Best practices for using Intel® Cilk™ Plus

Contents

Introduction:

Performance tuning of an existing application is truly a challenge and it depends on a lot of factors like the nature of algorithm the application works on, if the implementation is scalable to take advantage of thread/data parallelism etc. The most logical approach any developer would follow for tuning the performance of an application is to do a dynamic profiling of the application under different workloads, try to analyze the hotspots in that application, and then fine tune them to work best on a given hardware architecture. These hotspots could either be a function or loop which handles high computation load. Intel provides a dynamic profiling tool named Intel® Vtune Amplifier XE which is used for profiling any given application. Once the hotspots are identified, then the next approach is to analyze the corresponding algorithm and look for potential unexploited thread/data parallelism. Also it is a good programming practice to write the code scalable so that it makes use of all the available cores (thread parallelism) and SIMD (Single Instruction Multiple Data) registers in each core (data parallelism). This paper recommends the step by step approach to enable an application with both task parallelism and data parallelism using Intel® Cilk™ Plus. Also the usage of every explicit vectorization extension is explained in detail with examples which clearly gives a good understanding on how and when to use them.

Pre-requisites:

1. You have a serial working C/C++ application.
2. You are familiar with the following Intel® Cilk™ Plus tools:
    - Simd pragma
    - Array Notation
    - Elemental functions
    - Cilk keywords
3. You are familiar with Intel® VTune™ Amplifier XE and how to use it for hotspot analysis of your C/C++ application.

Code Samples attached:

1. Lower and Upper triangle assignments using Intel® Cilk™ Plus Implicit Index.
2. Manhattan Distance computation using Intel® Cilk™ Plus Implicit Index.
3. SIMD pragma reduction example

Intel® Cilk™ Plus:

Intel® Cilk™ Plus technology provides both threading solutions (using cilk_for, cilk_spawn and cilk_sync) and SIMD solution (using #pragma simd, Array Notation and Elemental functions). Cilk programming originated from MIT* Laboratory for Computer Science. In 2006, Cilk Arts licensed the Cilk technology from MIT with a goal of developing a commercial C++ implementation called Cilk++. In 2009, Intel acquired Cilk Arts and Cilk technology (task parallelism) was merged with Array Notation (vector parallelism) to provide a comprehensive language extension. Intel® Cilk™ Plus technology will be available as a part of gcc* 4.9 mainline and the work is currently in progress to make it available as part of LLVM, too. Pragma simd and SIMD functions concept have made their way into the OpenMP* 4.0 specification. Array Notation for C is already merged to the gcc mainline.

Strategy

1. Use Intel® VTune™ Amplifier XE to analyze the hotspots of your application.
2. Analyze the hotspot for potential thread/data parallelism.
3. Improve the single core utilization by enabling vectorization to make use of SIMD registers potential.
4. Improve the multi-core utilization by enabling threading to make use of multiple cores potential.
5. After every code change in step 3 and 4, the hotspot analysis is done again and made sure that the hotspots are taking lesser time since the optimizations will make the kernel work faster. It is an iterative process.

Improve single core utilization by enabling vectorization:

1. The first step to performance tuning as discussed above in the Introduction section is to identify hotspots and analyze the hotspot for potential parallelism.
2. Often the hotspots are loops with high computational load which include function calls within the loop body. The best approach to performance tuning a hotspot is by utilizing the threading/SIMD solution. If the algorithm is data parallel, the recommended approach to tune the performance of the loop is by ensuring the best usage of SIMD registers in one core before looking for opportunities to utilize all the available cores.

   Intel® Cilk™ Plus offers three different ways to enable SIMD in any application. The first recommended approach is to try enabling #pragma simd with appropriate clauses on the loops which are hotspots. When the Intel Compiler encounters #pragma simd, it will vectorize the loop as programmers asserted. It is a powerful tool for explicit vector programming which bypasses some of the compiler analysis and heuristics. Developers are expected to know what is

happening in the loop body and make a decision to use #pragma simd only if the loop is vectorizable in their code.

