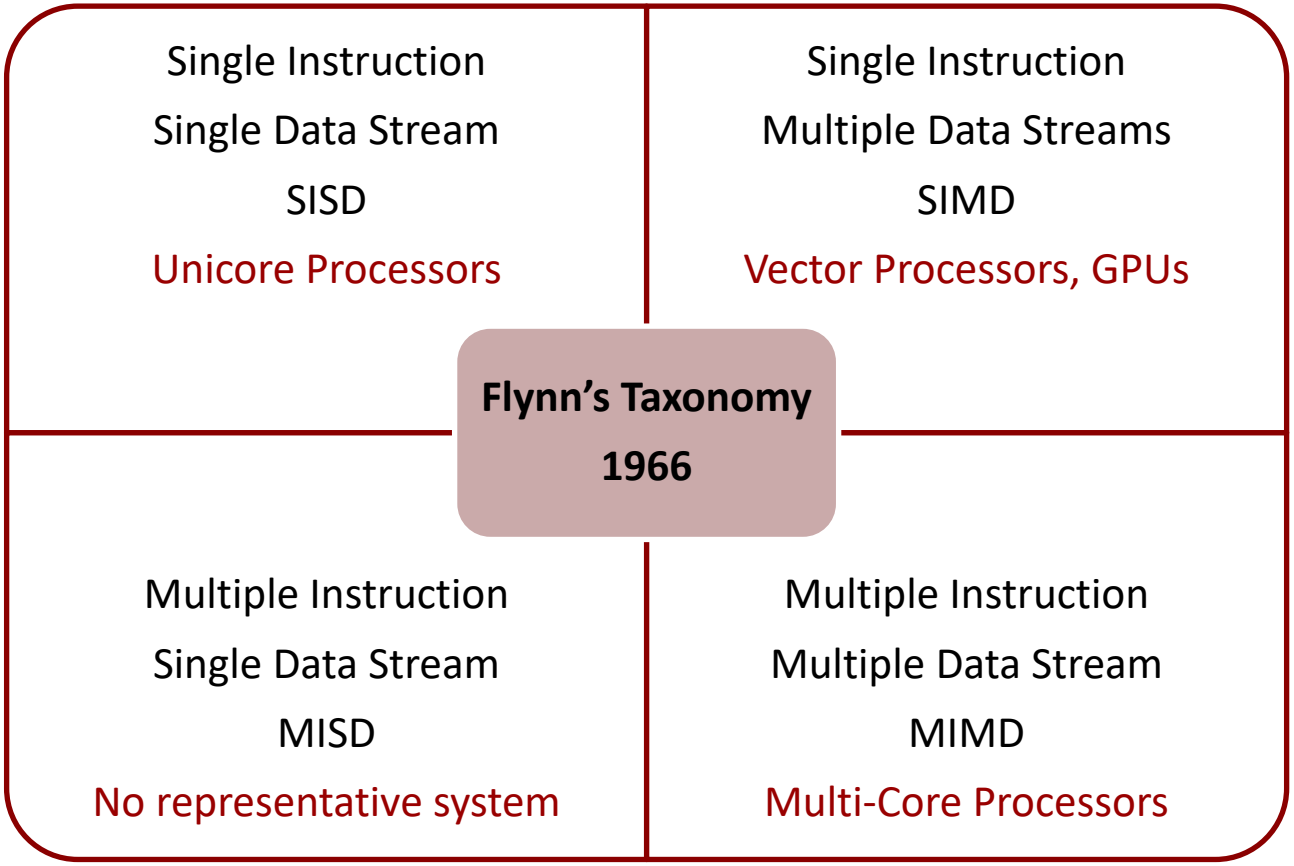
Vectorization

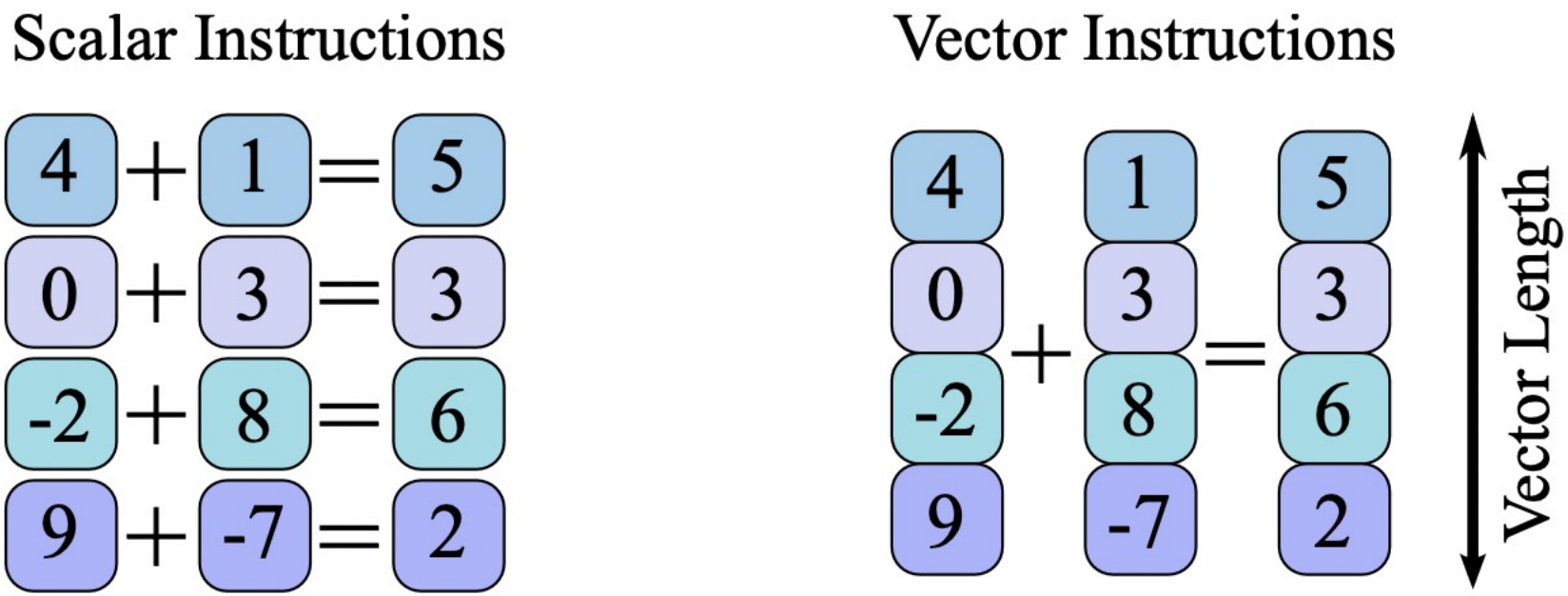
Spring 2022

Suresh Purini

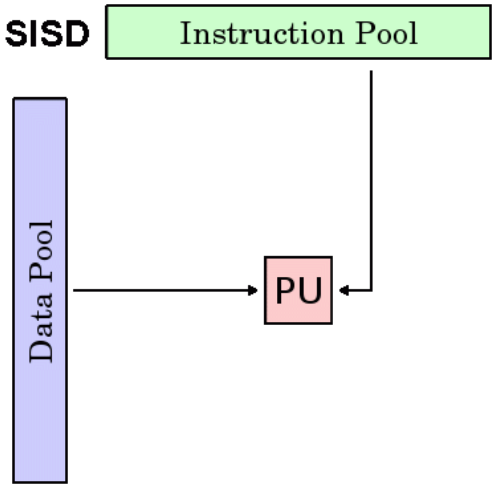
Computer Systems Group, IIIT , Hyderabad

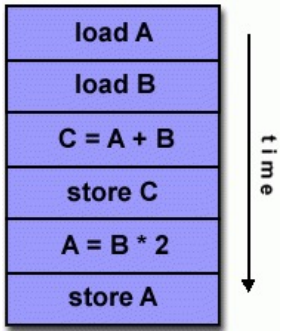

Single Instruction
Single Data Stream
SISD
Unicore Processors
Single Instruction
Multiple Data Streams
SIMD
Vector Processors, GPUs
Flynn's Taxonomy
1966
Multiple Instruction
Single Data Stream
MISD
No representative system
Multiple Instruction
Multiple Data Stream
MIMD
Multi-Core Processors

