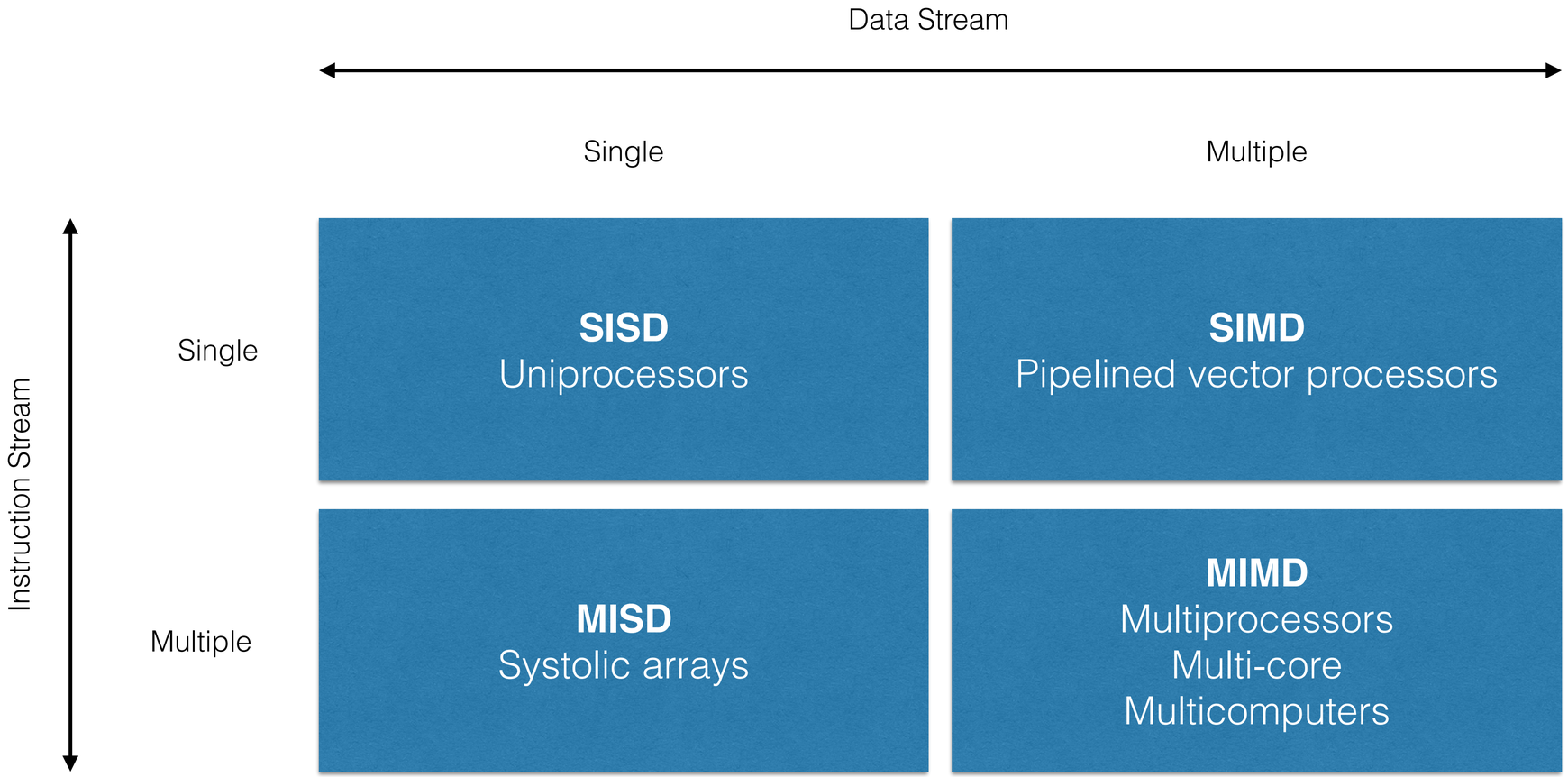
Improving Code Vectorization

Robert Kalescky
Adjunct Professor of Data Science
HPC Applications Scientist

DS 7347: High-Performance Computing and Data Science

Flynn's Parallel Architecture Taxonomy

- Single/multiple instruction streams
  - Number of types of instructions to be performed at once
- Single/multiple data streams
  - Number of data streams to be operated on at once
- Most modern parallel computers are MIMD
- SIMD was popular until 1990s
- MISD never used to large extent



