Vectorization

Silvio Luiz Stanzani , Raphael Mendes de Oliveira Cóbe , Rogério Luiz Iope

NCC/UNESP

silvio@ncc.unesp.br, rmcobe@ncc.unesp.br , rogerio@ncc.unesp.br

Agenda

- Hybrid Parallel Architectures **(10 min)**
- Memory System **(10 min)**
- Vector Processing Units **(10 min)**
- Profiling **(5 min)**
  - Optimization report
  - Intel Advisor
- Optimizing Memory Access **(25 min)**
  - Gather/scatter Pattern
  - Data Layout
    - Padding
    - AOS - SOA
    - Memory Alignment
    - Loop transformation
- Auto Vectorization **(10 min)**
  - Parameters for Compilation
- Guided Vectorization **(20 min)**
  - Compiler directives

Agenda

- Hybrid Parallel Architectures;
- Memory System;
- Vector Processing Units;
- Profiling;
- Optimizing Memory Access;
- Auto Vectorization;
- Guided Vectorization.

Agenda

- **Hybrid Parallel Architectures;**
- Memory System;
- Vector Processing Units;
- Profiling;
- Optimizing Memory Access;
- Auto Vectorization;
- Guided Vectorization.

Hybrid Parallel Architectures

- Heterogeneous computational systems:
  - Multicore processors;
  - Multi-level memory sub-system;
  - Input and Output sub-system;
- Multi-level parallelism:
  - Processing core;
  - Chip multiprocessor;
  - Computing node;
  - Computing cluster;
- Hybrid Parallel architectures
  - Coprocessors and accelerators;

Hybrid Parallel Architectures

- Heterogeneous computational systems:
  - Scalar and Vector Instructions

Vector Instructions (SIMD)

A7	A6	A5	A4	A3	A2	A1	A0
			+				
B7	B6	B5	B4	B3	B2	B1	B0
			=				
A7+B7	A6+B6	A5+B5	A4+B4	A3+B3	A2+B2	A1+B1	A0+B0

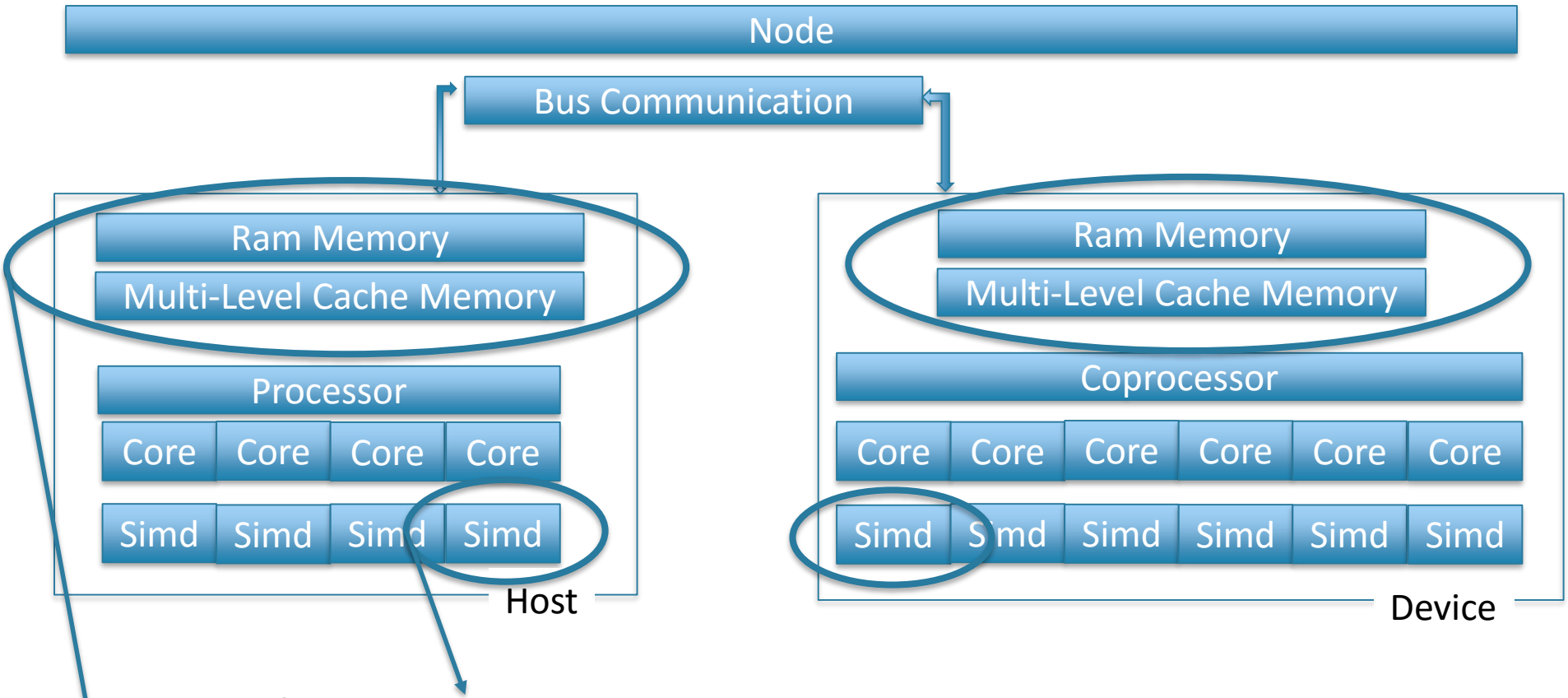
Scalar Instructions

A
+
B
=
A+B

  - Multi-level memory
    - Ram Memory;
    - Multi-level Cache.

Processor 1			Processor 2		
Core 1	Core 2	Core N	Core 1	Core 2	Core N
L1	L1	L1	L1	L1	L1
L2	L2	L2	L2	L2	L2
L3			L3		
Ram					