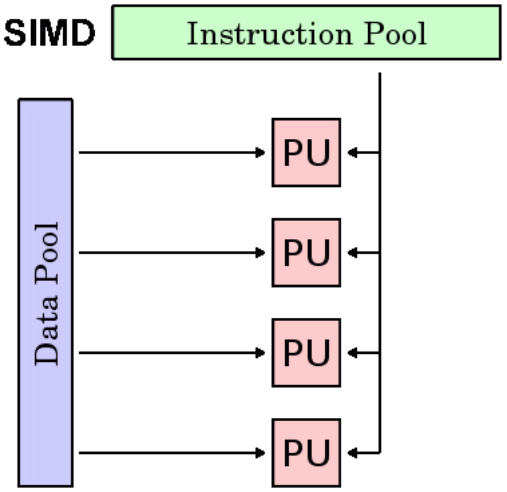
Scalar Vs Vector Instructions



SIMD and Data Parallelism

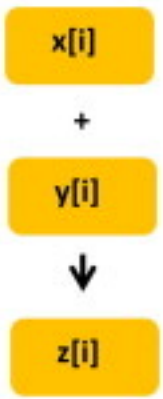


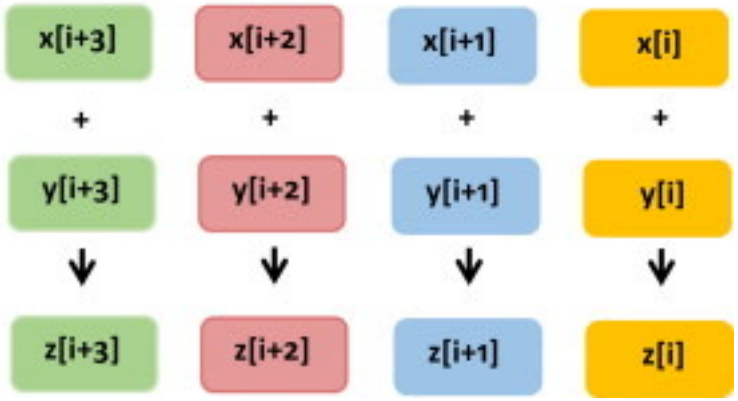




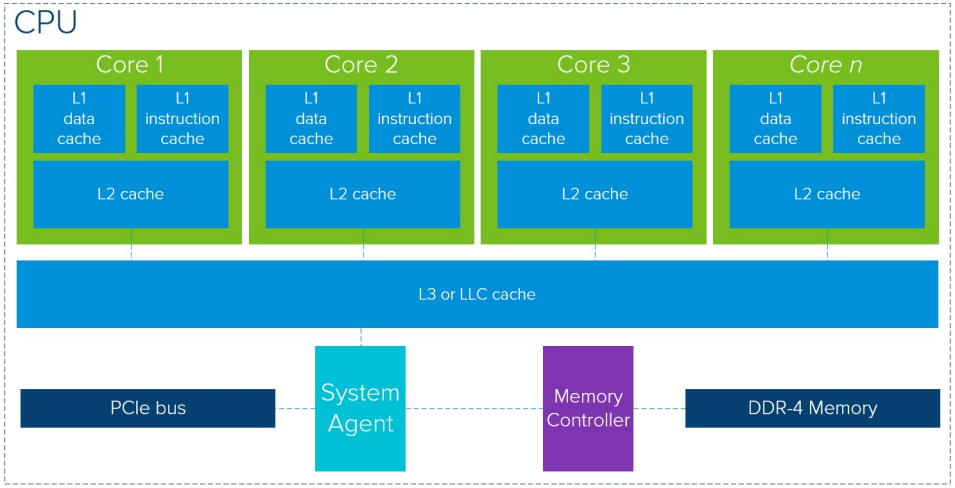

```
//scalar: 40 iterations
for (int i=0; i<40; i++)
   z[i] = x[i] + y[i];
```

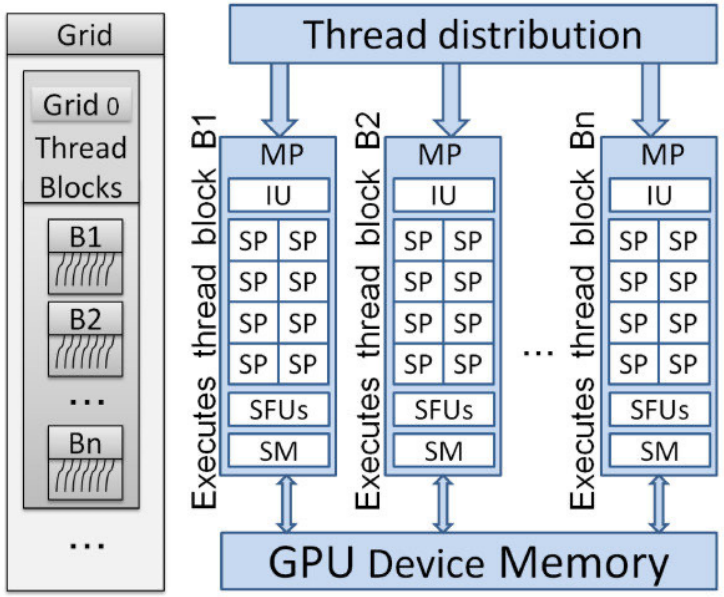
```
//vector: 10 iterations
for (int i=0; i<40; i+=4)
  z[i:i+3]= x[i:i+3]+ y[i:i+3];
```



