Guided Code Vectorization with Intel® Advisor XE

The Hands-On Workshop (HOW) Series "Tools"

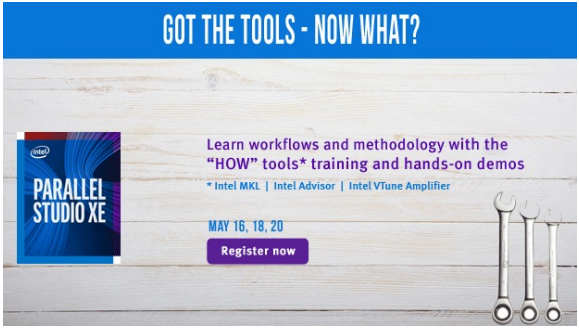
Ryo Asai — Colfax International
colfaxresearch.com

Disclaimer

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

About the Series

Hands-On Workshop (HOW “Tools” Series): webinars on efficient programming for the Intel architecture with the help of dedicated software development tools



colfaxresearch.com/how-tools-16-05

Learn More

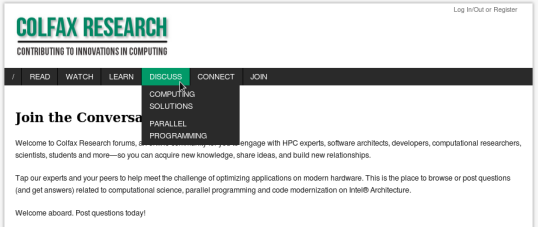
Interested? Sign-up at:

colfaxresearch.com/how-series

Get Your Questions Answered

Chat (for this course):
colfaxresearch.com/how-tools-16-05

Forums (general):
colfaxresearch.com/discussion



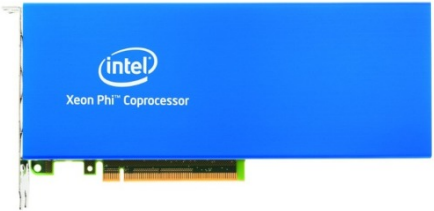
§2. Intel Architecture

Computing Platforms

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

* socket and coprocessor versions

Upcoming: Knights Landing (KNL)

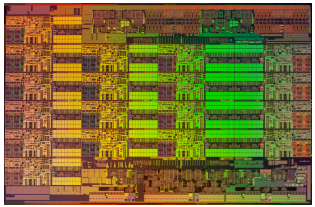
Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Intel Xeon CPU: Purpose and Specifications

General-purpose platform for demanding computing applications.

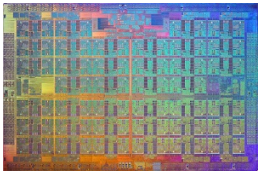
- Up to ~ 1 TFLOP/s in DP
- Up to ~ 2 TFLOP/s in SP
- Up to 768 GiB DDR4 RAM
- ~ 126 GB/s bandwidth
- Hardware-rich: forgiving of sub-optimal code



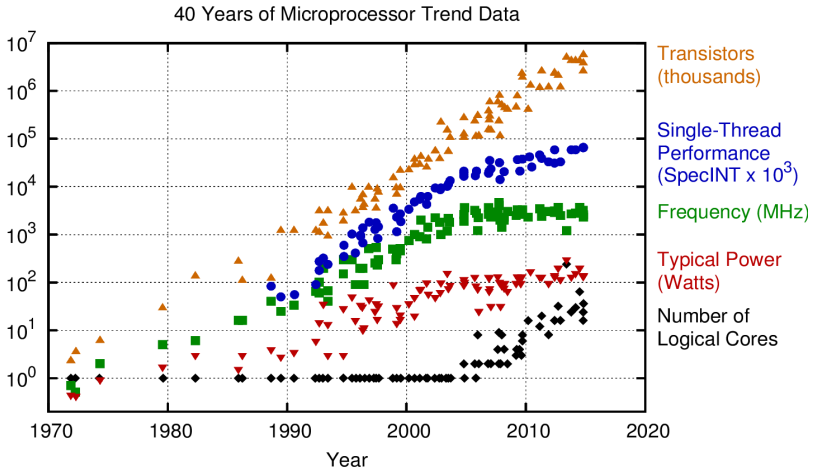
Intel Xeon Phi Processors (2nd Gen)

Specialized platform for demanding computing applications.

