<- This means this lab is on the hive

Lab 7: Parallelism

CS 61C Fall 2024

Contents

- Loop unrolling
- SIMD
- SIMD functions

Loop Unrolling

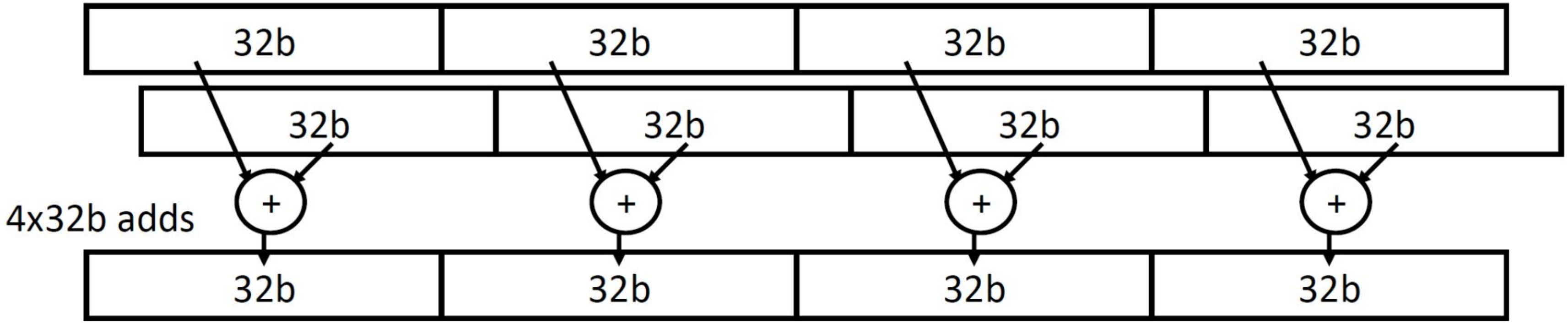
```
for(int i = 0; i < max; i++)
{
    arr[i] = i * i;
}
```

```
for(int i=0; i < max/4*4; i+=4)
{
    arr[i] = i * i;
    arr[i+1] = (i+1) * (i+1);
    arr[i+2] = (i+2) * (i+2);
    arr[i+3] = (i+3) * (i+3);
}
for(int i=max/4*4; i < max; i++)
{
    arr[i] = i * i;
}
```

tail case

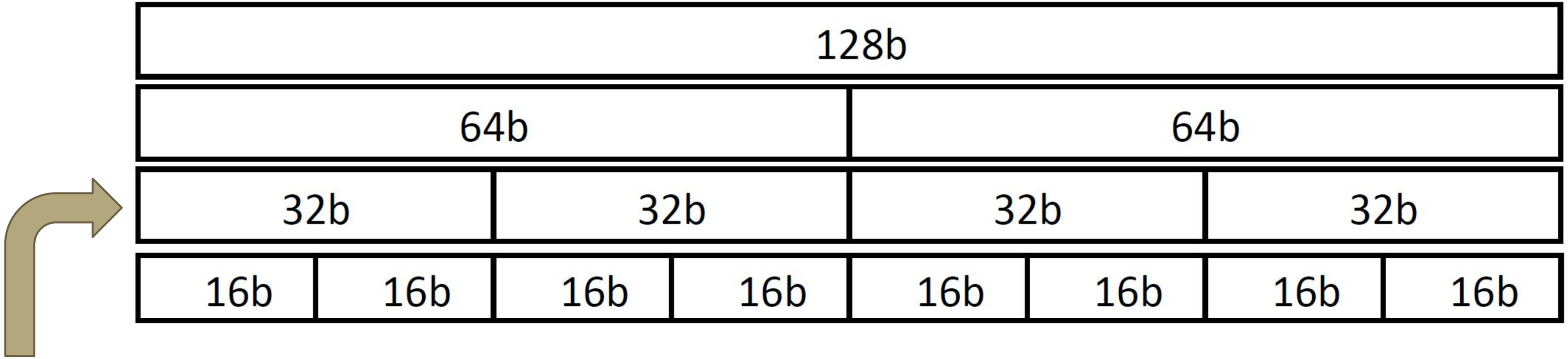
What is SIMD?

