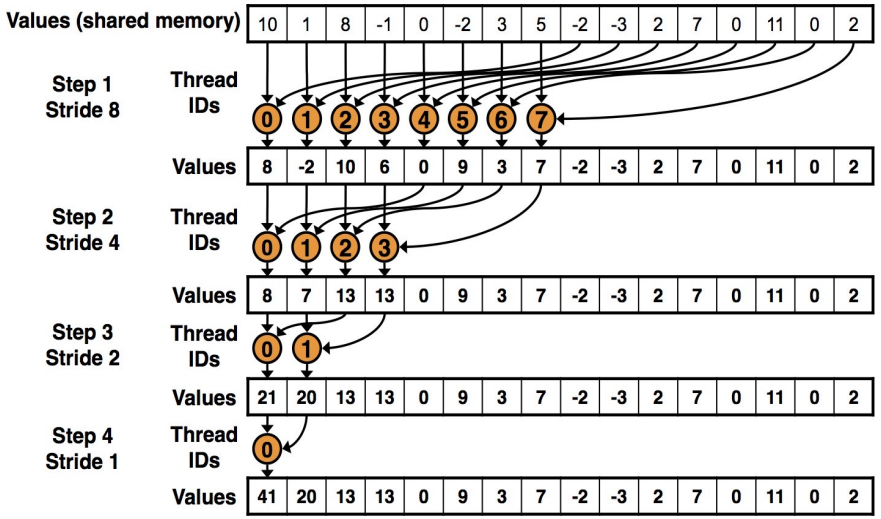
The `reduction` clause

```
sum = 0;
partial_sum [ NTHREADS ] = {0};
#pragma omp parallel for default (none) shared ( n, a, partial_sum )
 for (i = 0; i < n; ++i) {
  partial_sum [ omp_get_thread_num ()] += a[ i]
 } /* End of parallel region */
 for (i = 0; i < NTHREADS ; ++i){
   sum += partial_sum [ i];
}
```

The `reduction` clause

```
sum = 0;
partial_sum [ NTHREADS ] = {0};
#pragma omp parallel for default (none) shared ( n, a, partial_sum )
 for (i = 0; i < n; ++i) {
  partial_sum [ omp_get_thread_num ()] += a[ i]
 } /* End of parallel region */
 for (i = 0; i < NTHREADS ; ++i){
   sum += partial_sum [ i];
}
```

Is there a better algorithm?



The `reduction` clause

```
#pragma omp parallel for default (none) shared ( n, a) reduction (+: sum )
  for (i = 0; i < n; ++i) {
     sum += a[ i]
  } /* End of parallel reduction */
```

- `reduction(+:sum)`
  - Implements a reduction operation - The clause ensures that each thread maintains a private copy of the reduction variable, and at the end of the parallel region, the private copies are combined into a single result.
  - It is not necessary to specify the reduction variable as shared
  - The order in which thread-specific values are combined is unspecified
  - The operators supported are `+, ¡, *, &, , ˆ, && and`

- Some constraints
  - Aggregate types, pointer types and references types are not supported
  - A reduction variable must not be `const`
  - No overloaded operators with respect to the variable that appears in the clause

Nested Parallelism

```
#pragma omp parallel for num_threads (2)
  for (i = 0; i < Ni; i ++}
     #pragma omp parallel for num_threads (2)
     for (j = 0; j < Nj; j ++) {
```

- By default, the inner loop is serialized and run by one thread
- To enable multiple threads in nested parallel loops requires environment variable `OMP_NESTED` to be `TRUE`
- You can also control the maximum level of nesting through `OMP_MAX_ACTIVE_LEVELS`
- Each thread from the first `parallel for` will spawn a new team.
- Note - the use of synchronization constructions in nested parallel sections requires care (see OpenMP specs, *e.g.* there is no restriction on synchronizing threads across different teams!).

Nested Parallelism

```
#pragma omp parallel for collapse (2)
  for (i = 0; i < Ni; i ++) {
    for (j = 0; j < Nj; j ++) { …
```

A disadvantage of nested parallelism is parallel overhead

- `Collapse` is usually better way of parallelizing nested loops
- The `collapse` clause turns the two loops into a combined single loop.
- Threads work on a larger chunk of work, which can improve parallel efficiency.
- By collapsing loops, you reduce the overhead of managing multiple parallel loops.

The `barrier` directive

```
#pragma omp barrier
/* structured block */
```

- `barrier`: Each thread waits at the `barrier` until all threads arrive

Restrictions:

- Each barrier must be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for each thread in the team.

The `ordered` directive

```
#pragma omp ordered
/* structured block */
```

- `ordered`: threads execute the structured block in sequential order.

```
cumul_sum [0] = list [0];
#pragma omp parallel for ordered shared ( cumul_sum , list
, n)
for (i =1; i<n; i ++) {
 /* other processing on list [ i] if required */
 #pragma omp ordered
 {
  cumul_sum [ i] = cumul_sum [ i -1] + list [ i];
 }
}
```

- The first thread that encounters this directive enters the structured block without waiting.
- Any subsequenRestrictions: An `ordered`clause must be added to the parallel region in which this construct appears.
- threads wait until previous threads have completed the block execution.

The `atomic` directive

```
#pragma omp atomic [seq_cst[,]] atomic-clause [[,] seq_cst]
/* single expression statement */
#pragma omp atomic [seq_cst[,]]
/* single expression statement */
#pragma omp atomic [seq_cst[,]] capture [[,] seq_cst]
/* structured block *
```

- `atomic`: This is a hint for the compiler to use low-level atomic instructions if available.
- Atomic clauses
  - `read`: performs an atomic read of the input variable and stores it into the output. This is guaranteed regardless of the size of the variable.
  - `write`: the output variable is written atomically. This is guaranteed regardless of the size of the variable.

```
x = expr ; // x is written atomically
```

The `atomic` directive

Atomic clauses

- `update`: causes an atomic update of the location designated by x using the designated operator or intrinsic.

```
x++;
++x; x- -;
-- x;
x binop = expr ;
x = x binop expr ; x = expr binop x;
```

The following rules also apply

- The evaluation of `expr` need not be atomic with respect to the read or write of the location designated by x.
- No task scheduling points occur between the read and the write of the location designated by x.
- Binop (binary operators) is one of the following operators `+, * ,-, & , ^, <<, >>.`

The `atomic` directive

Atomic clauses

- `capture`: causes an atomic update of the location designated by x using the designated operator or intrinsic while also capturing the original or final value of the location (v) designated by x with respect to the atomic update.

```
v = x ++;
v = ++x;
v = x - -;
v = -- x;
v = x binop = expr;
v = x = x binop expr;
v = x = expr binop x;
```

The following rules also apply

- Only the read and write of the location designated by x are performed mutually atomically.
- The evaluation of expr, and the write to the location designated by v do not need to be atomic with respect to the read or write of the location designated by x.
- No task scheduling points occur between the read and the write of the location designated by x.

```
{ v = x; x += n; } // atomically update x, but capture original value in v
```

OpenMP: Library Functions and Environment Variables

Internal Control Variables

The OpenMP standard defines some internal control variables (ICVs) controlled by the implementations that govern the behaviour of a program at run time. Here are some of them:

- `nthreads-var` – stores the number of threads requested for the execution of future parallel regions.
- `dyn-var` – control whether dynamic adjustment of the number of threads to be used in future parallel regions is enabled.
- `nest-var` – controls whether nested parallelism is enabled for future parallel regions.
- `run-sched-var` – stores scheduling information to be used for loop regions using the runtime schedule.
- `def-sched-var` – stores implementation-dependent scheduling information to be used for loop regions.

These variables cannot be accessed directly, but via either library functions or environment variables.