Parallel Hardware

- Levels of parallelism
  - Vectorization
  - Multicore
  - Multi-CPU
  - Multi-Node
- Threads versus processes

Vectorization Example

```
1  for (int i=0; i<16; ++i) {
2     C[i] = A[i] + B[i];
3  }
4
5  for (int i=0; i<16; i+=4) {
6     addFourThingsAtOnceAndStoreResult(&C[i], &A[i], &B[i]);
7  }
```

Advanced Vector Extensions

- Advanced Vector Extensions (AVX) are a set of SIMD extensions to the x86 instruction set
- Initially developed by Intel in 2008
- Implemented in Intel and AMD CPU's since 2011

Advanced Vector Extensions

- AVX
  - Expanded many of the earlier SSE floating point instructions from 128 to 256 bits
  - Several new instructions
  - Intel Sandy Bridge and AMD Bulldozer and newer architectures

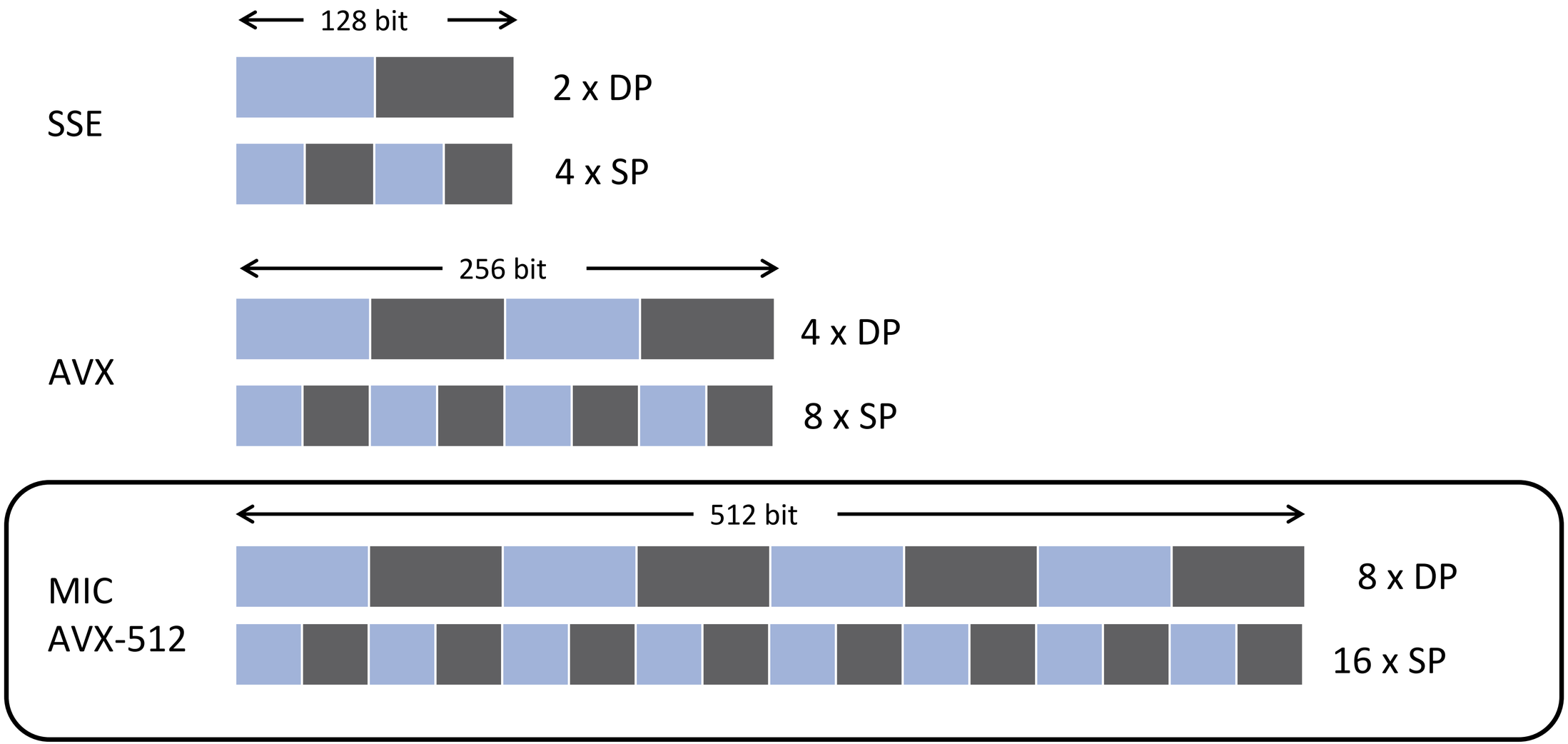
Advanced Vector Extensions

- AVX2
  - Expanded many of the earlier SSE integer instruction from 128 to 256 bits
  - Several new instructions
  - Intel Haswell and AMD Carrizo and newer architectures

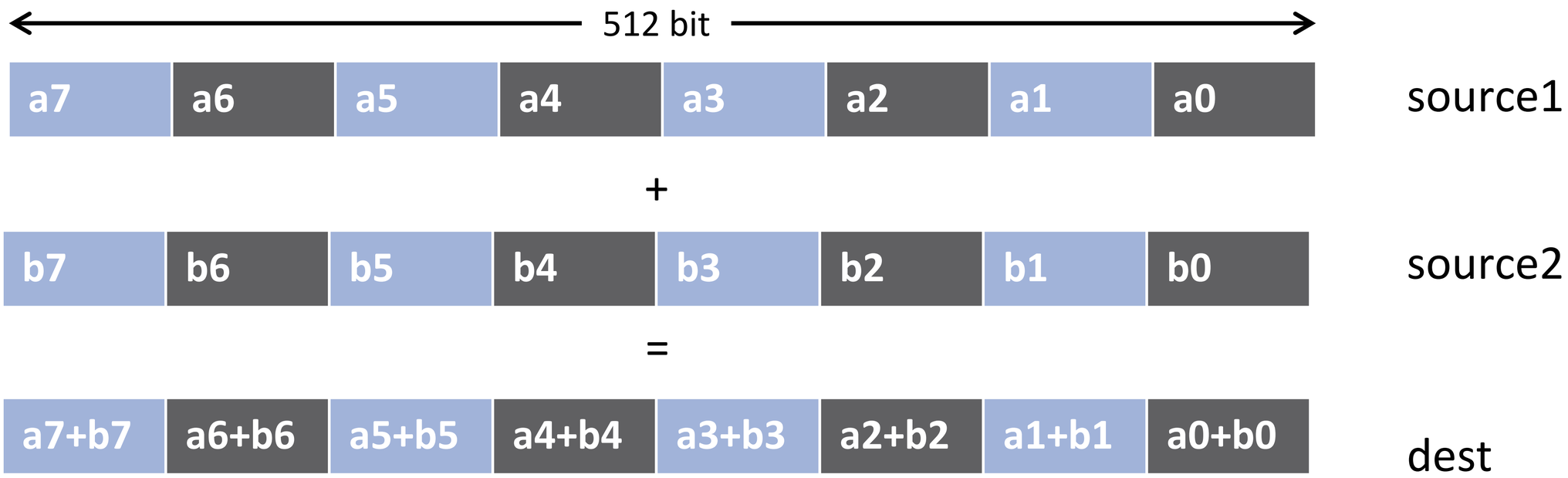
Advanced Vector Extensions

- AVX-512
  - Expanded AVX instructions from 256 to 512 bits
  - Several new instructions
  - Intel Xeon Phi x200 series CPU’s

