
COMP 364/464

High Performance Computing

Lecture 4:

Introduction to SIMD parallelism

Today's Lecture

- Conceptual understanding of SIMD
- Practical understanding of SIMD in the context of multimedia extensions
- Slide source:
 - Mary Hall, Parallel Programming, U. of Utah, Fall 2011, Lectures 11 & 12, <http://www.cs.utah.edu/~mhall/cs4961f11/>, (and sources within)

Review: SIMD Paradigm and history

- Single Instruction Multiple Data ... SIMD
- All processing elements (PE's) execute the same instruction (+, %, <) on private data concurrently and in lock-step.
 - One instruction controller, many workers.
 - Some workers can be deactivated for certain instructions: instruction or processor mask.
- Original data parallel machines used arrays of simple PE's and were designed to compute vector and matrix operations efficiently.
 - 1-d or 2-d network of PE's matches vector or matrix shape.
 - PE's could communicate (shift) private data to neighbors in lattice.

SIMD Processing Performance

- How fast is a SIMD operation?
- Legacy example:
 - 1024 PE's in 32^2 arrangement.
 - Each PE adds a pair of integers in $1\ \mu\text{s}$ (1 MHz)
 - What is the performance to add two vectors of 1024 elements?

$$\text{Throughput} = \frac{1024 \text{ operations}}{10^{-6} \text{ sec / operation}} = 1.024 \times 10^9 \text{ ops / sec} \\ \dots 1 \text{ GFLOP/s}$$

Overview of SIMD Programming

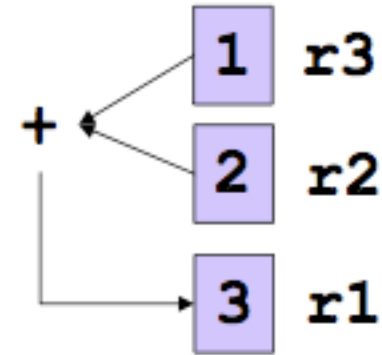
- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
 - Multimedia extensions such as SSE/AVX/AVX512 and AltiVec
 - Graphics and games processors (GPUs, “Accelerators”)
- Is there a dominant SIMD programming model
 - Unfortunately not!
- Why not?
 - Vector architectures were programmed by scientists
 - Multimedia extension architectures are programmed by systems programmers (almost assembly language!)
 - GPUs are programmed by games developers (domain- specific libraries)

Why SIMD?

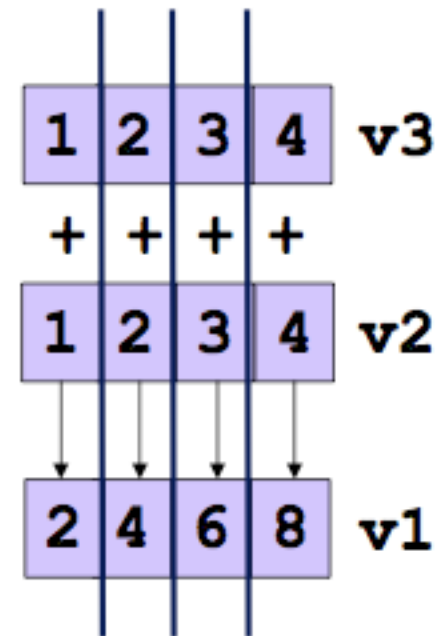
- Expose more parallelism
 - Good option when DATA parallelism is abundant
 - SIMD can be used with instruction level parallelism (ILP)
- Simple design:
 - replicate functional units
 - one control unit for many functional units
- Lower power: fewer instruction decodes
- Must be explicitly exposed to the hardware (unlike ILP).
- Programmed explicitly by developer or generated by compiler.