If the loop body includes some function calls and those functions do not have any vector version defined, then that will be a hindrance to vectorizing the loop. In those cases, it will be helpful if we can have a vector version of the same function.

3. Vector version of a user defined function can be created by declaring that function as an Intel® Cilk™ Plus Elemental function. When a function is declared as an Elemental function (by annotating the function with __declspec(vector) on Windows* and __attribute__((vector)) on Linux*/Mac OS X* with appropriate clauses), the Intel® Compiler generates both scalar and vector version of the same function. Vector version of the function accepts vector arguments and returns vector data unlike the traditional scalar functions that accept scalar arguments and return scalar data. Since the vector version of the function works with vector data rather than scalar data, vectorization is enabled. So if the functions called within the loop are vectorized, then the loop as a whole will be vectorized, too.

4. The last tool for explicit vectorization is Intel® Cilk™ Plus Array Notation. Array Notation is a language extension by which we can select vector operands from arrays (contiguous memory locations or linear non-unit stride memory locations) and use them in expressions. When the serial implementation of the code is such that the data parallelism is not evident for the compiler, we can rewrite the logic in a manner that takes vector operands as opposed to scalar operands by using Array Notation. Array notation will be the last resort as it demands good amount of code change by explicitly specifying the logic using vectors and thus forcing vectorization.

Improve multi-core utilization by enabling Threading:

Intel® Cilk™ Plus provides three different keywords as threading solution and they are:

1. cilk_for
2. cilk_spawn
3. cilk_sync

Vectorization enabled for loops makes sure to extract the best potential of the SIMD registers in a given single core. In order to take advantage of multiple cores on the machine, “for” in the for loop needs to be replaced with “cilk_for”. cilk_for will breakdown the overall iteration space of the for loop into independent tasks and Cilk runtime uses work stealing scheduler to schedule the ready task for execution across the execution units. Work Stealing means whenever a processor runs out of tasks, it tries to steal the task from other processors. The caveat behind using cilk_for is that, the developer needs to make sure that the computation load of the for loop is high enough and the iteration space is

large enough so that the spawning overhead does not play a significant role in the overall execution time of the program. Generally the spawning overhead is 3 -5 times the overhead of a function call. Body of the “for loop” marked with “cilk_for” will be executed by multiple cilk worker threads each working on mutually exclusive section of the overall iteration space. So there is an implicit sync point for cilk worker threads at the end of for loop marked with cilk_for keyword. Developers can either rely on the compiler heuristics for the work load division across multiple worker threads or specify the grain size to the cilk runtime using pragma as follows:

```
#pragma cilk grainsize = (expression)
cilk_for(){..}
```

Apart from the cilk_for keyword, developers have another handy keyword which is cilk_spawn. Unlike the cilk_for, cilk_spawn hints the cilk runtime that this section of code can run in parallel. So if there is a Cilk worker thread available, then the work stealing happens and the two different threads are launched for execution from that point. One is the original worker thread which continues executing the function called using cilk_spawn keyword and secondly the stealing worker thread takes up the executing the code section after the cilk_spawn until cilk_sync. Also unlike implicit sync point for the worker threads in the case of cilk_for, explicit syncing using cilk_sync keyword is expected when spawning a cilk worker thread using cilk_spawn keyword. Cilk_spawn keyword is used when a functional parallelism is seen in an application where the parallelism is not in for loop. An important point to be remembered is that by default Cilk runtime creates as many number of cilk worker threads as the number of processors. Also the cilk keywords just hint the runtime on the potential parallelism. Whether the worker threads spawn and start stealing the work depends on the how many worker threads are available at that moment. So executing the same program on a single core machine will be same as the serial equivalent of the program.

Following the above two sections of recommended steps, an application can be made to utilize all the cores on a machine and also make sure every core is utilized to its maximum potential.