Hybrid Parallel Architectures

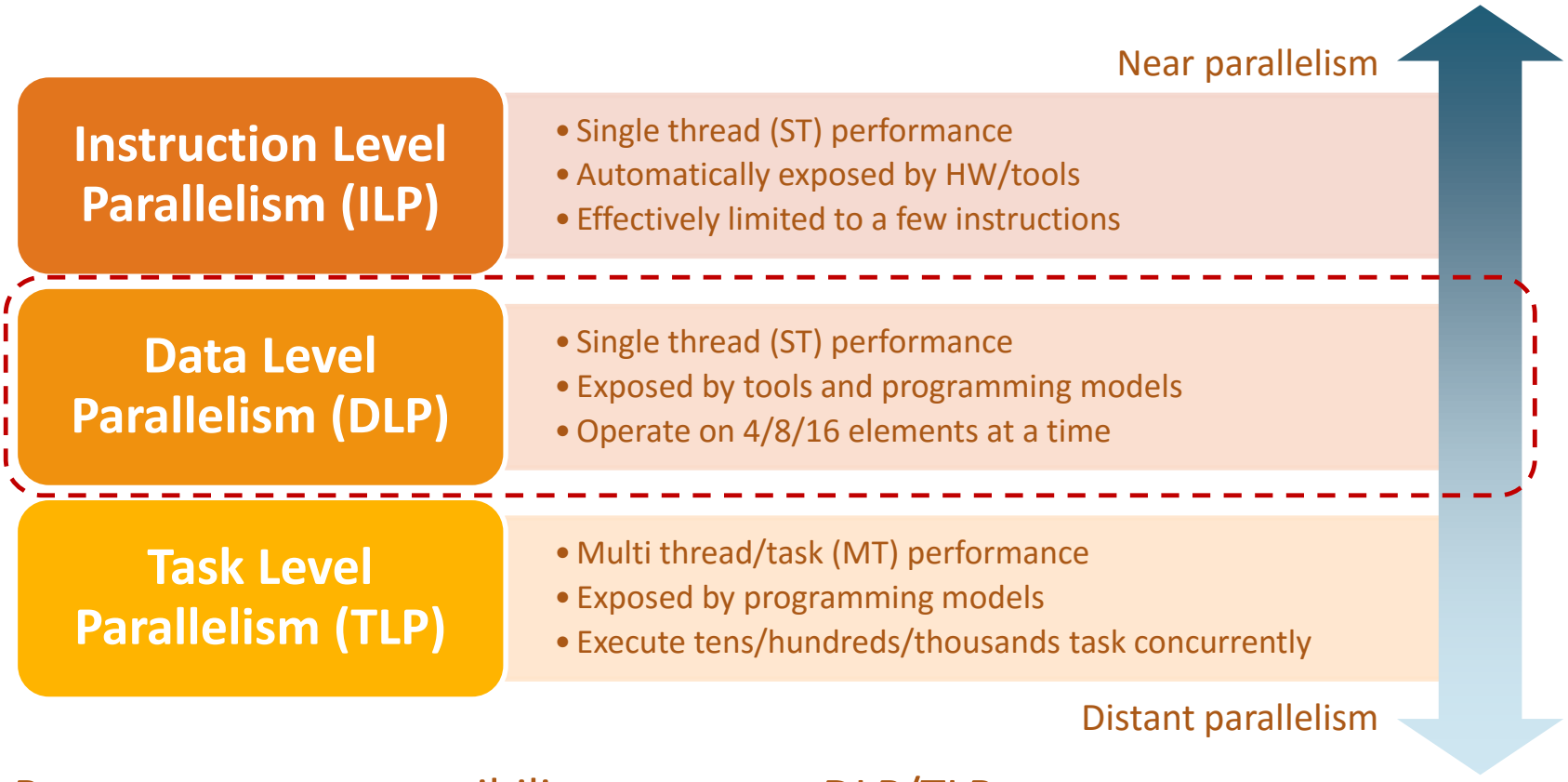


- Simd Instructions
- Optimized Memory Access

→ Vectorization!

Don’t use a single thread or vector lane

Exploiting the parallel universe



Programmers responsibility to expose DLP/TLP

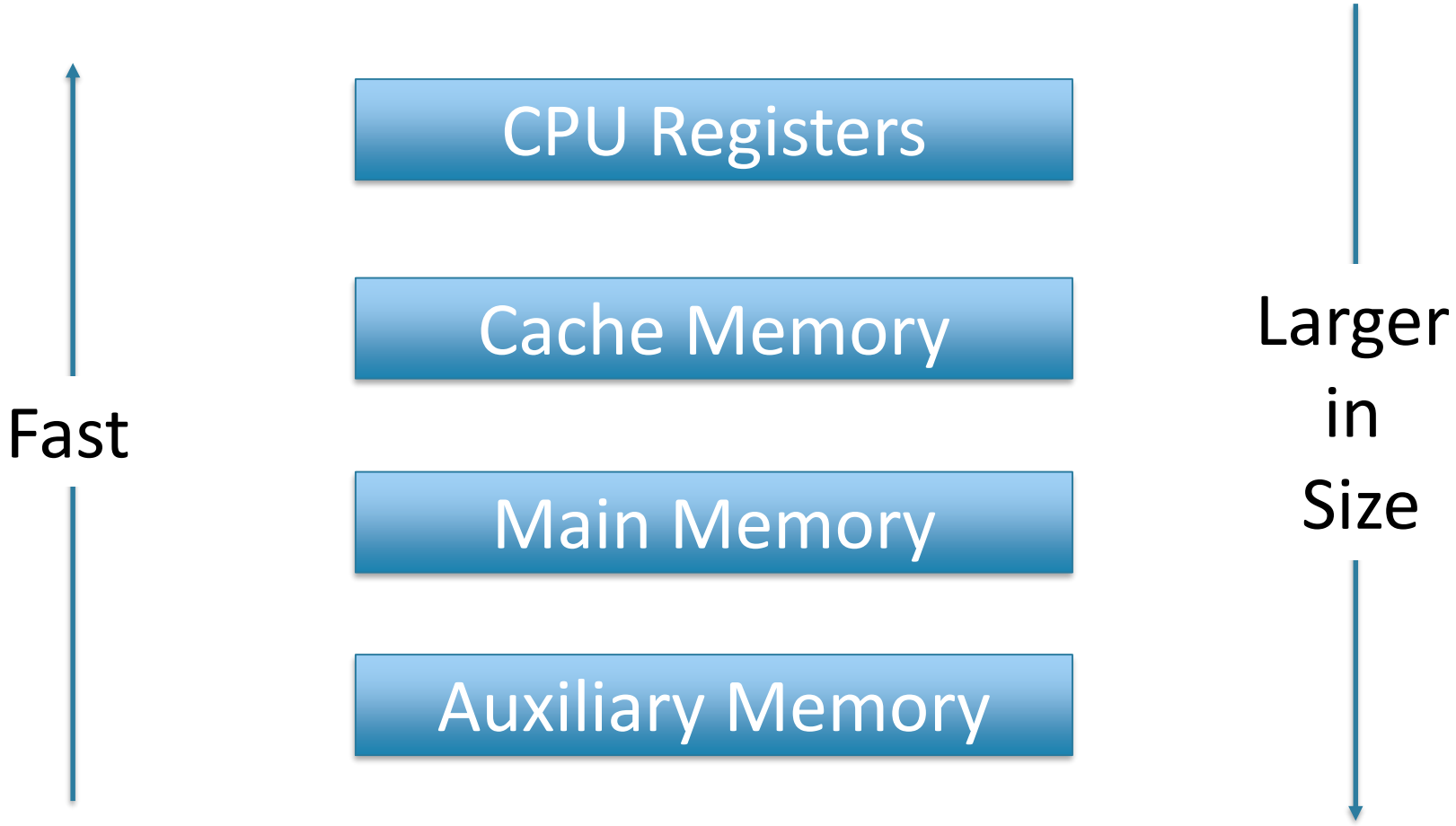
Agenda

- Hybrid Parallel Architectures**;**
- **Memory System;**
- Vector Processing Units;
- Profiling;
- Optimizing Memory Access;
- Auto Vectorization;
- Guided Vectorization.

Memory System

- CPU Register: internal Processor Memory. Stores the data or instruction which has to be executed;
- Cache: stores segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations;
- Main memory: only program and data currently needed by the processor resides in main memory;
- Auxiliary memory: devices that provides backup storage.

Memory Hierarchy



Cache Memory

- The Cache Memory is employed in computer system to compensate for the speed differential between main memory access time and processor logic.