Scalar vs. SIMD in Multimedia Extensions

Scalar: `add r1,r2,r3`

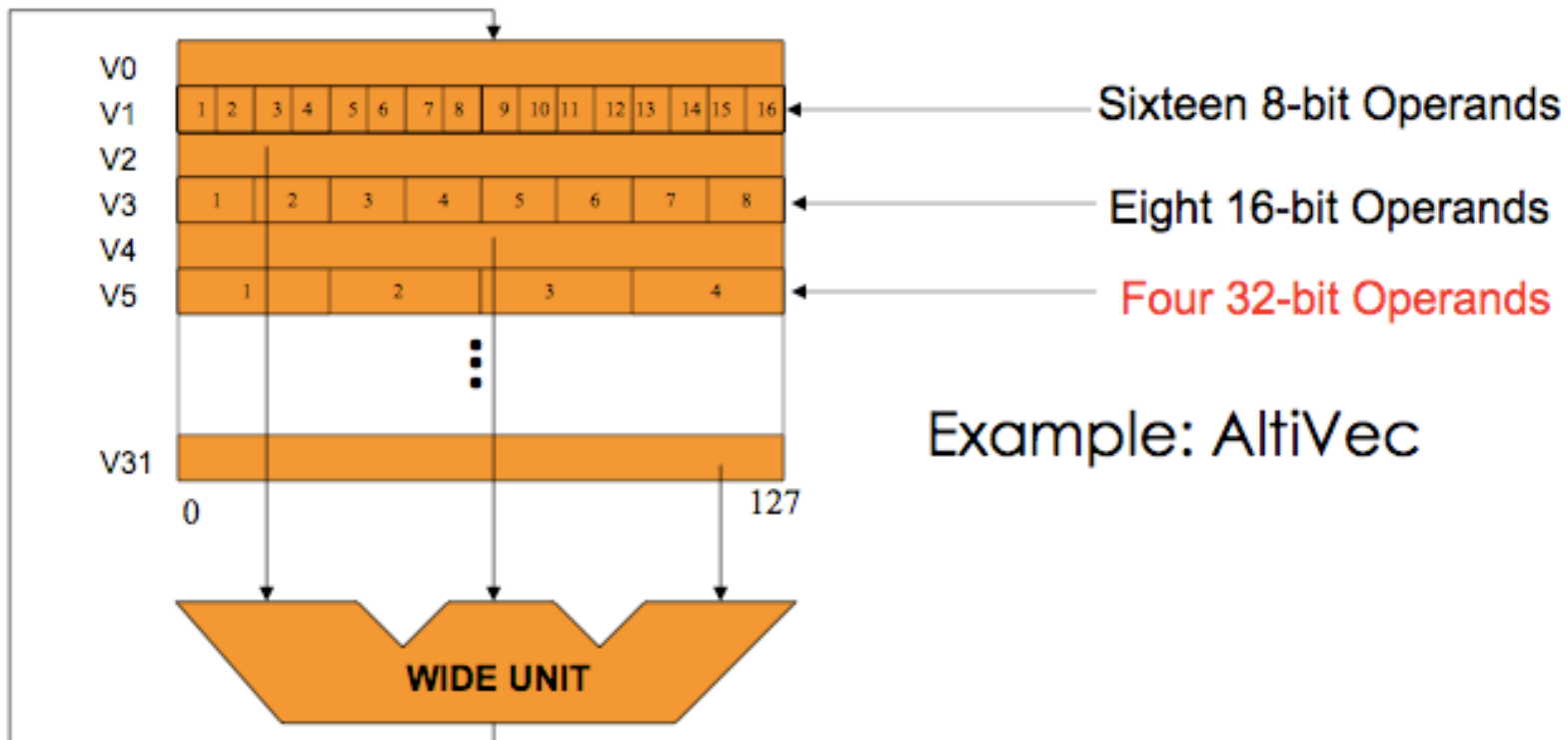


SIMD: `vadd<sws> v1,v2,v3`



MMX / SSE / AVX / AltiVec

- At the core of multimedia extensions (MMX) and Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)
 - SIMD parallelism
 - Variable-sized data fields:
 - Vector length = register width / type size



Multimedia / Scientific Applications

- Image

- Graphics : 3D games, movies
- Image recognition
- Video encoding/decoding : JPEG, MPEG4

- Sound

- Encoding/decoding: VoIP phone, MP3
- Speech recognition
- Digital signal processing: Cell phones

- Scientific applications

- Array-based data-parallel computation, another level of parallelism
- Sequence matching (another array view)

Characteristics of MMX/SSE Applications

- Regular data access pattern
 - Data items are contiguous in memory
- Short data types
 - 8, 16, 32 bits
 - Later support for 64-bit and even 128-bit words
- Data streaming through a series of processing stages
 - Some temporal reuse for such data streams