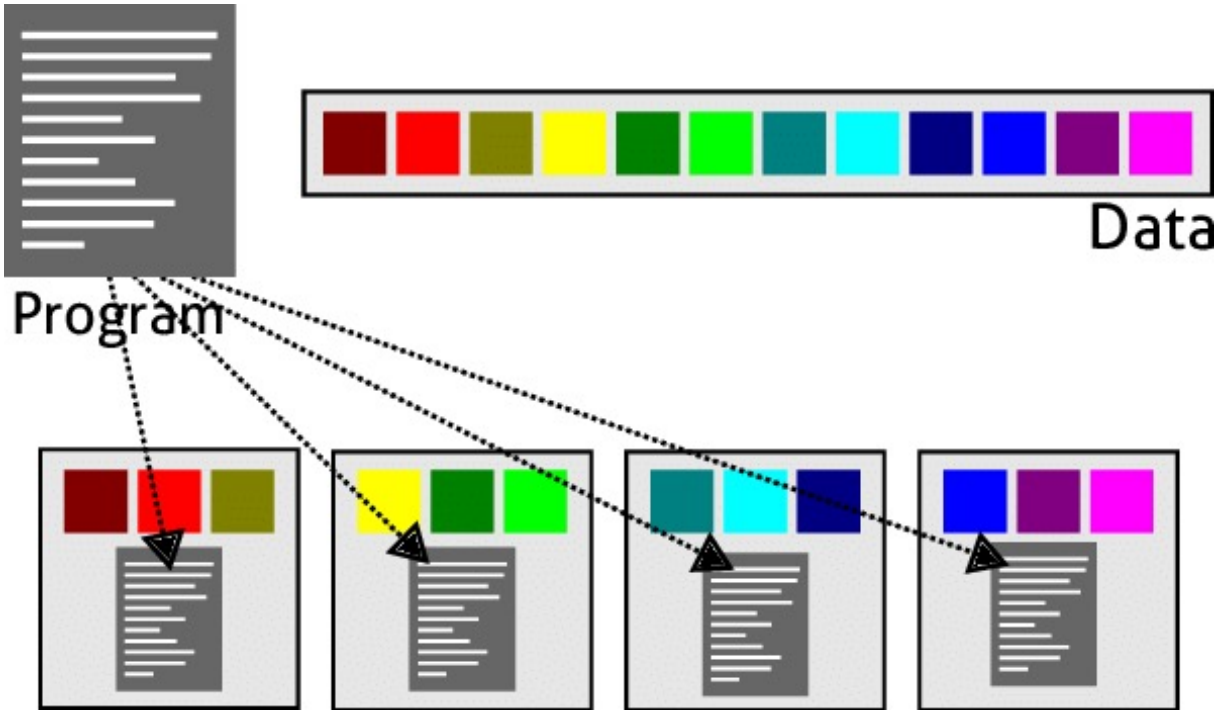


CPU vs GPU and Data vs Task Parallelism



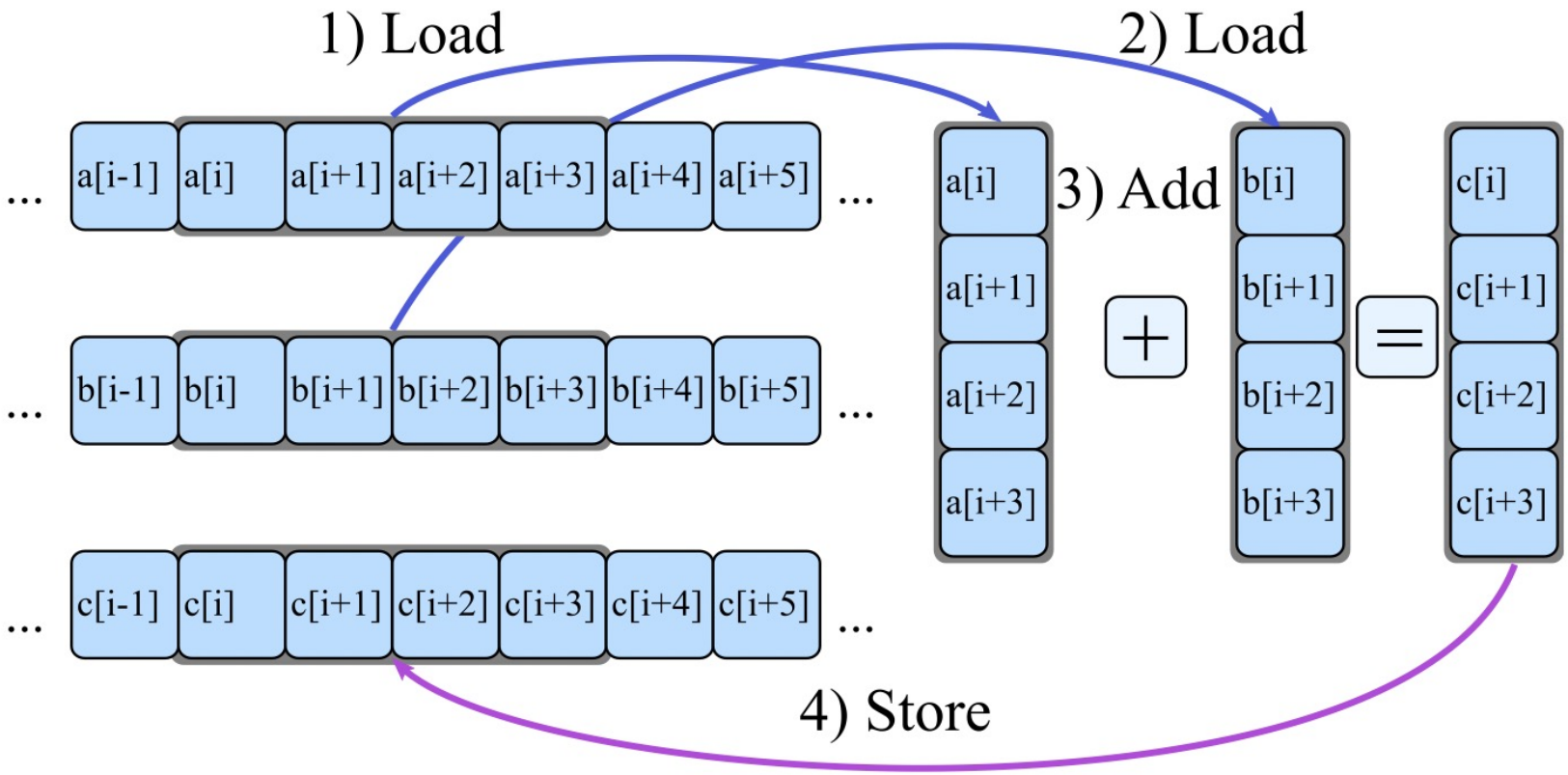


SIMD vs SPMD (Single Program Multiple Data)



SPMD Example: MAP Reduce

Workflow of Vector Compution



Using Vector Instructions: Two Approaches

Automatic Vectorization →

```
1 double A[vec_width], B[vec_width];
2 // ...
3 for(int i = 0; i < vec_width; i++)
4   A[i]+=B[i];
```

```
1 double A[8], B[8];
2 __m512d A_v = _mm512_load_pd(A);
3 __m512d B_v = _mm512_load_pd(B);
4 A_v = _mm512_add_pd(A_v,B_v);
5 _mm512_store_pd(A, A_v);
```

