Exploiting Parallelism

- Of the computing problems for which performance is important, many have inherent parallelism.

- E.g., computer games:
  - graphics, physics, sound, A.I. etc. can be done separately
  - Furthermore, there is often parallelism within each of these:
    - Each pixel on the screen's color can be computed independently
    - Non-contacting objects can be updated/simulated independently
    - Artificial intelligence of non-human entities done independently

- E.g., Google queries:
  - Every query is independent
    - Google searches are read-only!!

Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

single instruction
multiple data

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

- You could write assembly for this, something like:

```
lw    $t0, 0($a0)
lw    $t1, 0($a1)
add   $t0, $t1, $t2
sw    $t2, 0($a2)
```

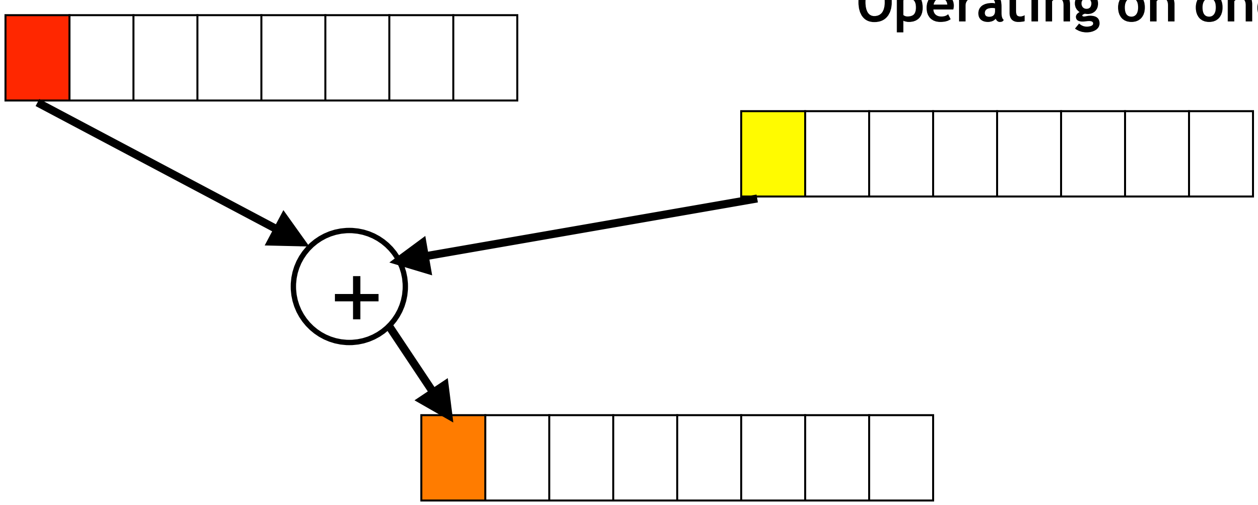
(plus all of the address arithmetic, plus the loop control)

Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

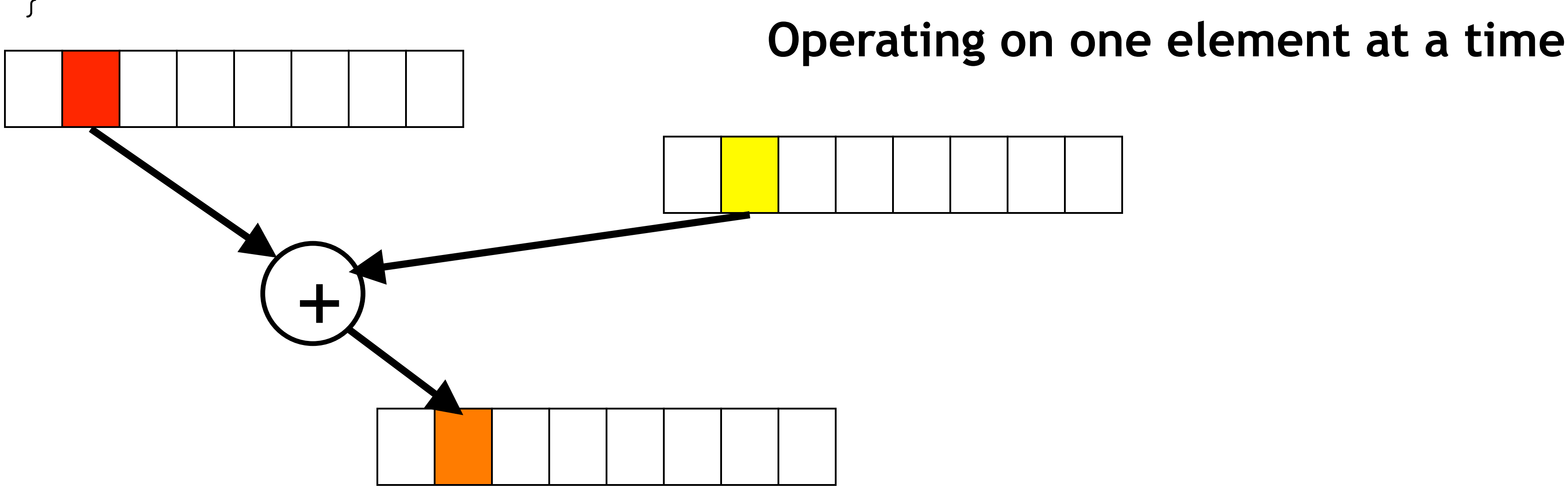
Operating on one element at a time



Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

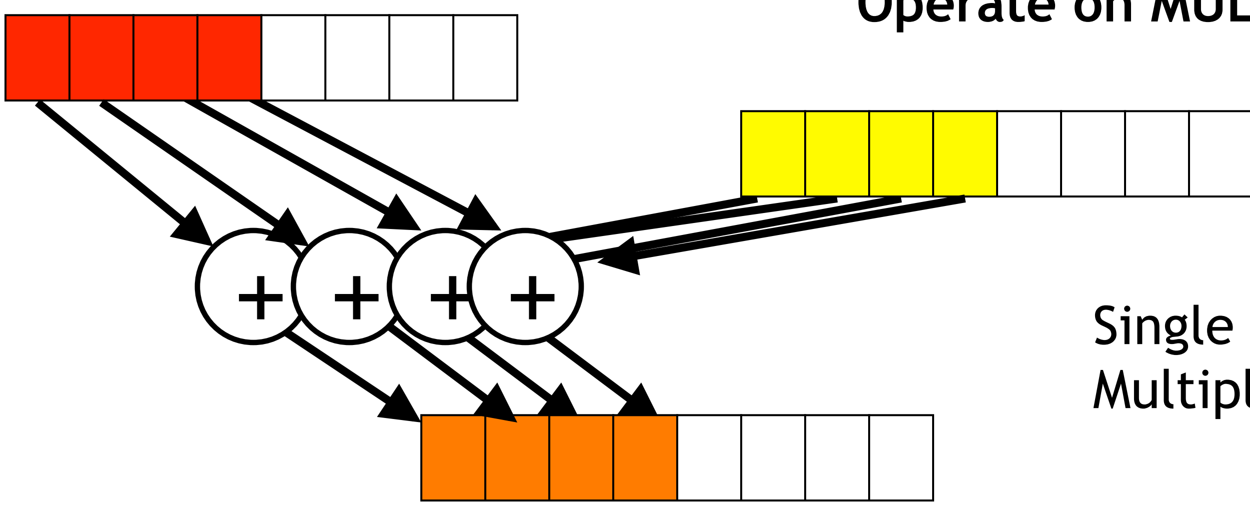


Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

Operate on MULTIPLE elements



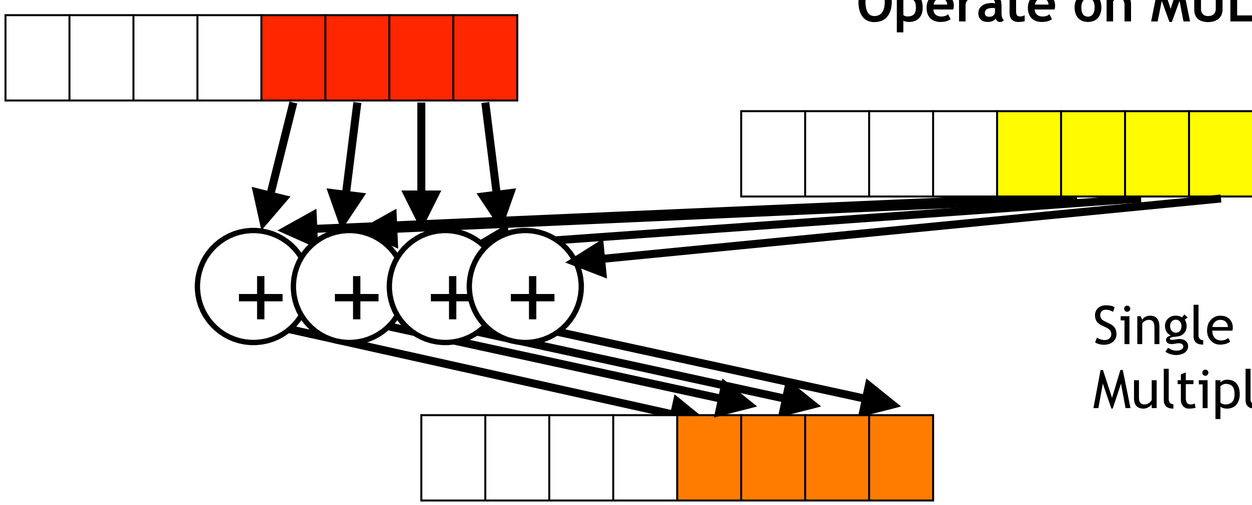
Single Instruction,
Multiple Data (**SIMD**)

Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
   C[i] = A[i] + B[i];
  }
}
```

Operate on MULTIPLE elements



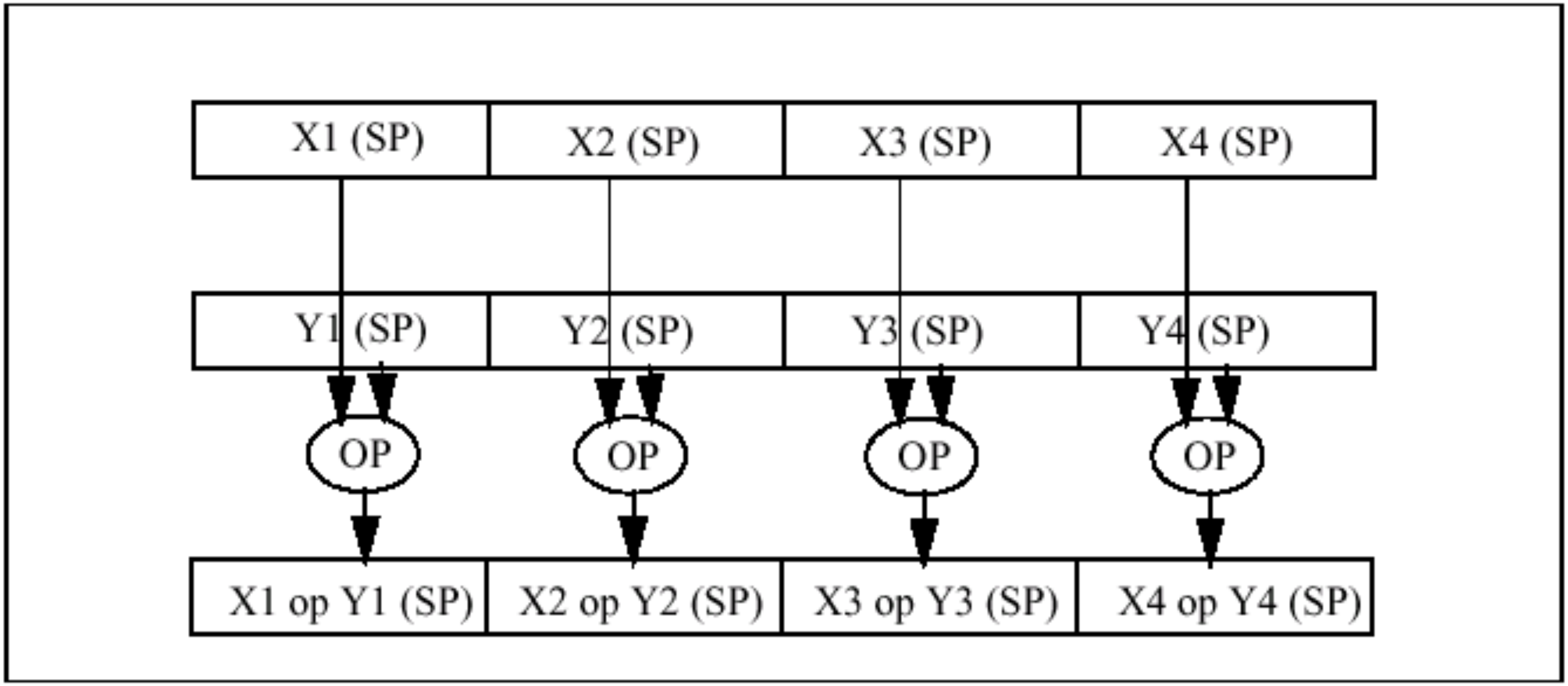
Single Instruction,
Multiple Data (**SIMD**)

Intel SSE/SSE2 as an example of SIMD

- Added new 128 bit registers (XMM0 – XMM7), each can store
  - 4 single precision FP values (SSE) 4 * 32b
  - 2 double precision FP values (SSE2) 2 * 64b
  - 16 byte values (SSE2) 16 * 8b
  - 8 word values (SSE2) 8 * 16b
  - 4 double word values (SSE2) 4 * 32b
  - 1 128-bit integer value (SSE2) 1 * 128b

	4.0 (32 bits)	4.0 (32 bits)	3.5 (32 bits)	−2.0 (32 bits)
+	−1.5 (32 bits)	2.0 (32 bits)	1.7 (32 bits)	2.3 (32 bits)
	2.5 (32 bits)	6.0 (32 bits)	5.2 (32 bits)	0.3 (32 bits)

SIMD Extensions



Packed Operations

More than 70 instructions. Arithmetic Operations supported: Addition, Subtraction, Mult, Division, Square Root, Maximum, Minimum. Can operate on Floating point or Integer data.

Annotated SSE code for summing an array

%eax = A
%ebx = B
