CS257 Advanced Computer Architecture

Topic 6: Compiler Optimisations and Parallelism

Matthew Leeke and Graham Martin

{matt,grm}@dcs.warwick.ac.uk

Department of Computer Science

University of Warwick

WARWICK

Topics - Optimisations

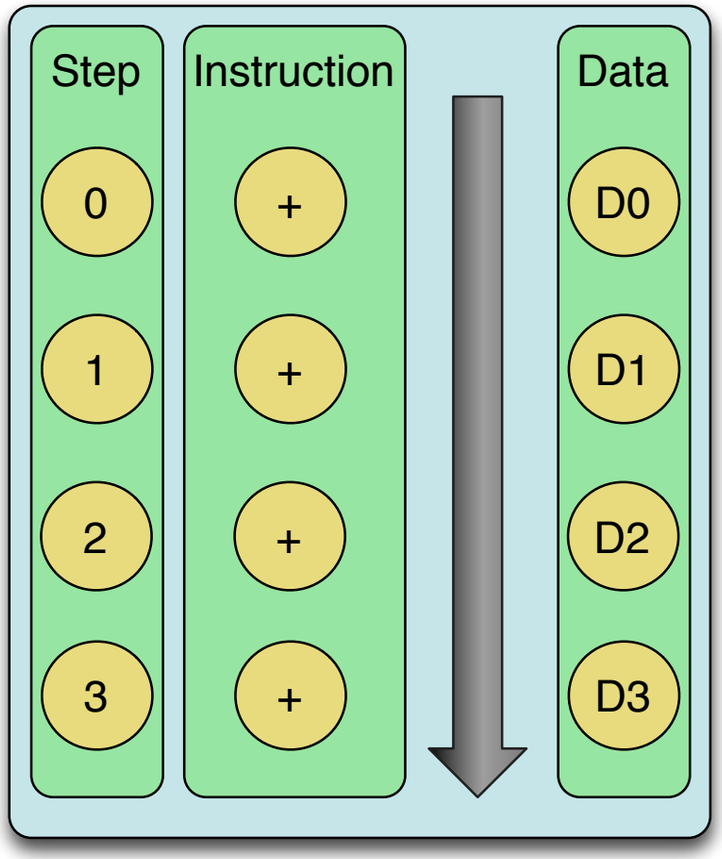
- **Workshop 2**
  - Flynn's Taxonomy.
  - Single Instruction Multiple Data (SIMD):
    - Vectorisation.
    - Intel SSE Intrinsics
  - Multicore Parallelism (MIMD):
    - Threading (OpenMP/Pthreads)
    - Concurrency Issues:
      - Critical Regions.
      - False Sharing.

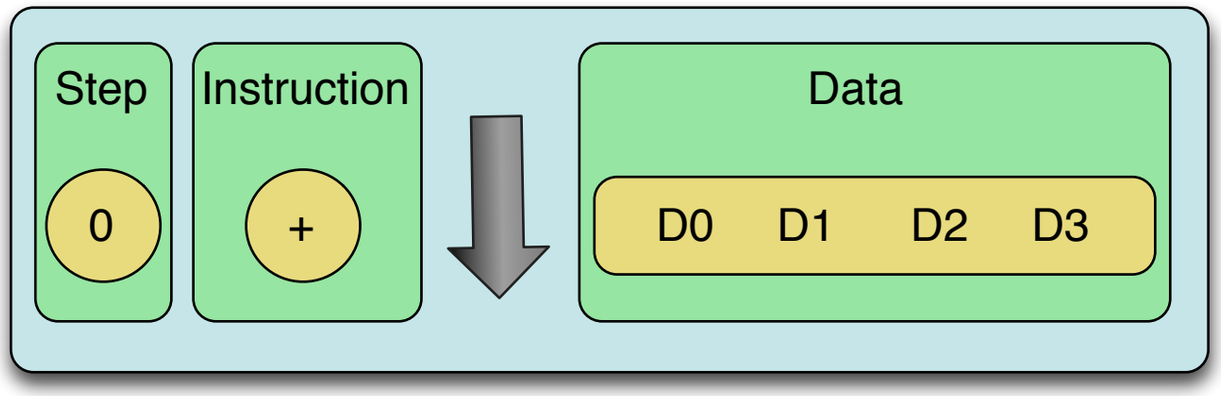
Flynn's Taxonomy

- Flynn's Taxonomy classifies programs into four different categories:
  - **SISD** - ***S****ingle* ***I****nstruction* ***S****ingle* ***D****ata*
    - Classic serial application, applies one instruction to one data stream at a time.
  - **SIMD** - ***S****ingle* ***I****nstruction* ***M****ultiple* ***D****ata*
    - One instruction can be applied to multiple data streams at the same time
  - **MISD** - ***M****ultiple* ***I****nstruction* ***S****ingle* ***D****ata*
    - Rarely encountered
  - **MIMD** - ***M****ultiple* ***I****nstruction* ***M****ultiple* ***D****ata*
    - Different instructions can be applied , across disparate data sets in parallel.

Vectorisation - Introduction

- Vector instructions utilise specific hardware dedicated to executing SIMD behaviour
- Examples include Streaming SIMD Extensions (SSE) on Intel/AMD chips and Altivec on PowerPC.
- Vector registers are special registers that contain multiple data elements at a time (amount dependent on register size.)
- Vector units apply a single instruction across all data elements simultaneously





Vectorisation - Approaches

- SIMD typically applied to loops. Three approaches:
  1. **Auto-vectorisation**:
     - Compiler implemented
     - Least control, but simplest to implement
     - Can provide hints before loop - #pragma ivdep, #pragma simd, #pragma vector always
  2. **Vectorisation Intrinsics**:
     - Intrinsics map to underlying assembly instructions
     - Greater control, but greater dependence on developer to implement
     - Can be used to apply vectorisation techniques that compiler may not be able to, or poorly implements - e.g. vectorising outer loops.
  3. **In-line assembly:**
     - Assembly instructions written directly in code.
     - Most control, most difficult to implement
     - Potentially less portable code.

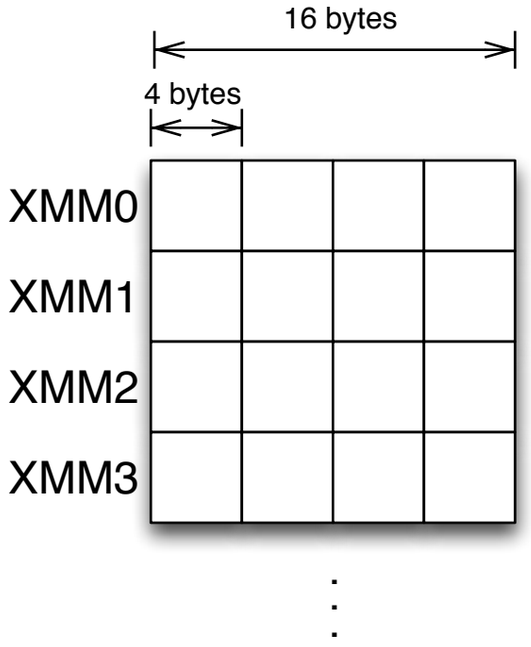
Vectorisation - Requirements

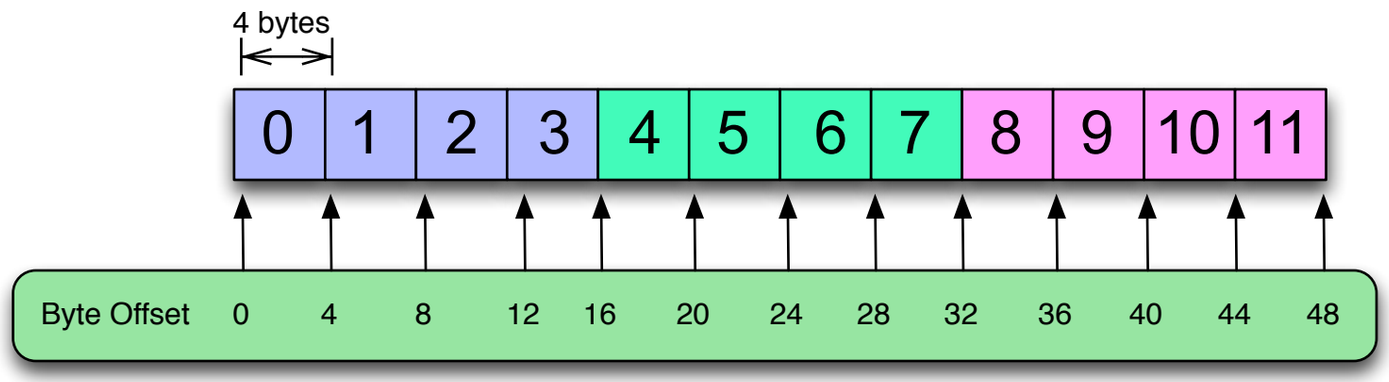
- The use of vector hardware units typically entails:
  - Loading in data from memory into one or more vector registers
  - The execution of vector operations on said registers
  - The write-back of updated data from registers to memory
- Some SIMD hardware units require aligned data (SSE can handle unaligned data, but impacts on performance) - more on this shortly
- Vectorisation cannot violate inter-loop dependancies - if operation ‘a’ depends upon the completion of operation ‘b’, they cannot be completed in parallel.
- Even if the code vectorises, speedup may be minimal if the code is memory bound

Vectorisation - Vector Registers

- The number of parallelised operations is dependent upon
  - The register width
  - The datatype size
- SSE has a register width of 16 bytes. Therefore a vector operation can consist of up to 4 data elements of size 4 bytes (e.g. int, float) or 2 data elements of size 8 bytes (e.g. double)
- SSE has 16 such vector registers, named xmm0 -> xmm15.
- For best results loads and stores to the registers should be kept at a minimum, with data already in the registers being used preferentially if possible

Vectorisation - Alignment

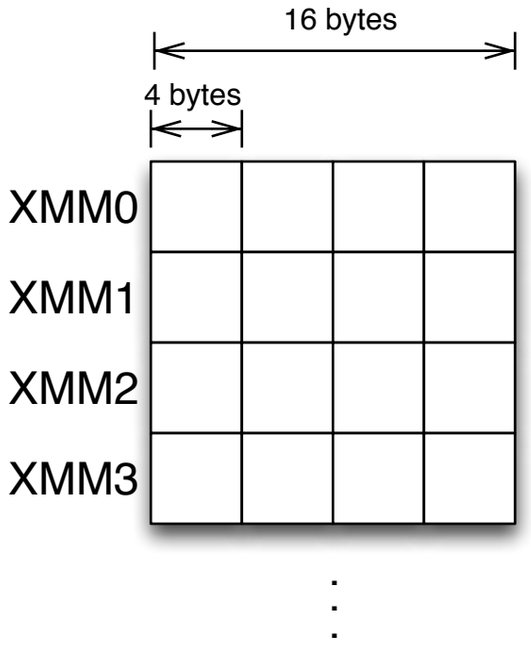


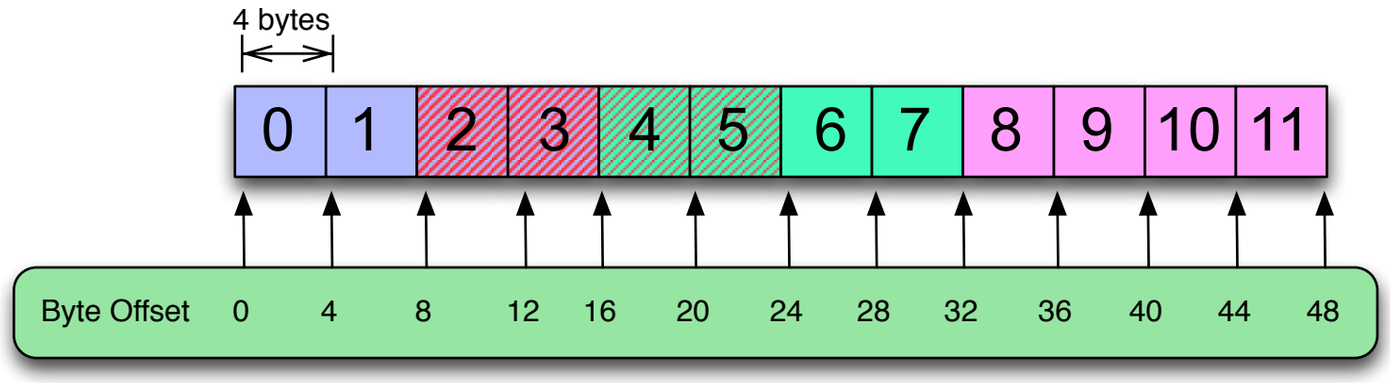


- **Memory Alignment:**
  A memory block is aligned to a value *m* if the byte offset is divisible by *m* (a power of 2).