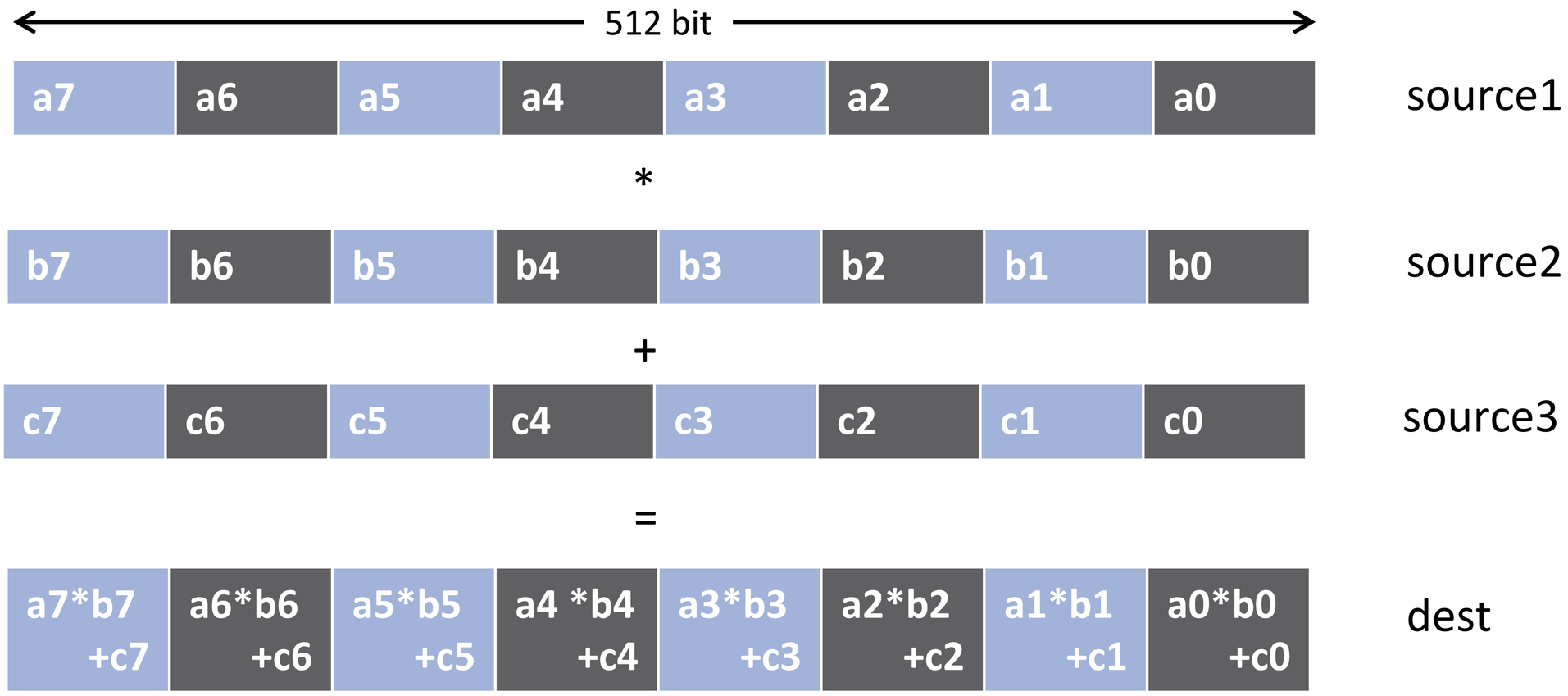
Vector Widths



Vector Widths



Specialized Hardware (FMA)



Parallel Programming Tools

- OpenMP is the primary approach for enabling shared-memory parallel computing
  - Compiler extension
  - Used via directives or pragmas in source code
- MPI (message passing interface) is the primary approach for enabling distributed-memory parallel computing
  - Library of functions and data types used to send messages among processes
- Many higher-level languages and libraries are built using OpenMP and MPI

Languages Supporting Parallelism

- Compiled languages supporting OpenMP and MPI include:
  - Fortran
  - C
  - C++
- Interpreted languages supporting parallelism include:
  - Python
  - MATLAB
  - Mathematica

Motivation

- Reduced run time
  - Faster iterations
  - Bigger/longer calculations
  - More efficient use of available resources
- Application should support diverse hardware

Profiling Your Application

- Hopefully you already know what your application does (but not a given)
- Know what your application is doing *as a function of time*
  - Determine what parts take the longest…
  - …and why
- Many available
  - Mostly delineated by application type and/or language

Options

- With knowledge of *how,* both performance-wise and algorithmically, the application runs:
  - Tune compile time choices
  - Improve code level choices
  - Extend runtime capabilities
- Need knowledge of what resources are available to your application to make some of these decisions
  - Where and on what will it run

Available Hardware - CPUs

- Number of cores on a node is important…
- …making sure that you're using the cores efficiently is more important
  - Vector extensions
  - Caches

Options for Using Accelerators

- Tune compile time choices
  - Vectorization
  - Link to a library
- Improve code level choices
  - Nondestructive directive-based APIs
- Extend runtime capabilities
  - Reimplementing algorithms to exploit specialized hardware

Increasing difficulty and implementation time

OpenMP

- Thread-based parallelism
  - Single process
  - Shared memory
- Explicit parallelism
  - Parallelism explicitly identified in source code via directives or pragmas
- Fork-Join Model
  - Compiled code follows serial → parallel → serial execution pattern
  - Capable of dynamic thread count based on workload

OpenMP Directives

- Compiler identifies explicitly indicated parallelizable code based on directives or pragmas
  - Ignored by compiler as comments if not used
- Directives are used to:
  - Indicate parallel region
  - Divide portions of source code among threads
  - Distributing loop iterations among threads
  - Serializing portions of code
  - Indicating synchronization among threads

OpenMP Directives

- Directives have three parts
  - Sentinel
  - Directive name
  - Clauses

Language	Sentinel	Directive	Clauses
Fortran	!$OMP	PARALLEL	DEFAULT(SHARED) PRIVATE(BETA,PI)
C/C++	#pragma omp	parallel	default(shared) private(beta,pi)

Compiling OpenMP Programs

- Compilers supporting OpenMP including:
  - Intel compilers
  - PGI compilers
  - GNU compilers
- OpenMP is used via a given compiler option
  - -fopenmp for GNU compilers
  - -mp for PGI compilers
  - -openmp for Intel compilers
  - Directives and pragmas are ignored if the compiler option is not given

Compiling OpenMP Programs

- GNU example

```
$ module purge

$ module load gcc

$ g++ –fopenmp example.cpp –o example
```

- PGI example

```
$ module purge

$ module load pgi

$ pgc++ –mp example.cpp –o example
```

Running OpenMP Programs

```
#!/bin/bash
#SBATCH -J example          # job name
#SBATCH -o example.out      # output/error file name
#SBATCH -p parallel-short # requested queue

# set compiler
module purge
module load gcc

# set the desired number of OpenMP threads
export OMP_NUM_THREADS=8

# run the code
./driver.exe
```

Vectorization

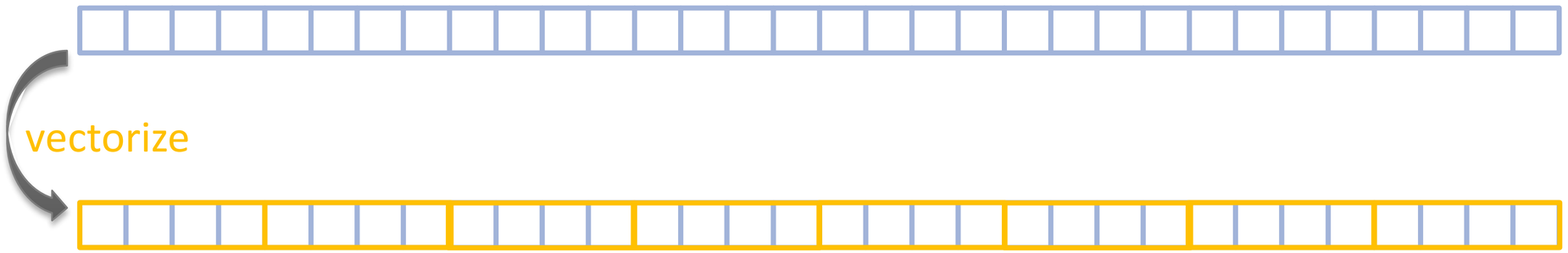
- Operations are exclusively done over vectors, matrices, and tensors instead of scalars
- Scalars in array programming are vectors with a single index
- Array programming simplifies programming linear algebra algorithms
- Array programming languages include MATLAB, Octave, R, Julia, and Python via NumPy

Single Instruction, Multiple Data

- A single operation is applied to multiple data simultaneously
- Allows some linear alegra operations to be done via vectors instead of single elements
- Modern systems are MIMD where each processor or core can perform independent SIMD operations
- SIMD instructions are CPU architecture specific, *i.e.* not all CPU's have the same SIMD instruction sets

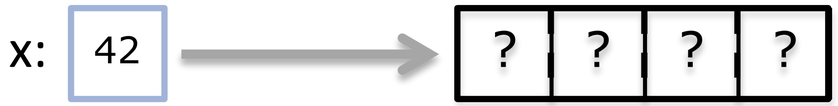
Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Data Sharing

- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*



- `reduction(`*`op`*`:`*`var-list`*`):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct

Data Sharing

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → in practice, maximum vector length
- `simdlen (length)`
  - → Specify preferred length of SIMD registers used
  - → Must be less or equal to `safelen` if also present
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture
- `collapse (n)`

SIMD Worksharing

- Parallelize and vectorize a loop nest
  - → Distribute a loop's iteration space across a thread team
  - → Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

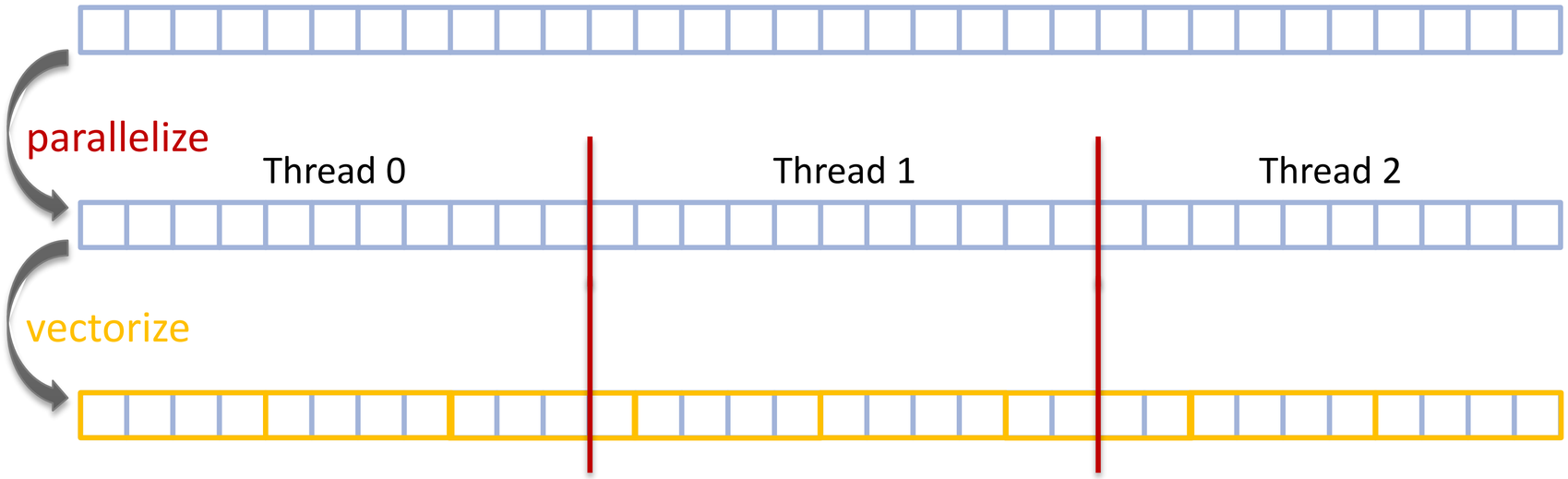
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp do simd [clause[[,] clause],…]
do-loops
[!$omp end do simd [nowait]]
```

SIMD Worksharing

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

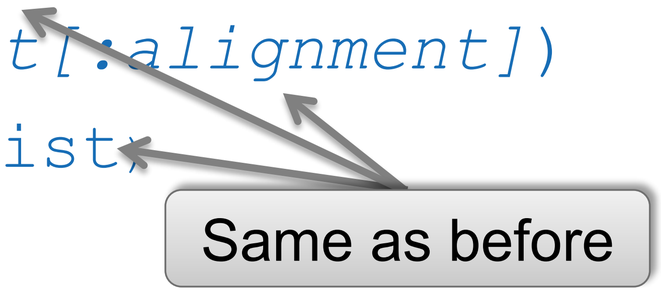
```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb), vc)
```

Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length
- `uniform (`*`argument-list`*`)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement
- `linear (`*`argument-list[:linear-step]`*`)`
- `aligned (`*`argument-list[:alignment]`*`)`
- `reduction (`*`operator`*`:list)`



Automatic Vectorization

- Automatic vectorization is when the compiler translates loops into SIMD instructions
- Most modern compilers support automatic vectorization
- In GCC, automatic vectorization is enabled with the `-O3` optimization flag
  - Specifically this enables `-ftree-loop-vectorize` and `-ftree-slp-vectorize`

Profile Guided Optimization

- Profile guided optimization is when trial runs are used to help the compiler vectorized code on subsequent builds
- Most modern compilers support profile guided optimization
- Profile guided optimization is a two step process
  - Build and run the code with the -fprofile-generate GCC flag
  - Rebuild and runt he code with -fprofile-use GCC flag

Intrinsic Functions

- Intrinsic functions are functions with implementations treated specially by the compiler
- They are frequently used as a method to access specific hardware instructions
  - Easier to implement and read than inline assembly
  - Improved code portability
- Most modern compilers support intrinsic functions that represent specific SIMD instructions

AVX Data Types

Type	Width [bits]	Contained Data
`__m128`	128	4 `floats`
`__m128d`	128	2 `doubles`
`__m128i`	128	16 `chars`, 8 `shorts`, 4 `ints`, 2 `longs`
`__m256`	256	8 `floats`
`__m256d`	256	4 `doubles`
`__m256i`	256	32 `chars`, 16 `shorts`, 8 `ints`, 4 `longs`
`__m512`	512	16 `floats`
`__m512d`	512	8 `doubles`
`__m512i`	512	64 `chars`, 32 `shorts`, 16 `ints`, 8 `longs`

AVX Function Naming Conventions