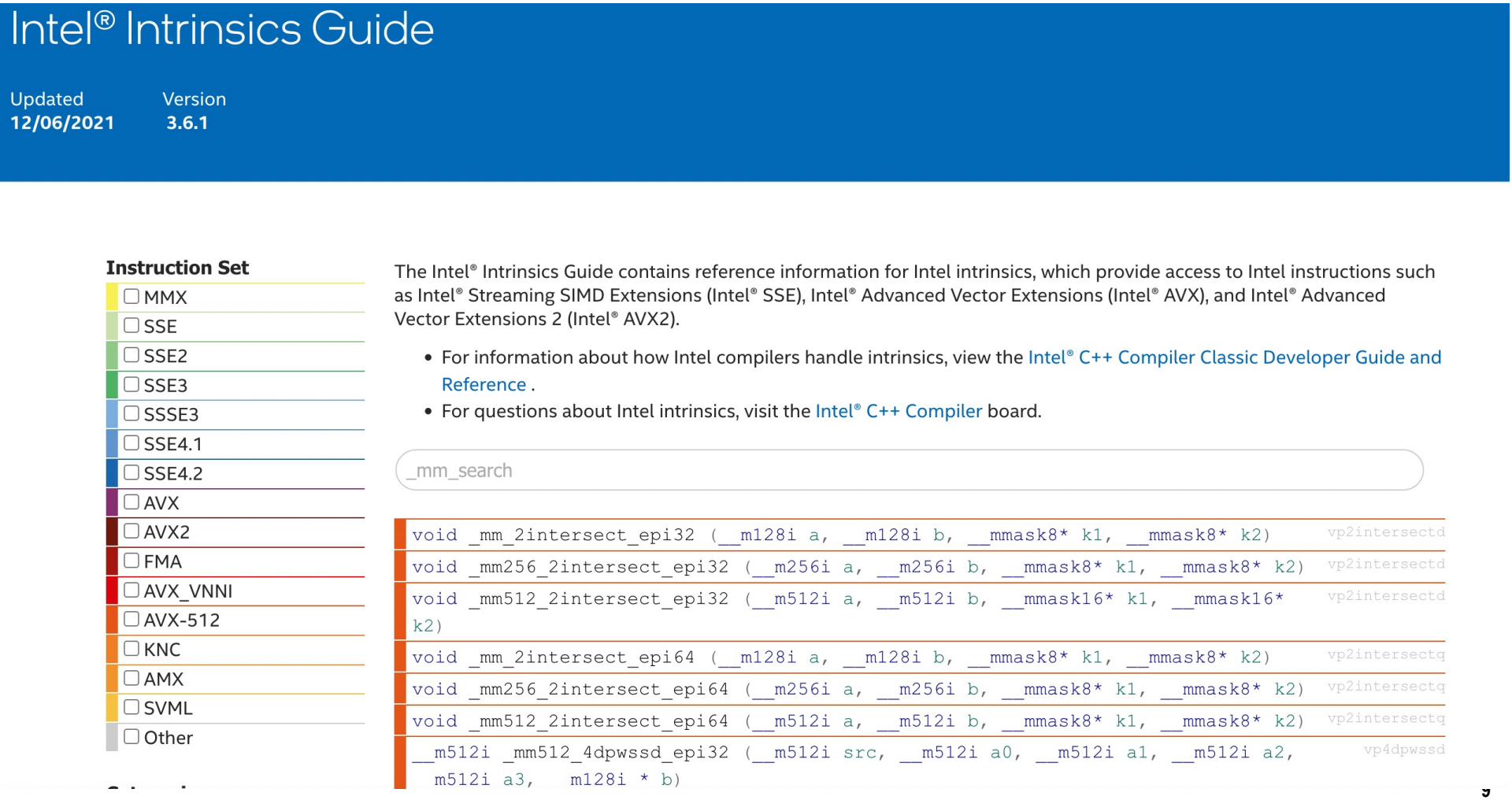
← Explicit Vectorization

Third Approach: Use Vectorized Libraries like Intel MKL

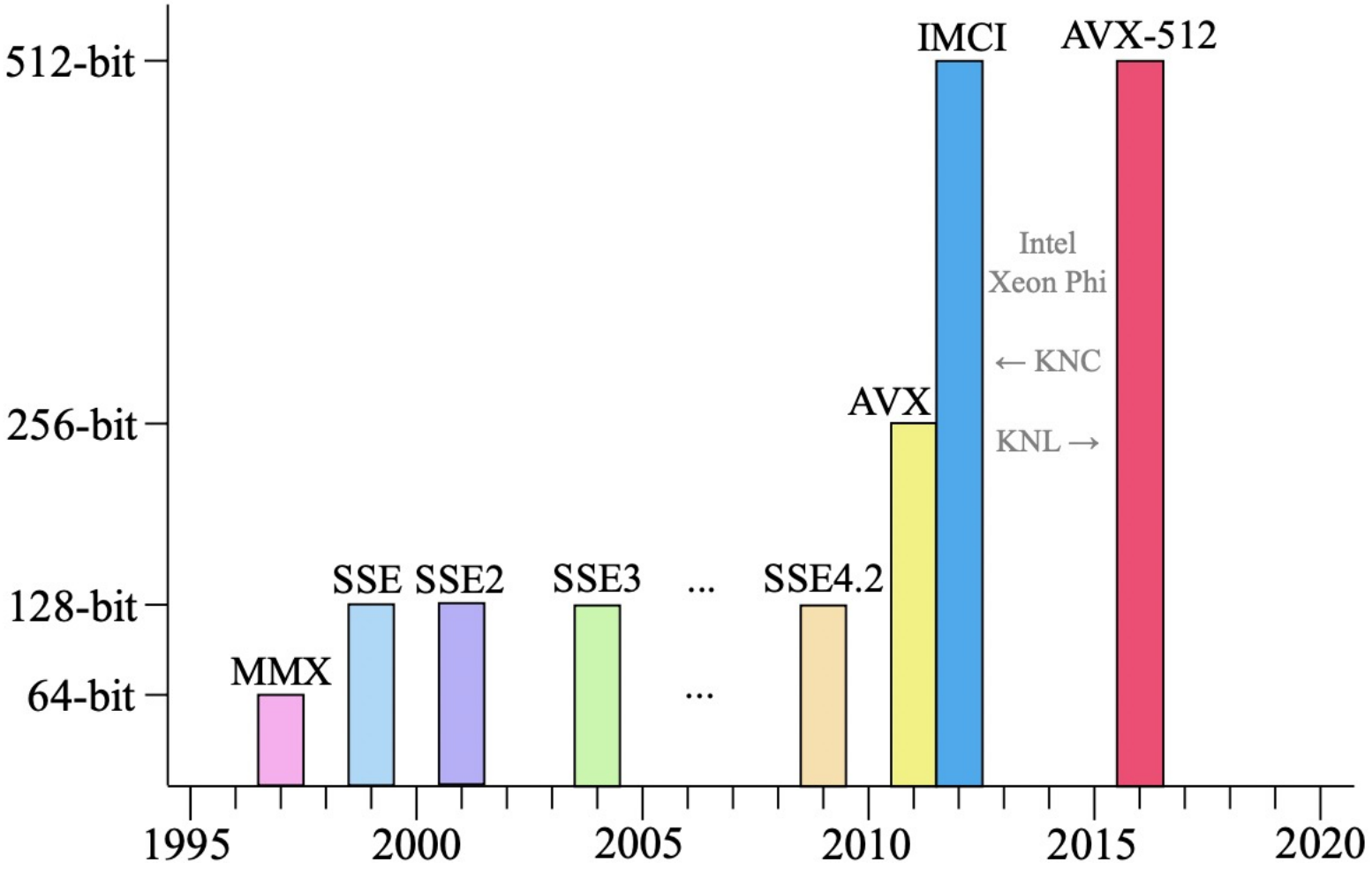
Fourth Approach: Use assembly code

Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide

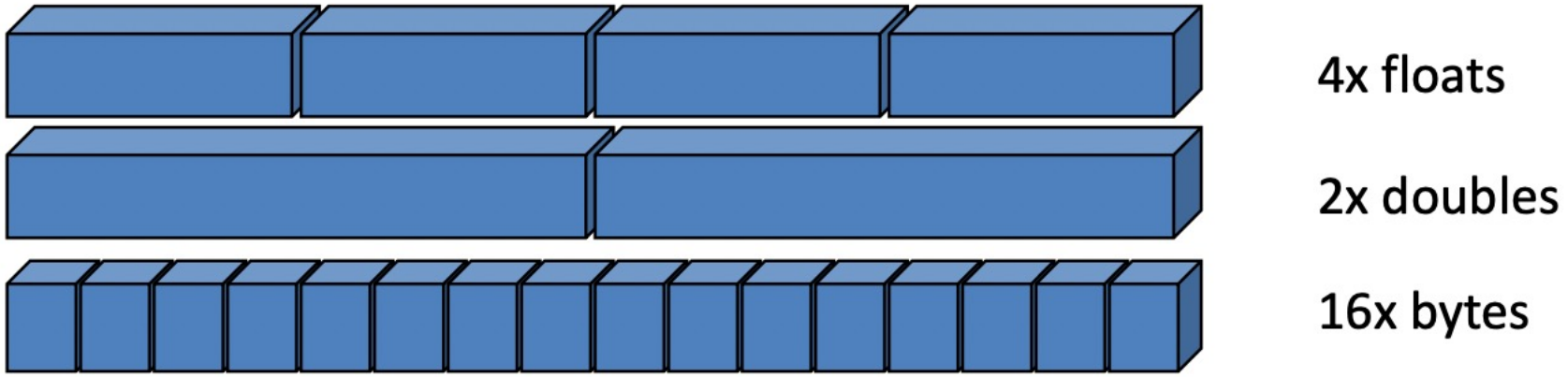


Intel SIMD Instruction Set



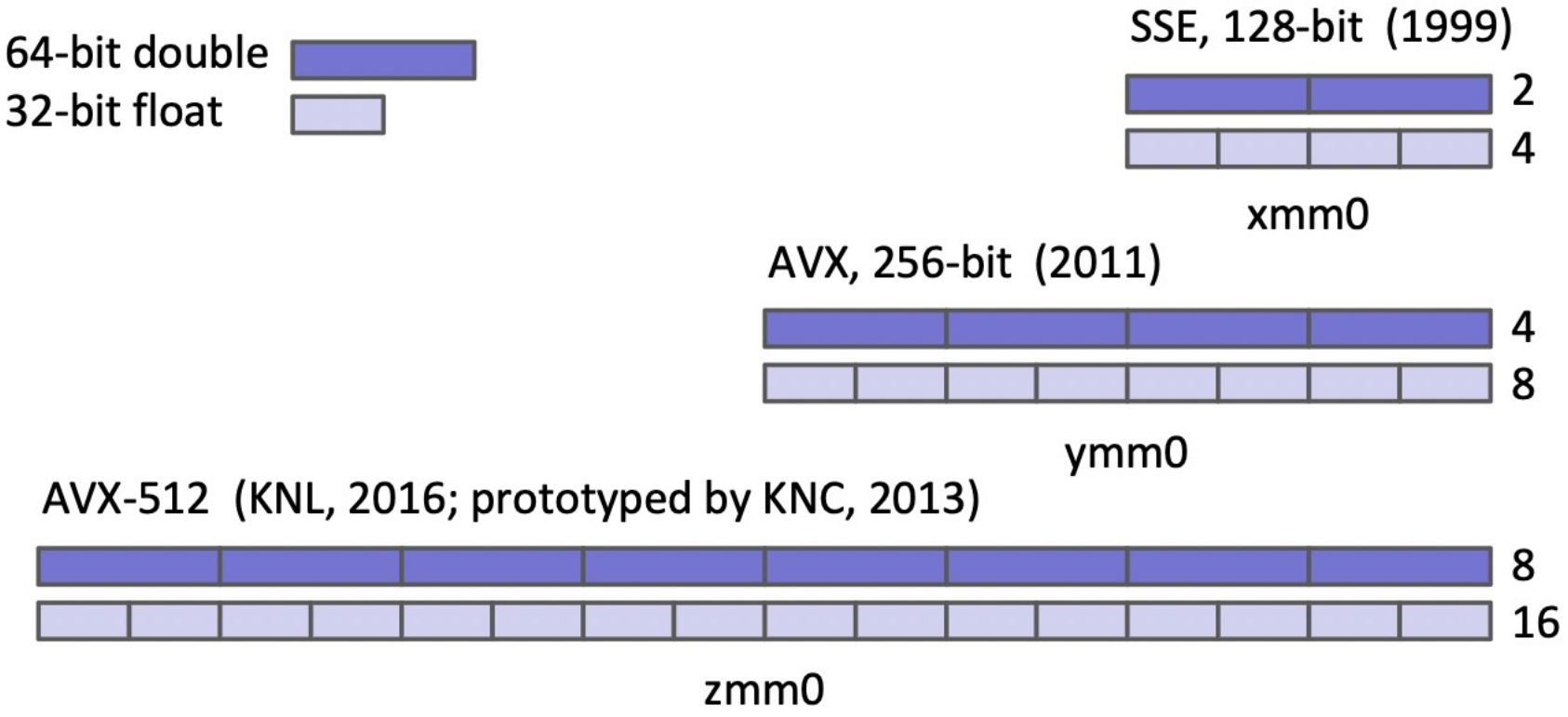
SSE/SSE2 ISA (128 bits)

- Eight 16 bit integers
- Four 32 bit integers
- Four floating point numbers
- 2 double precision floating point numbers



Evolution of Vector Registers and Instructions

SIMD Unit in a core has 16 (SSE, AVX) or 32 (AVX-512) vector registers



Peak FLOPs Computation

- **A core may have 2 Vector Processing Units (VPUs)**
- **Fused Multiply and Add instruction carried out in one cycle**
- **SIMD instructions are also pipelined**
- **Extra factor of 2 from Fused Multiply-Add (FMA):**
  - 128-bit SSE – 4x double, 8x single (pipelined, not truly fused)
  - 256-bit AVX – 8x double, 16x single (pipelined; FMA in AVX2)
  - 512-bit AVX – 16x double, 32x single (FMA)
- **Example: Intel Xeon Gold 6130 “Skylake-SP” @ 2.1 GHz**
  - Two AVX-512 units/core * 1 FMA/cycle = 64 flop/cycle SP