Library Functions

Defined in header file

```
#include <omp.h>
```

Controlling threads and processors

```
void omp_set_num_threads(int num_threads)
int omp_get_num_threads()
int omp_get_max_threads() // Number of threads used in the next parallel region
int omp_get_thread_num()
int omp_get_num_procs() // Returns the number of processor cores available
int omp_in_parallel() // Check if within a parallel region
```

Controlling thread creation

```
void omp_set_dynamic(int dynamic_threads) // Enable/disable dynamic thread adjustment
int omp_get_dynamic() // Check whether dynamic thread adjustment is enabled
void omp_set_nested(int nested) int omp_get_nested()
void omp_set_dynamic(int dynamic_threads) // Enable/disable dynamic thread adjustment
int omp_get_dynamic() // Check whether dynamic thread adjustment is enabled
void omp_set_nested(int nested) int omp_get_nested()
```

OpenMP Environment Variables

`OMP_NUM_THREADS`: default number of threads entering parallel region

`OMP_DYNAMIC`: if `TRUE` it permits the number of threads to change during execution, in order to optimize system resources

`OMP_NESTED`: if `TRUE` it permits nested parallel regions

`OMP_SCHEDULE`: determines scheduling for loops that are defined to have `runtime` scheduling

```
export OMP_SCHEDULE =" static ,4 "
export OMP_SCHEDULE =" dynamic "
export OMP_SCHEDULE =" guided "
```

OpenMP Environment Variables

`OMP_DISPLAY_ENV`: this is a very useful variable to verify all the settings. If set to `true` or `verbose` all the relevant environment variables are printed at the beginning of the program.

`OMP_STACKSIZE`: this variable allows to increase the default stack size allocated to each thread.

- The syntax supports a case-insensitive unit qualifier that is appended to the number: `B` for bytes, `K` for 1024 bytes, `M` for 1024 Kbytes, and `G` for 1024 Mbytes.

OpenMP and Pthreads

	OpenMP	Pthreads
Definition	Specification for compiler directives, library routines, and environment variables.	POSIX standard for libraries, providing low-level thread management.
Parallelism Model	**Shared Memory**: Utilizes multiple threads within a single process.	**Shared Memory**: Also operates within a single process, but with more manual control over threads.
Communication	Implicit (compiler handles thread synchronization).	Requires explicit handling of thread communication using functions like pthread_create and pthread_join.
Ease of Use	Easier to program and debug due to directives.	Requires more manual management of threads and synchronization.
Scalability	Limited scalability within a single node.	Scalable to multiple cores within a single machine.
Use Cases	Best for parallelizing loops and simple tasks within a single program.	Suitable for applications that need fine-grained control over threads and synchronization.

OpenMP and Pthreads

- OpenMP removes the need for a programmer to initialize task attributes, set up arguments to threads, partition iteration spaces, etc.
- OpenMP code can closely resemble serial code – (verification)
- OpenMP users require availability of an OpenMP compiler
  - performance dependent on quality of compiler — hardly a problem today
- Well-engineered OpenMP code causes no loss of performance with respect to lower-level APIs.
- Pthreads has a lower-level API that is slightly more flexible and rich, (e.g. condition waits, locks of different types etc) but also more error prone
- Pthreads is library based and not compiler-based
- OpenMP is now the *de facto* standard in High-Performance Computing

OpenMP and MPI

	OpenMP	MPI
Definition	Specification for compiler directives, library routines, and environment variables.	Library specification for message-passing, proposed as a standard by a committee of vendors, implementors, and users.
Parallelism Model	**Shared Memory**: Utilizes multiple threads within a single process.	**Distributed Memory**: Operates across a network of distributed nodes.
Communication	Implicit (compiler handles thread synchronization).	Explicit (programmer manages message passing using API calls like MPI_Send and MPI_Recv).
Architecture	Suitable for multi-core processors.	Works on both shared-memory and distributed-memory architectures.
Ease of Use	Easier to program and debug due to directives.	Requires more explicit handling of communication and synchronization.
Scalability	Limited to scalability within a single node.	Scalable to large clusters and supercomputers.
Use Cases	Best for parallelizing loops and simple tasks within a single program.	Ideal for distributed computing, large-scale simulations, and complex parallel applications.