- Four bytes per data element
- If the location is offset 16 bytes from 0, it has 1,2,4,8,16 byte alignment
- However, a location of offset 24 is only 1,2,4 and 8 aligned

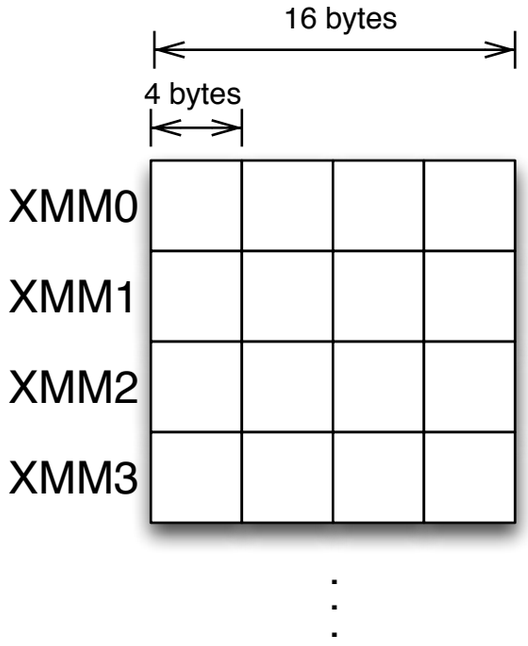
Vectorisation - Unaligned Load

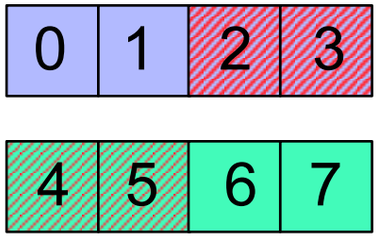


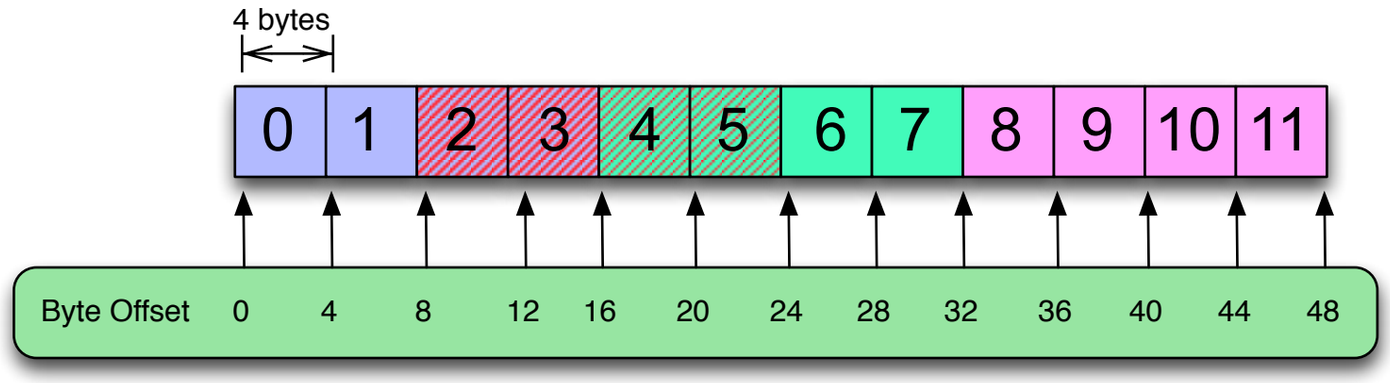


- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines

Vectorisation - Unaligned Load

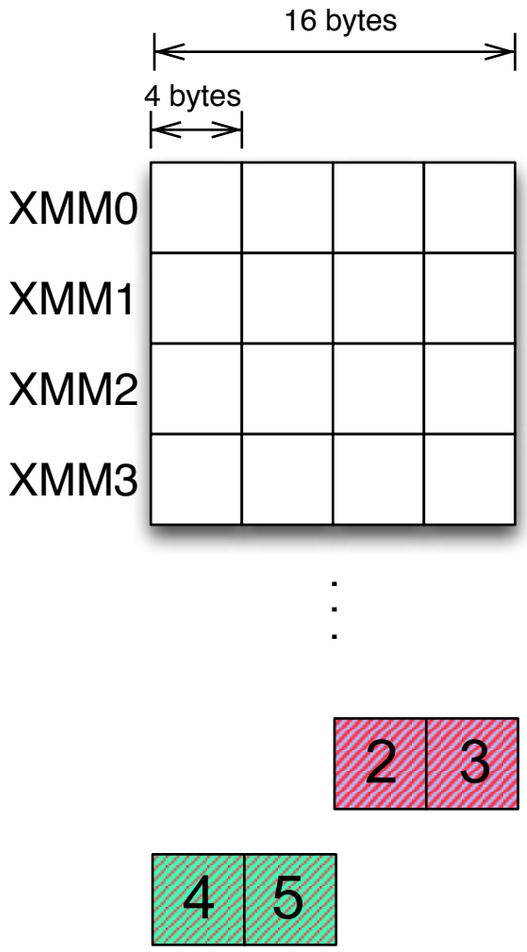


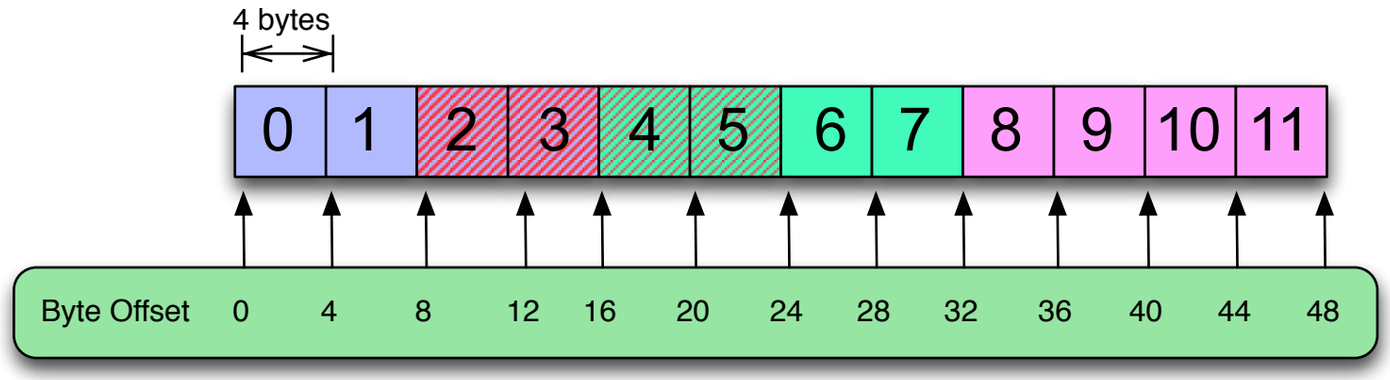




- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines
- System must load both lines
- The location of the relevant data is determined, and the rest unused, before moving to register.

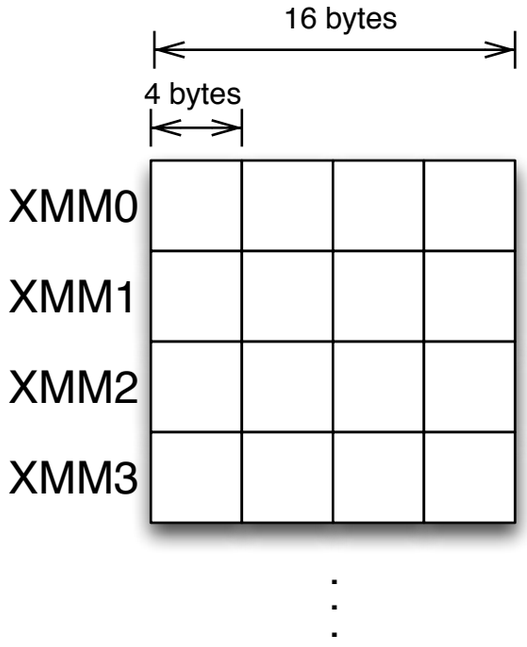
Vectorisation - Unaligned Load

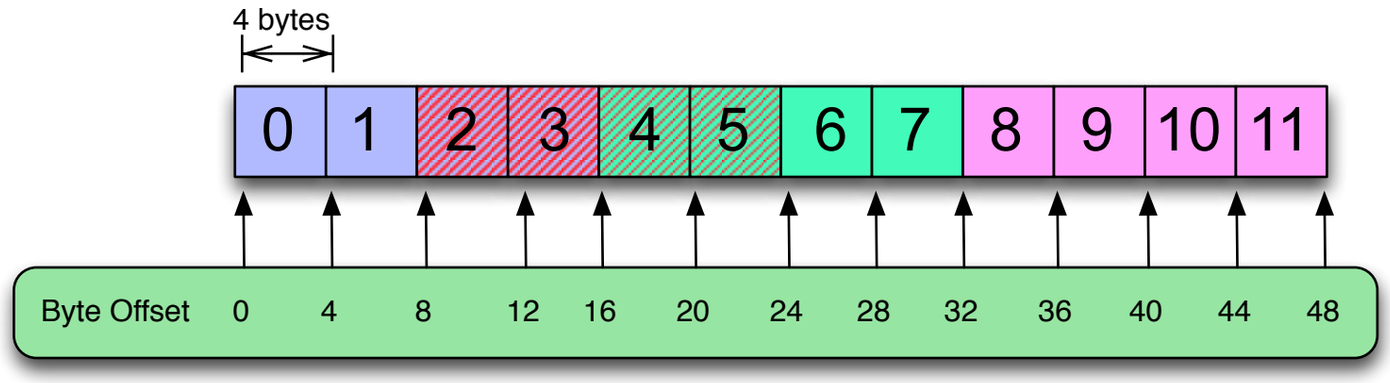




- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines
- System must load both lines
- The location of the relevant data is determined, and the rest unused, before moving to register.

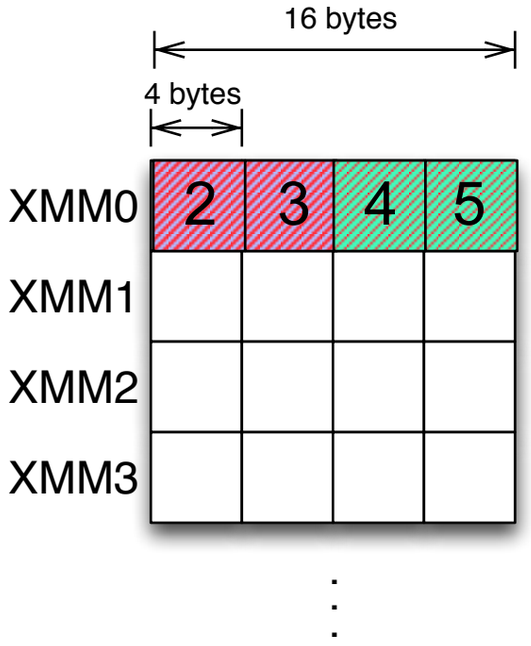
Vectorisation - Unaligned Load

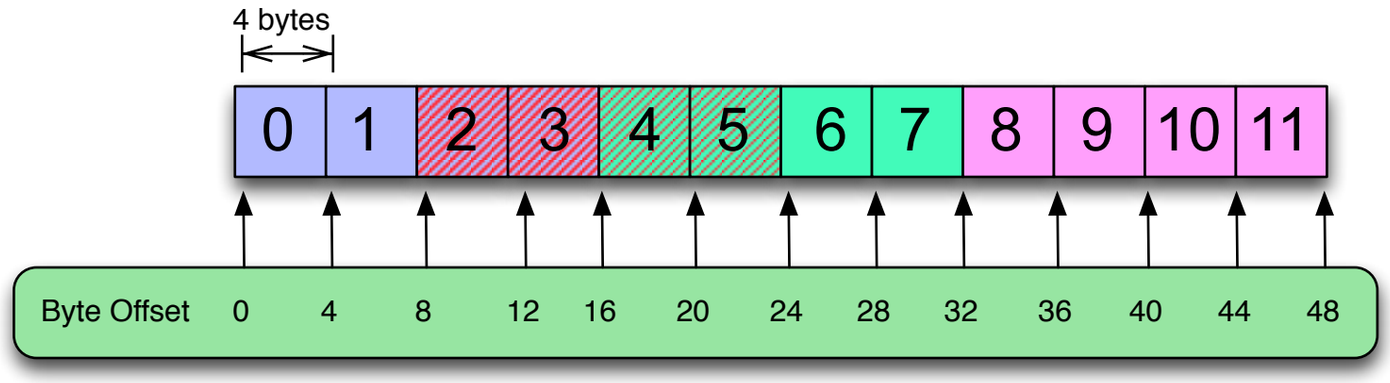




- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines
- System must load both lines
- The location of the relevant data is determined, and the rest unused, before moving to register.