- **64 flop/cycle/core * 16 cores * 1.9 GHz = 1950 Gflop/s peak SP**

Note, clock is throttled to 1.9 GHz due to heating at peak load

SIMD Array Processing

```
for each f in array
    f = sqrt(f)
```

```
for each f in array
{
    load f to floating-point register
    calculate the square root
    write the result from the
register to memory
}
```

```
for each 4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    store the 4 results from the register to memory
}
```

SIMD style

Automatic Vectorization of Loops

```cpp
#include <cstdio>

int main(){
  const int n=1024;
  int A[n] __attribute__((aligned(64)));
  int B[n] __attribute__((aligned(64)));

  for (int i = 0; i < n; i++)
    A[i] = B[i] = i;

  // This loop will be auto-vectorized
  for (int i = 0; i < n; i++)
    A[i] = A[i] + B[i];

  for (int i = 0; i < n; i++)
    printf("%2d %2d %2d\n",i,A[i],B[i]);
}
```

```
vega@lyra% icpc autovec.cc -qopt-report
vega@lyra% cat autovec.optrpt
...
LOOP BEGIN at autovec.cc(12,3)
remark #15399: vectorization support:
unroll factor set to 2 [autovec.cc(12,3)]
remark #15300: LOOP WAS VECTORIZED
[autovec.cc(12,3)]
LOOP END
...
vega@lyra% ./a.out
0  0  0
1  2  1
2  4  2
3  6  3
4  8  4