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

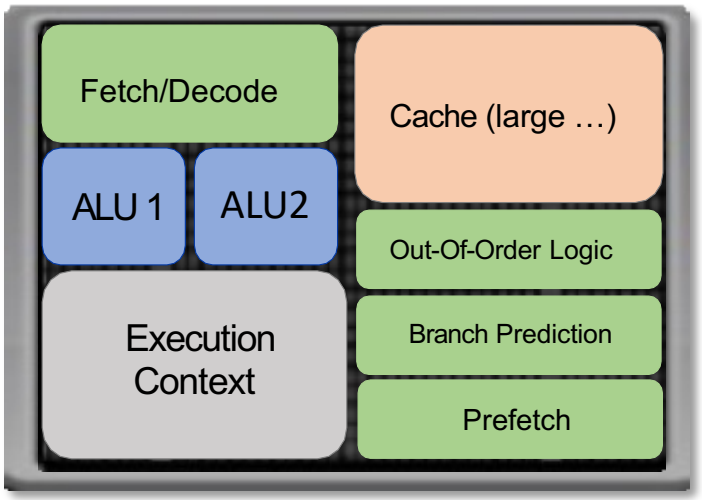
Reference Material

- *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**

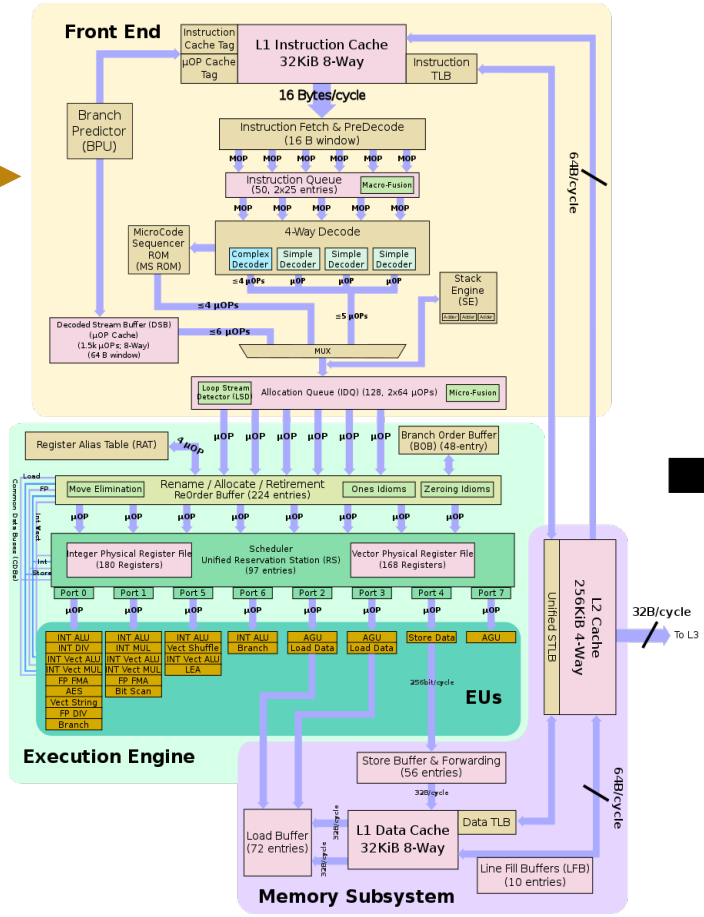


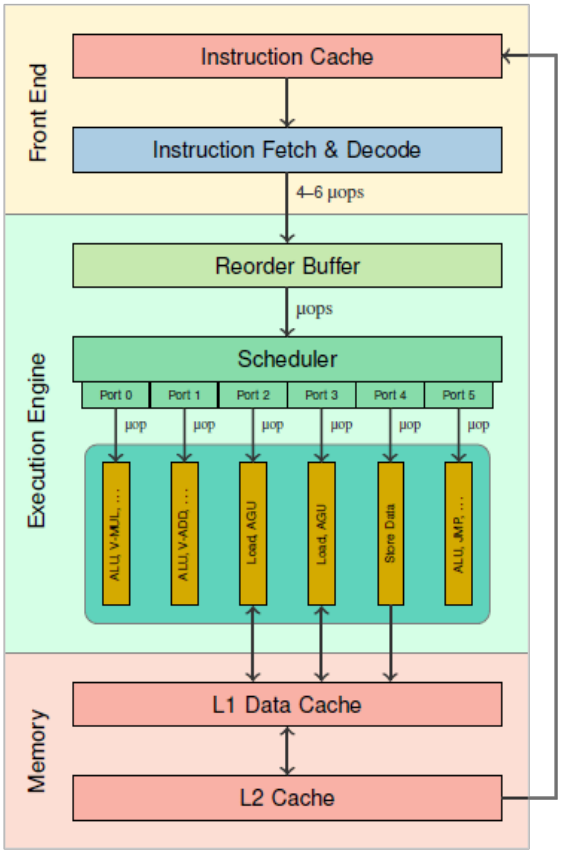
A real single core architecture

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

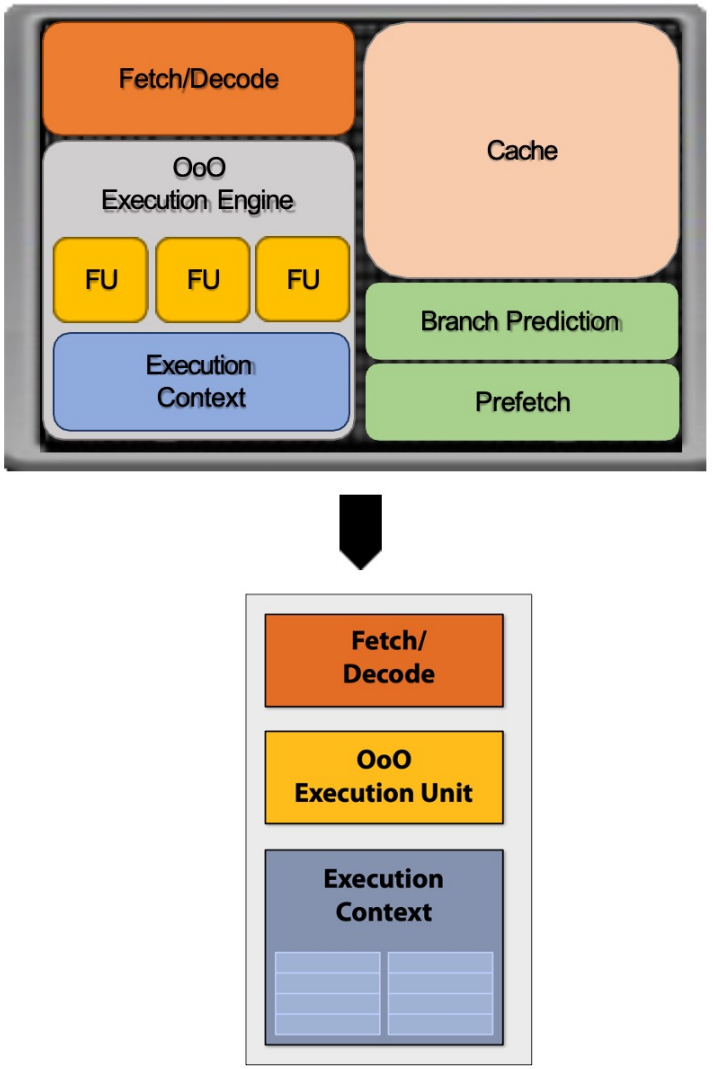
The execution unit is the part of the CPU that actually carries out the instructions





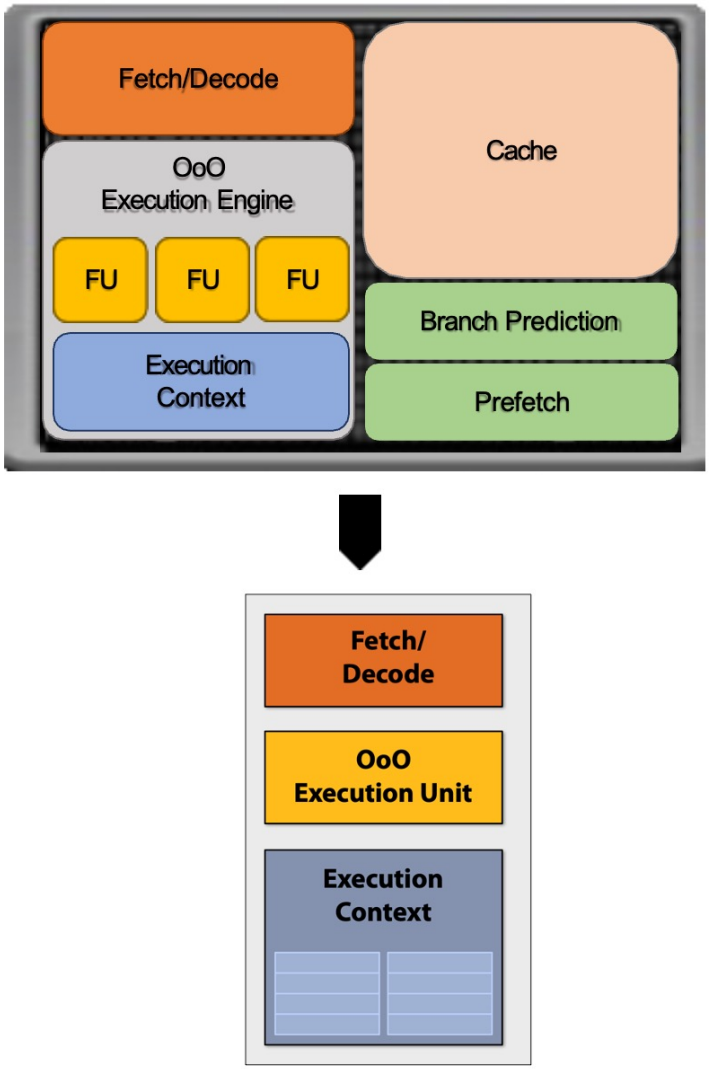
More schematic single core architecture

- 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
- We can leave the rest of the logic, Branch Predictor, Caches, Prefetching out of our representation