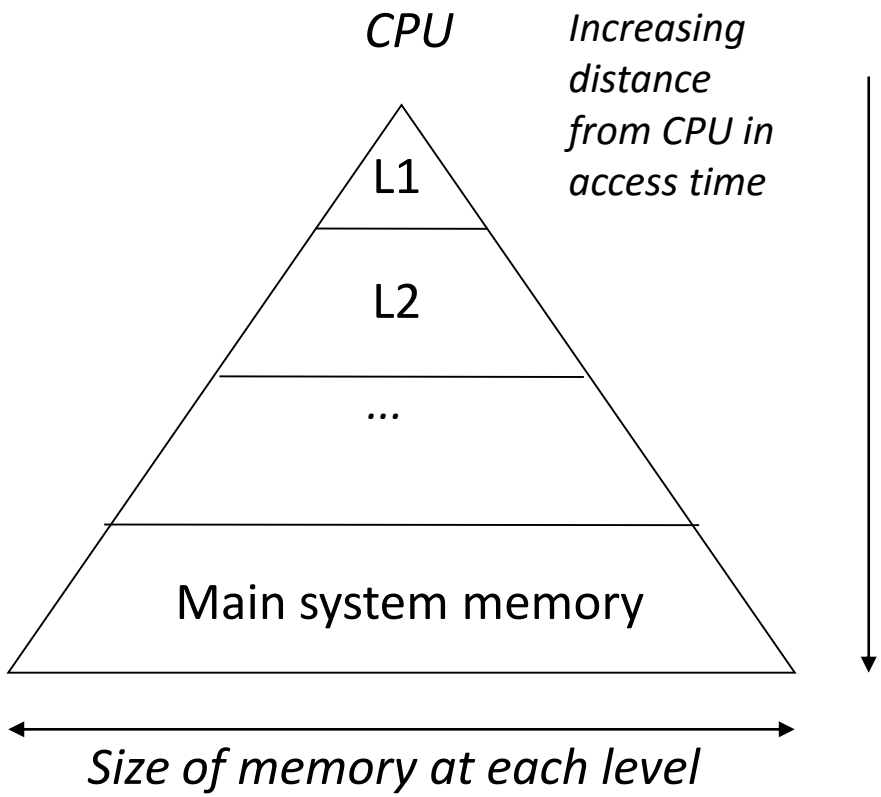
- Operational System Controls the load of Data to Cache, such load can be guided by the developer

Cache Memory

- The Performance of cache memory is frequently measured in terms of hit ratio.
  - When the CPU refers to memory and finds the word in cache, it is said to produce a **hit**.
  - If the word is not found in cache, it is in main memory and it counts as a **miss**

Locality

- Temporal locality: if an item was referenced, it will be referenced again soon (e.g. cyclical execution in loops);
- Spatial locality: if an item was referenced, items close to it will be referenced too (the very nature of every program – serial stream of instructions)



Agenda

- Hybrid Parallel Architectures**;**
- Memory System;
- **Vector Processing Units;**
- Profiling;
- Optimizing Memory Access;
- Auto Vectorization;
- Guided Vectorization.

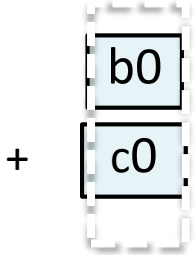
Scalar and Vector Instructions

- Scalar Code computes this one-element at a time.

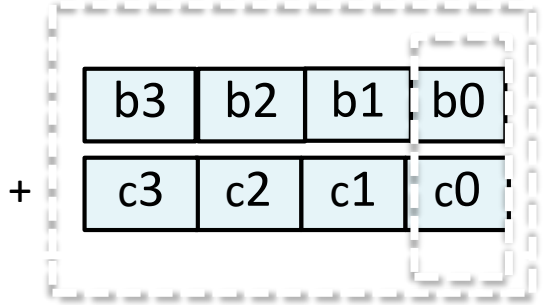
- Vector (or SIMD) Code computes more than one element at a time. SIMD stands for **S**ingle **I**nstruction **M**ultiple **D**ata.

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

- Scalar



- SIMD



Vectorization

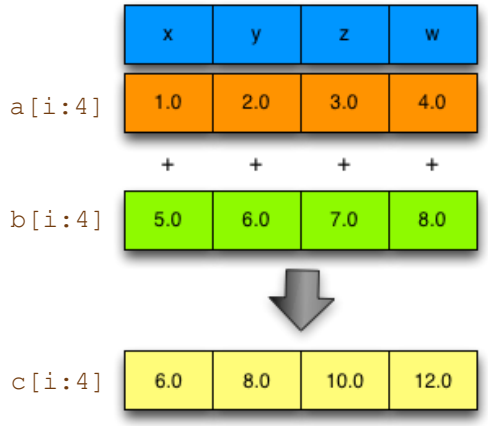
- Vectorization
  - Loading data into cache accordingly;
  - Store elements on SIMD registers or vectors;
  - Apply the same operation to a set of Data at the same time;
  - Iterations needs to be independent;
  - Usually on inner loops.