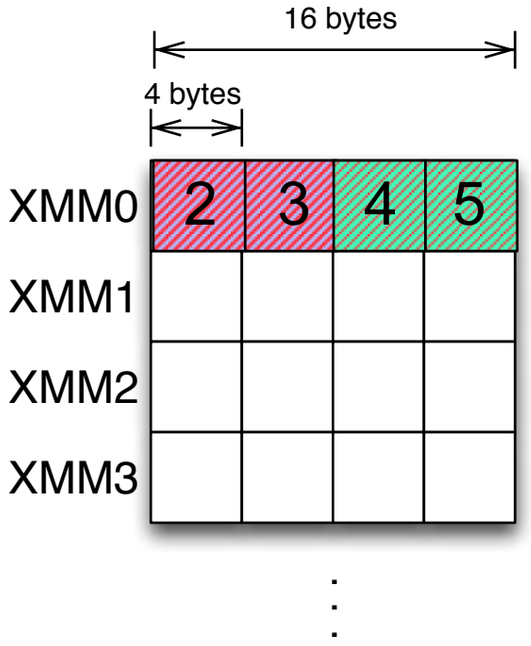
Vectorisation - Unaligned Load

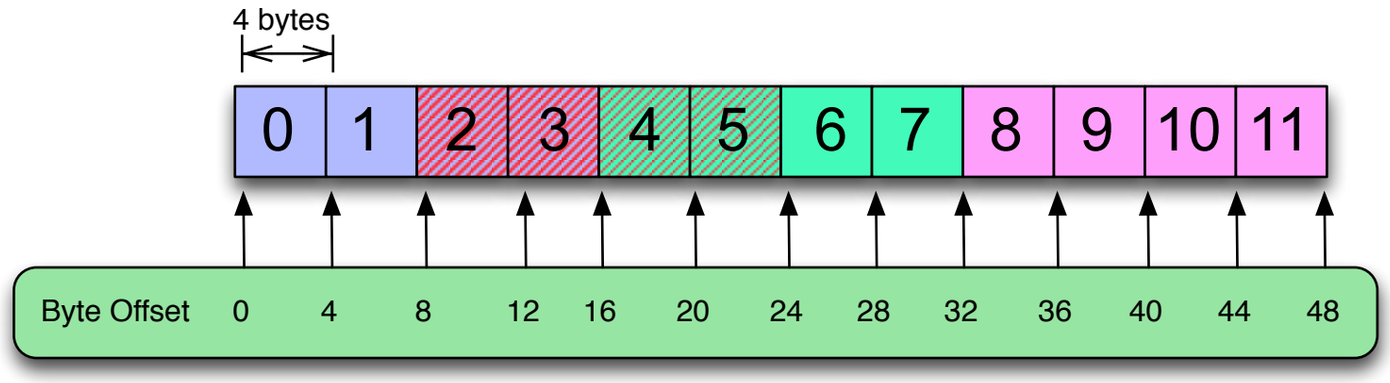




- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines
- System must load both lines
- The location of the relevant data is determined, and the rest unused, before moving to register.

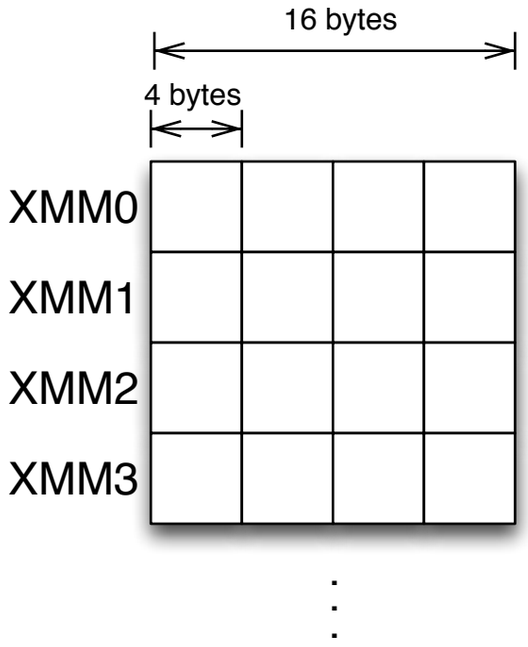
Vectorisation - Unaligned Load

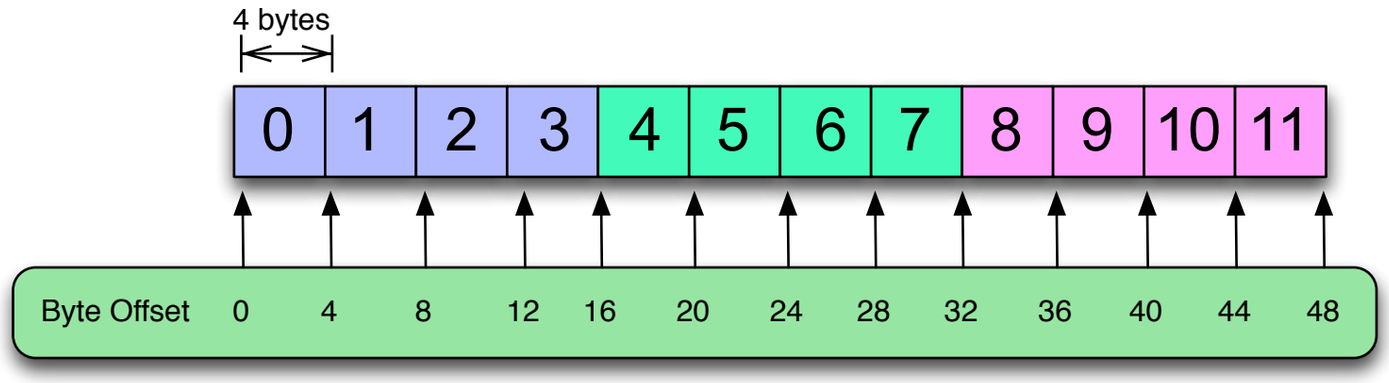




- Vector width is 16 bytes - load 4 values
- Load data elements 2,3,4,5
- Byte offset starts at 8 - 8 byte aligned
- Load traverses two separate cache lines
- System must load both lines
- The location of the relevant data is determined, and the rest unused, before moving to register.
- Two negative impacts on performance:
  - Load was unaligned
  - Load traversed multiple cache lines.

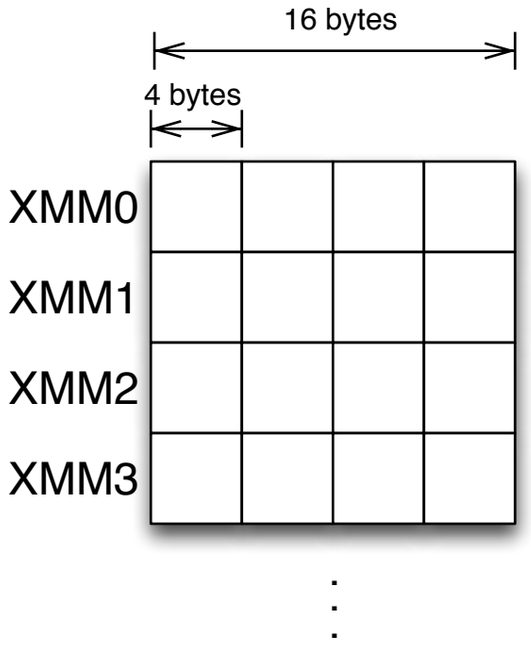
Vectorisation - Aligned Load

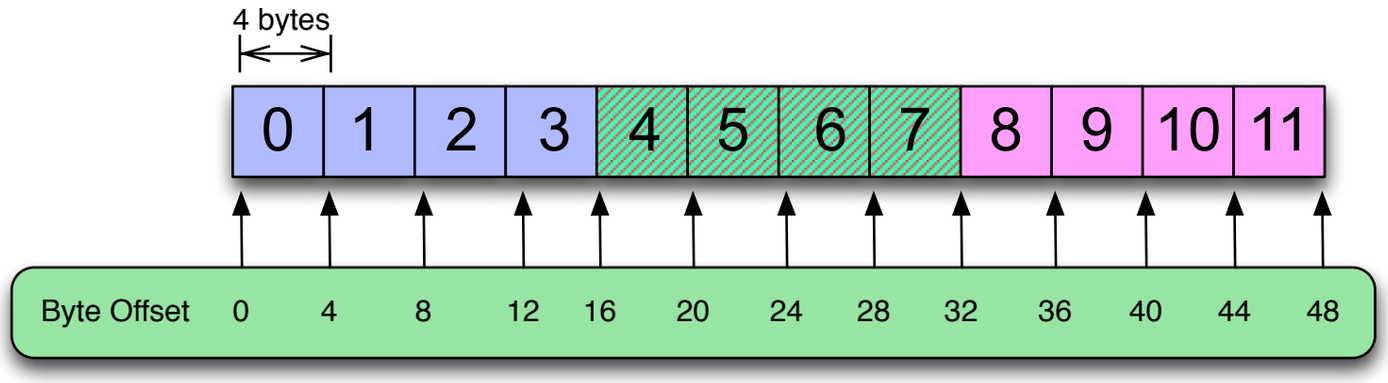




- Vector width is 16 bytes - load 4 values

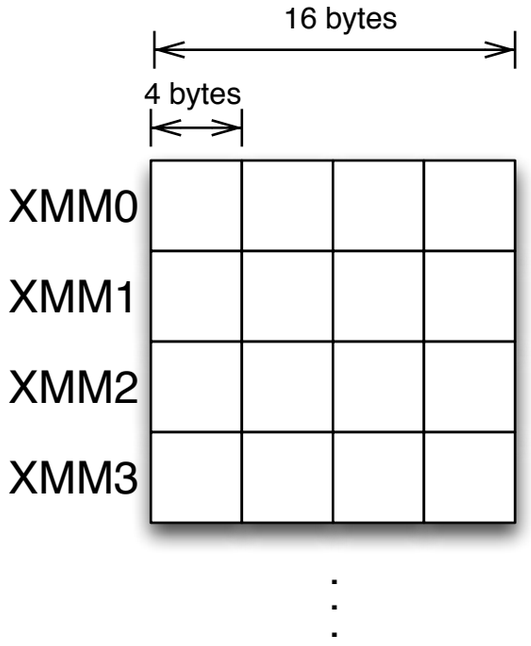
Vectorisation - Aligned Load

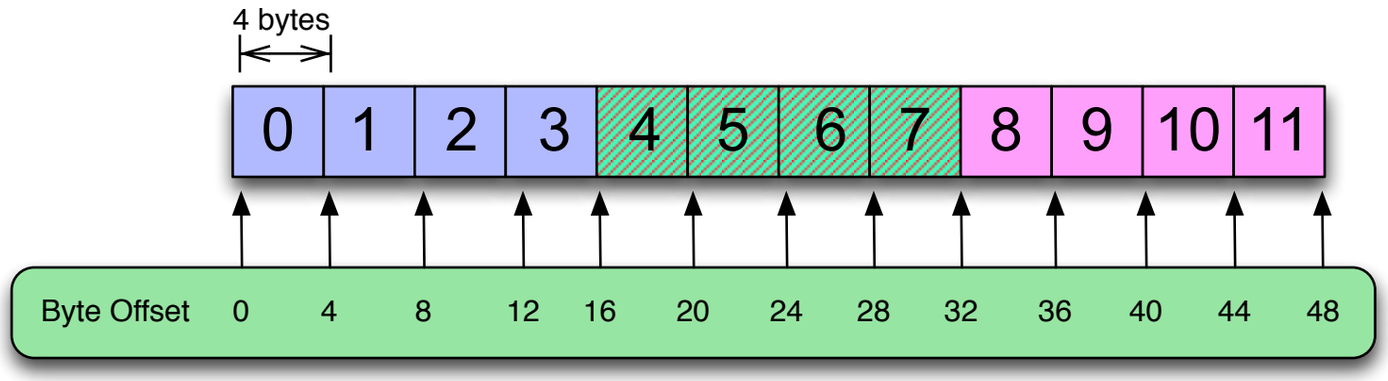




- Vector width is 16 bytes - load 4 values
- Load data elements 4,5,6,7

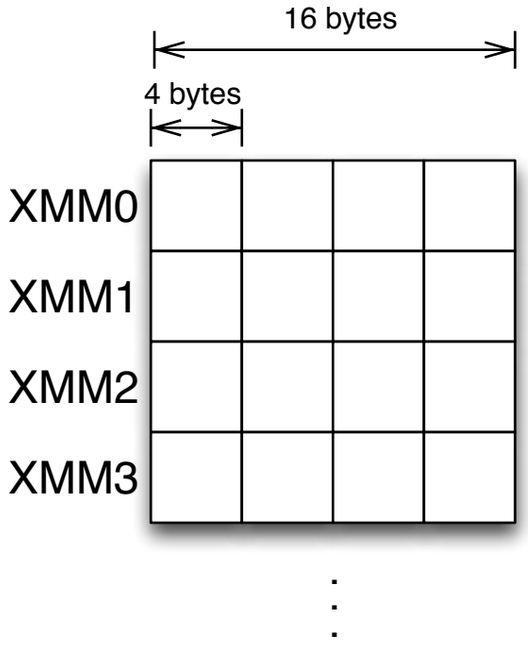
Vectorisation - Aligned Load

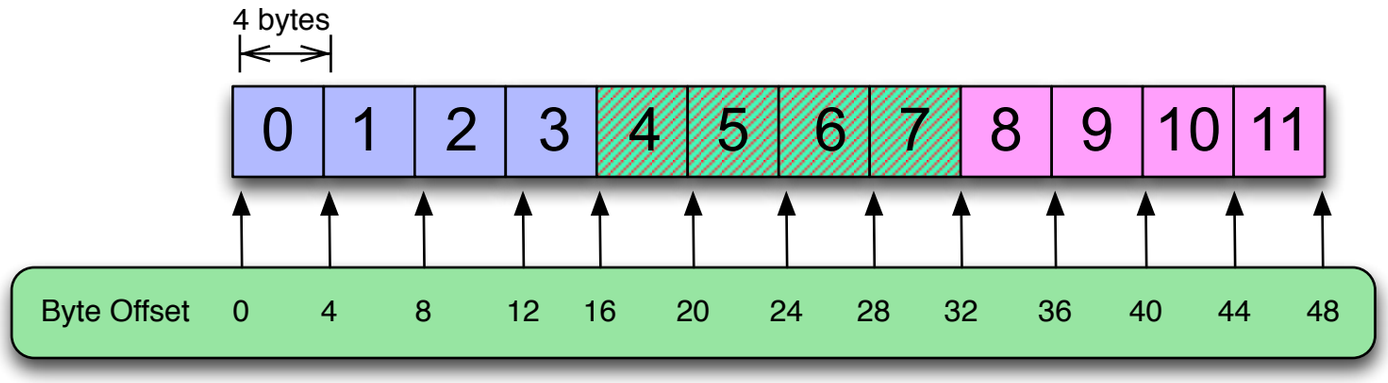




- Vector width is 16 bytes - load 4 values
- Load data elements 4,5,6,7
- Byte offset starts at 16 - 16 byte aligned