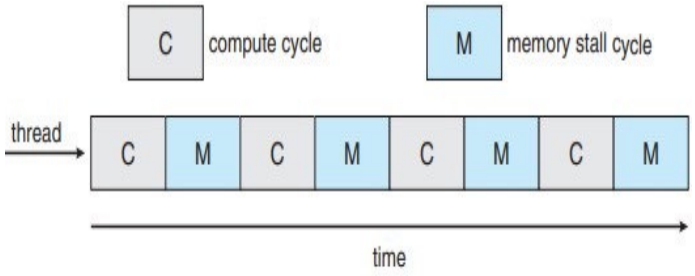
Under utilisation of ILP

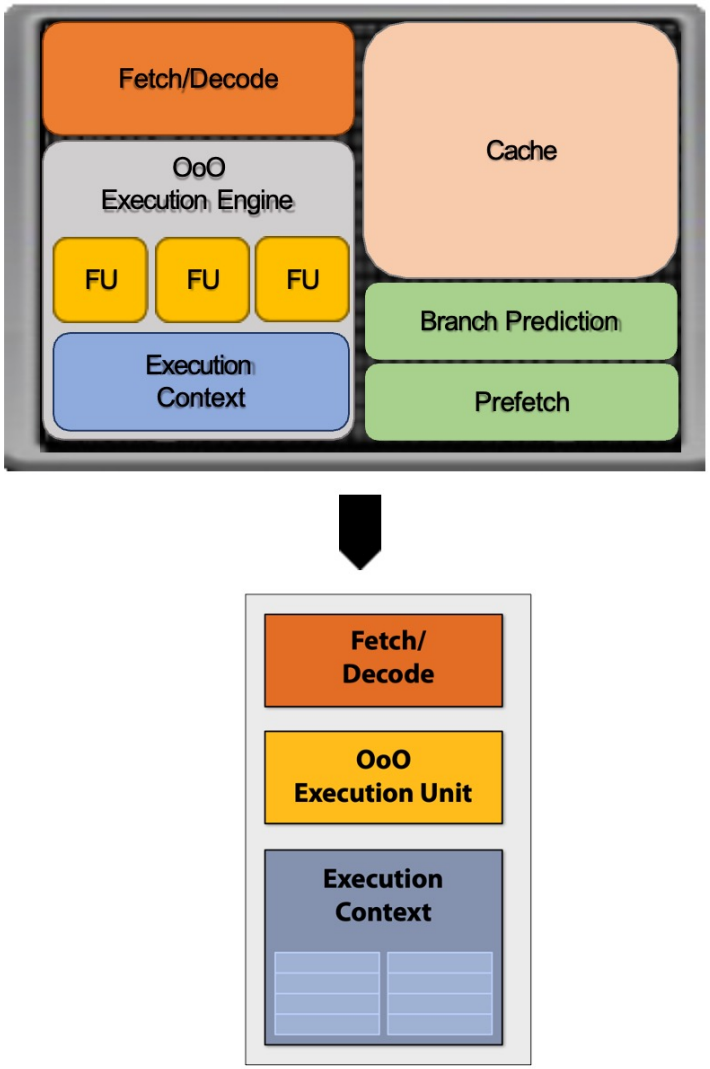
What are the main causes of under utilisation of ILP on this single-core architecture?



Under utilisation of ILP

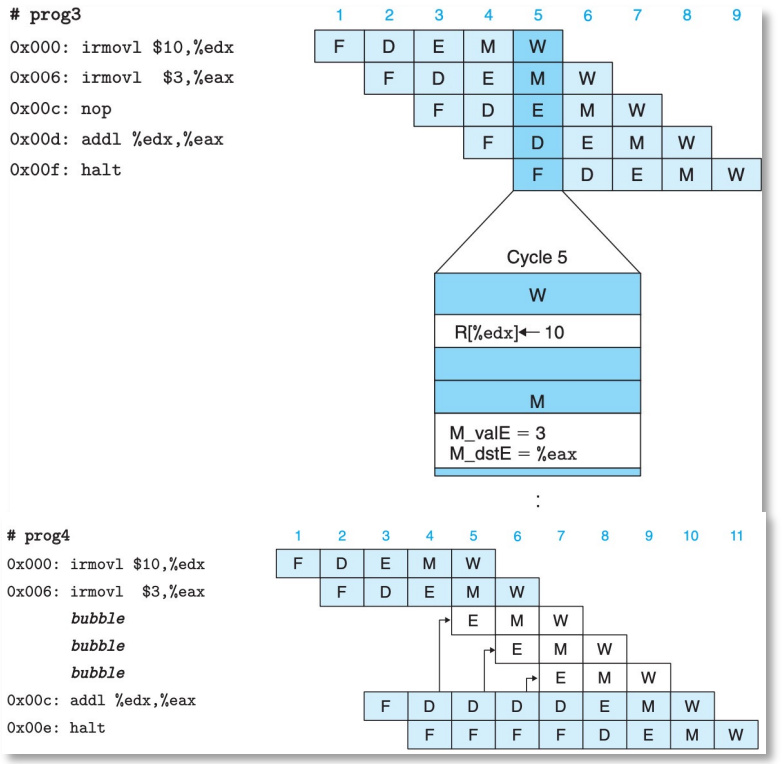
- The main causes of under utilisation of ILP on this single-core architecture: -
  - Branch misprediction
  - Bad instruction mix → cannot feed all replicates of the functional units
  - Thread stalls due to dependencies → thread cannot execute as it is waiting for operands to be fetched or dependencies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





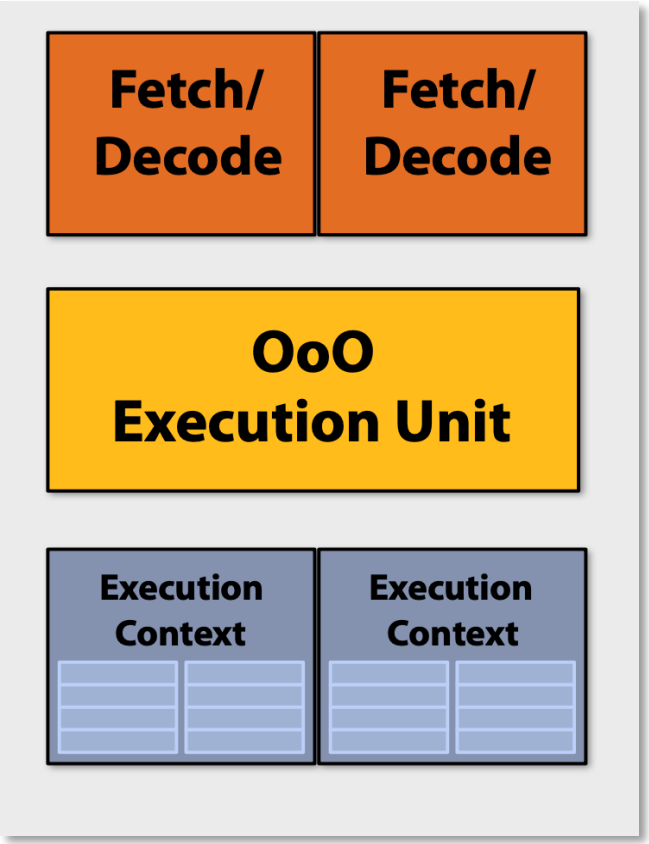
Under utilisation of ILP

- ***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



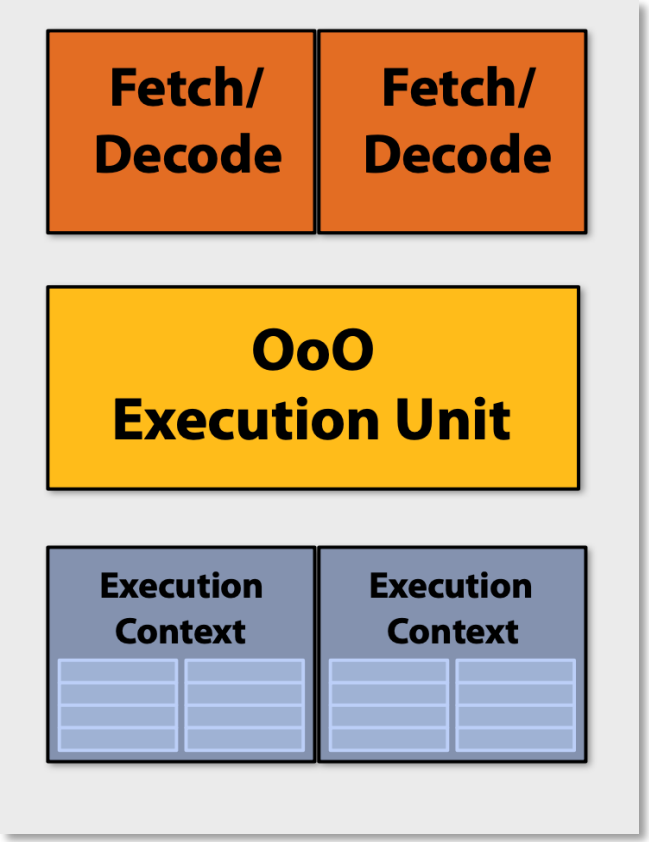
Hyperthreading (Simultaneous Multi-Threading)

- 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?