Best Practices of using Intel® Cilk™ Plus for performance with accuracy intact:

Usage of $pragma simd vectorlength clause:

Below is a simple code snippet which will be used to demonstrate usage and significance of pragma simd.

```
for(i = 4; i < n; i++)
{
        y[i] = y[i-2] + 1;
}
```

In the above for loop, there is a dependence in the operation performed (value at i^{th} location depends on value at $(i-2)^{th}$ location). This is one of those scenarios where auto-vectorizer of Intel® C++ Compiler generates a vectorization report stating the following:

```
remark: loop was not vectorized: existence of vector dependence.
```

When #pragma simd is used in the above loop as follows:

```
#pragma simd
for(int i = 4; i < n; i++)
{
    y[i] = y[i-2] + 1;
}
```

Intel® C++ Compiler goes ahead and vectorizes the loop generating the following vectorization report:

```
remark: LOOP WAS VECTORIZED.
```

Although the the loop is vectorized, the executable seems to produce wrong results. The reason why wrong results are generated is explained below:

By default when #pragma simd is used without any clauses, the compiler selects a vector length corresponding to the targeted architecture (SSE2, AVX etc). In this case we assume that the targeted architecture is SSE2 (default architecture targeted by Intel® C++ Compiler) and y is an array of floats. Hence in the process of generating vector code, a vector of 4 floats is considered instead of considering individual floats from the array as in the previous case (scalar mode). Obviously the next logical question is why 4 floats? Each float data consumes 4 bytes of memory and hence 4 floats will consume 32 bytes which is 128 bits (size of a SIMD register in SSE2 architecture). Considering that left operand and right operand of the expression will be of vector length 4, there is an evident overlap of 2 float locations in that expression. Overlap suggests that 2 input locations are read and used for computation before the latest values for those locations are updated and hence this leads to data corruption and eventually wrong results. So the root cause of the problem is the 2 overlapping memory location which is because the vector length is 4 floats.

In order to make sure there is no overlap of memory locations at any point between the vector version of the left operand and vector version of the right operand, the vector length needs to be changed from 4 floats to 2 floats. The way to express the above to the compiler is by using the vectorlength clause with #pragma simd as shown below:

```
#pragma simd vectorlength(2)
for(int i = 4; i < n; i++)
{
    y[i] = y[i-2] + 1;
}
```

In the above case, still vector code is generated but this time vector operands have 2 floats instead of 4 floats. By this approach, the performance is still better than scalar version of the loop by close to 2x and

results are also accurate. Thus it serves the purpose although the code is not fully utilizing the vector register bandwidth. That is simply because of the limitation posed by the nature of the algorithm.

The above example is a simple demonstration of a loop that does not vectorize by default, vectorizes but produces wrong results when using #pragma simd without any clauses and vectorizes/produces right results when using #pragma simd with the vectorlength clause. This example should give an idea about the significance of clauses that come along with #pragma simd. The clauses make #pragma simd more expressive by letting the compiler know that extra bit of information that it needs in order to generate the vector code that would generate the same results as the scalar version.

Usage of $pragma simd reduction and private clause:

Consider a for loop (as shown below) which happens to call a function defined outside the compilation unit where the loop resides. This is quite common scenario in practical applications.

```
#include<iostream>
#include"func.h"
using namespace std;
int main(int argc, char *argv[]){
float a[20], b[20], sum = 0.0f;
a[:] = __sec_implicit_index(0);
b[:] = 19 - __sec_implicit_index(0);
//#pragma simd //reduction(+:sum)
for(int i = 0; i < 20; i++)
     sum = sum + func(a[i], b[i]);
cout<<sum<<"\n";
return 0;
}
```

The above for loop invokes function func() which happens to be declared in func.h as shown below

```
#include<math.h>
float func(float, float);
```

and defined in a different .cpp file:

```
#include"func.h"
float func(float a, float b)
{
    a = a + b;
    b = a - b;
    a = a - b;
    return (a+b);
}
```

