SIMD and OpenMP

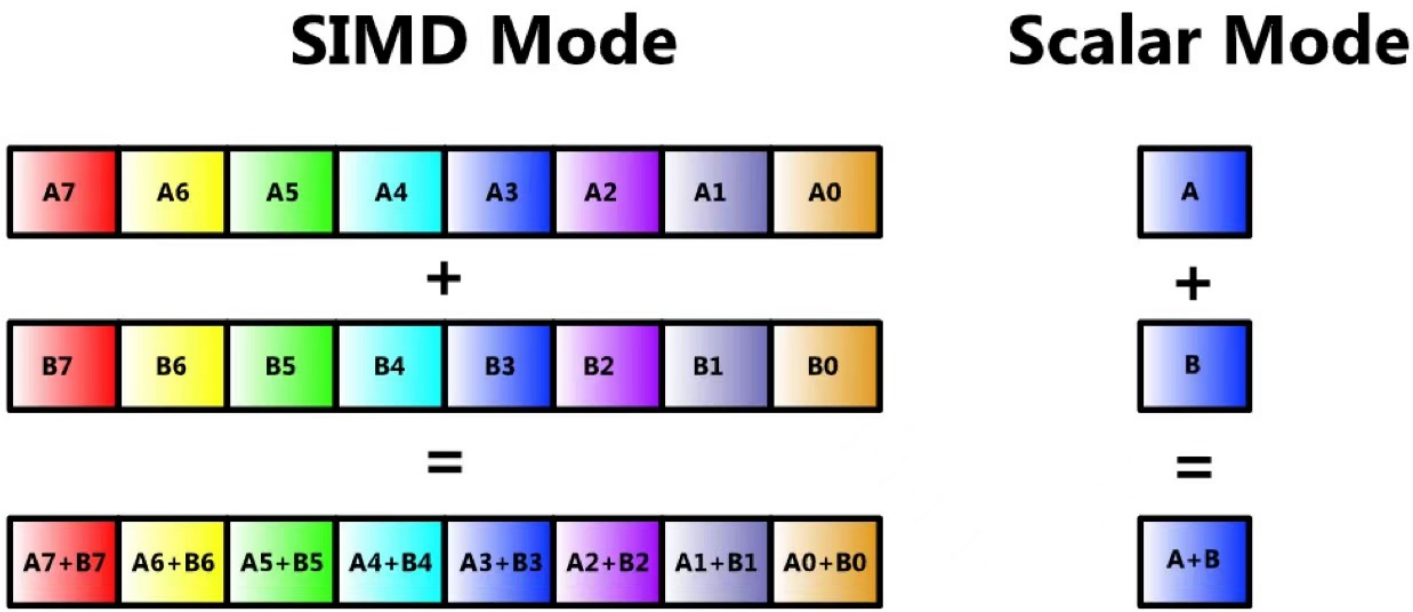
Disscussion 11

Hanjia Cui

May 9, 2024

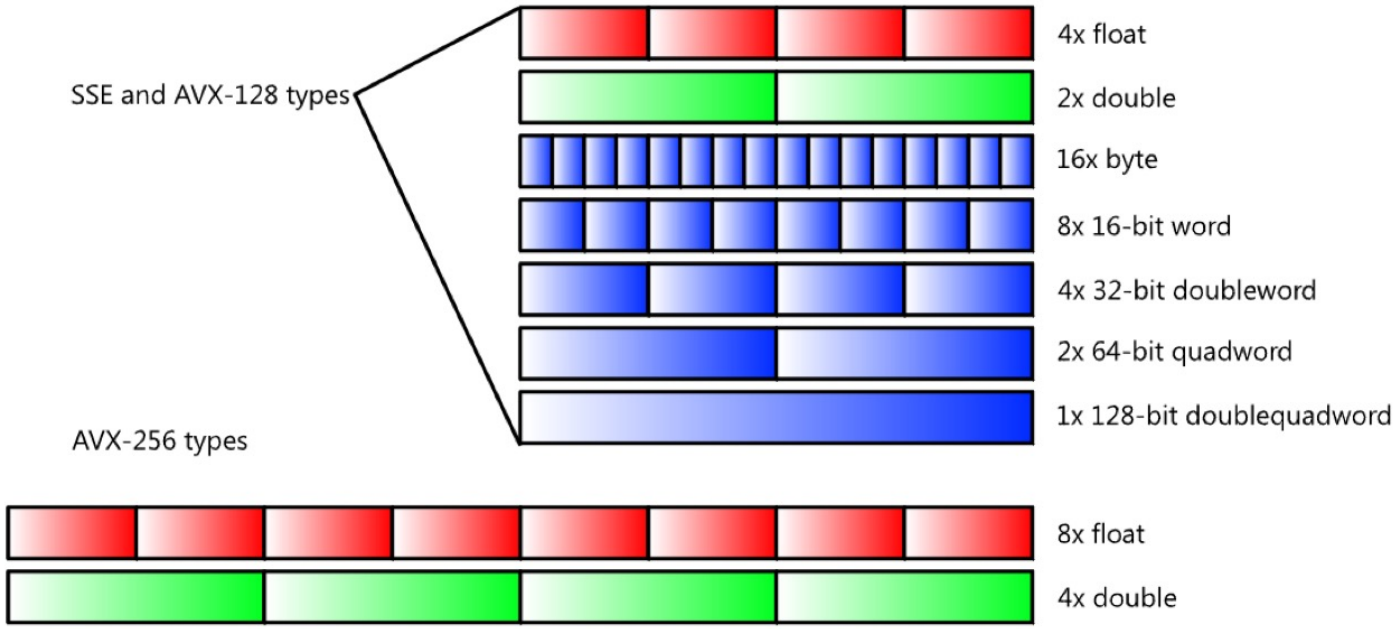
SIMD

- Single Instruction Multiple Data
- SIMD instructions allow processing of multiple pieces of data in a single step, speeding up throughput for many tasks
- SIMD extensions
  - ARM extensions - Neon
  - X86 extensions - SSE, AVX, AVX-512 etc



Vector Registers

- SSE support 128-bit XMM registers
- AVX support 256-bit YMM registers
- AVX-512 support 512-bit ZMM registers



Loop Unrolling

```
    void add(double *A, double *B, double *C, int N){
        for(int i=0; i<N; i++){
            C[i] = A[i] + B[i];
        }
    }

/* Suppose N % 4 = 0 */
void add_unrolling(double *A, double *B, double *C, int N){