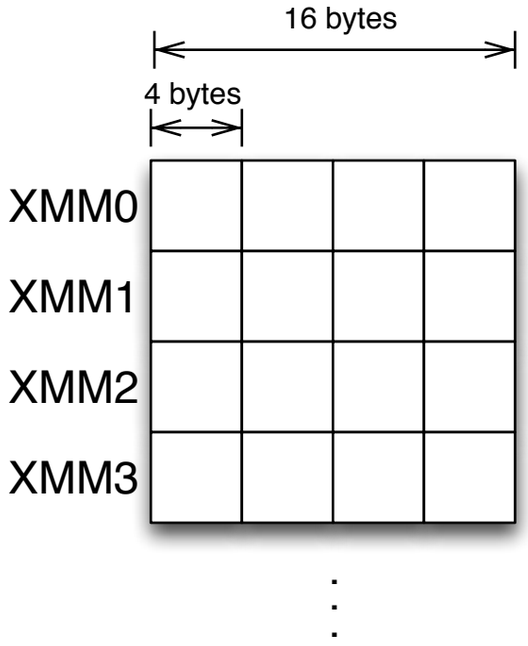
Vectorisation - Aligned Load

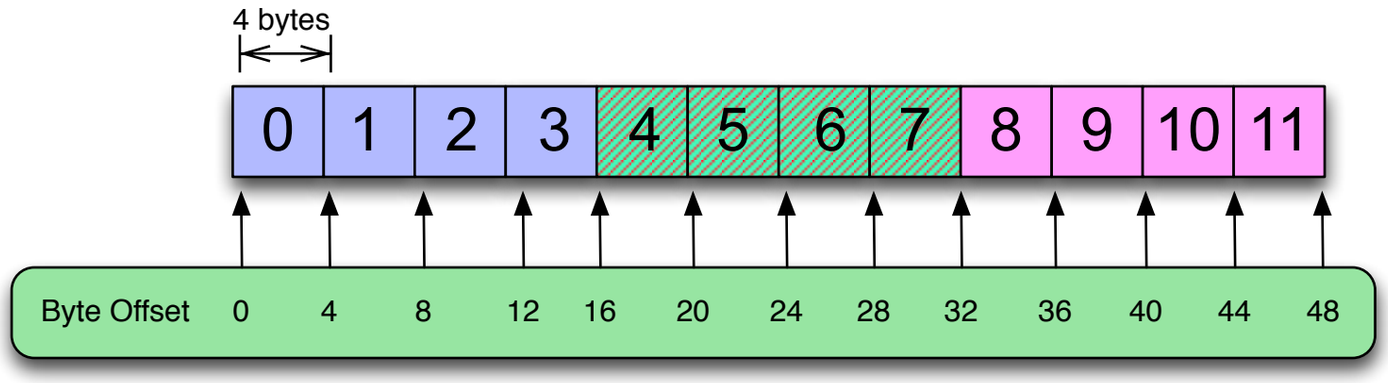




- Vector width is 16 bytes - load 4 values
- Load data elements 4,5,6,7
- Byte offset starts at 16 - 16 byte aligned
- Load only traverses one cache line

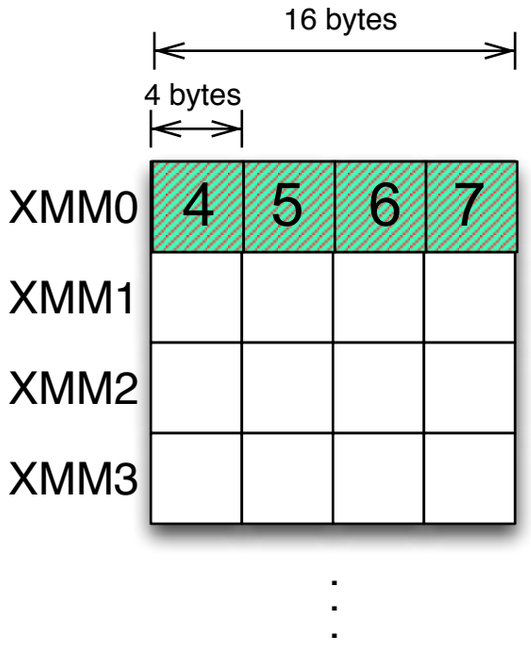
Vectorisation - Aligned Load

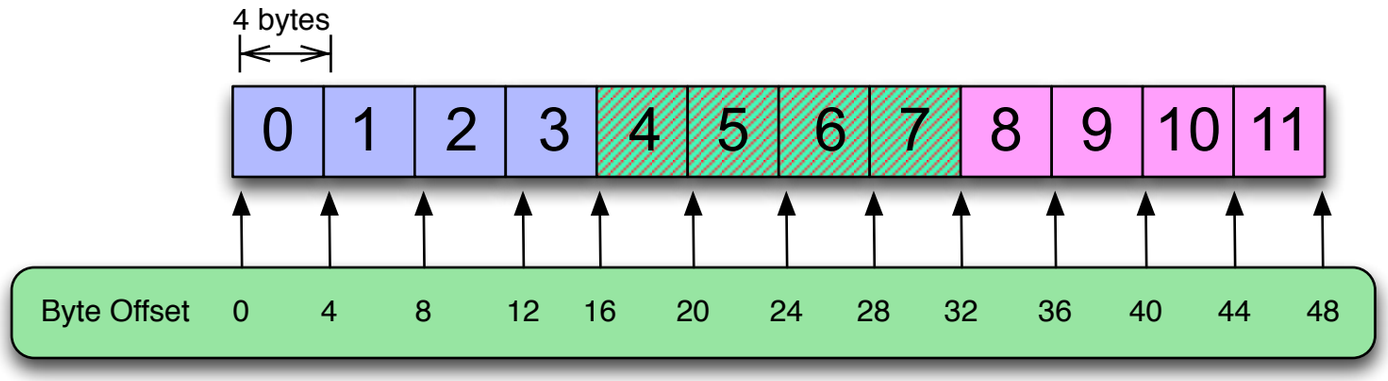




- Vector width is 16 bytes - load 4 values
- Load data elements 4,5,6,7
- Byte offset starts at 16 - 16 byte aligned
- Load only traverses one cache line
- We can load the 4 data elements into the SSE register

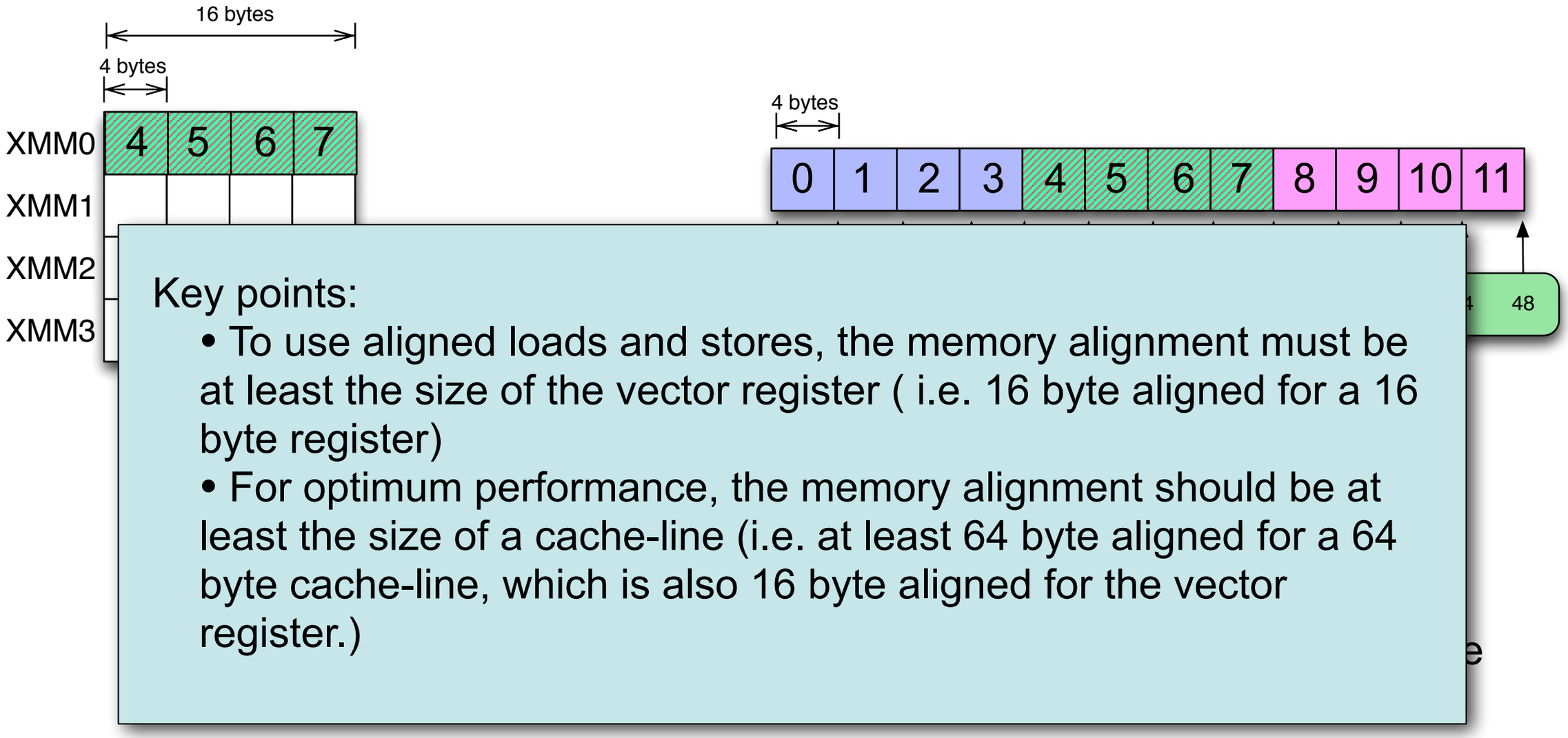
Vectorisation - Aligned Load





- Vector width is 16 bytes - load 4 values
- Load data elements 4,5,6,7
- Byte offset starts at 16 - 16 byte aligned
- Load only traverses one cache line
- We can load the 4 data elements into the SSE register

Vectorisation - Aligned Load



Key points:

- To use aligned loads and stores, the memory alignment must be at least the size of the vector register (i.e. 16 byte aligned for a 16 byte register)
- For optimum performance, the memory alignment should be at least the size of a cache-line (i.e. at least 64 byte aligned for a 64 byte cache-line, which is also 16 byte aligned for the vector register.)

Vectorisation - Available Implementations

- There are a wide-range of SIMD implementations:
  - **SSE - S**treaming **S**IMD **E**xtensions:
    - Intel based, found on most modern Intel and AMD x86 CPUs
  - **AVX - A**dvanced **V**ector E**x**tensions:
    - Twice the register width of SSE.
    - Found on Intel Sandybridge chips and above.
  - **Altivec:**
    - Sun implementation of SIMD
    - Found on PowerPC architectures
  - **MIC** - ***M**any **I**ntegrated **C**ores*
    - Xeon Phi
  - **GPUs** - ***G**raphics **P**rocessing **U**nits*
    - Essentially very large SIMD units
    - CUDA for Nvidia, OpenCL for AMD/Nvidia

Vectorisation - SSE Intrinsics

- Intel provides an intrinsics guide detailing intrinsics for multiple versions of SSE and AVX at:
  - http://software.intel.com/en-us/articles/intel-intrinsics-guide
- The header <immintrin.h> provides access to SSE intrinsics and custom vector types (both single and double precision variants)
- Sample datatypes include:
  - __m128 - 128bit single precision (float) vector unit
  - __m128d - 128bit double precision (double) vector unit

SSE - Intrinsics and Functions

Intrinsic	Input Values	Return Value	Operation
_mm_malloc	size_t memory size in bytes int alignment	(void*) p	Allocates a block of memory in a similar manner to malloc, but pointer p will point to a memory location with the requested alignment.
_mm_free	(void*) p	None	Aligned memory allocated with _mm_malloc must be freed using _mm_free.
_mm_load_ps	float* p	__m128 v	Aligned load - Loads contents of *p, *(p+1), *(p+2) and *(p+3) into the vector register, returned as v. Aligned memory only.
_mm_loadu_ps	float* p	__m128 v	Unaligned load - Loads contents of *p, *(p+1), *(p+2) and *(p+3) into the vector register, returned as v.
_mm_store_ps	float* p, __m128 v	None	Aligned Store - Stores the contents of the vector v into *p, *(p+1), *(p+2) and *(p+3). Aligned memory only.
_mm_storeu_ps	float* p, __m128 v	None	Unaligned Store - Stores the contents of the vector v into *p, *(p +1), *(p+2) and *(p+3).

SSE - Intrinsics and Functions

Intrinsic	Input Values	Return Value	Operation
_mm_add_ps	__m128 v1, __m128 v2	__m128 v3	Adds the contents of v1 and v2, returns a vector v3 containing the result
_mm_mul_ps	__m128 v1, __m128 v2	__m128 v3	Multiples the contents of v1 and v2, returns a vector v3 containing the result
_mm_rsqrt_ps	__m128 v1	__m128 v2	Computes the reciprocal square root of the contents of vector v1, returns the results in vector v2
_mm_sqrt_ps	__m128 v1	__m128 v2	Computes the square root of the contents of vector v1, returns the results in vector v2

SSE - Intrinsics and Functions

Intrinsic	Input Values	Return Value	Operation
_mm_set1_ps	float val	__m128 v1	Sets the value of each element in the returned vector v1 to that of val.
_mm_shuffle_ps	__m128 a, __m128 b, int i	__m128 v1	Selects some elements of a and some elements of b, and copies into new vector. Which elements is dependent on i.
_mm_cmpgt_ps	__m128 a, __m128 b	__m128 mask	Populates all the bits for a data element in mask with either 1's or 0's, depending upon whether that data element in a is greater than that data element in b.
_mm_blendv_ps	__m128 a, __m128 b, __m128 mask	__m128 v1	For each data element of the vector, if the value in mask has a significant bit of 1, copy the element from b into that position of v1, else copy the element from a.

SSE - Example 1 - Vector Sum

- Simple kernel that computes the sum of two arrays.
- Each iteration accesses a sequential memory location for each array.
- No inter-loop dependancies
- Amenable to vectorisation

```
float *a = malloc(sizeof(float) * n);
float *b = malloc(sizeof(float) * n);
float *c = malloc(sizeof(float) * n);
...
{
    for(i=0;i<n;i++){
        c[i] = a[i] + b[i];
    }
}
free(a);
free(b);
free(c)
```

SSE - Example 1 - Vector Sum

- Step 1 - Unroll the loop to a depth that matches our vector length.
- Vector length is 16 bytes / sizeof(float) = 4
- Unrolling allows us to substitute 4 unrolled, identical operations acting on sequential memory locations with a single vectorised operation

```
float *a = malloc(sizeof(float) * n);
float *b = malloc(sizeof(float) * n);
float *c = malloc(sizeof(float) * n);
...
{
    int vec_size = (n/4)*4;
    for(i=0;i<vec_size;i++){
        c[i] = a[i] + b[i];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
    for(;i<n;i++){
        c[i] = a[i] + b[i];
    }
}
free(a);
free(b);
free(c)
```

SSE - Example 1 - Vector Sum

- Step 2 - Update malloc to aligned memory allocation (to aid in performance)
- Step 3 - Substitute in vectorisation intrinsics.

1. Vector load for data a: i,i+1,i+2,i+3
2. Vector load for data b: i,i+1,i+2,i+3
3. Vector add on registers containing data for a and b
4. Store results from vector (containing 4 result values) in c: i,i+1,i+2,i+3

```
#include <immintrin.h>

float *a = _mm_malloc(sizeof(float) * n,64);
float *b = _mm_malloc(sizeof(float) * n,64);
float *c = _mm_malloc(sizeof(float) * n,64);
...
{
    int vec_size = (n/4)*4;
    for(i=0;i<vec_size;i+=4)
    {
        __m128 a_v = _mm_load_ps(a+i);
        __m128 b_v = _mm_load_ps(b+i);
        __m128 c_v = _mm_add_ps(a_v, b_v);
        _mm_store_ps(c+i,c_v);
    }
    for(;i<n;i++){
        c[i] = a[i] + b[i];
    }
}
_mm_free(a);
_mm_free(b);
_mm_free(c)
```

SSE - Example 2 - Matrix Multiply

- Recall our blocking matrix-multiply from workshop 1

```
matrix_multiply(a,b,c,n){
    int b = BLOCK_SIZE;
    for(i0=0; i<n; i+=b){
        for(j0=0; j<n; j+=b){
            for(k0=0; k<n; k+=b){
                for(i=i0; i<min(i0+b,n); i++){
                    for(j=j0; j<min(j0+b,n); j++){
                        for(k=k0; k<min(k0+b,n); k++){
                            c[i,j] = c[i,j] + (a[i,k] * b[k,j]);
                        }
                    }
                }
            }
        }
    }
}
```

In the previous workshop we tackled the issue of cache behaviour.

However, matrix-multiply performs a large number of floating-point operations, and this blocking version is operating mostly on data in cache - this makes it a prime candidate for vectorisation!

SSE - Example 2 - Matrix Multiply - Unroll

```
matrix_multiply(a,b,c,n){
    int b = BLOCK_SIZE;
    for(i0=0; i<n; i+=b){
        for(j0=0; j<n; j+=b){
            for(k0=0; k<n; k+=b){
                for(i=i0; i<min(i0+b,n); i++){
                    vecsize = (min(j0+b,n)/4)*4;
                    for(j=j0; j<vecsize; j+=4){
                        for(k=k0; k<min(k0+b,n); k++){
                            c[i,j]     = c[i,j]     + (a[i,k] * b[k,j]);
                            c[i,j+1] = c[i,j+1] + (a[i,k] * b[k,j+1]);
                            c[i,j+2] = c[i,j+2] + (a[i,k] * b[k,j+2]);
                            c[i,j+3] = c[i,j+3] + (a[i,k] * b[k,j+3]);
                        }
                    }
                    for(; j<min(j0+b,n); j++){
                        for(k=k0; k<min(k0+b,n); k++){
                            c[i,j] = c[i,j] + (a[i,k] * b[k,j]);
                        }
                    }
}}}}}}
```

Problem: If we vectorise the k inner loop, we would need 4 separate loads for b (traversing k is not sequential.)

Solution: Vectorise a loop further out instead!

If we move up to the j inner loop, we can traverse *c* and *b* sequentially with vectors, reusing the vector for each full k loop, while still traversing *a* sequentially.

To vectorise, we first unroll!

SSE - Example 2 - Matrix Multiply - Intrinsics

```
matrix_multiply(a,b,c,n){
    int b = BLOCK_SIZE;
    for(i0=0; i<n; i+=b){
        for(j0=0; j<n; j+=b){
            for(k0=0; k<n; k+=b){
                for(i=i0; i<min(i0+b,n); i++){
                    vecsize = (min(j0+b,n)/4)*4;
                    for(j=j0; j<vecsize; j+=4){
                        vec_c = _mm_load_ps(c + j + (i*n));
                        for(k=k0; k<min(k0+b,n); k++){
                            vec_a = _mm_set1_ps(a+k+(i*n));
                            vec_b = _mm_load_ps(b+j+(k*n));
                            vec_c = _mm_add_ps(vec_c, _mm_mul_ps(vec_a,vec_b));
                        }
                        _mm_store_ps(c + j + (i*n), vec_c);
                    }
                    // mopup loop
                }
            }
        }
    }
}}}
```

SSE - Conditionals

- Branching is an impediment to SSE - cannot selectively run different operations in parallel
- A solution is to compute the result of all branches, and select the correct value from each vector of results using a mask.
- Such an approach can undermine the performance gains of SSE if there are many branches - two branches means twice as much computation as the serial version!
- Best to eliminate or move branches if possible
- N.B. Other SIMD technologies can take a different approach - e.g. some can apply a mask to which instructions to run, rather than to the outcomes.

SSE - Conditionals - Example

```
for(i=0;i<4;i++)
{
    if(a[i] > 0.5){
        a[i] = a[i] * 2.5;
    }
    else {
        a[i] = a[i] * 1.5;
    }
}
```

```
__m128 vec_a = _mm_load_ps(a);
__m128 half = _mm_set1_ps(half);
__m128 mask = _mm_cmpgt_ps(a,half);

__m128 branch1 = _mm_mul_ps(a_vec,_mm_set1_ps(2.5));
__m128 branch2 = _mm_mul_ps(a_vec,_mm_set1_ps(1.5));

__m128 result = _mm_blendv_ps(branch2, branch1,mask);
_mm_store_ps(a,result);
```

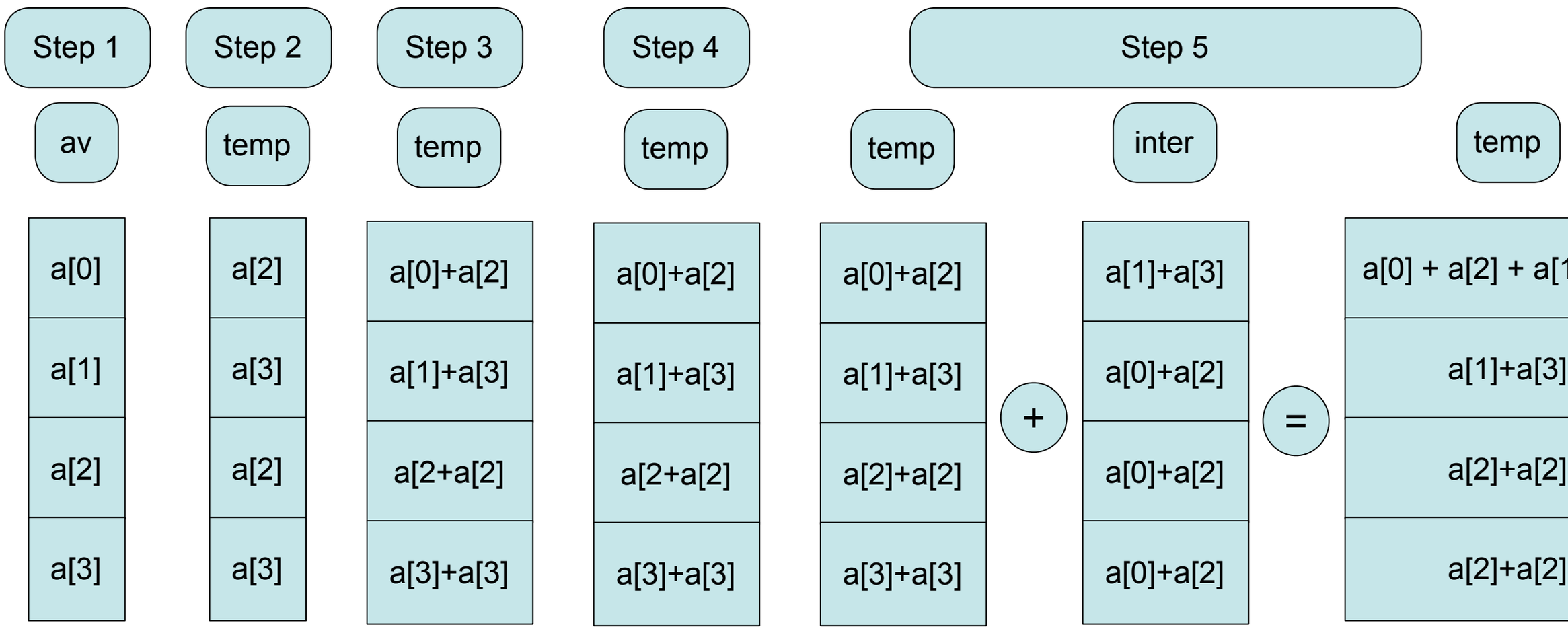
a	Mask (Sig bit)	branch1	branch2	result
1.0	1	2.5	1.5	2.5
0.2	0	0.5	0.3	0.3
0.3	0	0.75	0.45	0.45
1.2	1	3.0	1.8	3.0

SSE - Shuffle

- Shuffle operations rearrange the order of data elements within vector registers (or copy parts of vectors from one register to another.)
- E.g. _mm_movehl_ps, _mm_shuffle_ps ....
- Outcome typically depends on operation and control value - consult Intel documentation for output behaviour.
- Example: horizontal add (adds elements within a vector, rather than two vectors.) Outcome stored in a[0].

SSE - Shuffle - Example

```
__m128 av = _mm_load_ps(a);
__m128 temp = _mm_movehl_ps(a_vec,a_vec);
temp = _mm_add_ps(a_vec,temp);
temp = _mm_add_ss(temp,_mm_shuffle_ps(temp,temp,_MM_SHUFFLE(0,0,0,1))
_mm_store_ss(&sum,temp);
```



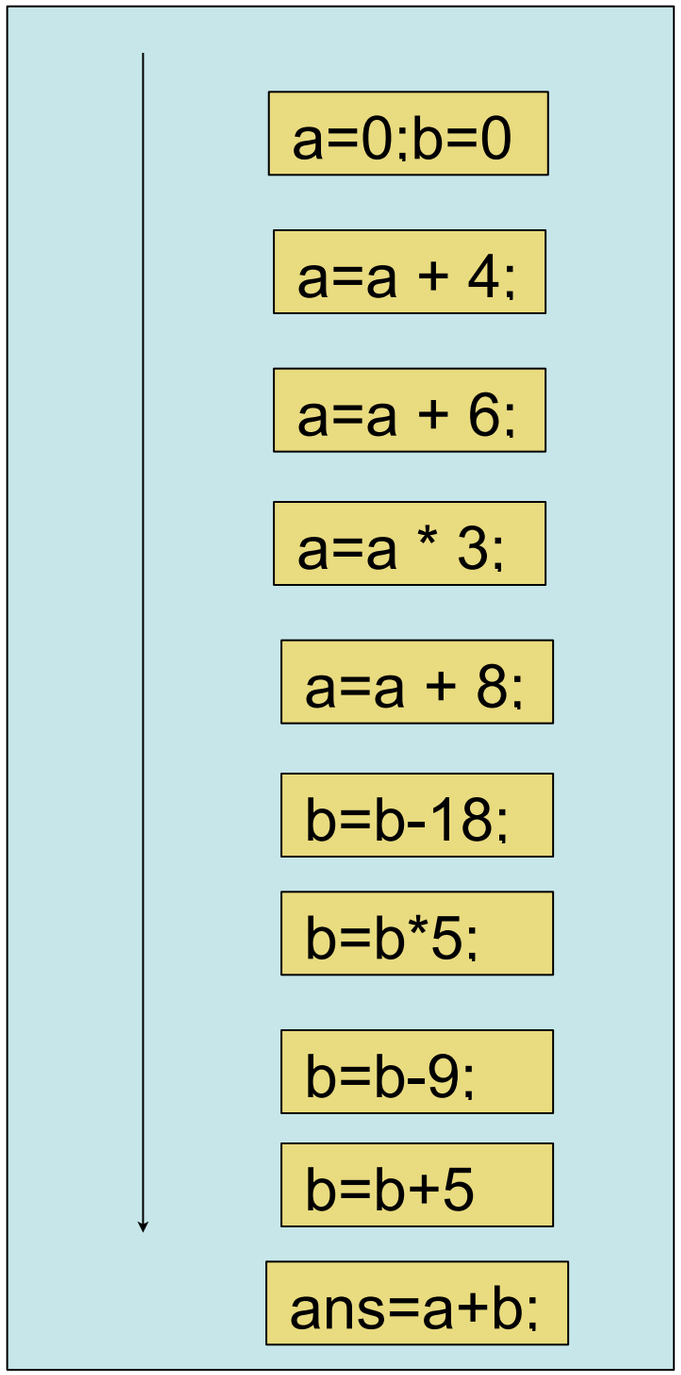
Threading - Introduction

- Increasing clock speed has previously been a “cheap” way to gain improved performance, by increasing the number of cycles (and by extension, instructions) per second.
- However, increasing clock speeds also leads to a greater power draw, increased heat generation, and higher cooling requirements.
- Instead of increasing clock speeds further, manufacturers started adding multiple cores per chip to enable parallel algorithms. Multiple instructions could be run simultanously by distributing tasks across the cores.
- To use these cores, the developer must explicitly implement parallel algorithms (the compiler cannot auto-thread code). A number of programming language extensions/libraries exist to aid in this task.