In such cases vectorization report generated while building the code is as follows:

```
testelem.cpp(9): message : loop was not vectorized: nonstandard loop is not a vectorization
candidate
```

It is for obvious reason that the function func() is not a vector function rather it accepts only scalar arguments. In order to create a vector version of func(), annotate the function func() with __declspec(vector) as shown below in func.h where the function is declared:

```
#include<math.h>
__declspec(vector) float func(float, float);
```

Annotating a function with __declspec(vector) will convert that to a SIMD function. Thus Intel® C++ Compiler will generate both scalar and vector version of func(). Now rebuilding the application and generating the vectorization report does not state that the loop is not a vectorization candidate since the vector version of the function is available. But now compiler thinks there is vector dependence and generates the following vectorization report:

```
testelem.cpp(9): message : loop was not vectorized: existence of vector dependence
```

On further investigation all the function does is swap the values of its arguments and return the sum of them. This operation should not have dependence across iterations and they can execute in any order. Now that vector dependence is not an issue, #pragma simd can be used to force the compiler to vectorize the loop. The #pragma simd is introduced as shown below:

```
#pragma simd //reduction(+:sum)
for(int i = 0; i < 20; i++)
     sum = sum + func(a[i], b[i]);
```

And Intel C++ Compiler vectorizes the loop generating the following vectorization report:

```
testelem.cpp(9): message : SIMD LOOP WAS VECTORIZED
```

But when the executable is run, it generates wrong result (result doesn't match with the result generated by non-vectorized version). So in the process of vectorizing the loop using #pragma simd without any clauses, the compiler generated code which produced wrong results. Every time #pragma simd is used, it is advised to specify the kind of operation happening inside the loop explicitly so that compiler when generating the code has all the required information to generate correct vectorized code. In the above loop, the sum variable is a reduction variable and the reduction operation happens to be addition. This information needs to explicitly specified to make sure the right code is generated. The way to specify this information is by using the reduction clause of #pragma simd as shown below:

```
#pragma simd reduction(+:sum)
for(int i = 0; i < 20; i++)
     sum = sum + func(a[i], b[i]);
```

This time the compiler vectorizes the code and produces the right result. Also since variable "i" is private for each iteration, the private clause can be used to declare the variable i as private. In order to do that we need to move the declaration of i outside the for loop as shown below:

```
int i;
#pragma simd reduction(+:sum) private(i)
for(i = 0; i < 20; i++)
     sum = sum + func(a[i], b[i]);
```

It is highly recommended that clauses are rightly used along with #pragma simd to ensure a deterministic behavior across multiple versions of compiler. Clauses can be considered as a set of switches which are latched to enable some optimization and not specifying them will leave everything to the compiler heuristics which need not be the same in future. A code sample for the same is attached with this paper. Please open the attached Code Samples.zip and check the folder "Using #pragma simd reduction example".

Usage of Array Notation:

Intel® Cilk™ Plus Array Notation is a language extension used for selecting vector operands from arrays (either contiguous in memory or memory locations separated by linear non-unit stride). Intel® C++ Compiler assumes there is no aliasing of the vector operands when the expression is written using array notation. Since compiler is clear about the fact that there is no aliasing, it is a way to explicitly force the vectorization to happen. All the logic involving arrays can be rewritten using Array Notation. But while dealing with applications involving huge data set, it is not a good idea to use expression which operates on full domain of the array (especially at places where array data is used in multiple places down the computation). In such cases, if the code is written to operate on full domain of the array in each statement, there is always a high probability that the same data section is required in the next statement but the data is evicted from cache when required. So cache miss rate will increase. The Best workaround is to rewrite the algorithm to do the computation on a shorter section of array (short vector) rather than operating on the full domain of the array (long vector). Thus operating on a short vector will make sure that the data is fully used in computations across statements before getting evicted from the cache bringing the cache miss rate down. A simple illustration of this concept is shown below with a code snippet. For detailed article please refer to "Array Notation Tradeoffs".

```
__declspec(noinline) void longvector(T *R,T *A, T *B, T k) {
  __assume_aligned(R,64);
  __assume_aligned(A,64);
  __assume_aligned(B,64);
  T tmp[S];
  tmp[0:S] = A[0:S] * k - B[0:S];
  if (tmp[0:S] > 5.0f) {
    tmp[0:S] = tmp[0:S] * sin(B[0:S]);
  } A[0:S] = tmp[0:S];
}
```