- Single Instruction, Multiple Data
- Data are packed together then same operations can be done in parallel



What is SIMD?

- Single Instruction, Multiple Data
- Data are packed together then same operations can be done in parallel

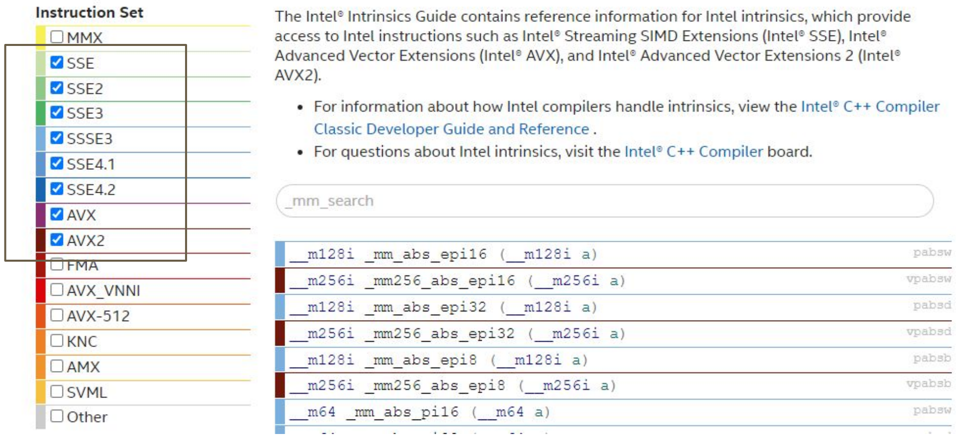


4 32-bit integers combined to form 128-bits

SIMD Functions

- For this class, we'll use Intel's SIMD instructions

Supported by hive machines



List of all possible functions that you can use for SIMD operations

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL&avxnewtechs=AVX,AVX2

Some SIMD Functions

- `__m128i _mm_setzero_si128()`
  - returns a 128-bit zero vector
- `__m128i _mm_loadu_si128(__m128i *p)` ← *remember to cast
  - returns 128-bit vector stored at pointer p
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`
  - returns vector ($a_0 + b_0$, $a_1 + b_1$, $a_2 + b_2$, $a_3 + b_3$)
- `void _mm_storeu_si128(__m128i *p, __m128i a)`
  - stores 128-bit vector a into pointer p

Careful of the number of '_' in your code! (__m128i has 2 underscores)

Other Useful SIMD Functions for This Lab

- `__m128i _mm_cmpgt_epi32(__m128i a, __m128i b)`
  - returns the vector `(a_i > b_i ? 0xffffffff : 0x0 for i from 0 to 3)` .
- `__m128i _mm_and_si128(__m128i a, __m128i b)`
  - returns vector `(a_0 & b_0, a_1 & b_1, a_2 & b_2, a_3 & b_3)` , where & represents the bitwise and operator

OpenMP

- **O**pen specification for **m**ulti**p**rocessing
- Enables us to easily parallelize code
- Invoked using compiler directives

OMP Example

declares that the directive is for OpenMP

says that the following block should be executed in parallel by different threads

Tells the compiler that this is a compiler directive

```
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("hello world from thread %d\n", thread_id);
    }
}
```

Every single thread is going to execute this block!