    for(int i=0; i<N; i+=4){
        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];
    }
 }
```

Intrinsics

```
#include <immintrin.h>

void add_intrinsic(double *A, double *B, double *C, int N){
    for(int i=0; i<N; i+=4){
        __m256d a = _mm256_load_pd(&A[i]);
        __m256d b = _mm256_load_pd(&B[i]);
        __m256d c = _mm256_add_pd(a, b);
        _mm256_store_pd(&C[i], c);
    }
}
```

OpenMP

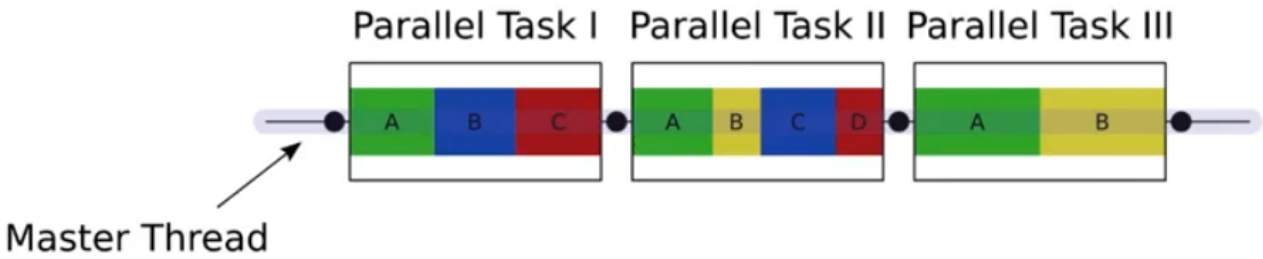
- Scalable programming model that gives parallel programmers a simple and flexible interface for developing portable parallel applications
- Fortran, C, C++ support
- Use compiler directives and library routines
  - `#include <omp.h>`
  - `#pragma omp construct [clause [clause]...]`