Above is a code snippet from the "Array Notation Tradeoffs" article. Here the longvector() function gets the pointers of 3 arrays each of size S. Depending on the value of S, the performance of the program changes. If S happens to be larger than L2 cache, then the program may not perform in the optimal way. Every time a computation is specified in terms of long vectors, the compiler has to decide on how to

split the data set depending on the targeted architecture so that the vector data fits in the SIMD registers. Instead the developer can make it easier for the compiler to split the data such that it fits on the target architecture's SIMD registers. The data splitting is done using the short vector Array Notation approach as demonstrated below:

```
__declspec(noinline) void shortvector(T *R,T *A, T *B, T k) {
  __assume_aligned(R,64);
  __assume_aligned(A,64);
  __assume_aligned(B,64);
  for (int i = 0; i < S; i += VLEN) {
    T tmp[VLEN];
    tmp[:] = A[i:VLEN] * k - B[i:VLEN];
    if (tmp[:] > 5.0f) {
      tmp[:] = tmp[:] * sin(B[i:VLEN]);
    } A[i:VLEN] = tmp[:];
  }
```

The above code snippet shows the short vector implementation of the same logic. The extra work is about introducing a for loop which increments by VLEN for every iteration, where VLEN is the vector length (number of scalars that will fit in the SIMD register of the targeted architecture). For instance if the arrays were of type integer and the targeted architecture is SSE2 (128 bit registers), it can fit in vectors of 4 integers in one SIMD register and thus VLEN will be 4.

Short vector Array Notation makes sure that cache blocking happens.

Developers have to make sure the vector operands selected using Array Notations ought to have the same rank. If the ranks aren't the same then the compiler will report the following error:

```
catastrophic error: section length mismatch in array expression
```

So what a developer should not do is as follows:

```
int a[10], b[20], c[10];
a[:] = b[:] + c[:];
```

In the above code, two vectors b[:] and c[:] are used but they are not of same rank since b[:] has a length of 20 while c[:] has a length of 10.

Usage of Implicit index with Array Notation:

The most commonly used initialization loop in many applications is shown below:

```
for(int i = 0; i < 20; i++)
    b[i] = i;
```

Array Notations provides a less verbose way of implementing the same using implicit index feature as shown below:

```
b[:] = __sec_implicit_index(0);
```

The function __sec_implicit_index() creates array indexes and stores them in the destination array. The function takes one argument which has two possible values:

1. 0 – This will create the indexes along the row.
2. 1 – This will create the indexes along the column.

Other use cases of Implicit indexes includes assigning values to the lower or upper triangle of a matrix, generating the Manhattan distance on a 2D grid given the starting point.

Lower and Upper Triangle assignment: The following code snippet shows how implicit index can be used to assign values in the lower and upper triangle of a given matrix.

```
int main(int argc, char *argv[]){
    int grid[10][10];
    grid[:][:] = 0;
    cout << "Lower Triangle\n";
   grid[0:10][0:__sec_implicit_index(0)] = 1;
   for(int i = 0; i < 10; i++) {
      for(int j = 0; j < 10; j++)
         cout<<grid[i][j]<<"\t";
      cout<<"\n";
   }
   cout<<"\n";
   cout << "Upper Triangle\n";
   grid[:][:] = 0;
   cout<<"\n";
   grid[9:10:-1][9:__sec_implicit_index(0)+1:-1] = 1;
   for(int i = 0; i < 10; i++) {
      for(int j = 0; j < 10; j++)
         cout<<grid[i][j]<<"\t";
      cout<<"\n";
   }
   cout<<"\n";
return 0;
}
```

Consider the Lower triangle construction code shown above. The statement

grid[0:10][0:__sec_implicit)index(0)] = 1;

can be interpreted as:

Row 1: grid[0][0]
Row 2: grid[1][0], grid[1][1]
.
.