Programming MMX/SSE

- Language extensions
 - Programming interface similar to function call
 - C/C++: built-in functions
 - Most C compilers support their own extensions
 - GCC: `-faltivec`, `-msse2`, `-mavx`
 - But are platform dependent!!!
 - AltiVec: `dst = vec_add(src1, src2);`
 - SSE: `dst = _mm_add_ps(src1, src2);`
 - BG/L: `dst = __fpadd(src1, src2);`
 - Easier to program if wrapped within a C++ class with overloaded operators. (But may not be as efficient.)
- Only known standard is OpenCL (introduced in '08)
 - Permits native operations

```
float4 a, b, c;  
c = a + b;
```

Rest of Lecture

1. Understanding SIMD execution
2. Understanding the overheads
3. Understanding how to write code to deal with the overheads

Note:

A few people write very low-level code to access this capability.

Today the compilers do a pretty good job, at least on relatively simple codes.

When we study CUDA (GPUs) later, we will see a higher-level notation for SIMD execution on a different architecture ... SIMT.

Exploiting SLP with SIMD Execution

- Benefit:
 - Multiple ALU ops on individual words → One SIMD op on superword (data parallelism – superword level parallelism)
 - Multiple load/store ops → One wide mem op per superword
- What are the overheads:
 - Packing and unpacking:
 - Pack (gather) data so that it is contiguous in memory (i.e., vec register).
 - Unpack (scatter) data back to destination.
 - Alignment overhead
 - Accessing data from the memory system so that it is aligned to a “superword” boundary
 - Control flow
 - Control flow may require executing all paths
 - Remainder / Peel
 - Loop limits may not be multiple of superword length.

1. Independent ALU Ops

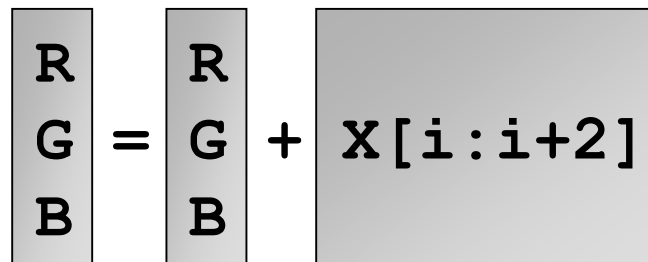
$$\begin{aligned} R &= R + XR * 1.08327 \\ G &= G + XG * 1.89234 \\ B &= B + XB * 1.29835 \end{aligned}$$



$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} XR \\ XG \\ XB \end{bmatrix} * \begin{bmatrix} 1.08327 \\ 1.89234 \\ 1.29835 \end{bmatrix}$$

2. Adjacent Memory References

```
R = R + X[i+0]  
G = G + X[i+1]  
B = B + X[i+2]
```



3. Vectorizable Loops

```
for (j = 0; j < 100; j += 1)  
    A[j] = A[j] + B[j]
```


3. Vectorizable Loops

```
for (j = 0; j < 100; j += 4)
    A[j] = A[j] + B[j]
    A[j+1] = A[j+1] + B[j+1]
    A[j+2] = A[j+2] + B[j+2]
    A[j+3] = A[j+3] + B[j+3]
```



```
for (j = 0; j < 100; j += 4)
    A[j:j+3] = B[j:j+3] + C[j:j+3]
```

4. Partially Vectorizable Loops

```
for (j = 0; j < 16; j += 1)
    L = A[j+0] - B[j+0]
    D = D + abs(L)
```

4. Partially Vectorizable Loops

```
for (j = 0; j < 16; j += 2)
    L = A[j+0] - B[j+0]
    D = D + abs(L)
    L = A[j+1] - B[j+1]
    D = D + abs(L)
```



```
for (j = 0; j < 16; j += 2)
    

|    |
|----|
| L0 |
| L1 |

 = 

|          |
|----------|
| A[j:j+1] |
|----------|

 - 

|          |
|----------|
| B[j:j+1] |
|----------|


    D = D + abs(L0)
    D = D + abs(L1)
```

4. Partially Vectorizable Loops

```
D0 = 0; D1 = 0;
for (j = 0; j < 16; j += 2)
    L = A[j+0] - B[j+0]
    D0 = D0 + abs(L)
    L = A[j+1] - B[j+1]
    D1 = D1 + abs(L)
D = D0 + D1;
```

```
DV[2] = {0,0};
for (j = 0; j < 16; j += 2)
    L[:] = A[j:j+1] - B[j:j+1]
    DV[:] = DV[:] + vabs(L[:])
D = DV[0] + DV[1];
```

Programming Complexity Issues

- High level: Use compiler
 - may not always be successful
- Low level: Use intrinsic functions or inline assembly
 - tedious and error prone ... but powerful.
- Data must be aligned, and adjacent in memory
 - Unaligned data may produce incorrect results
 - May need to copy to get adjacency (gather/scatter overhead)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked

Packing/Unpacking Costs

$$\begin{array}{l} C = A + 2 \\ D = B + 3 \end{array}$$



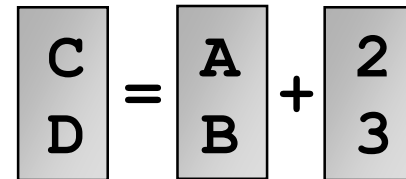
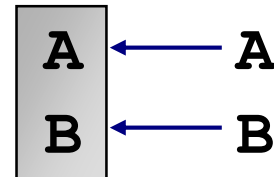
$$\begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} A \\ B \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Packing/Unpacking Costs

- Packing source operands
 - Copying into contiguous memory

A = f()
B = g()

C = A + 2
D = B + 3



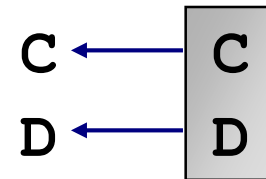
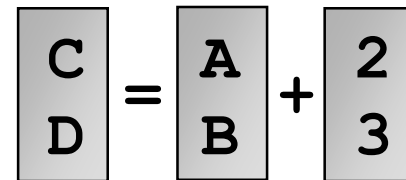
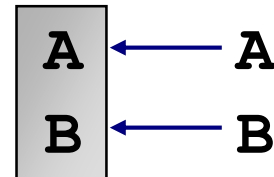
Packing/Unpacking Costs

- Packing source operands
 - Copying into contiguous memory
- Unpacking destination operands
 - Copying back to location

A = **f** ()
B = **g** ()

C = **A** + 2
D = **B** + 3

E = **C** / 5
F = **D** * 7



Alignment Code Generation

- Aligned memory access
 - The address of the superword is a multiple of 16b for SSE
 - 32b for AVX, or 64b for AVX512
 - Just one superword load or store instruction

```
float a[64];  
for (i = 0; i < 64; i += 4)  
    Va = a[i:i+3];
```

0	4	8	12	16	20	24	28	32	36	40	44	48	...
---	---	---	----	----	----	----	----	----	----	----	----	----	-----

Alignment Code Generation (cont.)

- Misaligned memory access

- The address is always a non-zero constant offset away from the 16 byte boundaries.
- Static alignment: For a misaligned load, issue two adjacent aligned loads followed by a merge.
 - All is not lost: can reuse V2 on the next iteration but still have to merge the two superwords together.

```
float a[64];  
for (i = 0; i < 60; i += 4)  
    Va = a[i+2:i+5];
```



```
float a[64];  
for (i = 0; i < 60; i += 4)  
    V1 = a[i:i+3];  
    V2 = a[i+4:i+7];  
    offset = 8; // bytes  
    Va = merge(V1, V2, offset);
```



Alignment Code Generation (cont.)

- Unaligned memory access

- The offset from 16 byte boundaries is varying or not enough information is available to the compiler.
- Dynamic alignment: The merging point is computed during run time.

```
float a[64];  
start = read();  
for (i = start; i < 60; i++)  
    Va = a[i:i+3];
```



```
float a[64];  
start = read();  
for (i = start; i < 60; i++)  
    V1 = a[i:i+3];  
    V2 = a[i+4:i+7];  
    offset = (&a[i]) % 16;  
    Va = merge(V1, V2, offset);
```