- Socket version or coprocessor
- 3+ TFLOP/s in DP
- 6+ TFLOP/s in SP
- Up to 16 GiB MCDRAM
- ~ 400 GB/s MCDRAM bandwidth
- Up to 384 GiB DDR4 RAM
- ~ 90 GB/s DDR4 bandwidth
- Supports common OS
- Public disclosures



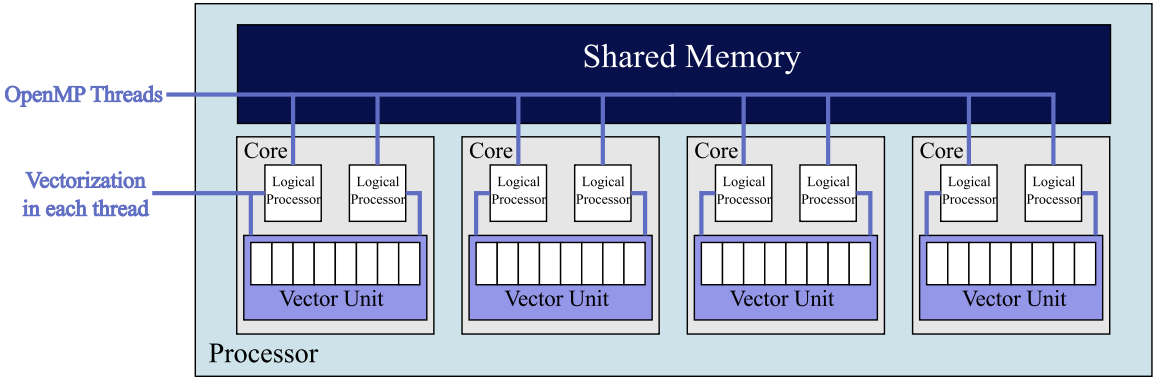
Trend Towards of Parallelism



Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Hardware Details of Parallelism

- **Data Parallelism** - Single Instruction Multiple Data (SIMD)
- **Task Parallelism** - Multiple tasks across multiple processor cores



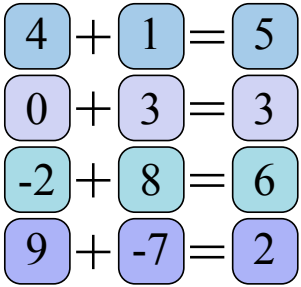
Intel® Advisor™ covers both: today we focus on Data Parallelism

§3. Background: Vectorization

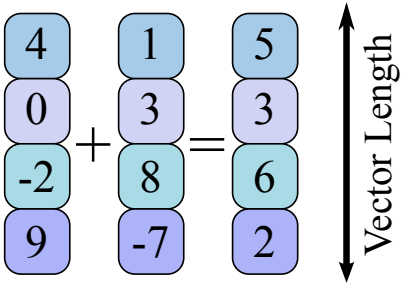
Short Vector Support

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

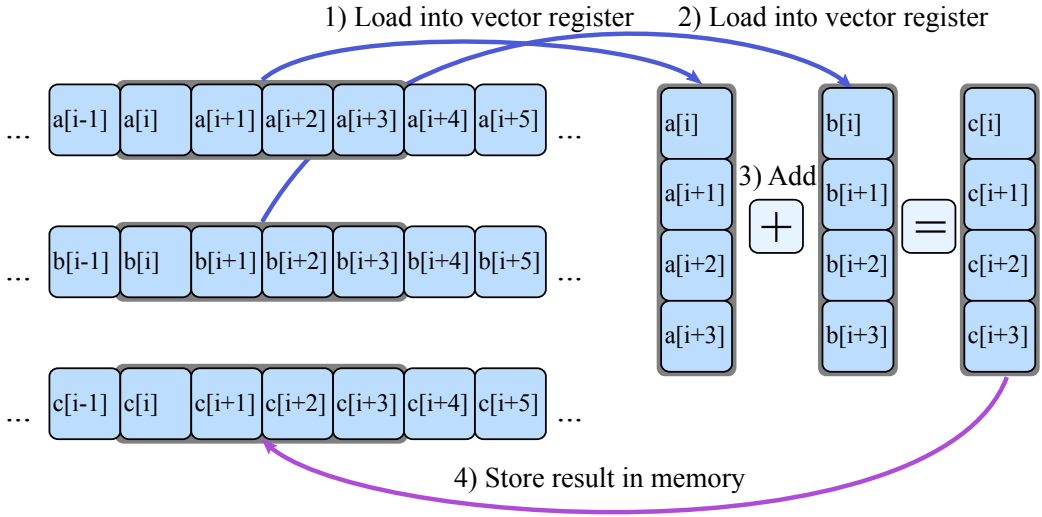
Scalar Instructions



Vector Instructions



Workflow of Vector Computation



Two Approaches To Vectorization

Automatic Vectorization:

- Vectorization w/ compiler.
- Portable: just recompile.
- Tuning with directives.

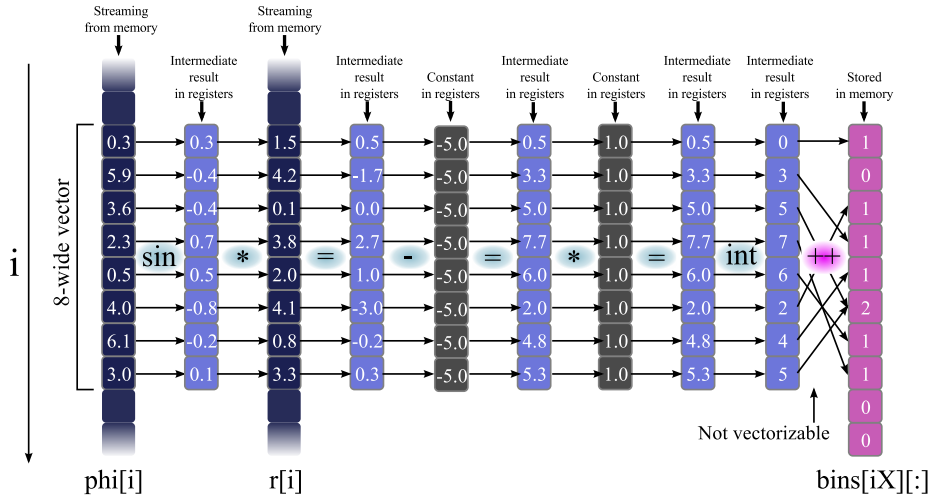
```
1 double A[vec_width], B[vec_width];
2 // ...
3
4
5 // This loop will be auto-vectorized
6 for(int i = 0; i < vec_width; i++)
7   A[i]+=B[i];
```

```
1 double A[vec_width], B[vec_width];
2 // ...
3 // This is explicitly vectorized
4 __m512d A_vec = _mm512_load_pd(A);
5 __m512d B_vec = _mm512_load_pd(B);
6 A_vec = _mm512_add_pd(A_vec,B_vec);
7 _mm512_store_pd(A,A_vec);
```

Explicit Vectorization:

- Vectorization w/ intrinsics
- Full control over instructions
- Limited portability

Auto-Vectorized Loops May Be Complex



See this paper for more details

Limits of Automatic Vectorization

The compiler will do the best it can. But there are issues it can't resolve.

Unvectorizable loops

```
vega@lyra% cat worker.optrpt
...
LOOP BEGIN at worker.cc(11,5)
 remark: loop was not vectorized: ...
 remark: vector dependence: ...
 remark: vector dependence: ...
LOOP END
```

Inefficient loops

```
vega@lyra% cat worker.optrpt
...
LOOP BEGIN at worker.cc(12,3)
 remark: LOOP WAS VECTORIZED
 remark: masked strided loads: 4
 remark: type converts: 2
LOOP END
```

Advisor guides developers in resolving these issues.

§4. Advisor Basics

Intel® Parallel Studio XE

Software	Composer	Professional	Cluster
Intel C, C++ and Fortran compilers	x	x	x
Intel Math Kernel Library (MKL)	x	x	x
Intel Threading Building Blocks (TBB)	x	x	x
Intel Performance Primitives (IPP)	x	x	x
Intel VTune Amplifier XE		x	x
Intel Inspector XE		x	x
Intel Advisor XE		**x**	**x**
Intel MPI Library			x
Intel Trace Analyzer and Collector (ITAC)			x

Setting Up

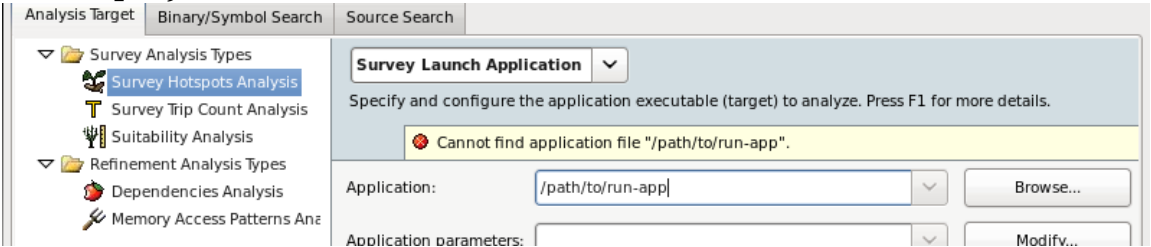
Starting Advisor GUI:

```
vega@lyra% source /opt/intel/advisor_xe/advixe-vars.sh
vega@lyra% advixe-gui
```

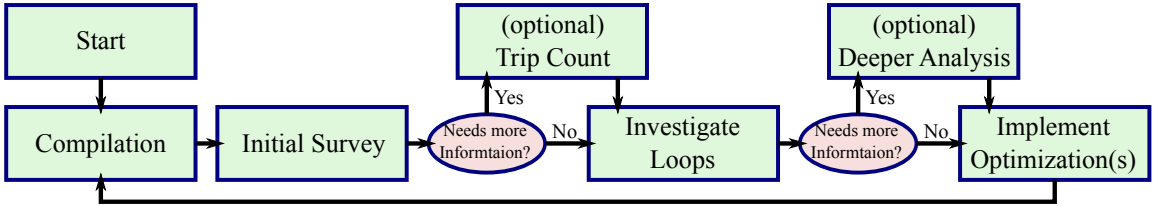
Compile your code:

```
vega@lyra% icpc -g -O3 -xCORE-AVX2 myapp.cc -o run-app
vega@lyra% #alteratively, use -xhost instead of -xCORE-AVX2
```

Create project:



Advisor Workflow



- **Compilation** – Compile the application with the appropriate flags
- **Initial Survey** – General overview analysis for finding hots-spots
- **Trip Count** – Additoinal information on loop/function call count
- **Investigate Loops** – View analysis details from the Advisor
- **Deeper Analysis** – Gather any addiotional information requested
- **Implement Optimization** – One optimization per iteration

CLI

For running on remote systems without GUI, use the `advixe-cl` command to run from command line.

```
vega@lyra% # run Survey Analysis
vega@lyra% advixe-cl -c survey ./app-CPU
vega@lyra% # find LoopID to mark
vega@lyra% advixe-cl -report survey -project-dir [proj]
vega@lyra% # run Deeper Analysis with marked loops
vega@lyra% advixe-cl -c map -mark-up-list=[LoopID],[LoopID]
```

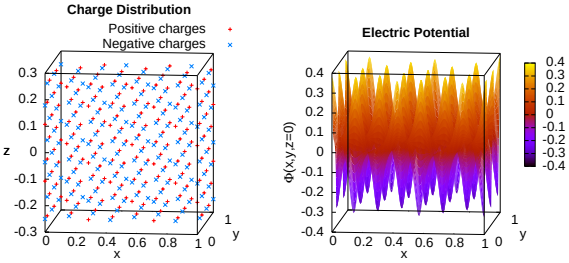
Advisor documentaion page on CLI

§5. Example Optimization

Example: Coulomb's Law Application

$$\Phi\left(\vec{R}_j\right) = -\sum_{i=1}^{m} \frac{q_i}{\left|\vec{r}_i - \vec{R}_j\right|}, \tag{1}$$

$$\left|\vec{r}_i - \vec{R}\right| = \sqrt{\left(r_{i,x} - R_x\right)^2 + \left(r_{i,y} - R_y\right)^2 + \left(r_{i,z} - R_z\right)^2}. \tag{2}$$



Paper: http://xeonphi.com/papers/autovec

Initial Implementation

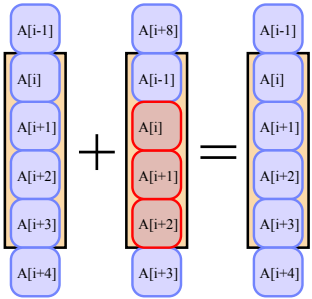
```
struct Charge {
  float x, y, z, q;
} chgs[m]; // Coordinates and value of this charge
```

```
for (int i=0; i<m; i++)  { // Coulomb's law
  // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
  const float dx=chg[i].x - Rx;
  const float dy=chg[i].y - Ry;
  const float dz=chg[i].z - Rz;
  phi -= 8.98755178737e9 * chg[i].q / sqrt(dx*dx+dy*dy+dz*dz);
}
```


Vector Dependence

True Vector Dependence

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



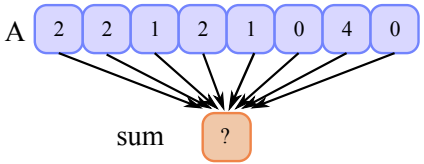
Vector Dependence:
A memory location that is written to is also read from in the same vector operation.