What is Threading?

- A serial program will run a set of operations in a deterministic, predictable order.
- A threaded program will run blocks of code in parallel asynchronously.
- On a multi-core machine, each thread will have access to:
  - Private memory, accessible only by itself;
  - Global shared memory, accessible by all threads.
- Operation order between threads is non-deterministic
- Typical thread assignment is to have one thread per core (multiple threads on a core will have to share the core's resources.)
- Threading is an example of MIMD - Each thread can run a different set of instructions across a separate block of data.

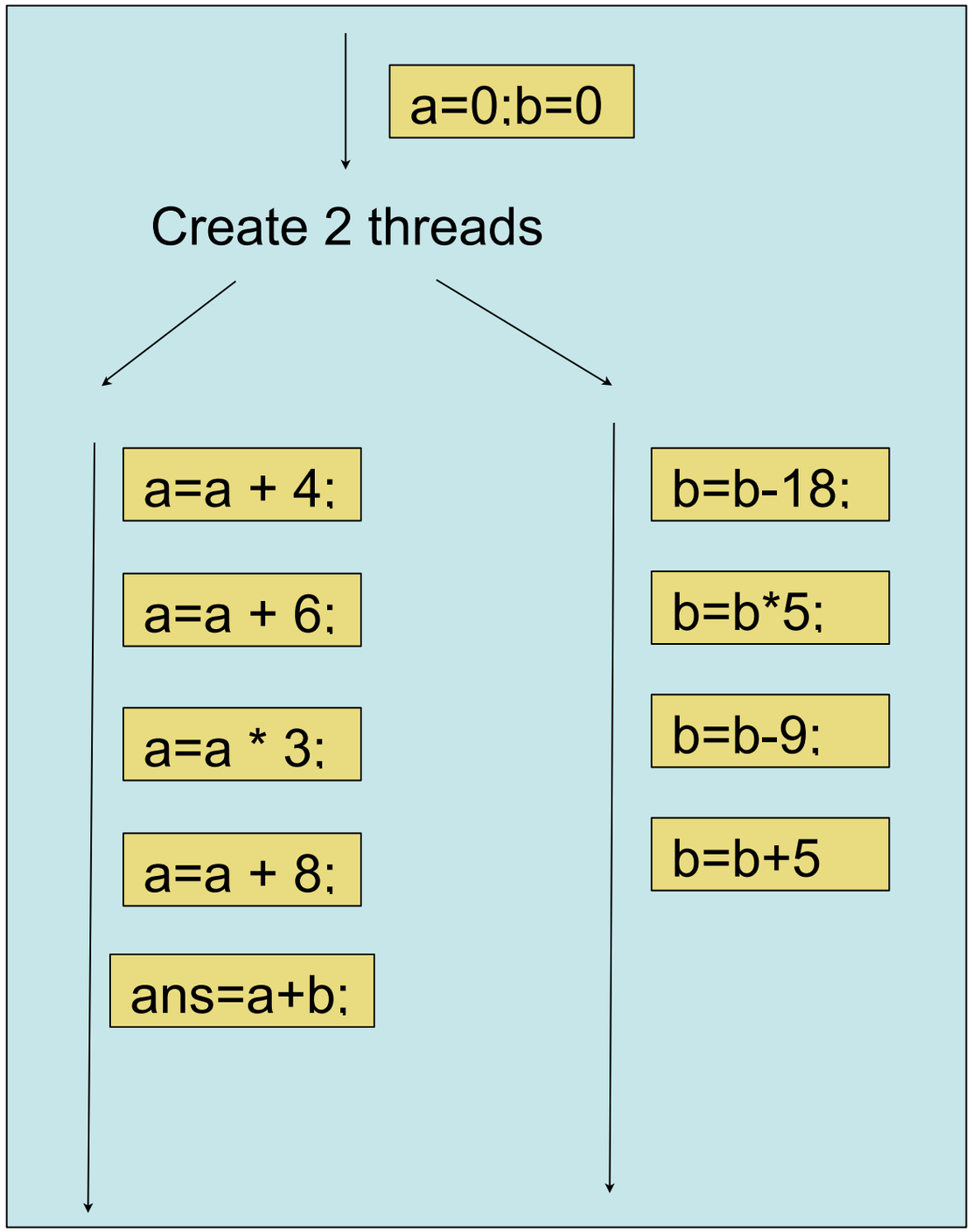
What is Threading?



Serial Application:

- Order is deterministic
- Can guarantee at end of execution value of ‘ans’ will be -38
- 9 operations, plus initialisation
- 9 operations run serially

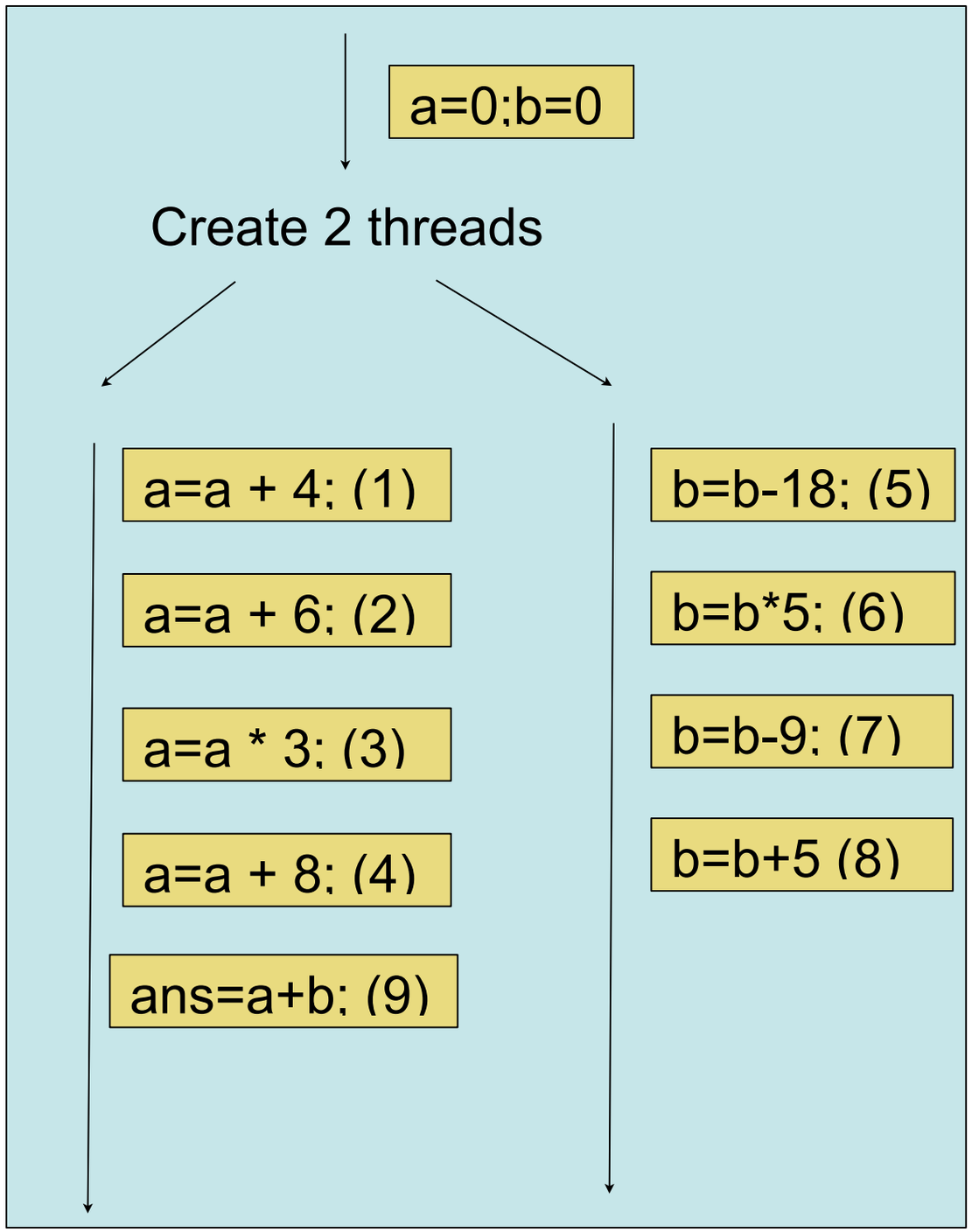
What is threading?



Threaded Application:

- Initially starts with one master thread
- Splits into two, work is distributed between threads
- 8 operations are completed in 4 steps

What is threading?

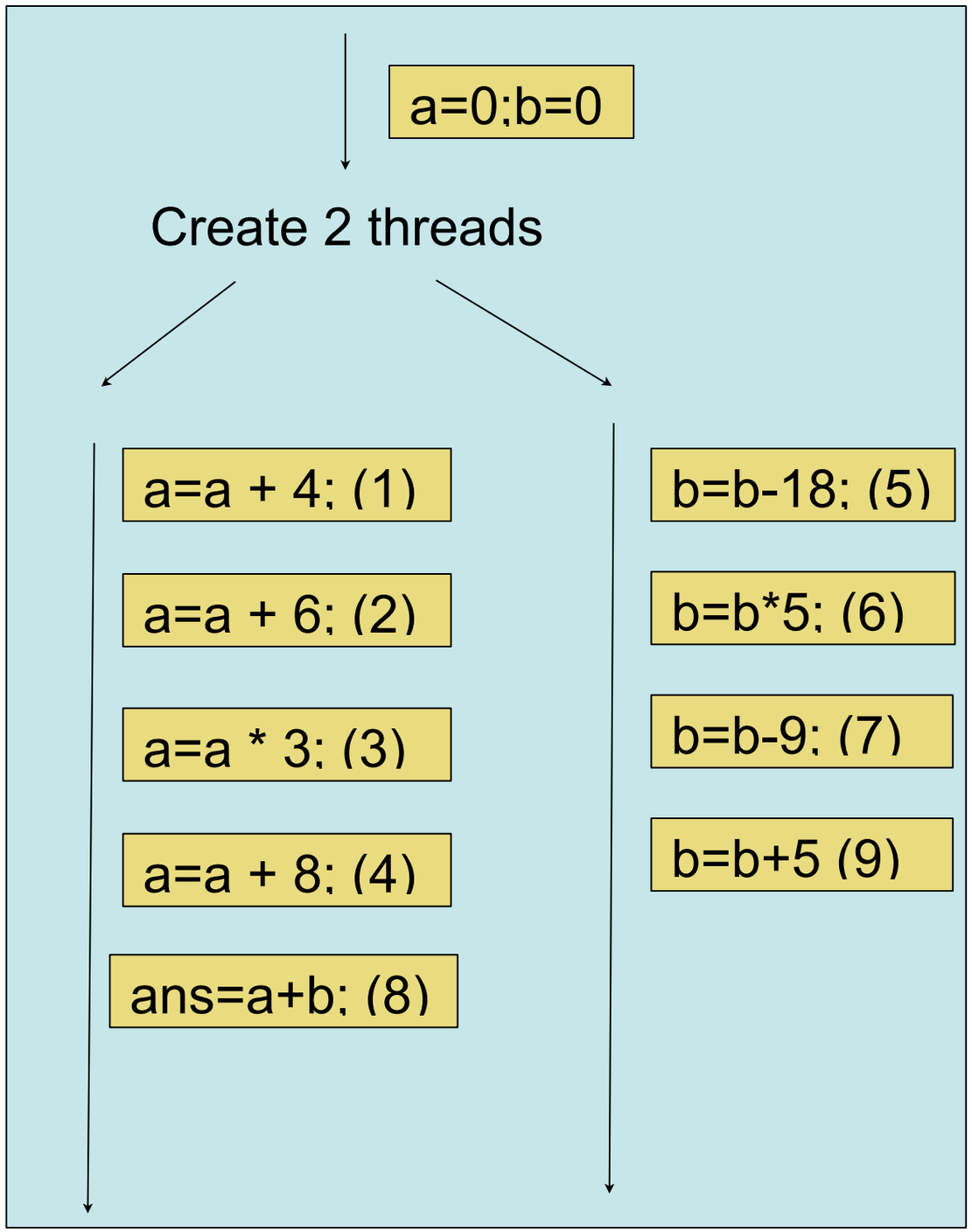


Threaded Application:

- Initially starts with one master thread
- Splits into two, work is distributed between threads
- 8 operations are completed in 4 steps

- Non-deterministic behaviour
- Both threads are updating a shared value
- Order of Operations 1: ans = -38

What is threading?