Summary of dealing with alignment issues

- Worst case is dynamic alignment based on address calculation (previous slide)
- Compiler (or programmer) can use analysis to prove data is already aligned
 - We know that data is initially aligned at its starting address by convention.
 - If we are stepping through a loop with a constant starting point and accessing the data sequentially, then it preserves the alignment across the loop.
- We can adjust computation to make it aligned by having a sequential portion until aligned, followed by a SIMD portion, possibly followed by a sequential cleanup.
 - Peeled loop, main SIMD loop, Remainder loop
- Sometimes alignment overhead is so significant that there is no performance gain from SIMD execution

Last SIMD issue: Control Flow

- What if I have control flow?
 - Both control flow paths must be executed.
 - Could be more expensive than sequential version.
- 1. Compute predicate (mask) of logical test.
- 2. Form solution for both true and false conditions.
- 3. Blend (select) the proper solutions using predicate.
 - “where true, select A, else select B.”
 - Masks are often bit sequences equal to superword length but may be byte or word sequences.

```
for (i = 0; i < 16; i++)  
    if (a[i] != 0)  
        b[i]++;
```



```
for (i = 0; i < 16; i += 4)  
    vzero = {0,0,0,0};  
    vmask = (a[i:i+3] != vzero);  
    v0 = b[i:i+3];  
    v1 = v0 + 1;  
    b[i:i+3] = select(vmask, v1, v0);
```

Last SIMD issue: Control Flow

- What if I have control flow?
 - Both control flow paths must be executed.
 - Could be more expensive than sequential version.
1. Compute predicate (mask) of logical test.
 2. Form solution for true and false conditions.
 3. Blend (select) the proper solutions using predicate.
 - “where true, select A, else select B.”
 - Masks are often bit sequences equal to superword length but may be byte or word sequences.

```
for (i = 0; i < 16; i++)  
    if (a[i] != 0)  
        b[i] = b[i] / a[i];  
    else  
        b[i]++;
```



```
for (i = 0; i < 16; i += 4)  
    vzero = {0,0,0,0};  
    vmask = (a[i:i+3] != vzero);  
    v0 = b[i:i+3] / a[i:i+3];  
    v1 = b[i:i+3] + 1;  
    b[i:i+3] = select(vmask, v1, v0);
```

Can we improve on this?

- Suppose that all control flow paths are not executed with the same frequency.
- Code could be optimized for the one with the highest frequency.
- The other would execute with the default general behavior.
 - Branch-on-superword jump (... ugly)
 - Or superword “reduction”: `vall()`, `vany()` functions test each predicate word and return a reduced yes/no. (... prettier)

```
for (i = 0; i < 16; i++)  
    if (a[i] != 0)  
        b[i]++;
```



```
for (i = 0; i < 16; i += 4)  
    vzero = {0,0,0,0};  
    vmask = (a[i:i+3] != vzero);  
    if (vany(vmask)) {  
        v0 = b[i:i+3];  
        v1 = v0 + 1;  
        b[i:i+3] = select(vmask, v1, v0);  
    }
```

Control Flow

- Not likely to be supported in older compilers
 - Increases complexity of compiler
 - Potential for slowdown
 - Performance is dependent on input data
- Many are of the opinion that SIMD is not a good programming model when there is control flow.
- But speedups are possible!
- Newer compilers support this as SIMD is becoming more powerful and critical.
 - Wasn't that big a loss for 128-bit SSE with doubles.
 - But superwords are 256 or 512-bit now. Loss is too large to accept.

Nuts and Bolts

- What does a piece of code really look like?
 - Example using SSE2/SSE3

```
for (i = 0; i < 100; i += 4)
```

```
    A[i:i+3] = B[i:i+3] + C[i:i+3]
```



```
for (i = 0; i < 100; i += 4) {
```

```
    __m128 vb = _mm_load_ps( &B[i] );
```

```
    __m128 vc = _mm_load_ps( &C[i] );
```

```
    __m128 va = _mm_add_ps( vb, vc );
```

```
    _mm_store_ps( &A[i], va );
```

```
}
```

What is possible?

- Mainly impacted media: 1997-2011
 - MMX introduced in 1997 (64-bit):
 - 2 32-bit int's, 4 16-bit short's, 8 8-bit char's
 - SSE (1999)-SSE4 (2007) (128-bit):
 - 2 double's, 4 float's, 4 int's, ...
 - For small words, big possible performance boost.
 - Not so great for scientific computing: max 2x faster ... but big gains still for byte data (genomics, encryption)
- But ... SIMD width growing fast ... Moore & Denard
 - AVX (2011), AVX2 (2013) (256-bit):
 - What are the operand sizes?
 - 4-8x faster for common math operations.
 - MIC-512 (2011), AVX-512 (2016) (512-bit):
 - We can't ignore this anymore. Only $1/8^{\text{th}}$ - $1/16^{\text{th}}$ of peak performance if we don't vectorize now.