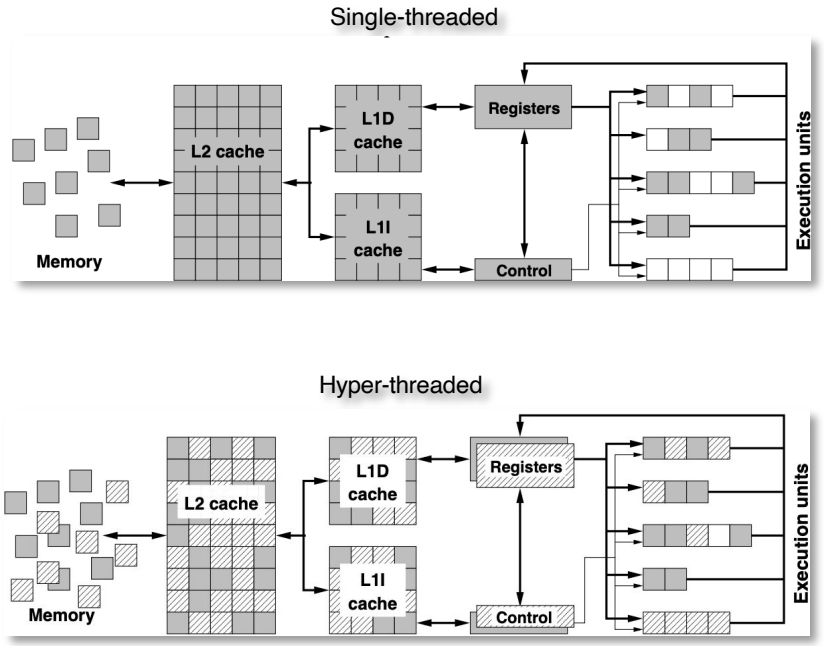
Hyperthreading (Simultaneous Multi-Threading)

- 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*



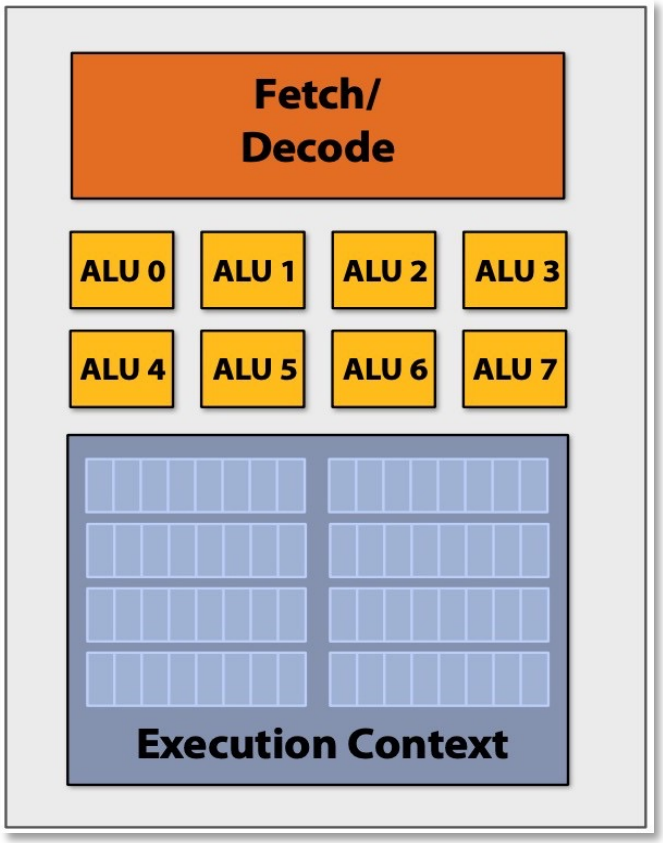
Hyperthreading: A More Detailed View

- 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)

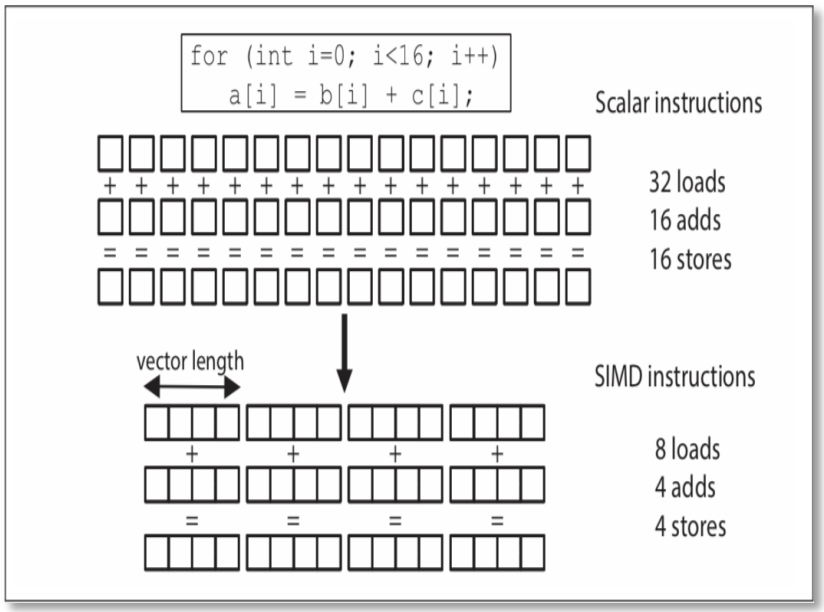
- Idea “2”: Amortize cost/complexity of managing an instruction stream across many ALUs
- Single Instruction Multiple Data (SIMD) processing
- Fetch and decode one instruction
- Same instruction broadcast and executed in parallel on ALUs operating on different data elements
- Execution context must be 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 with 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

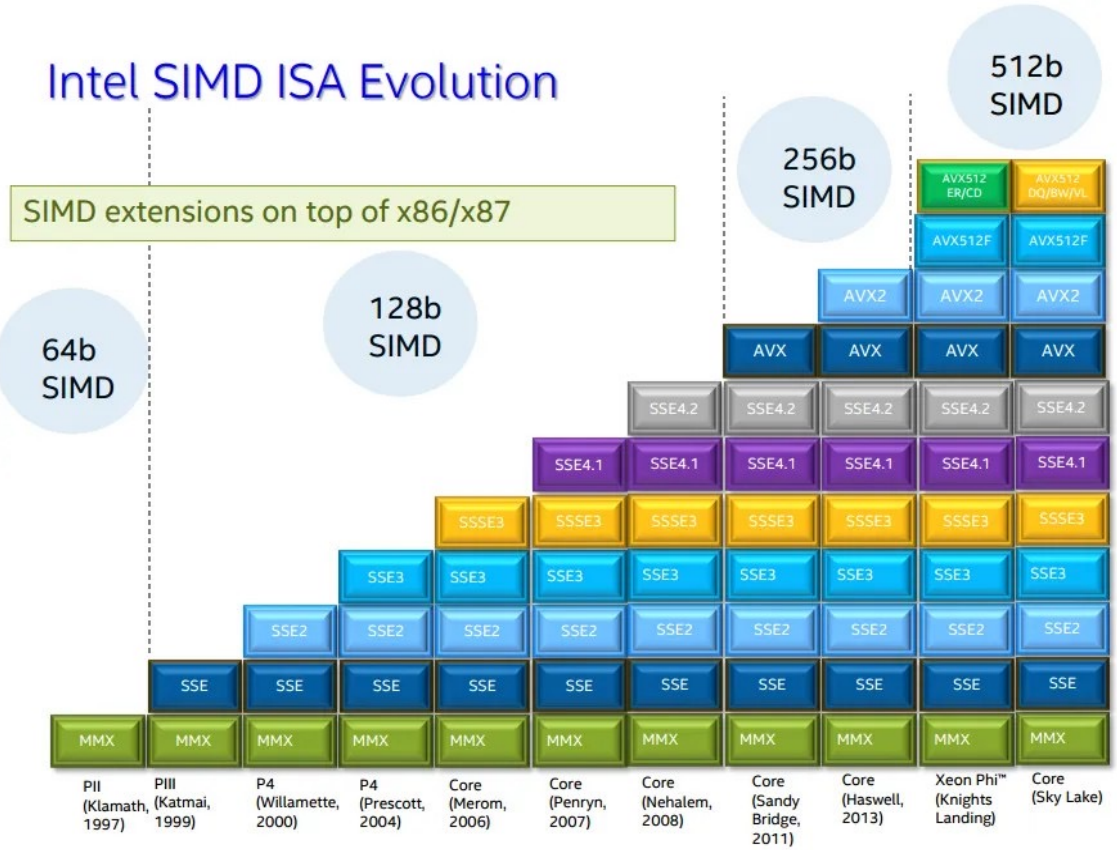
- **OpenMP SIMD**
  - Pragmas in OpenMP for implementing SIMD parallelism
  - *Advantages*: Portable, concise and easy to use
  - *Disadvantages*: Performance compiler-dependent, correctness left to programmer, no low-level control on implementation