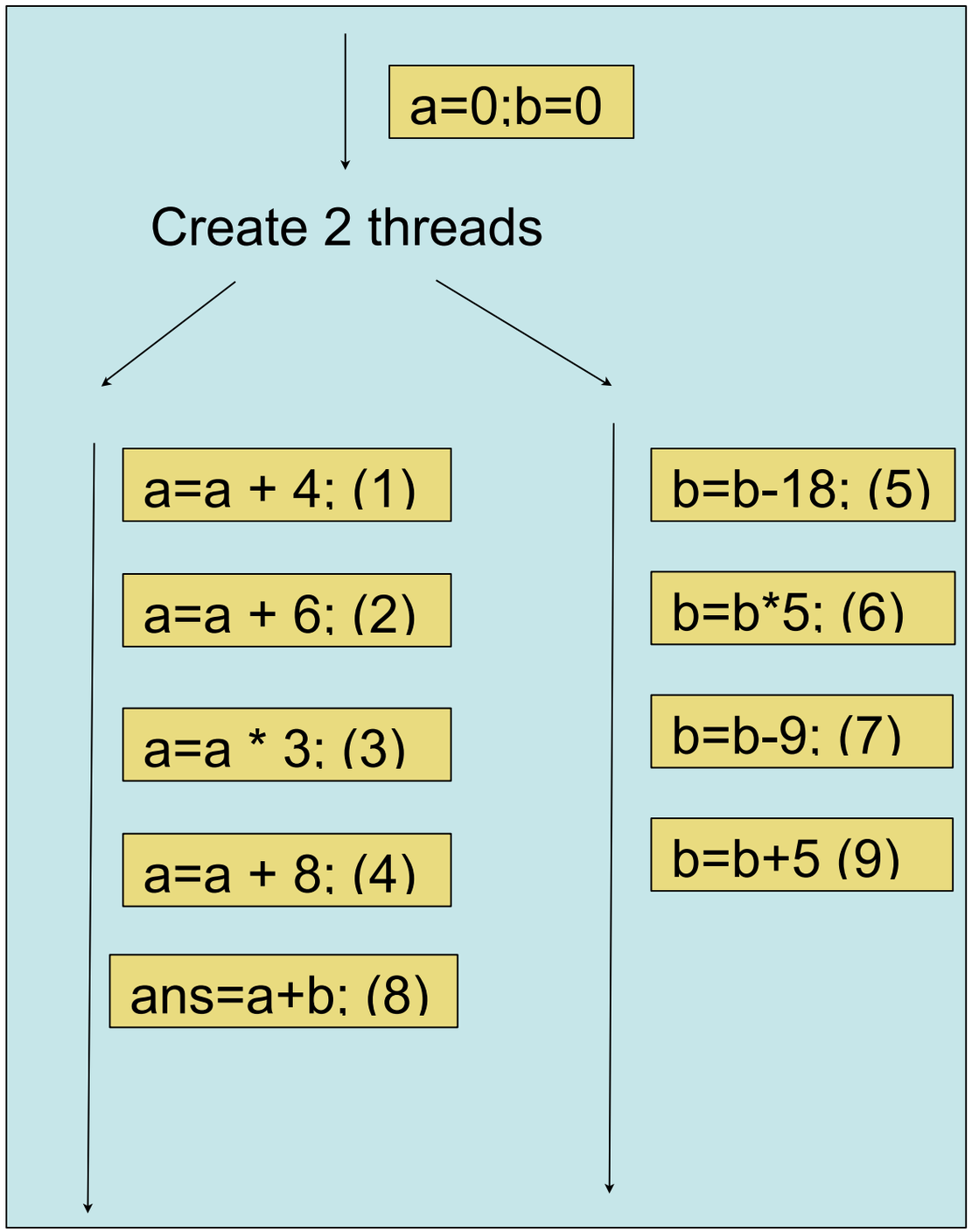
Threaded Application:

- Initially starts with one master thread
- Splits into two, work is distributed between threads
- 8 operations are completed in 4 steps

BUT

- Non-deterministic behaviour
- Both threads are updating a shared value
- Order of Operations 1: ans = -38
- Order of Operations 2: ans = -61

What is threading?



Threaded Application:

- Initially starts with one master thread
- Splits into two, work is distributed between threads
- 8 operations are completed in 4 steps

BUT

- Non-deterministic behaviour
- Both threads are updating a shared value
- Order of Operations 1: ans = -38
- Order of Operations 2: ans = -61
- This is known as a *race condition* - to be avoided at all costs!
- A parallel algorithm must end with a deterministic result, even if the order of operations between threads is non-deterministic

Race Conditions

- **Race condition:**
  - Operations between threads are executed in a different order.
  - A race condition is where the final result is dependent upon the order of execution - this is unreliable!
  - E.g. Two threads update a cumulative sum. Order of operation is:
    1. Thread 0 reads sum
    2. Thread 1 reads sum
    3. Thread 0 adds to sum and writes back
    4. Thread 1 adds to sum and writes back
  - Thread 1 has used the old value of sum, rather than the value written by thread 0, overwriting thread 0's contribution!

Threading - Implementation

- More than one option exists:
  - pThreads
  - OpenMP

- Will cover OpenMP, but you are free to use either for the coursework.

OpenMP - Usage

- **Compilation:**
  - OpenMP uses language extensions and a threading library.
  - OpenMP functions: include <omp.h> header, link library.
  - OpenMP library: -fopenmp flag for gcc.
- **Execution:**
  - Number of threads, n, must be set:
    - Set environment variable OMP_NUM_THREADS=n.
    - Call OpenMP function omp_set_num_threads(n).

OpenMP - Usage

- **OpenMP Language Extensions:**
  - #pragma omp <keywords>
    - *parallel* - Start a threaded region
    - barrier - Synchronise threads in a parallel region
    - *for, schedule* - Work distribution
    - *default, private, firstprivate, shared* - variable scope.
    - *reduction* - reduction operation
- **OpenMP Functions:**
  - omp_get_num_threads() - Get the maximum number of threads
  - omp_set_num_threads(n) - Set the maximum number of threads
  - omp_get_thread_num() - Get the id of the current thread

OpenMP - Flow Control

- **Flow Control:**
  - *parallel, barrier, critical, atomic*
  - These deal with the parallel execution, blocking and synchronisation of threads

- **Usage:**
  - Control the creation of threads, and prevent the violation of dependencies and/or race conditions.

OpenMP - Flow Control - Keyword *parallel*

- ***parallel:***
  - Signifies the start of a parallel, threaded region.
  - Region bounded by { } following pragma declaration is threaded.

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    printf("Hello World from thread %d!\n",tid);
}
```

OpenMP - Flow Control - Keyword *barrier*

- **barrier:**
  - Prevents any one thread from continuing until all threads have reached the barrier point and sychronised.
  - Useful to prevent violation of dependencies.

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    if(tid==0) sleep(10);
    printf("I'm thread %d\n",tid);
#   pragma omp barrier
    printf("I'm thread %d after the barrier!\n");
}
```

Thread 1 will print the first statement immediately, thread 0 will print after 10 seconds.
Neither will print the second until they both reach the barrier, forcing thread 1 to wait on thread 0

OpenMP - Flow Control - Keyword *critical*

- **critical:**
  - Serialises a portion of a parallel region - only one thread may execute it at a time (all others are blocked until first thread exist region.)
  - Order in which threads enter is still non-deterministic.
  - Critical region delimited by { ... }
  - Prevents race conditions
  - Caution: negatively impacts performance, minimise use.

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
#  pragma omp critical
    {
        printf("I am thread %d\n",tid);
        printf("I am still thread %d\n",tid);
        printf("I am thread %d, about to leave\n",tid);
    }
}
```

OpenMP - Flow Control - keyword *atomic*

- **atomic:**
  - Allows the following update to be conducted atomically, preventing race conditions from multiple writing threads
  - Only applies to the immediately following statement
  - Lower overhead than critical