What can impede vectorization?

- Explicit SIMD programming is very difficult and (often) not portable across platforms.
- Instead, we rely upon the compiler to get us most of the performance with minimal effort.
- But, the compiler does not know what you intended, only what you coded.
 - Sometimes we can restructure code to help.
 - Other times, the algorithm is fundamentally not vectorizable.
 - Common impediments or outright roadblocks:
 1. Data dependencies: read-after-write (RAW), write-after-read (WAR), write-after-write (WAW)
 2. Aliasing: Compiler can't tell if two arrays overlap and must be conservative. May be a data dependency; maybe not.
 3. Branches: Simple conditions can be handled. Complex branching impossible or just too expensive.

Impediments to SIMD: Countable

- Loops should be countable.
 - For loops with predictable increments good.

```
for (i = 0; i < n; i++)  
    c[i] = a[i] * b[i];
```

- Unpredictable for/while loops bad.

```
for (ptr = head; ptr != NULL; ptr = ptr->next)  
    ptr->value = 1 / ptr->value;
```

```
for (i = 0; i < nrows; ++i)  
    for (j = 0; j < ncols[i]; j += stride[i])  
        y[i] = y[i] + A[i][j] * x[j];
```

Impediments to SIMD: Branching

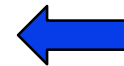
- Loops should not contain jumps or branches.
 - Simple conditionals can be vectorized.

```
for (i = 0; i < n; i++)  
    if (a[i] != 0)  
        b[i]++;
```

- Jumps or deep branches aren't.

```
for (i = 0; i < n; i++)  
    b[i] = func(a[i], b[i]);
```

```
for (i = 0; i < n; i++)  
    if (a[i] != 0)  
        b[i]++;  
    else if (a[i] == 2)  
        b[i] = a[i];  
    else if (a[i] % 2)  
        b[i] = 0;  
    else  
        b[i] = 1;
```



Can fix this by
in-lining function!
Supported in many
languages.

Impediments to SIMD: Data Dependency

- Read after write (RAW): *FLOW* dependency.
 - Using data written in previous iteration.

```
for (i = 1; i < n; i++)  
    A[i] = A[i-1] + B[i];
```



```
A[1] = A[0] + B[1]  
A[2] = A[1] + B[2]  
A[3] = A[2] + B[3]  
A[4] = A[3] + B[4]
```

Not vectorizable!

- Can work if offset is predictable and greater than superword (SIMD width).

```
for (i = 4; i < n; i++)  
    A[i] = A[i-4] + B[i];
```

May be
vectorizable.

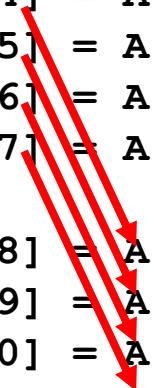
Impediments to SIMD: Data Dependency

- If stride is known greater than width, RAW or FLOW dependencies doesn't happen.

```
for (i = 4; i < n; i++)  
    A[i] = A[i-4] + B[i];
```



```
A[ 4] = A[0] + B[ 4]  
A[ 5] = A[1] + B[ 5]  
A[ 6] = A[2] + B[ 6]  
A[ 7] = A[3] + B[ 7]  
  
A[ 8] = A[4] + B[ 8]  
A[ 9] = A[5] + B[ 9]  
A[10] = A[6] + B[10]  
A[11] = A[7] + B[11]
```



Impediments to SIMD: Data Dependency

- Write after read (WAR): *ANTI* dependency.
 - Writing data just read in previous iteration.

```
for (i = 0; i < n-1; i++)  
    A[i] = A[i+1] + B[i];
```



```
A[0] = A[1] + B[0]  
A[1] = A[2] + B[1]  
A[2] = A[3] + B[2]  
A[3] = A[4] + B[3]
```

Vectorizable!