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)

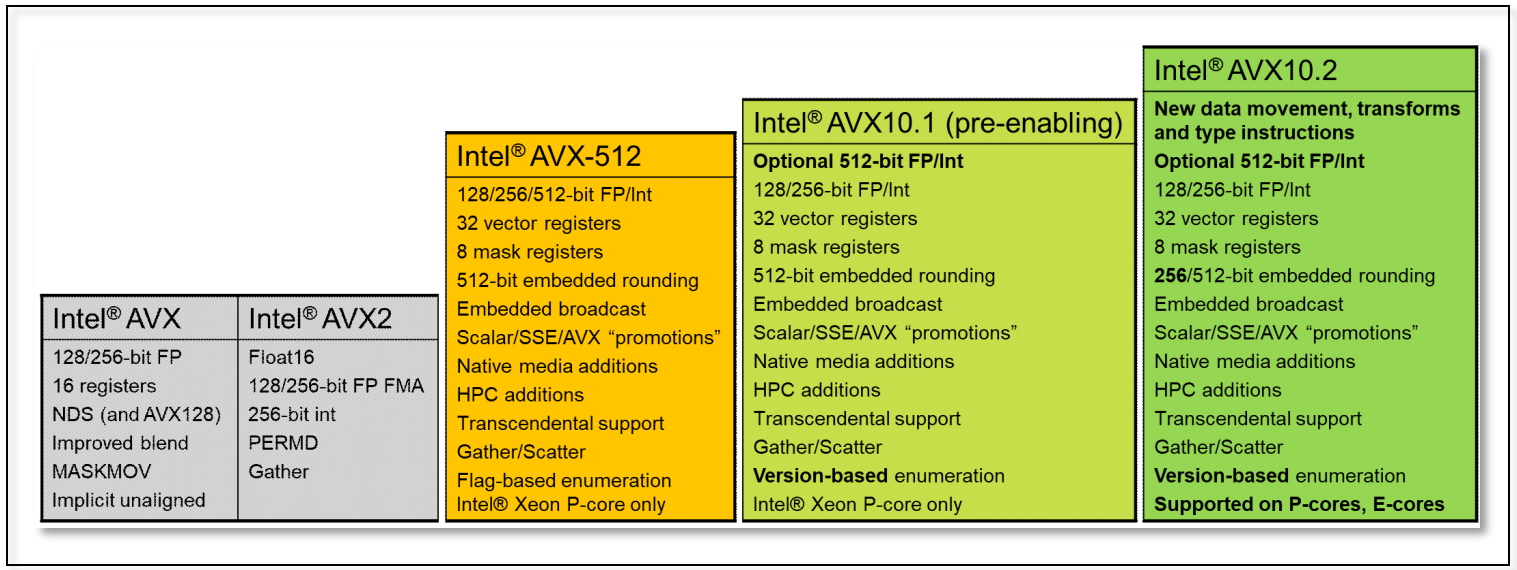


Figure 1-2. Intel® ISA Families and Features

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;
  }
}
```

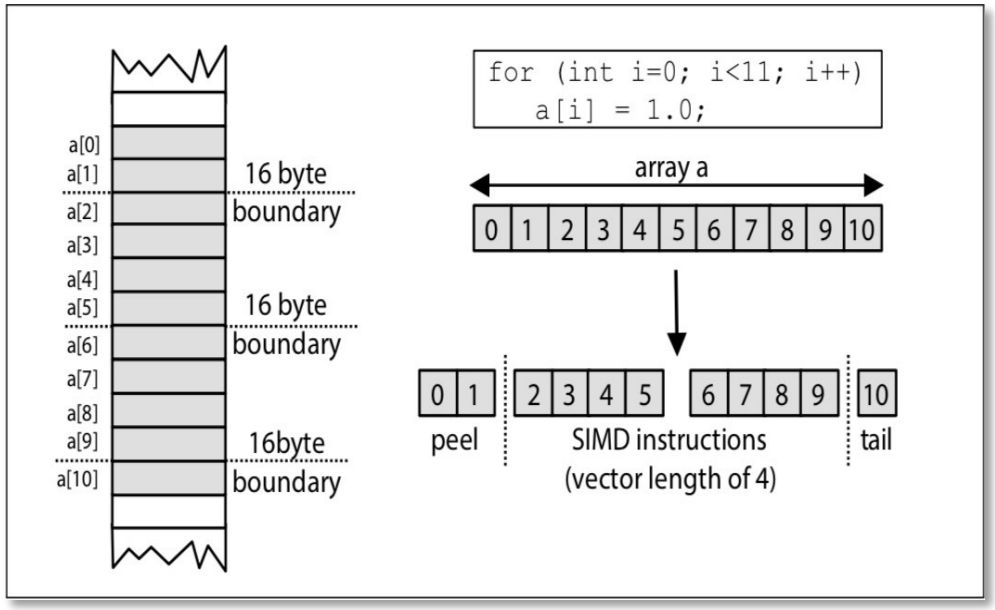
Vector Program Using AVX2 Intrinsics

```
#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_set1ps(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_set1ps(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_set1ps((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 OpenMP 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 OpenMP 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 it 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

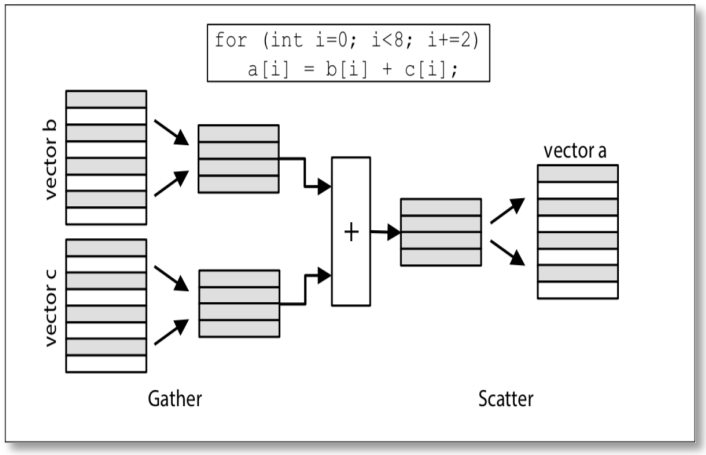
- The `reduction` and `collapse` clauses works for SIMD loop as well!
- For each variable in the `reduction` list, a private instance is used during the execution of the SIMD loop, with all private instances being combined using the reduction operator
- When using `collapse` use the compiler commentary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