Scalar loop

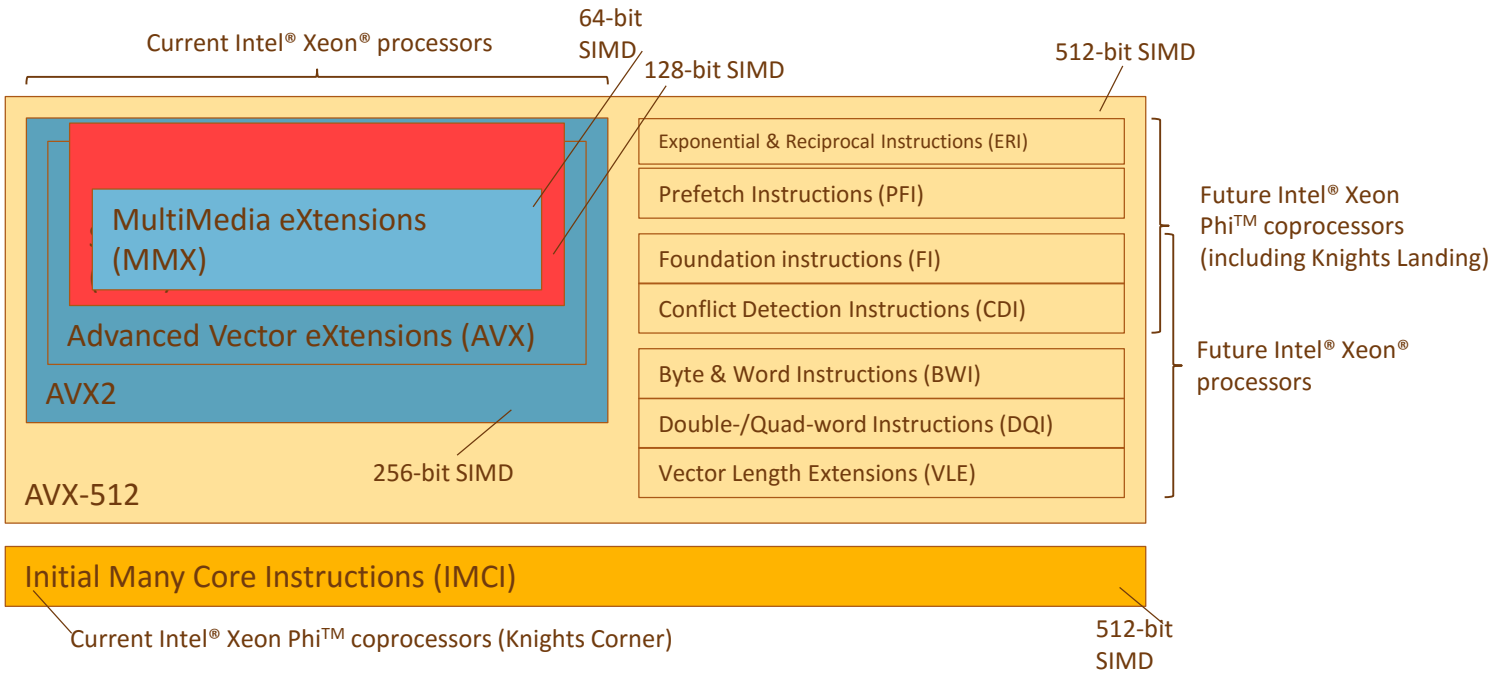
```
for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

SIMD loop (4 elements)

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



Past, present, and future of Intel SIMD types



Intel® AVX2/IMCI/AVX-512 differences

	Intel® Initial Many Core Instructions **IMCI**	Intel® Advanced Vector Extensions 2 **AVX2**	Intel® Advanced Vector Extensions 512 **AVX-512**
Introduction	2012	2013	2015
Products	Knights Corner	Haswell, Broadwell	Knights Landing, future Intel® Xeon® and Xeon® Phi™ products
Register file	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x 16-bit mask registers	SP/DP/int32/int64 data types 16 x 256-bit SIMD registers No mask registers (instr. blending)	SP/DP/int32/int64 data types 32 x 512-bit SIMD registers 8 x (up to) 64-bit mask
ISA features	Not compatible with AVX*/SSE* No unaligned data support Embedded broadcast/cvt/swizzle MVEX encoding	Fully compatible with AVX/SSE* Unaligned data support (penalty) VEX encoding	Fully compatible with AVX*/SSE* Unaligned data support (penalty) Embedded broadcast/rounding EVEX encoding
Instruction features	Fused multiply-and-add (FMA) Partial gather/scatter Transcendental support	Fused multiply-and-add (FMA) Full gather	Fused multiply-and-add (FMA) Full gather/scatter Transcendental support (ERI only) Conflict detection instructions PFI/BWI/DQI/VLE (if applies)

Agenda

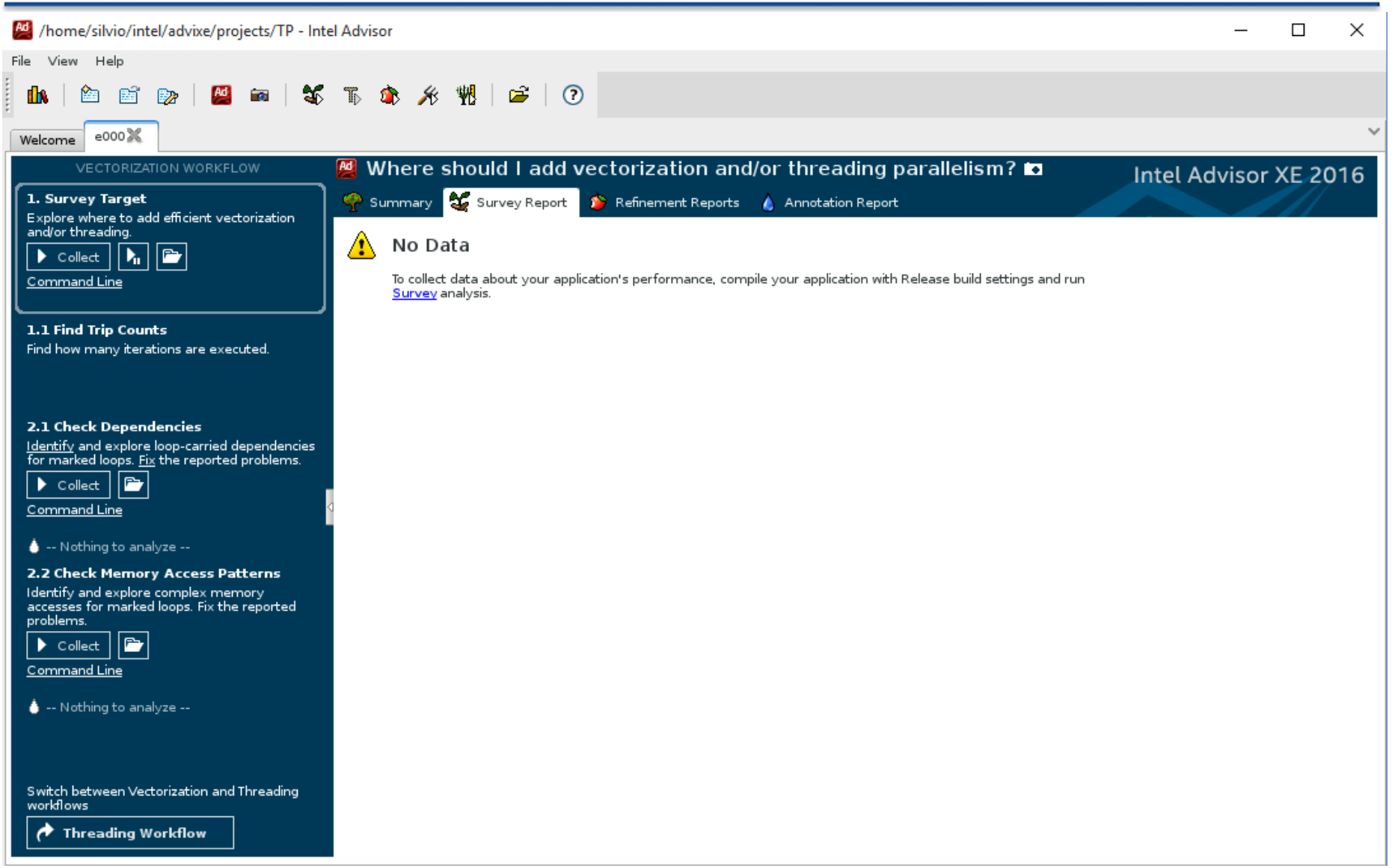
- Hybrid Parallel Architectures**;**
- Memory System;
- Vector Processing Units;
- **Profiling;**
- Optimizing Memory Access;
- Auto Vectorization;
- Guided Vectorization.

Intel Advisor

- Evaluate multi-threading parallelization

- Intel® Advisor XE
  - ❑ Performance modeling using several frameworks for multi-threading in processors and co-processors:
    - OpenMP, Intel® Cilk ™ Plus, Intel® Threading Bulding Blocks
    - C, C++, Fortran (OpenMP only) and C# (Microsoft TPL)
  - ❑ Identify parallel opportunities
    - Detailed information about vectorization;
    - Check loop dependencies;
  - ❑ Scalability prediction: amount of threads/performance gains
  - ❑ Correctness (deadlocks, race condition)

Intel Advisor



Intel Advisor

- Survey Target;
  - Vectorization of loops: detailed information about vectorization;
  - Total Time: elapsed time on each loop considering the time involved in internal loops;
  - Self Time: elapsed time on each loop not considering the time involved in internal loops;
- Find Trip Counts;
  - Analysis to identify how many time particular loops run;
- Check Dependencies;
  - Analysis it there are many loop-carried dependencies;
- Check Memory Access Patterns.
  - Analysis to identify how your code is iterating through memory.

Agenda

- Hybrid Parallel Architectures**;**
- Memory System;
- Vector Processing Units;
- Profiling;
- **Optimizing Memory Access;**
- Auto Vectorization;
- Guided Vectorization.

Stride (array elements)

- Stride:
  - Step size between consecutive access of array elements;

- Strided access with stride k means touching every kth memory element
  - Unit Stride :
    - Sequential access (0, 1, 2, 3, 4, 5, 6, ...)
  - Non-unit stride
    - Constant Stride =
      - 2 is (0, 2, 4, 6, 8, ...)
    - k is (0, k, 2k, 3k, 4k, ...)
    - Random Access;

- Strides > 1 commonly found in multidimensional data
  - Row accesses (stride=N) & diagonal accesses (stride=N+1)
  - Scientific computing (e.g., matrix multiplication)

Padding

- Data structures may have members with different sizes.
- To maintain proper alignment the translator normally inserts additional unnamed data members so that each member is properly aligned.
- Example:

```
struct stu_a {
    int i;
    char c;
};
```

- Actual size 4+1 (5)

- After Padding size 4+4 (5)

- ...

Padding

- Vectorization more efficient with unit strides
  - Non-unit strides will generate gather/scatter
  - Unit strides also better for data locality

Demo: padd.c

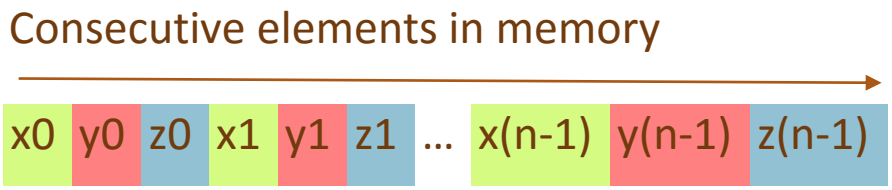
Icc padd.c –o padd

./padd

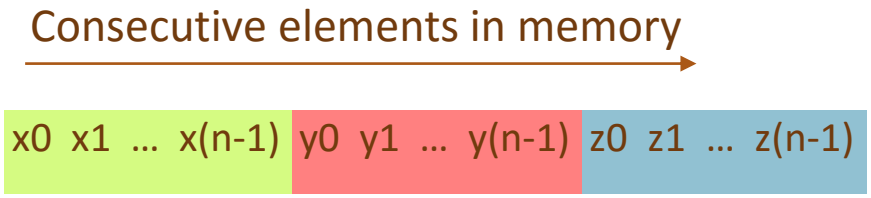
Data layout

- AoS vs SoA (Array of Structures vs Structure of Arrays)
  - Layout your data as Structure of Arrays (SoA)

```
// Array of Structures (AoS)
struct coordinate {
    float x, y, z;
} crd[N];
    …
for (int i = 0; i < N; i++)
    … = … f(crd[i].x, crd[i],y, crd[i].z);
```




```
// Structure of Arrays (SoA)
struct coordinate {
    float x[N], y[N], z[N];
} crd;
    …
for (int i = 0; i < N; i++)
    … = … f(crd.x[i], crd.y[i], crd.z[i]);
```



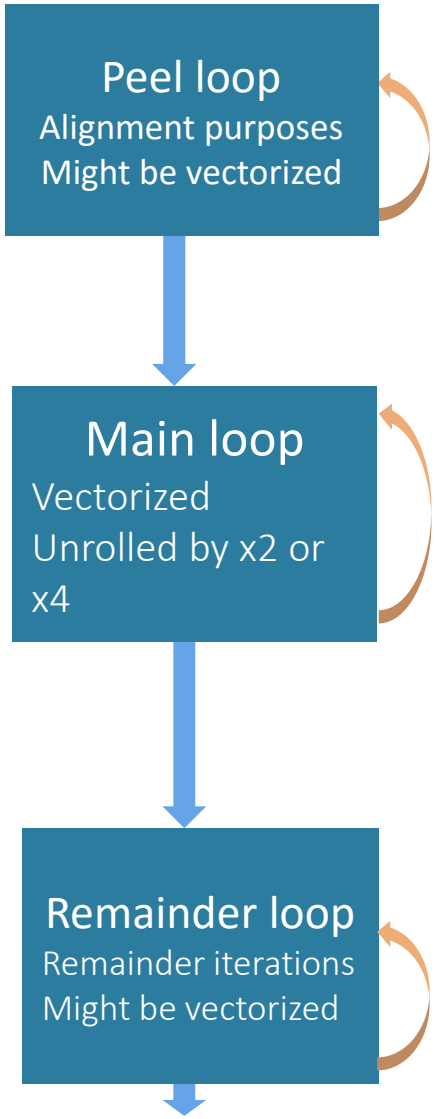
Data Alignment

How to…	Syntax	Semantics
…align data	`void* _mm_malloc(int size, int n)` `void* _mm_free(int size)`	Allocate memory on heap aligned to *n* byte boundary.
	`int posix_memalign (void **p, size_t n, size_t size)`	
	`__declspec(align(n)) array`	Alignment for variable declarations.
…tell the compiler about it	`#pragma vector aligned`	Vectorize assuming all array data accessed are aligned (may cause fault otherwise).
	`__assume_aligned(array, n)`	Compiler may assume array is aligned to *n* byte boundary.

Loop Spliting

- Loop Spliting
  - Set of techniques to breaking the loop into multiple loops which have the same body, but iterate over different contiguous portions of the index range.
    - Body
    - Peel Loop: beginning of loop
    - Remainder Loop: end of loop
- Loop Unrolling
  - Execute a set of iterations as a single iteration;

Vectorization with multi-version loops

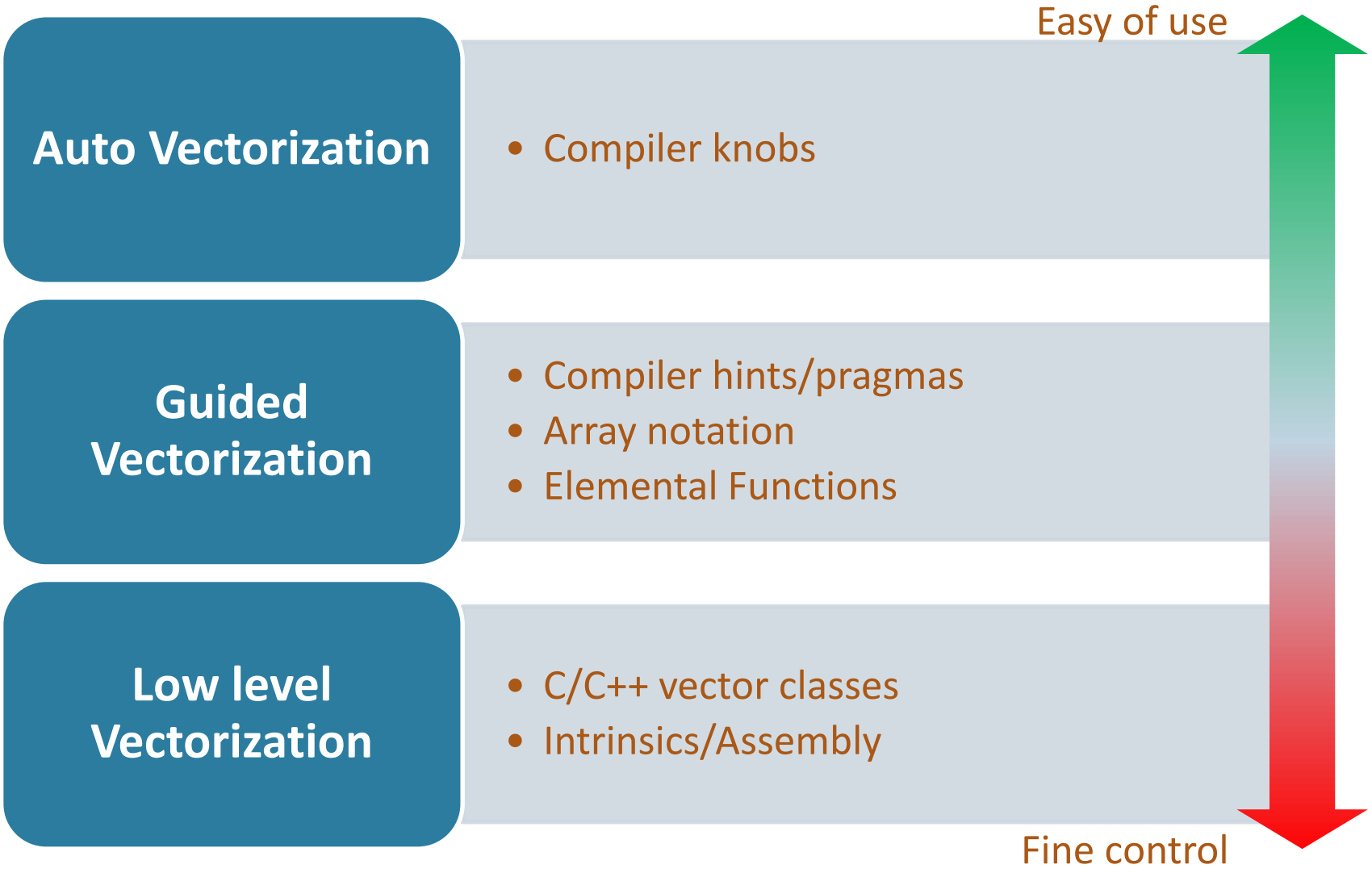