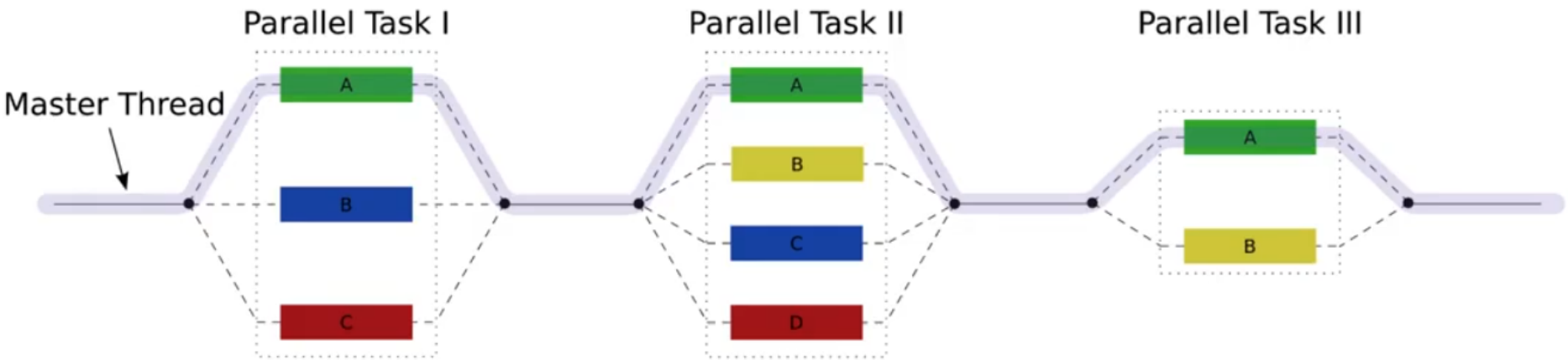
```
{
    /* Structured Block */
}
```

- Compile with flag: `-fopenmp`
- Intend to support programs that will execute correctly both as
  - parallel programs (multiple threads of execution and a full OpenMP support library)
  - sequential programs (directives ignored and a simple OpenMP stubs library)

OpenMP

- Parallelization in OpemMP employs multi-thread and shared-memory
- **fork-join model**
  - If there is parallel work, master thread forks off slave threads.
  - When slave threads finish, they merge back into matser thread.





Runtime

- Library calls
  - `omp_set_num_threads()`
    - Affects the number of threads used for subsequent `parallel` constructs not specifying a `num_threads` clause
  - `omp_get_num_threads()`
    - Returns the number of threads in the current team
  - `omp_get_thread_num()`
    - Returns the thread number of the calling thread
- Environment variables
  - `OMP_NUM_THREADS`
    - sets the number of threads to use for parallel regions

parallel construct

- Creates a team of OpenMP threads that execute the region

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

```
Hello from thread 7 out of 8
Hello from thread 3 out of 8
Hello from thread 5 out of 8
Hello from thread 6 out of 8
Hello from thread 2 out of 8
Hello from thread 4 out of 8
Hello from thread 0 out of 8
Hello from thread 1 out of 8
```

Work-sharing

- When any team encounters a worksharing construct, the work inside the construct is divided among the members of the team, and executed **cooperatively** instead of being executed by every thread.
- An **implicit barrier** occurs at the end of any region that corresponds to a worksharing construct for which the `nowait` clause is not specified.

Work-sharing

loop constructs: `for`

- Specifies that the iterations of associated loops will be executed in parallel by threads in the team.

```
int i, tid;
#pragma omp parallel for private(tid)
for(i=0; i<10; i++){
    tid = omp_get_thread_num();
    printf("Iteration %d by thread %d\n", i, tid);
}
```

```
Iteration 2 by thread 1
Iteration 3 by thread 1
Iteration 6 by thread 4
Iteration 9 by thread 7
Iteration 5 by thread 3
Iteration 0 by thread 0
Iteration 1 by thread 0
Iteration 4 by thread 2
Iteration 7 by thread 5
Iteration 8 by thread 6
```

Schedule

- `schedule(static[,chunk_size])`
  - Iterations are divided into chunks of size `chunk_size`
  - Chunks assigned in round-robin fashion
  - Default: chunks approximately equal
- `schedule(dynamic[,chunk_size])`
  - Iterations are divided into chunks of size `chunk_size`
  - Grab chunk each time finished
  - Default: `chunk_size = 1`
- `schedule(guided[,chunk_size])`
  - Start with large size of chunks and then shrink down to `chunk_size`
  - Grab chunk each time finished

Work-sharing

sections

- A non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { /* Structured Block */ }
        #pragma omp section
        { /* Structured Block */ }
    }
}
```

Work-sharing

single & master

- single associated structured block is executed by only one of the threads in the team

```
#pragma omp parallel
{
    #pragma omp single
    { /* Structured Block */ }
}
```

- master associated structured block is executed by only master thread and **no implicit barrier**

```
#pragma omp parallel
{
    #pragma omp master
    { /* Structured Block */ }
}
```

Data-sharing

`private(variable_list)`

- Creates a new variable for each item in list that is private to each thread

```
int i = 1;
#pragma omp parallel private(i)
{
    /* undefined initial value `i` */
}
```

Data-sharing

`firstprivate(variable_list)`

- Subject to the `private` clause semantics, except as noted
- Initialized from the value the original variable has at the time the construct is encountered

```
int i = 1;
int *ptr_i = &i;
#pragma omp parallel firstprivate(i)
{
    assert(i == 1);
    i = 2;
    assert(*ptr == 1);
}
assert(i == 1);
```

Data-sharing

lastprivate(variable_list)

- Subject to the `private` clause semantics, except as noted
- Values of the variables are the same as when the loop is executed sequentially

```
int k = 0;
#pragma omp parallel for lastprivate(k)
for(int i=1; i<=100; i++){
    if(i % 33 == 0){
        k = i;
    }
}
assert(k == 99);
```

Data-sharing

shared(variable_list)

- All references to a shared variable within a task refer to the storage area of the original variable at the point the directive was encountered.

```
/* Sum of 1, 2, ..., 10000000 */
double sum = 0;
double start = omp_get_wtime( );
#pragma omp parallel for shared(sum)
for(int i=1; i<=10000000; i++){
    sum += i;
}
double end = omp_get_wtime( );
printf("Sum: %.1f in %fs\n", sum, end-start);
```

```
Sum: 6654243197488.0 in 0.050225s
```

critical

- Restricts execution of the associated structured block to a single thread at a time.

```
/* Sum of 1, 2, ..., 10000000 */
double sum = 0;
double start = omp_get_wtime( );
#pragma omp parallel for shared(sum)
for(int i=1; i<=10000000; i++){
#pragma omp critical
    sum += i;
}
double end = omp_get_wtime( );
printf("Sum: %.1f in %fs\n", sum, end-start);
```

```
Sum: 50000005000000.0 in 1.625899s
```

Data-sharing

`reduction(op : variable_list)`

- Private copy is created for each thread and is initialized with the initializer value of the specified operator
- Combining original value with the final value of each private copies using the combiner of the specified operator
- Operators: `+`, `*`, `&`, `|`, `^`, `&&`, `||`

```
/* Sum of 1, 2, ..., 10000000 */
double sum = 0;
double start = omp_get_wtime( );
#pragma omp parallel for reduction(+ : sum)
for(int i=1; i<=10000000; i++){
    sum += i;
}
double end = omp_get_wtime( );
printf("Sum: %.1f in %fs\n", sum, end-start);
```

```
Sum: 50000005000000.0 in 0.051570s
```

Reference

- Intel Intrinsics
- OpenMP Specification
- Slides from ShanghaiTech CS110
- Slides from ShanghaiTech CS121
- Slides from MIT 12.950