Vector Addition

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel
    {
        for(int i=0; i<ARRAY_SIZE; i++)
        {
            z[i] = x[i] + y[i];
        }
    }
}
```

Every single thread is going to execute this loop!

This is not what we want - we want the threads to split up the work of the loop

Vector Addition

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel for
    for(int i=0; i<ARRAY_SIZE; i++)
    {
        z[i] = x[i] + y[i];
    }
}
```

This will split up the loop for us

Useful OMP Functions

- There are several ways to parallelize for loops
  - You can use `#pragma omp parallel for`
  - You can use `#pragma omp for` within a `#pragma omp parallel {}` block
- Useful functions
  - `int omp_get_num_threads()` - returns the current total number of OpenMP threads. Note that the number of threads will be 1 outside of an OpenMP parallel section
  - `int omp_get_thread_num()` - returns the thread number of the current thread, commonly used as thread ID

Synchronization

- Sometimes our threads need to write to the same location
- If multiple threads try to write to the same location at the same time, it will lead to a **data race**
    - The order of accesses is non-deterministic which can lead to different results each time you execute the program

```
double dotp_race(double* x, double* y, int arr_size) {
    double global_sum = 0.0;
    #pragma omp parallel for
        for (int i = 0; i < arr_size; i++) {
            global_sum += x[i] * y[i];
        }
    return global_sum;
}
```

What's the problem here?

Each spawned thread can overwrite the global_sum values written by other threads

Return value will be wrong!

Synchronization

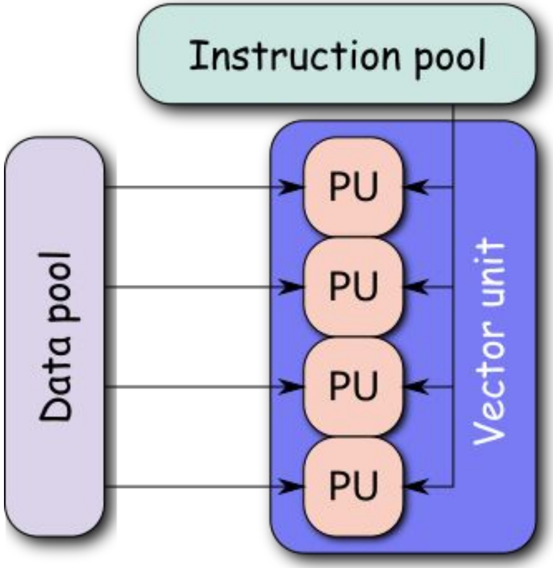
- OMP provides two methods to deal with this
  - `#pragma omp critical`
    - only one thread can execute this section at a time
  - `#pragma omp parallel for reduction(+:var_name)`
    - Whenever you execute this operation on the given variable, accumulation occurs into a private copy of var_name which is then combined with the original var_name.

Data Race Case Study

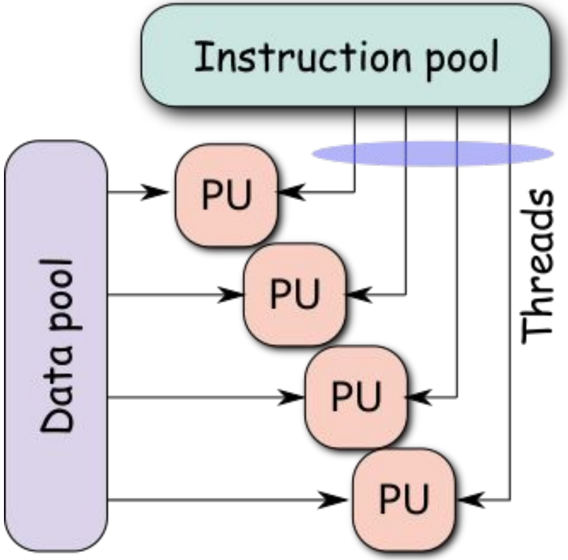
- Therac-25 was a machine to deliver radiation to a human body
- It killed multiple people due to software bugs
- These bugs included (among many others)
  - Data race conditions
  - Not having atomic read-writes

Take CS195 or CS162 to learn more!!!

Data-level (SIMD) vs Thread-level (OpenMP) Parallelism



- 1 core, parallel ALUs



- >1 thread, 1 ALU/thread
- Threads can run on different cores

https://en.wikipedia.org/wiki/Single_instruction,_multiple_data