```
LOOP BEGIN at gas_dyn2.f90(2330,26)
<Peeled>
   remark #15389: vectorization support: reference AMAC1U has unaligned
access
   remark #15381: vectorization support: unaligned access used inside loop
body
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)
   remark #25084: Preprocess Loopnests: Moving Out Store
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 0.620
   remark #15478: estimated potential speedup: 15.890
   remark #15479: lightweight vector operations: 5
   remark #15488: --- end vector loop cost summary ---
   remark #25018: Total number of lines prefetched=4
   remark #25019: Number of spatial prefetches=4, dist=8
   remark #25021: Number of initial-value prefetches=6
LOOP END

LOOP BEGIN at gas_dyn2.f90(2330,26)
<Remainder>
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15388: vectorization support: reference AMAC1U has aligned access
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Agenda

- Hybrid Parallel Architectures**;**
- Memory System;
- Vector Processing Units;
- Profiling;
- Optimizing Memory Access;
- **Auto Vectorization;**
- Guided Vectorization.

Vectorization on Intel® compilers



Auto vectorization

- Relies on the compiler for vectorization
  - No source code changes
  - Enabled with `-vec` compiler knob (default in `-O2` and `-O3` modes)
- Compiler smart enough to apply loop transformations
  - It will allow to vectorize more loops

Option	Description
`-O0`	Disables all optimizations.
`-O1`	Enables optimizations for speed which are know to not cause code size increase.
`-O2/-O` (default)	Enables intra-file interprocedural optimizations for speed, including: • **Vectorization** • **Loop unrolling**
`-O3`	**Performs O2 optimizations and enables more aggressive loop transformations such as:** • **Loop fusion** • **Block unroll-and-jam** • **Collapsing IF statements** This option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. However, it might incur in slower code, numerical stability issues, and compilation time increase.

Vectorization: target architecture options

Option	Description
-mmic	Builds an application that runs natively on Intel® MIC Architecture.
-xfeature -xHost	Tells the compiler which processor features it may target, referring to which instruction sets and optimizations it may generate (not available for Intel® Xeon Phi™ architecture). Values for *feature* are: • COMMON-AVX512 (includes AVX512 FI and CDI instructions) • **MIC-AVX512 (includes AVX512 FI, CDI, PFI, and ERI instructions)** • **CORE-AVX512 (includes AVX512 FI, CDI, BWI, DQI, and VLE instructions)** • **CORE-AVX2** • CORE-AVX-I (including RDRND instruction) • AVX • SSE4.2, SSE4.1 • ATOM_SSE4.2, ATOM_SSSE3 (including MOVBE instruction) • SSSE3, SSE3, SSE2 **When using -xHost, the compiler will generate instructions for the highest instruction set available on the compilation host processor.**
-axfeature	Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel® processors if there is a performance benefit. Values for *feature* are the same described for -xfeature option. Multiple features/paths possible, e.g.: -axSSE2,AVX. It also generates a baseline code path for the default case.

Auto vectorization: not all loops will vectorize

- Data dependencies between iterations
  - Proven Read-after-Write data (i.e., loop carried) dependencies
  - Assumed data dependencies
    - Aggressive optimizations

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

- Vectorization won't be efficient
  - Compiler estimates how better the vectorized version will be
  - Affected by data alignment, data layout, etc.

Inefficient vectorization

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

- Unsupported loop structure
  - While-loop, for-loop with unknown number of iterations
  - Complex loops, unsupported data types, etc.
  - (Some) function calls within loop bodies

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Validating vectorization

- Generate compiler report about optimizations

  `-qopt-report[=n]` Generate report (level [1..6], default 2)

```
LOOP BEGIN at gas_dyn2.f90(193,11) inlined into gas_dyn2.f90(4326,31)
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 53
   remark #15477: vector loop cost: 14.870
   remark #15478: estimated potential speedup: 2.520
   remark #15479: lightweight vector operations: 19
   remark #15481: heavy-overhead vector operations: 1
   remark #15488: --- end vector loop cost summary ---
   remark #25456: Number of Array Refs Scalar Replaced In Loop: 1
   remark #25015: Estimate of max trip count of loop=4
LOOP END
```

Vectorized loop

```
LOOP BEGIN at gas_dyn2.f90(2346,15)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between IOLD line 376 and IOLD line 354
   remark #25015: Estimate of max trip count of loop=3000001
LOOP END
```

Non-vectorized loop

Agenda

- Hybrid Parallel Architectures**;**
- Memory System;
- Vector Processing Units;
- Profiling;
- Optimizing Memory Access;
- Auto Vectorization;
- **Guided Vectorization.**

Intel® compiler directives for vectorization

Directive	Clause	Description
`ivdep`		Instructs the compiler to ignore assumed vector dependencies.
`vector`	`always`	Force vectorization even when it might be not efficient.
	`[un]aligned`	Use [un]aligned data movement instructions for all array vector references.
	`[non]temporal(var1[,…])`	Do or do not generate non-temporal (streaming) stores for the given array variables. On Intel® MIC architecture, generates a cache-line-evict instruction when the store is known to be aligned.
	`[no]vecreminder`	Do (not) vectorize the remainder loop when the main loop is vectorized.
	`[no]mask_readwrite`	Enables/disables memory speculation causing the generation of [non-]masked loads and stores within conditions.
`simd`	`vectorlength(n1[,…])` `vectorlengthfor(dtype)`	Assume safe vectorization for the given vector length values or data type.

	`private(var1[,…])`
	`firstprivate(var1[,…])`
	`lastprivate(var1[,…])`
		Which variables are private to each iteration; *firstprivate*, initial value is broadcasted to all private instances; *lastprivate*, last value is copied out from the last instance.

	`linear(var1:step1[,…])`	Letting know the compiler that *var1* is incremented by *step1* on every iteration of the original loop.
	`reduction(oper:var1[,…])`	Which variables are reduction variables with a given operator.
	`[no]assert`	Warning or error when vectorization fails.
	`[no]vecremainder`	Do (not) vectorize the remainder loop when the mail loop is vectorized.