...
```

Limitations of Automatic Vectorization

- Innermost loops
- Known number of iterations
- No Vector dependence
- Functions must be SIMD enabled

Use #pragma omp simd

Targeting a Specific Instruction Set

`-x[code]` to target specific processor architecture

`-ax[code]` for multi-architecture dispatch

code	Target architecture
MIC-AVX512	Intel Xeon Phi processors (KNL)
CORE-AVX512	Future Intel Xeon processors
CORE-AVX2	Intel Xeon processor E3/E5/E7 v3, v4 family
AVX	Intel Xeon processor E3/E5 and E3/E5/E7 v2 family
SSE4.2	Intel Xeon processor 55XX, 56XX, 75XX and E7 family
host	architecture on which the code is compiled

Vectorization Directives

- `#pragma omp simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `__assume_aligned` keyword
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `restrict` qualifier and `-restrict` command-line argument
- `#pragma loop count`

Compiler driven Auto-Vectorization

Use Loop Unrolling

```
for (i=0; i<N; i++) {
    c[i]=a[i]+b[i];
}
```

```
for (i=0; i<N; i+=4) {
    c[i+0]=a[i+0]+b[i+0];
    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];
}
```

```
Load a(i..i+3)
Load b(i..i+3)
Do 4-wide a+b->c
Store c(i..i+3)
```