```
__mm<bit_width>_<name>_<data_type>
```

`<bit_width>`	Width of vectors, which for 128-bit vectors is absent
`<name>`	Description of what the intrinsic function does
`<data_type>`	Data type of intrinsic function's arguments

AVX Argument Naming Conventions

Name	Data Type
`ps`	`float`
`pd`	`double`
`epi{8,16,32,64}`	{8,16,32,64}-bit signed integers
`epu{8,16,32,64}`	{8,16,32,64}-bit unsigned integers
`si{128,256}`	Unspecified {128,256}-bit vector
`m128{d,i}`	Indicates input type when differing from return type (128-bit)
`m256{d,i}`	Indicates input type when differing from return type (256-bit)

Example AVX Instructions Program

```c
#include <immintrin.h>
#include <stdio.h>

int main() {

  /* Initialize the two argument vectors */
  __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
  __m256 odds = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

  /* Compute the difference between the two vectors */
  __m256 result = _mm256_sub_ps(evens, odds);

  /* Display the elements of the result vector */
  float* f = (float*)&result;
  printf("%f %f %f %f %f %f %f %f\n",
    f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);

  return 0;
}
```

Compiling AVX Programs

GCC

- Version 4.6 and 4.7 or newer is required for AVX and AVX2 respectively
- The flags `-mavx` or `-mavx2` is required for AVX and AVX2 respectively
- Example: `gcc -mavx -o hello_avx hello_avx.c`

Intel

- Version 11.1 and 14.0 or newer is required for AVX and AVX2 respectively
- The flags `-march=native` (if compiling on AVX(2) enabled CPU), `-axAVX`, or `-axAVX2` is required for AVX and AVX2 respecively
- Example: `icc -march=native -o hello_avx hello_avx.c`

PGI

- Example: `pgcc -o hello_avx hello_avx.c`

Readings and Assignments

- Readings
  - None
- Project
  - Report three targets for optimization
  - Report baseline performance, i.e. the pre-optimization performance
  - Commit details to your project repo
  - Due 12:00 AM Tuesday, July 19