Guided vectorization: disambiguation hints

- Assume function arguments won't be aliased
  - C/C++: Compile with `-fargument-noalias`
- C99 "restrict" keyword for pointers
  - Compile with `-restrict` otherwise

```
void v_add(float *restrict c,
           float *restrict a,
           float *restrict b)
{
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

```
void v_add(float *c, float *a, float *b)
{
for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Guided vectorization:

- `#pragma simd or #pragma ivdep`
  - Force loop vectorization ignoring **all** dependencies
    - Additional clauses for specify reductions, etc.

```
void v_add(float *c, float *a, float *b)
{
#pragma simd
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

SIMD loop

```
__declspec(vector)
void v_add(float c, float a, float b)
{
    c = a + b;
}
    …
for (int i = 0; i < N; i++)
    v_add(C[i], A[i], B[i]);
```

SIMD function

Example

- Particle Binning Problem**[1]**

- Optimizations:
  - Automatic Vectorization
  - Data Alignment

[1] http://colfaxresearch.com/optimization-techniques-for-the-intel-mic-architecture-part-2-of-3-strip-mining-for-vectorization/

Particle Binning - Serial

```
for (int i = 0; i < inputData.numDataPoints; i++) {

    // Transforming from cylindrical to Cartesian coordinates:
    const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
    const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);

     // Calculating the bin numbers for these coordinates:
     const int iX = int((x - xMin)*binsPerUnitX);
     const int iY = int((y - yMin)*binsPerUnitY);

     // Incrementing the appropriate bin in the thread-private counter:
     threadPrivateBins[iX][iY]++;
  }
```

Particle Binning - Vectorized

```
for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {

    int iX[STRIP_WIDTH];
    int iY[STRIP_WIDTH];

    const FTYPE* r   = &(inputData.r[ii]);
    const FTYPE* phi = &(inputData.phi[ii]);

    // Vector loop
    for (int c = 0; c < STRIP_WIDTH; c++) {
       // Transforming from cylindrical to Cartesian coordinates:
       const FTYPE x = r[c]*COS(phi[c]);
     const FTYPE y = r[c]*SIN(phi[c]);

       // Calculating the bin numbers for these coordinates:
       iX[c] = int((x - xMin)*binsPerUnitX);
       iY[c] = int((y - yMin)*binsPerUnitY);
    }

}
```

Particle Binning – Data Alignment

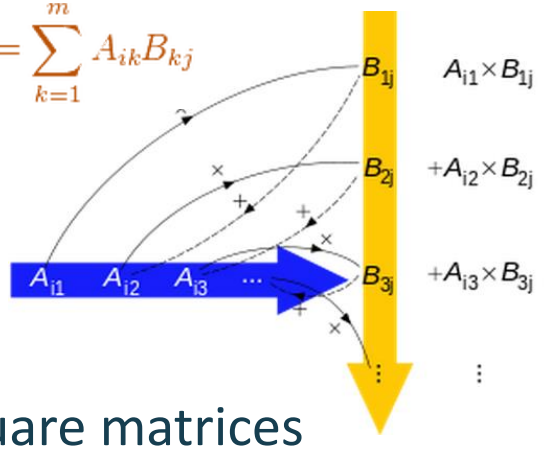
```
for (int ii = 0; ii < inputData.numDataPoints; ii += STRIP_WIDTH) {

    int iX[STRIP_WIDTH] __attribute__((aligned(64)));
    int iY[STRIP_WIDTH] __attribute__((aligned(64)));

    const FTYPE* r   = &(inputData.r[ii]);
    const FTYPE* phi = &(inputData.phi[ii]);

    // Vector loop
#pragma vector aligned
    for (int c = 0; c < STRIP_WIDTH; c++) {
        // Transforming from cylindrical to Cartesian coordinates:
        const FTYPE x = r[c]*COS(phi[c]);
      const FTYPE y = r[c]*SIN(phi[c]);

        // Calculating the bin numbers for these coordinates:
        iX[c] = int((x - xMin)*binsPerUnitX);
        iY[c] = int((y - yMin)*binsPerUnitY);
    }

  }
```

Matrix multiplication (SGEMM)

- Problem definition
- $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix} \qquad (\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}$$

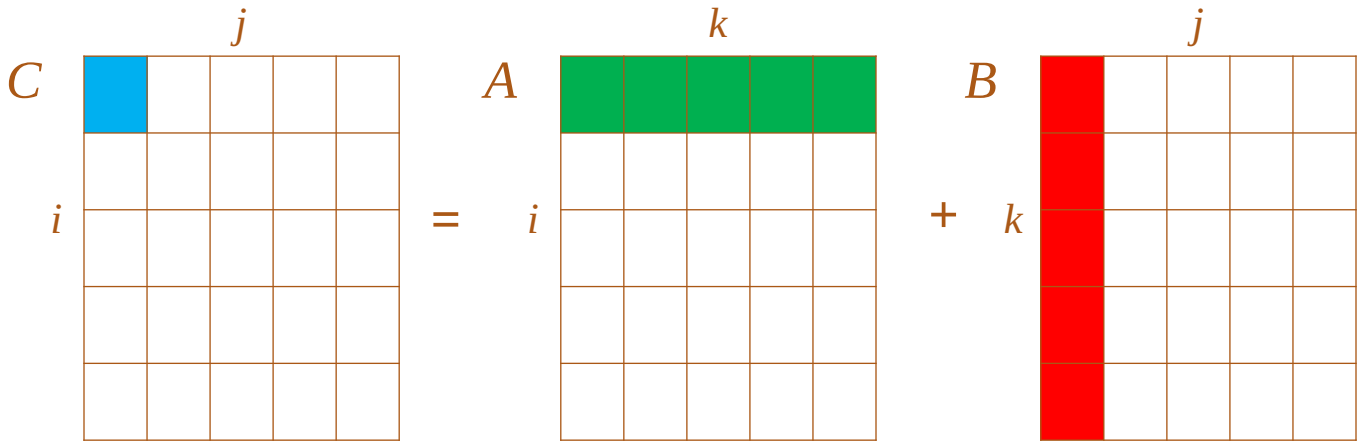


- Implementation
  - For simplification, no α, β parameters, square matrices
  - Based on "*Many faces of parallelism*", by Michael Hebenstreit

v0. Base version

```
void my_sgemm(float *a, float *b, float *c, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```

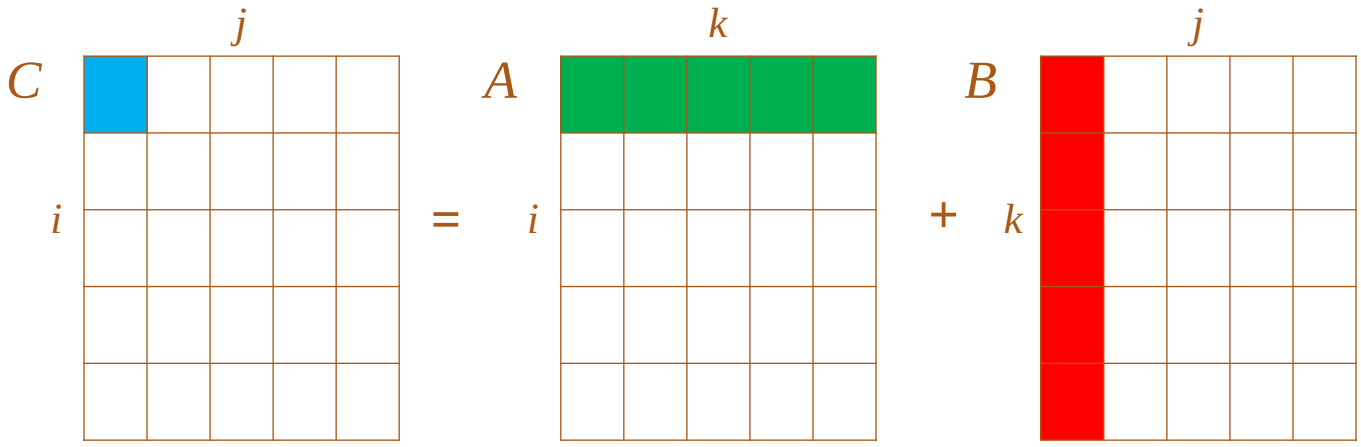
```
float *my_allocate(int n, int s)
{
    return (float *)malloc(s * n);
}
```