```
int count=0;   // shared across threads
#pragma omp parallel
{
#   pragma omp atomic
      count++;
}
printf("Thread Count:%d\n",count);
```

OpenMP - Scope

- **Variable Scope:**
  - *private, firstprivate, shared, default*
  - *reduction* - part flow control, part scope
  - Each thread has it's own private scope, plus access to shared memory.
  - Variables declared in a parallel region are private to a thread.
  - All other variables, declared outside the parallel region, must have a scope type declared if used inside the region.

OpenMP - Scope - keyword private(var1,var2...)

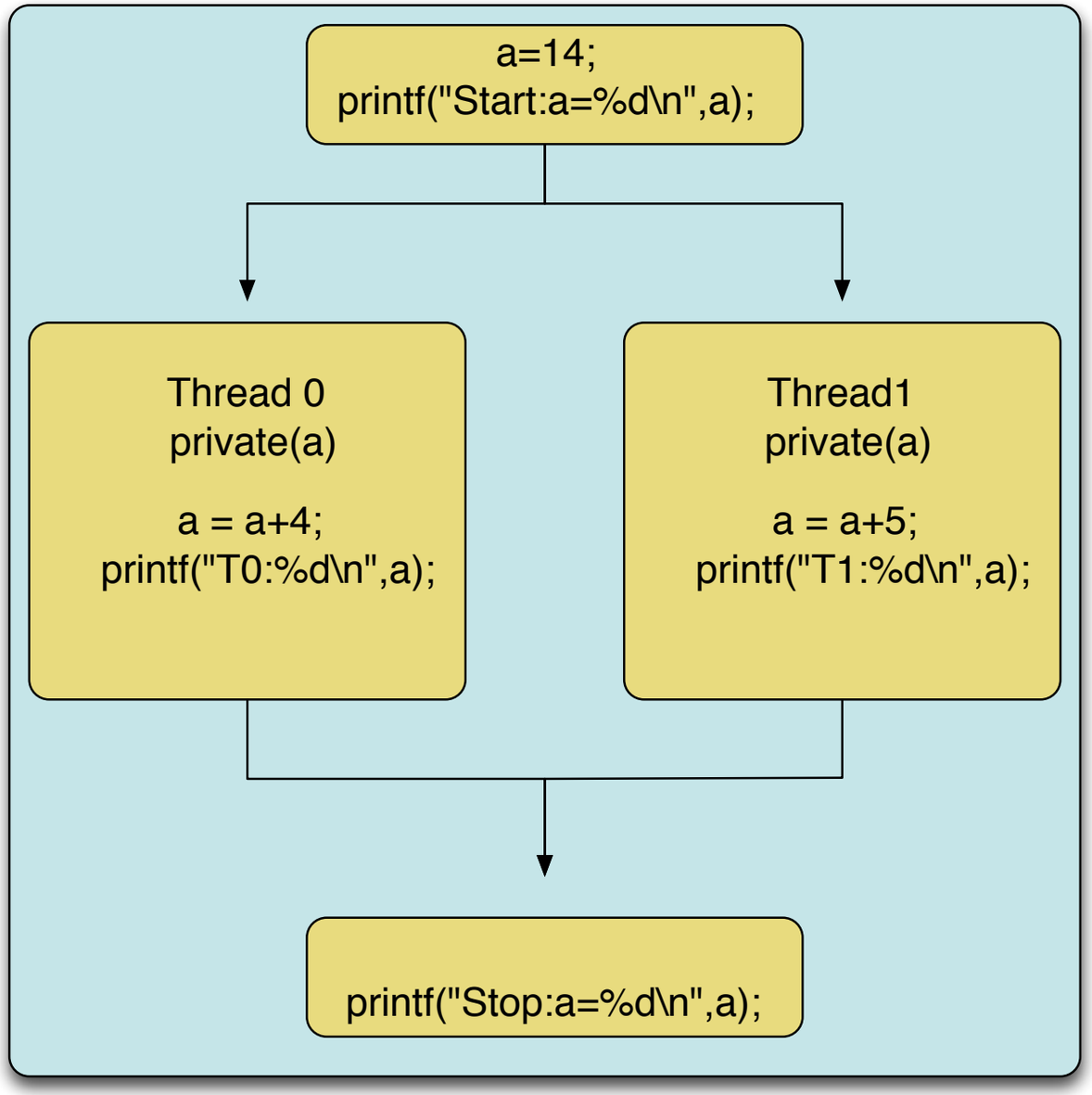
private:

Separate variable instance per thread, does not retain original value.

Usage:

- Original value does not matter - e.g. loop indexes.
- # pragma omp private(a,b) ....

Output:
Start: a=14
T0: 4
T1: 5
Stop: a = 14



OpenMP - Scope - keyword firstprivate(var1,var2...)

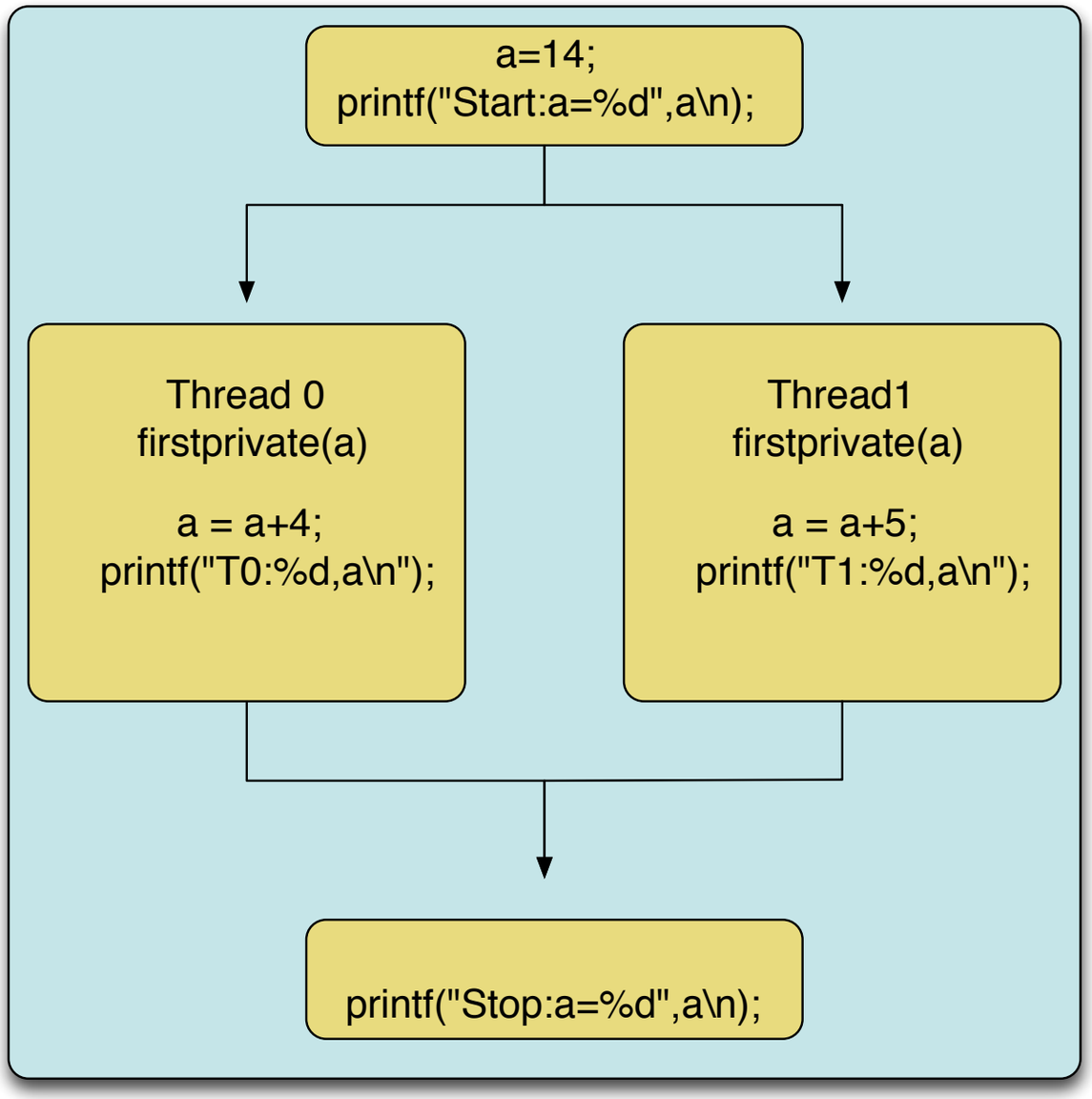
firstprivate:

Separate variable instance per thread, copies original value.

Usage:

- Original value does matter - e.g. constant calculated outside loop.
- # pragma omp firstprivate(a) ....

Output:
Start: a=14;
T0: 18;
T1: 19;
End: a = 14



OpenMP - Scope - keyword shared(var1,var2...)

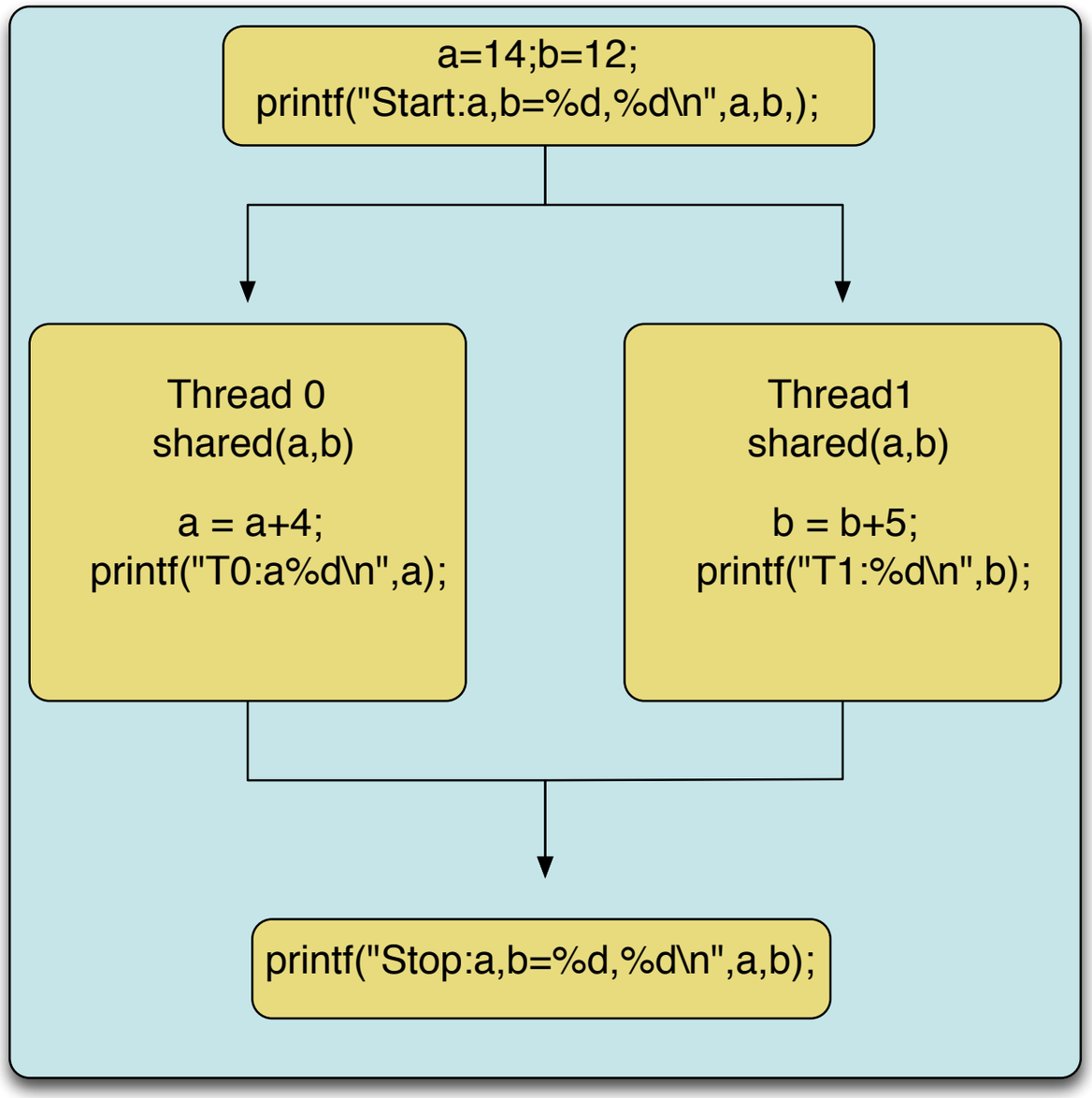
shared:

Uses shared global memory, accessible by all threads.

Usage:

- Multiple threads need to access same data
- Caution - potential for race condition
- # pragma omp shared(n) ....

Output:
Start: a,b =14,12;
T0:18;
T1:17;
End: a,b = 18,17



OpenMP - Scope - keyword default(*option)*

- ***default*** :
  - Default behaviour of variable scope for any variable declared outside a parallel region, but used within.
- Available options:
  - *default(none)* - Make no assumptions about variable scope, any variable used in a parallel region (and not declared within) must be specified as private, firstprivate or shared, else raise a compiler error.
  - *default(shared)* - Assume all variables not declared in the parallel region are shared (unless declared as otherwise)
  - no default keyword - use the behaviour of *default(shared)*
  - #pragma omp default(none)

OpenMP - Scope - keyword reduction(op:var)

- ***reduction:***
  - Process a dataset to update a single value shared across all threads.
- **Usage:**
  - A reduction operation typically processes a large dataset to produce some form of “aggregate” value - e.g. min, max, sum etc.
  - This involves the update of a shared, cumulative value - potential race condition.
  - Create a private variable for storing partial result per thread, apply *op* across all partial results at end (to minimise serialisation.)
  - The *reduction* keyword enables us to declare both the variable and operation to apply, with OpenMP handling potential race conditions.
  - #pragma omp reduction(+:sum)

OpenMP - Scope - keyword reduction(var:op)

```
#pragma omp parallel, shared(sum)
{
    sum += 2;
    sum += 3;
    sum += 4;
}
```

Race condition

```
#pragma omp parallel, shared(sum)
{
#   pragma omp critical
    {
        sum += 2;
        sum += 3;
        sum += 4;
    }
}
```

Safe, but serial

```
#pragma omp parallel, shared(sum)
{
    int sump=0; // private variable;
    sump +=2;
    sump +=3;
    sump +=4;
#   pragma omp atomic
    sum += sump;
}
```

Safe, but can be improved...

OpenMP - Scope - keyword reduction(var:op)

```
#pragma omp parallel, shared(sum)
{
    sum += 2;
    sum += 3;
    sum += 4;
}
```

Race condition

```
#pragma omp parallel, shared(sum)
{
#   pragma omp critical
    {
        sum += 2;
        sum += 3;
        sum += 4;
    }
}
```

Safe, but serial

```
#pragma omp parallel, shared(sum)
{
    int sump=0; // private variable;
    sump +=2;
    sump +=3;
    sump +=4;
#   pragma omp atomic
    sum += sump;
}
```

Safe, but can be improved...

OpenMP - Scope - keyword reduction(var:op)

```
#pragma omp parallel, reduction(+:sum)
{
    sum +=2;
    sum +=3;
    sum +=4;
}
```

Equivalent of the previous atomic implementation.

Reduction Op handles creation of private variable per thread.

Final shared value updated, applying the op specified in the reduction declaration, while preventing race condition.

Code is more readable.

OpenMP - Workload Decomposition

- **Workload Decomposition:**
  - *for, schedule*
  - Distribute work between all threads
  - Ideal goal is each thread does an equal amount of work

- **Usage:**
  - Saves the effort of manual decomposition
  - Better distribution of work at runtime (for an overhead)

omp keywords - parallel for

- **for:**
  - Shares iterations of a loop across threads
  - Distribution depends on *schedule* behaviour
  - Applies to loop following *for* statement

```
#pragma omp parallel for schedule(static) shared(n)
for(i=0;i<n;i++)
{
    a[i] = a[i] * 2;
}
```

Iteration shared amongst threads - E.g. for two threads, thread 0 will perform iterations 0 -> 4, thread 1 will perform iterations 5->9.

OpenMP keywords - schedule(type,chunksize)

- ***schedule(type, chunksize):***
  - *type* - Select one of three behaviours, static, dynamic or guided
    - static - divide the loop iterations into blocks of chunksize iterations, then assign to threads in a round-robin fashion.
    - dynamic - divide the loop iterations into blocks of chunksize iterations. Assign a new block to a thread each time it completes a block.
    - guided - like dynamic, but each subsequent chunk decreases in size.
- **Usage:**
  - Balance of overhead and data distribution.
  - If each iteration takes roughly same time, static with default chunk size.
  - If iteration time can vary, dynamic with smaller chunk size.

OpenMP - Performance

- **Threading Performance:**
  - Typically one thread per core, else each thread shares a cores resources and all run slow (are exceptions - e.g. hyperthreading.)
  - Good workload decomposition.
  - Avoiding serialising thread execution where possible.
  - Thread outer loops rather than inner loops where possible.
  - Avoiding *false sharing.*
  - NUMA.

Threading - False Sharing

- **False Sharing:**
  - A thread accesses a memory location that shares a cache-line with a memory location accessed by another thread, resulting in poor performance.
- **Why it Occurs:**
  - Each core has it's own cache.
  - One thread per core = one cache per thread.
  - *Cache-coherency* - Updating a cache-line forces an update for all other caches containing that cache-line, even if that thread does not use that value.
  - *Cache-thrashing* - If threads frequently share a cache-line, each thread's cache is frequently updating to maintain cache-coherency - performance hit.

Threading - OpenMP - False Sharing - Example

False Sharing

```
# pragma omp parallel for default(none) shared(a,result,n) private(i)
  for(i=0;i<n;i++) {
        int threadid = omp_get_thread_num();
        result[threadid] += a[i];
  }
```

No False Sharing

```
# pragma omp parallel for default(none) shared(a,result,n) private(i)
  for(i=0;i<n;i++)
  {
        int gap =omp_get_num_threads()*16;
        int threadid = omp_get_thread_num();
        result[threadid*gap] += a[i];
  }
```

Threading - NUMA

- **NUMA -** Non Uniform Memory Access
  - Architectures where each processor may have its own memory, as well as the shared memory.
  - Accessing it's own memory is faster than accessing memory belonging to another processor
  - As such, the location where memory is allocated for a processor matters (or the core to which a thread is binded matters.)