- Write after write (WAW): *OUTPUT* dependency.
 - Multiple iterations can write to the same location.

```
for (i = 0; i < n; i++)  
    A[i%k] = B[i] + C[i];
```

Not vectorizable!

Impediments to SIMD: Data Dependency

- Can not write to the same location (lane) in the SIMD superword.

```
for (i = 0; i < n; i++)  
    A[i%2] = B[i] + C[i];
```



```
A[0] = B[0] + C[0]  
A[1] = B[1] + C[1]  
A[0] = B[2] + C[2]  
A[1] = B[3] + C[3]
```

```
A[0] = B[4] + C[4]  
A[1] = B[5] + C[5]  
A[0] = B[6] + C[6]  
A[1] = B[7] + C[7]
```

Not vectorizable!

```
for (i = 0; i < n; i++)  
    A[i%4] = B[i] + C[i];
```



```
A[0] = B[0] + C[0]  
A[1] = B[1] + C[1]  
A[2] = B[2] + C[2]  
A[3] = B[3] + C[3]
```

```
A[0] = B[4] + C[4]  
A[1] = B[5] + C[5]  
A[2] = B[6] + C[6]  
A[3] = B[7] + C[7]
```

Vectorizable!

Impediments to SIMD: Aliasing

- Compiler can not determine if arrays overlap.
 - Only a problem in languages with very flexible pointers (C/C++).
 - Compiler must be conservative and protect against RAW.

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

```
...
float a[100], b[100], c[100];
...
// Call add() without aliasing.
add (100, a, b, c);
```

Vectorizable!

```
// Call add() with aliasing.
add (99, a+1, a, c+1);
```

Not vectorizable!
 $A[i] = A[i-1] + C[i];$

Modern SIMD Programming

- Intrinsic functions lock us into one instruction set ... not portable.
- Options:
 - Many/most compilers support some pragma's to guide the compiler. Usually very limited and not (always) portable.
 - OpenCL provides SIMD datatypes and operations for 2 to 16 elements (double2 to double16).
 - Portable and “standardized” but OpenCL hasn't been accepted well and is fading. Very sad ... I have a lot of OCL code.
 - OpenMP introduced SIMD parallelism in v4 (2013-14).
 - Standard set of compiler pragma's. Limited but at least portable.

SIMD Pragma's

- Most common is `ivdep` – “ignore vector dependency”

```
void add (int n, float *A, float *B, float *C)
{
    #pragma ivdep /* Tells Compiler that this is
                  vectorization code. It's only a,
                  and may be ignored though. */
    for (i = 0; i < n; i++)
        A[i] = B[i] + C[i];
}
```

```
...
float a[100], b[100], c[100];
...
// Call add() without aliasing.
add (100, a, b, c);
```

Vectorized! And correct.

```
// Call add() with aliasing.
add (99, a+1, a, c+1);
```

Vectorized!
... And may be wrong.

OpenCL SIMD Datatypes

- OpenCL has built-in SIMD datatypes. Alignment is ensured if allocated by the built-in allocation system.

```
void add4 (int n, float *A, float *B, float *C)
{
    float4 *A4 = A, *B4 = B, *C4 = C; /* Ignoring alignment! */
    int n4 = n / 4;
    for (i = 0; i < n4; ++i)
        A4[i] = B4[i] + C4[i];

    /* Remainder */
    for (i = n4; i < n; ++i)
        A[i] = B[i] + C[i];
}

...
float a[100], b[100], c[100];
...
add4 (100, a, b, c);
```

OpenMP SIMD Pragma's

- OpenMP v4 standardized SIMD parallel pragma's.

```
void add (const int n, float *A, float *B, float *C)
{
    const int vlen = 8;
    bool vectorize = (abs(A - B) > vlen) &&
                    (abs(A - C) > vlen);
    #pragma omp simd if (vectorize) safelen(vlen)
    for (i = 0; i < n; i++)
        A[i] = B[i] + C[i];
}
```

```
...
float a[100], b[100], c[100];
...
// Call add() without aliasing.
add (100, a, b, c);
```

Vectorized! And correct.

```
// Call add() with aliasing.
add (99, a+1, a, c+1);
```

Vector safety test failed!
Loop isn't vectorized.

```
// Call add() with potential aliasing.
add (92, a+8, a, c);
```

Vector safety test passed!
Loop is vectorized.