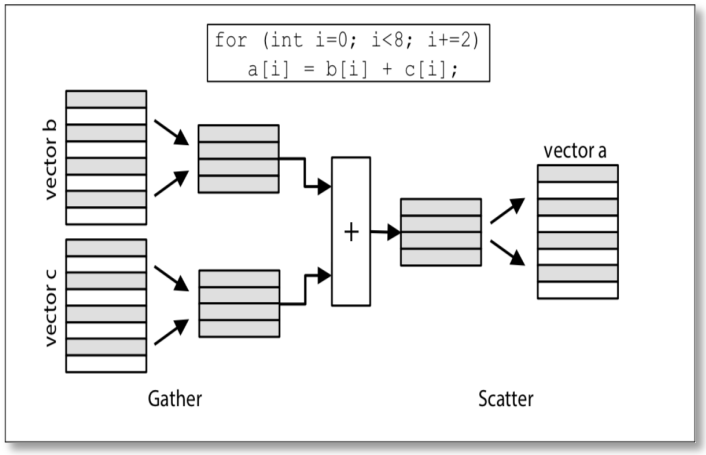
Gather/scatter data in/out of vectors

- 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



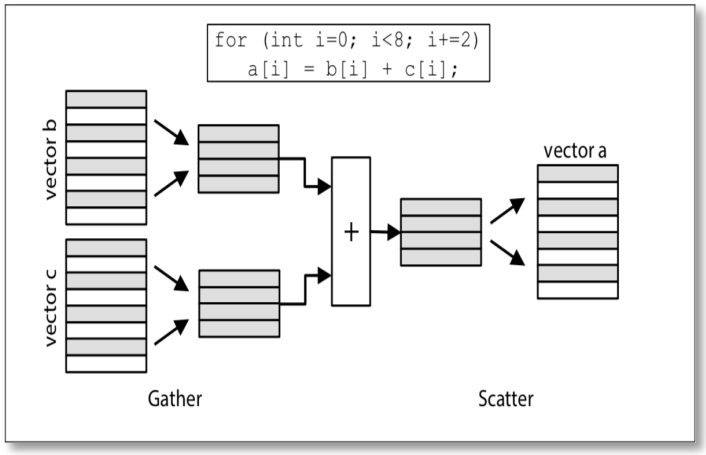
Gather/scatter data in/out of vectors

- 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.



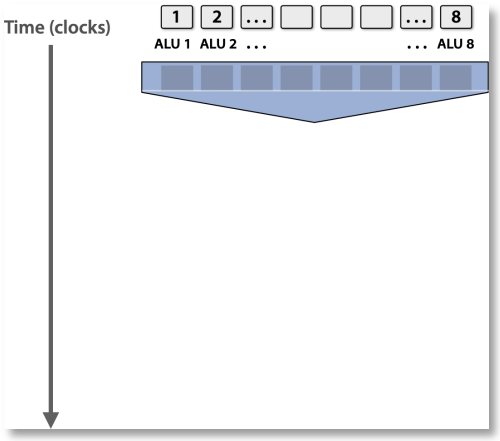
The `aligned` clause

```
#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

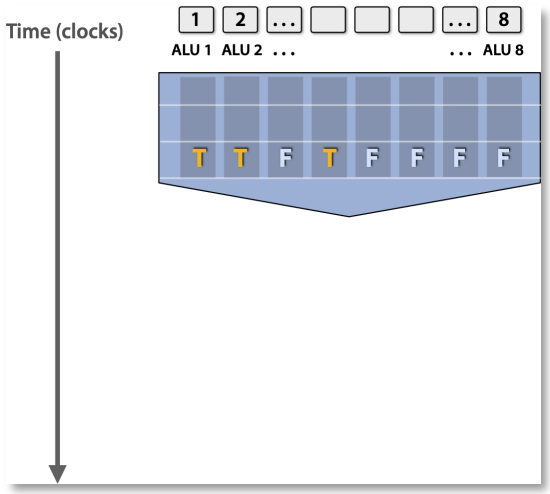
```
<unconditional code>
float x = A[i];
if (x > 0) {
  float tmp = exp(x,5.f);
  tmp *= kMyConst1;
  x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp; }
<resume unconditional code>
result[i] = x;
```



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

Conditional Execution

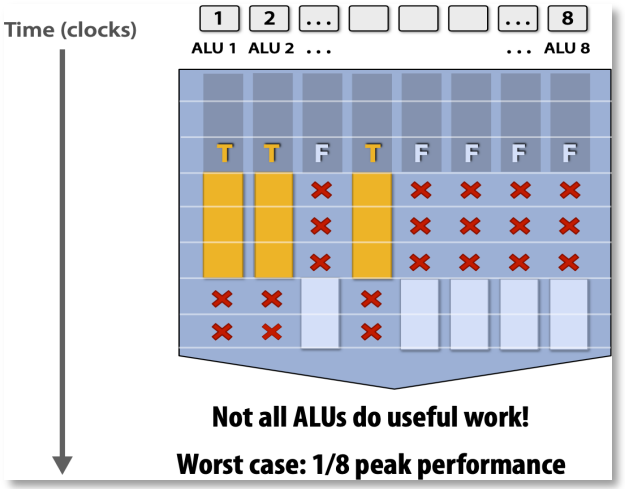
```
<unconditional code>
float x = A[i];
if (x > 0) {
  float tmp = exp(x,5.f);
  tmp *= kMyConst1;
  x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp; }
<resume unconditional code>
result[i] = x;
```



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;
```

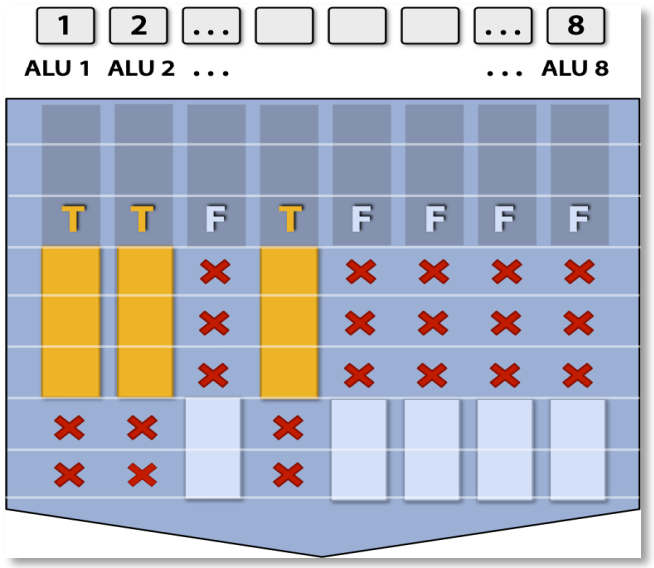


Many processing elements might have to remain idle, which can reduce the overall performance (throughput)

Terminology

Instruction stream coherence (“**coherent**” 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