Generalized Loop Unrolling

- **A loop of n iterations**
- **k copies of the body of the loop**
- **Assuming (n mod k) != 0**
  - Then we will run the loop with 1 copy of the body (n mod k) times
  - and then with k copies of the body floor(n/k) times

Data Alignment

Aligning the data to 64 byte boundaries is beneficial

- Align the Data.
- Use pragmas/directives and clauses in performance critical regions (where the data is used) to tell the compiler that memory accesses are aligned.

https://www.intel.com/content/www/us/en/developer/articles/technical/data-alignment-to-assist-vectorization.html

```
float A[1000] __attribute__((aligned(64)));
```

Assumed Vector Dependency

Not enough information to confirm or rule out vector dependence:

```
1 void AmbiguousFunction(int n, int *a, int *b) {
2   for (int i = 0; i < n; i++)
3     a[i] = b[i];
4 }
```

- If a, b are not aliased or b>a, then safe to vectorize
- If a, b are aliased (e.g., b==a-1), requires scalar computation

Pointer Disambiguation

Prevent multiversioning or allow vectorization with a directive:

```
1 #pragma ivdep
2   for (int i = 0; i < n; i++)
3       // ...
```

Pointer Disambiguation

```
void updatePtrs(size_t *ptrA, size_t *ptrB, size_t *val)
{
  *ptrA += *val;
  *ptrB += *val;
}
```

```
void updatePtrs(size_t *restrict ptrA, size_t *restrict ptrB, size_t *restrict val);
```

Dependency Analysis

- True (flow) dependence (RAW = read after write)
- Anti dependence (WAR = write after read)
- Output dependence (WAW = write after write)

Anti and output dependences are called false dependences.

```
1:  a = 1;
2:  b = a;
3:  a = a + b;
4:  c = a;
```

If S_j is dependent on S_i, we write $S_1 \ \delta \ S_2$. Sometimes we also indicate the kind of dependence.

$$S_1 \ \delta^f \ S_2 \quad S_1 \ \delta^o \ S_3 \quad S_2 \ \delta^a \ S_3 \quad \ldots$$

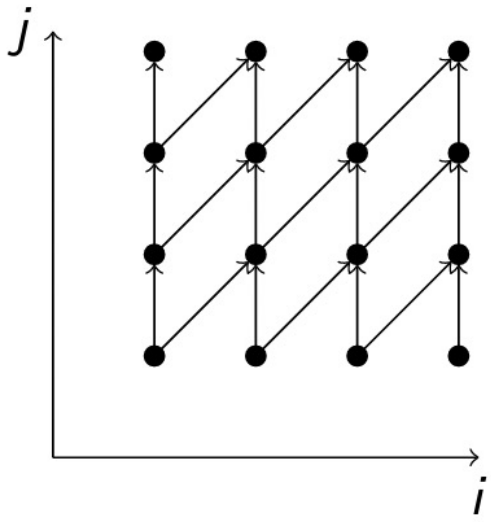
Dependences in Loops

```
for i = 1 to 3
    1: X[i] = Y[i] + 1
    2: X[i] = X[i] + X[i-1]
```

- loop-independent flow dependence from S_1 to S_2
- loop-carried flow dependence from S_2 to S_2
- loop-carried anti dependence from S_2 to S_2

Dependence Distance Vectors

```
for i = 1 to 3
  for j = 1 to 3
    X[i,j] = X[i,j-1]
           + X[i-1,j-1]
```



Dep. vectors $(0,1),(1,1)$

- One way to represent dependences are distance vectors
- If statement instance $\vec{t}$ is dependent on instance $\vec{s}$ the distance vector for these two instances is

$$\vec{d} = \vec{t} - \vec{s}$$

- Uniform dependences are described by distance vectors that do not contain index variables.

Loop Carried Dependencies

- True vector dependence – vectorization impossible:

```
1 for (int i = 1; i < n; i++)
2   a[i] += a[i-1]; // dependence on the previous element
```

- Safe to vectorize:

```
1 for (int i = 0; i < n-1; i++)
2   a[i] += a[i+1]; // no dependence on the previous element
```

- May be safe to vectorize:

```
1 for (int i = 16; i < n; i++)
2   a[i] += a[i-16]; // no dependence if vector length <=16
```

Loop Unswitching

```
for i = 1 to N
  for j = 1 to M
    if X[i] > 0
      S
    else
      T
```

```
for i = 1 to N
  if X[i] > 0
    for j = 1 to M
      S
  else
    for j = 1 to M
      T
```

- Hoist conditional as far outside as possible
- Enable other transformations

Loop Peeling

```
for i = 1 to N
  S
```

```
if N ≥ 1
  S
  for i = 2 to N
    S
```

- Align trip count to a certain number (multiple of N)
- Peeled iteration is a place where loop invariant code can be executed non-redundantly

Index Set Splitting

```
for i = 1 to N
  S
```

```
assert 1 ≤ M < N
for i = 1 to M
  S
for i = M + 1 to N
  S
```

- Create specialized variants for different cases
  e.g. vectorization (aligned and contiguous accesses)
- Can be used to remove conditionals from loops

Strip Mining

```
for i = 1 to N
  S
```

```
for (i = 0; i < n; i += U)
  for (j = 0; i < U; j++)
    S(i + j)
```

- strip-mine + interchange = tiling
- Vectorization is a kind of strip mining