%ecx = C
%edx = i

mov = data movement
dq = double-quad (128b)
a = aligned

A + 4*i

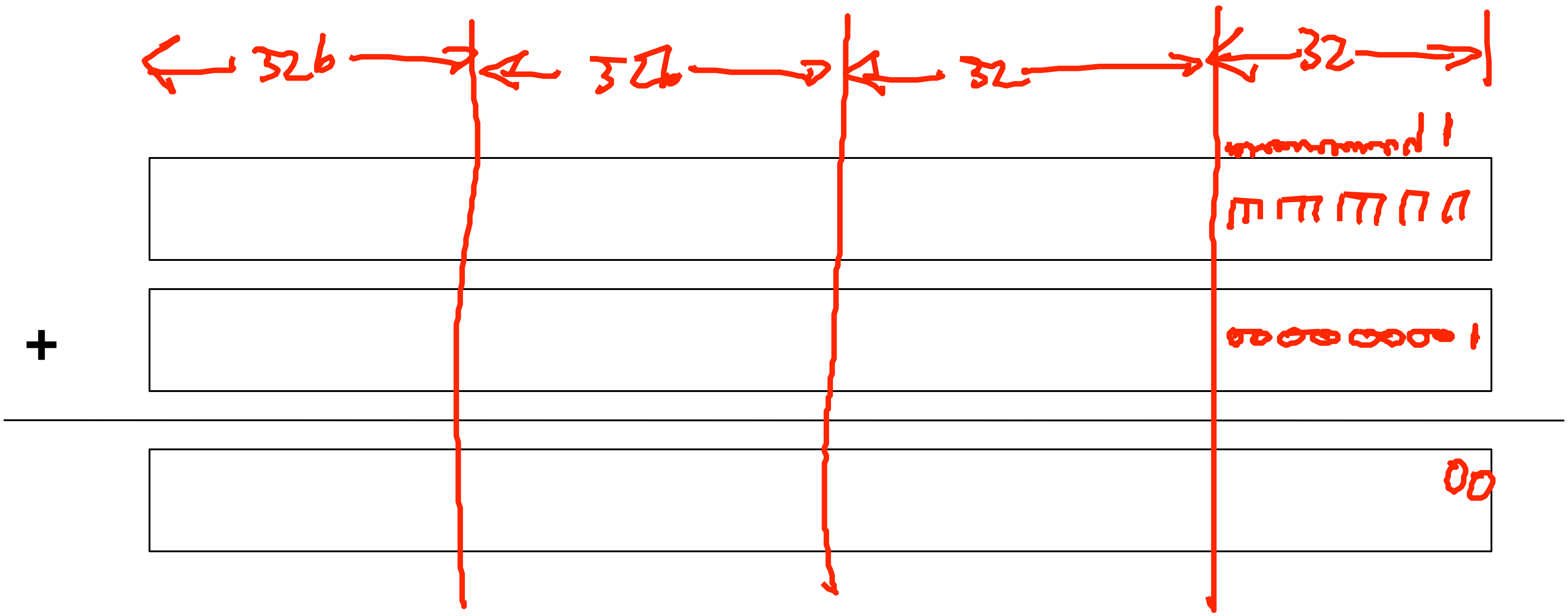
```
movdqa    (%eax,%edx,4), %xmm0      # load A[i] to A[i+3]
movdqa    (%ebx,%edx,4), %xmm1      # load B[i] to B[i+3]
paddd     %xmm0, %xmm1          # CCCC = AAAA + BBBB
movdqa    %xmm1, (%ecx,%edx,4)      # store C[i] to C[i+3]
addl      $4, %edx              # i += 4
(loop control code)
```

16b

SIMD

32b

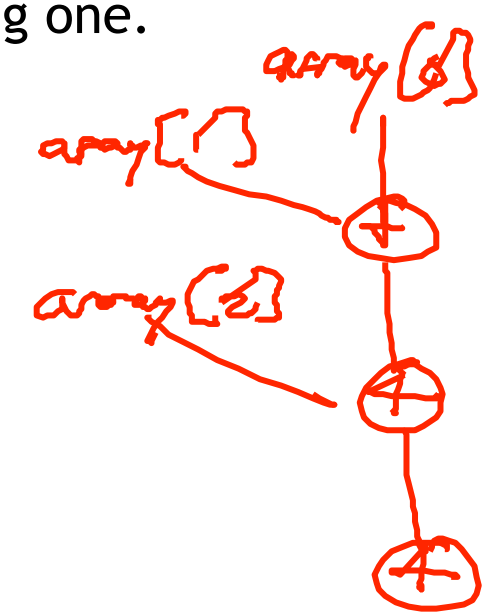
p = packed
add = add
d = double (i.e., 32-bit integer) why?


32b
32b
32
32
+

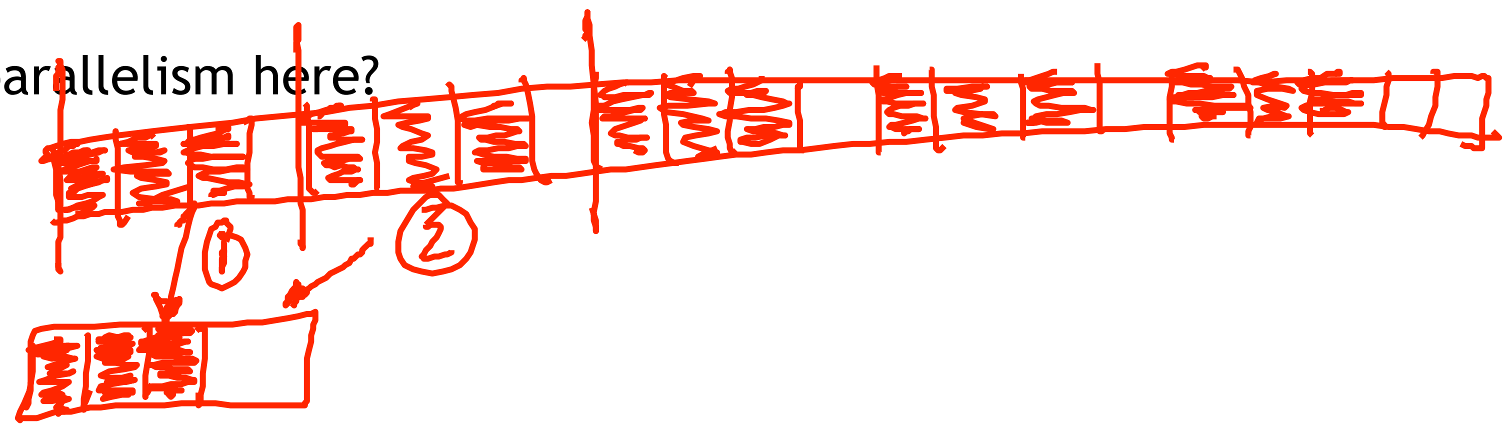
Is it always that easy?

- No. Not always. Let's look at a little more challenging one.

```
unsigned
sum_array(unsigned *array, int length) {
  int total = 0;
  for (int i = 0 ; i < length ; ++ i) {
      total += array[i];
  }
  return total;
}
```



- Is there parallelism here?



total

Exposing the parallelism

```
unsigned
sum_array(unsigned *array, int length) {
  int total ~~= 0;~~ [4] = {0,0,0,0};

                                        i = i+4
  for (int i = 0 ; i < length ; ~~++ i~~) {
     total[0] += array[i];
     total[1] += array[i+1];
     total[2] += array[i+2];

3
  return total;
}
```

We first need to restructure the code

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

Independent

Then we can write SIMD code for the hot part

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

Summary

- Performance is of primary concern in some applications
  - Games, servers, mobile devices, super computers

- Many important applications have parallelism
  - Exploiting it is a good way to speed up programs.

- Single Instruction Multiple Data (SIMD) does this at ISA level
  - Registers hold multiple data items, instruction operate on them
  - Can achieve factor or 2, 4, 8 speedups on kernels
  - May require some restructuring of code to expose parallelism
    - Create temporary vectors, which are then reduced
    - Deal with remainder of array (if not evenly divisible)