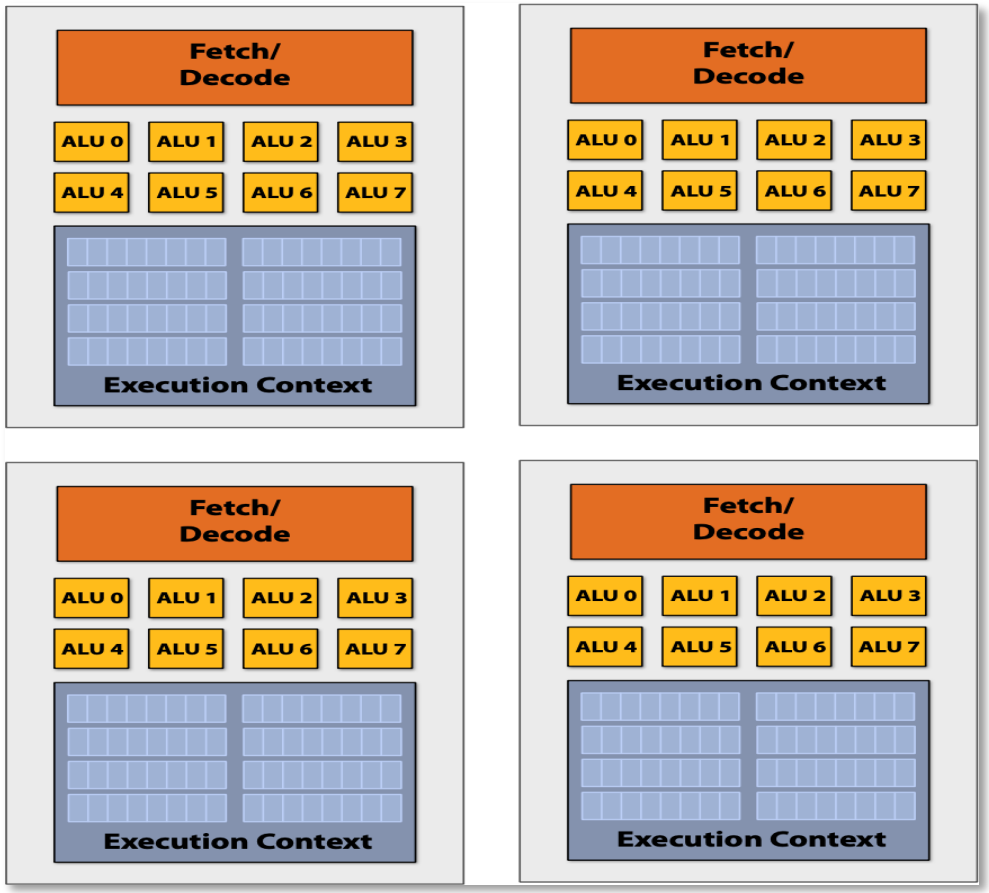
- **Explicit SIMD execution**: SIMD parallelization is performed at compile time. Instructions are generated by the *compiler* (e.g., AVX512 instructions)
  - Parallelism explicitly requested by programmer using intrinsics
  - Parallelism conveyed using parallel language semantics (e.g., `pragma omp 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

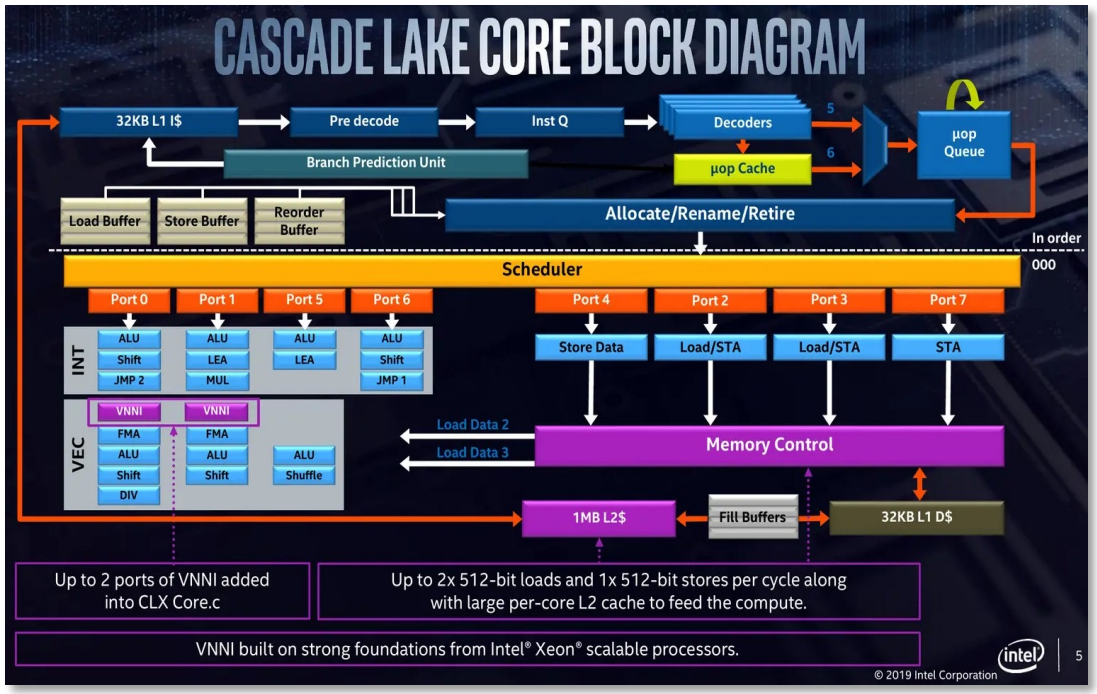
- 4 cores
- Each core has 8 SIMD ALUs
- = 32 operations



SIMD Multi-Core

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

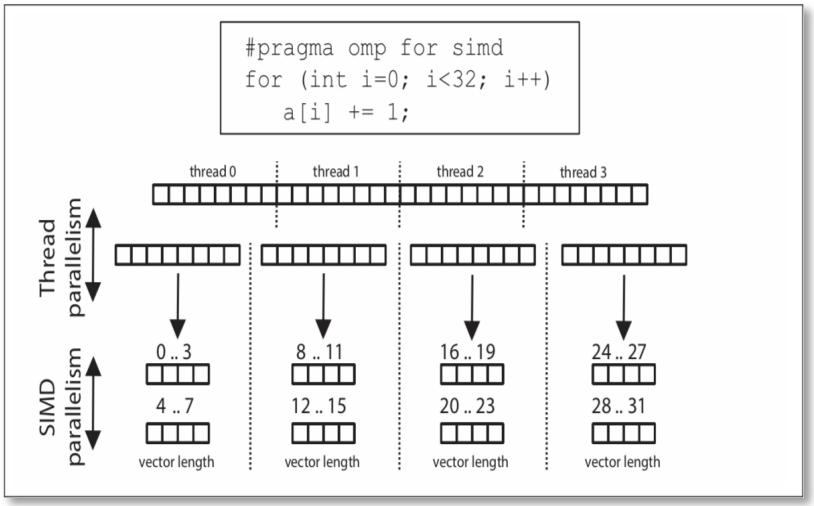
- 24 cores, each core 2 × AVX512 units, equivalent to 16 × 64-bit ALUs
- Hyperthreading level 2 (enabled by default)



The composite `for simd` construct

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

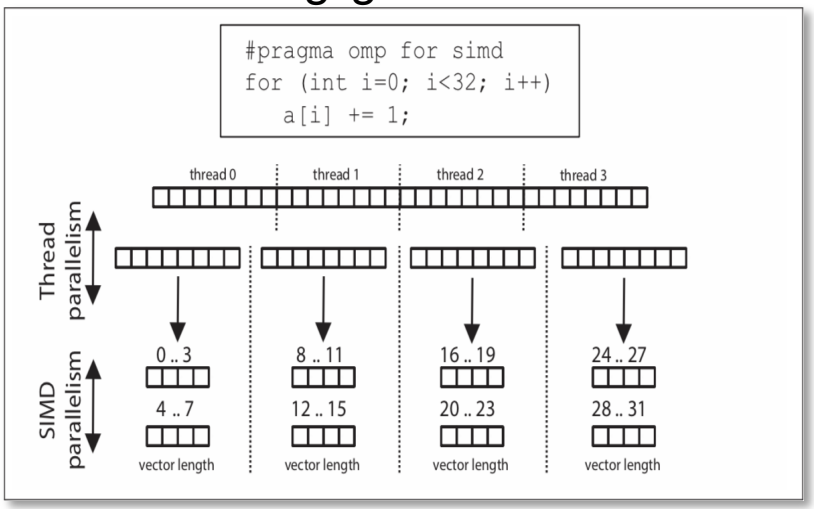
- Chunks of loop iterations are first distributed across the threads in a team according to the clauses on the `for` directive (e.g. `schedule`)
- Each chunk of loop iterations may be converted into SIMD loops in a way that is determined by any clauses that apply to the `simd` construct



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

SIMD Chunk Size = Size of Data Element (in bits) / SIMD Register Width (in bits)

eg on AVX2 for *data element = 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 }
```