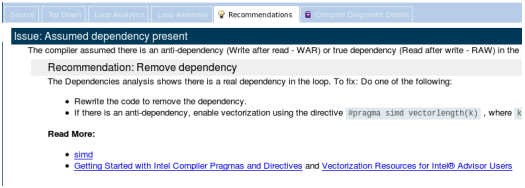
`A[i]` is written into in the first SIMD lane
but it is also read from in the second SIMD lane.
→ Can not be done in data parallel manner

Unsafe to Vectorize: Compiler will not vectorize this

Reduction: Form of Vector Dependence

```
1 for(int i = 0; i < n; i++)
2   sum = A[i];
```





One way to resolve it is to use the OpenMP SIMD reduction.

```
1 #pragma omp simd reduction({op}:{list})
```

- `op` – supported reduction operator (e.g. +, *)
- `list` – reduced scalar variables (e.g. phi, a)

Type Conversion

Consistency of Precision: Constants

```
// Bad: 2 is "int"
long b=a*2;

// Bad: overflow
long n=100000*100000;

// Bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q=2*p;

// Bad: 1e9 is "double"
float r=1e9*p;

// Bad: 1 is "int"
double t=s+1;
```

```
// Good: 2L is "long"
long b=a*2L;

// Good: correct
long n=100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2.0f is "float"
float q=2.0f*p;

// Good: 1e9f is "float"
float r=1e9f*p;

// Good: 1.0 is "double"
double t=s+1.0;
```

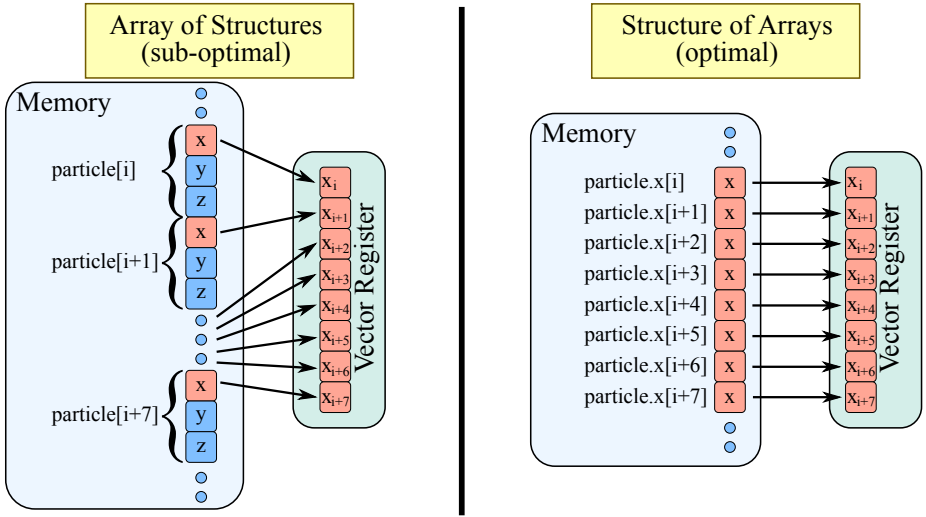
Consistency of Precision: Functions

```
// Bad: 3.14 is a double
float x = 3.14;

// Bad: sin() is a
// double precision function
float s = sin(x)

// Bad: round() takes double
// and returns double
long v = round(x);

// Bad: abs() is not from IML
// it takes int and returns int
int v = abs(x);
```

```
// Good: 3.14f is a float
float x = 3.14f;

// Good: sin() is a
// single precision function
float s = sinf(x)

// Good: lroundf() takes float
// and returns long
long v = lroundf(x);

// Good: fabsf() is from IML
// It takes and returns a float
float v = fabsf(x);
```


Data Structure

Arrays of Structures versus Structures of Arrays



Optimized Solution: Structure of Arrays, Unit-Stride Access

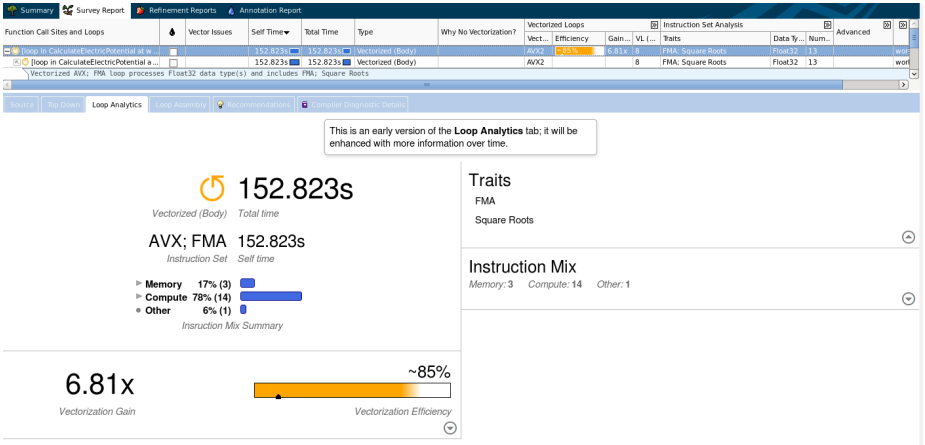
```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
};
```

```
// This version vectorizes better thanks to unit-stride data access
float phi_p = 0.0f;
#pragma omp simd reduction(+: phi_p)
for (int i=0; i<m; i++)  {
  // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
  const float dx=chg.x[i] - Rx;
  const float dy=chg.y[i] - Ry;
  const float dz=chg.z[i] - Rz;
  phi_p -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
}
phi = phi_p;
```

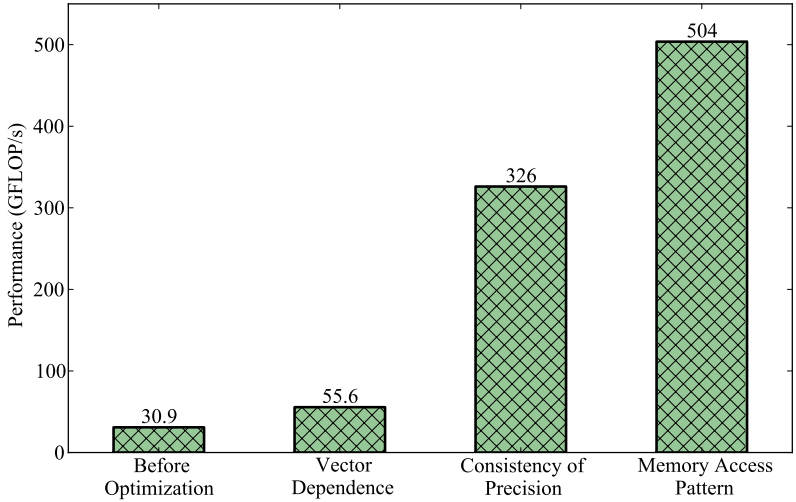

§6. Closing Words

Are We Done?

Vector Efficiency: one metric for determining if you are done.



Final Performance Data



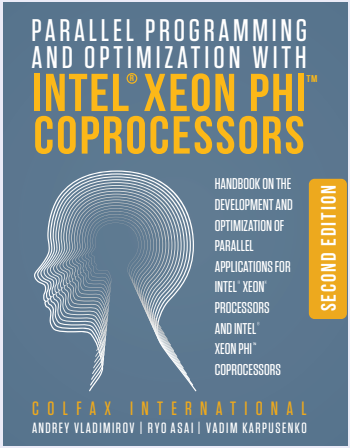
§7. Resources

Supplementary Materials: Textbook

ISBN: 978-0-9885234-0-1 (2nd edition, 508 pages, Electronic or Print)

Parallel Programming
and Optimization with
Intel® Xeon Phi™
Coprocessors

Handbook on the Development and
Optimization of Parallel Applications
for Intel® Xeon® Processors
and Intel® Xeon Phi™ Coprocessors



http://xeonphi.com/book

Colfax Research

http://colfaxresearch.com/

Learn More

Interested? Sign-up at:

colfaxresearch.com/how-series

Slides, Code, Video

You can download slides, code and watch the video recording of this webinar here (requires registration for a free Colfax Research account):

colfaxresearch.com/how-tools-16-05