Row 10: grid[9][0], grid[9][1], grid[9][2], grid[9][3], grid[9][4], grid[9][5], grid[9][6], grid[9][7], grid[9][8], grid[9][9]

It is clear from the above demonstration that __sec_implicit_index(0) generates array indexes starting from 0 for every row. A code sample for the same is attached with this paper. Please open the attached Code Samples.zip and check the folder “Lower and upper triangle using implicit index”.

Manhattan distance: This is commonly used distance metrics in grid algorithms and a simple way to implement the same using implicit index is as follows:

```
#include<stdlib.h>
using namespace std;
int main(int argc, char *argv[]){
    int grid[10][10], initialX, initialY;
    if(argc != 3)
    {
        cout<<"Wrong number of arguments; Please type <executable> <initial x location> <initial y location> \n";
        cout<<"initial x location and initial y location should be between 0 and 9 since the grid size here is 10 \n";
        return 0;
    }
    grid[:][:] = 0;
    initialX = atoi(argv[1]);
    initialY = atoi(argv[2]);
    //Constructing the bottom right part of the grid
    grid[initialX:(10 - initialX)][initialY:(10 - initialY)] = __sec_implicit_index(0) + __sec_implicit_index(1);
    //Constructing the bottom left part of the grid
    grid[initialX:(10 - initialX)][initialY:initialY+1:-1] = __sec_implicit_index(0) + __sec_implicit_index(1);
    //Constructing the top left part of the grid
    grid[initialX:initialX+1:-1][initialY:initialY+1:-1] = __sec_implicit_index(0) + __sec_implicit_index(1);
    //Constructing the top right part of the grid
    grid[initialX:initialX+1:-1][initialY:(10 - initialY)] = __sec_implicit_index(0) + __sec_implicit_index(1);
    for(int i = 0; i < 10; i++)
    {
        for(int j = 0; j < 10; j++)
            cout<<grid[i][j]<<"\t";
        cout<<"\n";
    }
    return 0;
}
```

A code sample for the same is attached with this paper. Please open the attached Code Samples.zip and check the folder “Manhattan Distance using implicit index example”.

Support of Vectorization for Array Iterator based loops:

Intel® C++ Compiler supports vectorization of loops based on array iterators starting from version 14.0. Since programming using iterators is considered as an industry standard coding practice, this feature helps developers to stick to coding standards and at the same time get vectorized code. Below is a code snippet which demonstrates the concept:

```
#include<iostream>
#include<array>
using namespace std;
int main(){
std::tr1::array<float,20> y;
auto y_iter = y.begin();
float *y_ptr = &y_iter[0];
y_ptr[0:20] = __sec_implicit_index(0);
#pragma simd vectorlength(2)
for(int i = 2; i < 20; i++)
    y_iter[i] = y_iter[i-2] + 1;
for(int i = 0; i < 20; i++)
    cout<<y_iter[i]<<"\n";
return 0;
}
```

The code snippet demonstrates some important concepts:

1. Given an array iterator, the equivalent pointer can be fetched by using the following statement:

```
auto y_iter = y.begin();
float *y_ptr = &y_iter[0];
```

2. Array Notation can be applied only on arrays by using the pointer and not using the iterators. So the above conversion comes in handy to use Array Notations on arrays. Also here since the pointer is used, the section of the array to be selected needs to be explicitly specified even if the full domain of the array needs to be selected.