v0. Base version (cont'd)

```
void my_sgemm(float *a, float *b, float *c, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```

Not vectorized due to assume dependencies

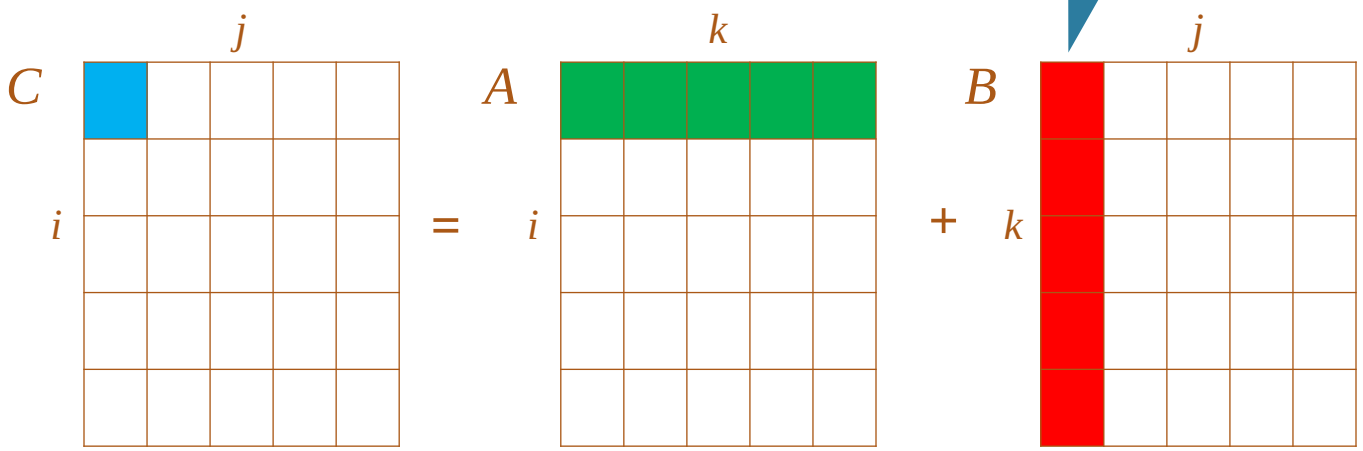


v1. Pragma SIMD for “bad” vectorization

```
void my_sgemm(float *a, float *b, float *c, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
#pragma simd
            for (int k = 0; k < n; k++)
                c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```

Vectorized on the “wrong” *k* dimension

Non-unit stride for B matrix!!!

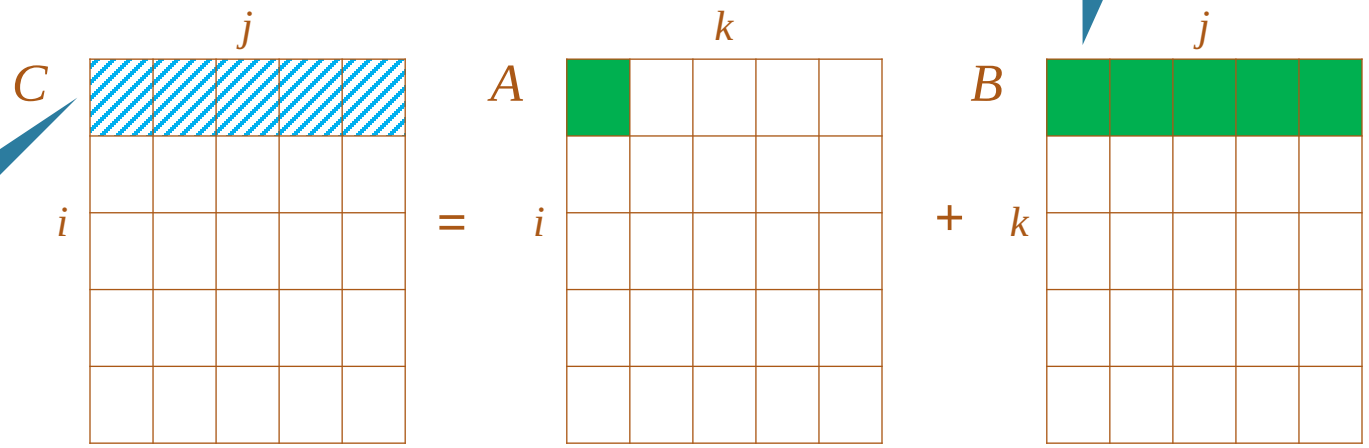


v2. Pragma SIMD on the right dimension

```
void my_sgemm(float *a, float *b, float *c, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
#pragma simd
            for (int j = 0; j < n; j++)
                c[i * n + j] += a[i * n + k] * b[k * n + j];
}
```

Vectorized on the right "j" dimension (loop interchange)

Unit stride for B matrix



Row *i* is partially updated on every *k* iteration

v3. Align matrix rows

```
float *my_allocate(int n0, int s)
{
    return (float *)_mm_malloc(s * n0, 64);
}
```

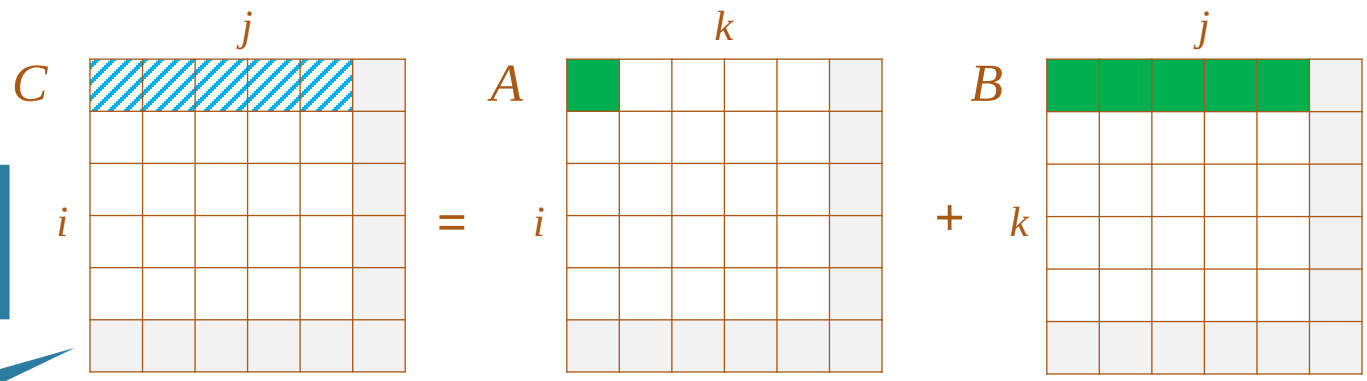
Every matrix will be aligned

```
void my_sgemm(float *a, float *b, float *c, int n, int n0)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
        for (int k = 0; k < n; k++)
#pragma simd
#pragma vector aligned
            for (int j = 0; j < n; j++)
                c[i * n0 + j] += a[i * n0 + k] * b[k * n0 + j];
}
```

```
/* padded dimension */
int nepc = 64 / sizeof(float);
int n0 = ((n - 1)/nepc + 1) * nepc;
```

Every row will be aligned

Compiler knows everything is aligned



Padding