3. The above for loop without the #pragma simd does not vectorize because of the vector dependence. When #pragma simd is used without clauses, this leads to data corruption because the default vector length assumed will be 4 which will lead to overlapping input and output vectors in the same statement (Very similar to the case discussed in the section “Using pragma simd with vectorlength clause”). Introducing the pragma simd with vectorlength(2) clause vectorizes the loop involving array iterators and the vectorization report generated will be as shown below:

```
itervec.cpp(11): message : SIMD LOOP WAS VECTORIZED
```

Usage of SIMD function with uniform clause:

Traditional C/C++ functions accept scalar arguments and return scalar data. This convention hinders the vectorization of loops whose body has function calls. The vectorization will go fine if the functions called from the loop body can handle vector arguments and return vector data. This is possible when a regular function made an SIMD function by annotating the function with __declspec(vector(clauses)) on Windows* and __attribute__((vector(clauses))) on Linux*/ Mac OS X*.

When Intel® C++ Compiler encounters the annotation __declspec(vector), it creates both scalar and vector version of the same function. But like the usage of #pragma simd, SIMD function comes with a bunch of clauses among which uniform and linear will be demonstrated with an example.

Consider a function declaration:

```
__declspec(vector) void foo(int *a, int i);
```

This declaration will lead to a scalar and vector version foo(). The vector version generated by default (without any clauses) will convert the scalar arguments/return types to its equivalent vector versions. So in the above case it will pass an integer pointer array “a” and an integer array “i”. But that is not the behavior a developer expects from the vector version of the function. Rather the developer expects the integer pointer “a” to be broadcasted across all iterations thus maintaining a single copy and variable “i” will be used as a iteration count measure which increments by 1 for every iteration.

Just mentioning the __declspec(vector) does not communicate these above mentioned conditions while generating the vector version of the function. Thus clauses play an important role in explicitly defining the specifics which the compiler must consider while generating the vector version of the function. The recommended way to convey the above information to the compiler is as shown below:

```
__declspec(vector(uniform(a), linear(i:1))) void foo(int *a, int i);
```

The uniform(a) clause specifies that the argument “a” to the function is to considered as a single pointer (value of “a” is shared across all iterations) rather than as an array of integer pointers and linear(i:1) specifies the argument “i” as a variable which increments by 1 for every iteration. Above is the declaration of the function being invoked. The call site for the same is as shown below:

```
    for(int i = 0; i < n; i++)
        foo(a, i);
```

The call site for function foo() and the function definition site can be in two separate compilation units. Depending on the availability of the vector version foo(), the compiler either calls the vector version or scalar version of the function. In some cases, you might have function calls inside a condition. In those cases, the compiler calls the appropriate masked vector version of the function which is automatically created during the compilation phase. A masked version of the function can be generated using the following declaration.

```
__declspec(vector(uniform(a), linear(i), mask)) void foo(int *a, int i);
```

Conclusion:

Intel® Cilk™ Plus package is a one stop solution for both task parallelism and data parallelism. The three cilk keywords (cilk_for, cilk_spawn and cilk_sync) can be used to create multiple cilk worker threads

which will work on mutually exclusive workloads. Apart from the threading solution, the package comes with 3 language extensions for explicit vectorization which are Pragma SIMD, Elemental functions and Array Notations.

References:

1. http://cilkplus.org – Official website for Intel® Cilk™ Plus
2. Article on Array Notation Tradeoffs - http://software.intel.com/en-us/articles/array-notation-tradeoffs
3. Documentations on Intel® C++ Compiler - http://software.intel.com/en-us/articles/intel-c-composer-xe-2011-documentation
4. Webinar on Introduction to Vectorization using Intel® Cilk™ Plus Extensions - http://software.intel.com/en-us/intro-to-vectorization-using-intel-cilk-plus
5. A guide to Auto-vectorization using Intel® C++ Compiler - http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers