Vector Programming: The no longer MIA part of parallel programming

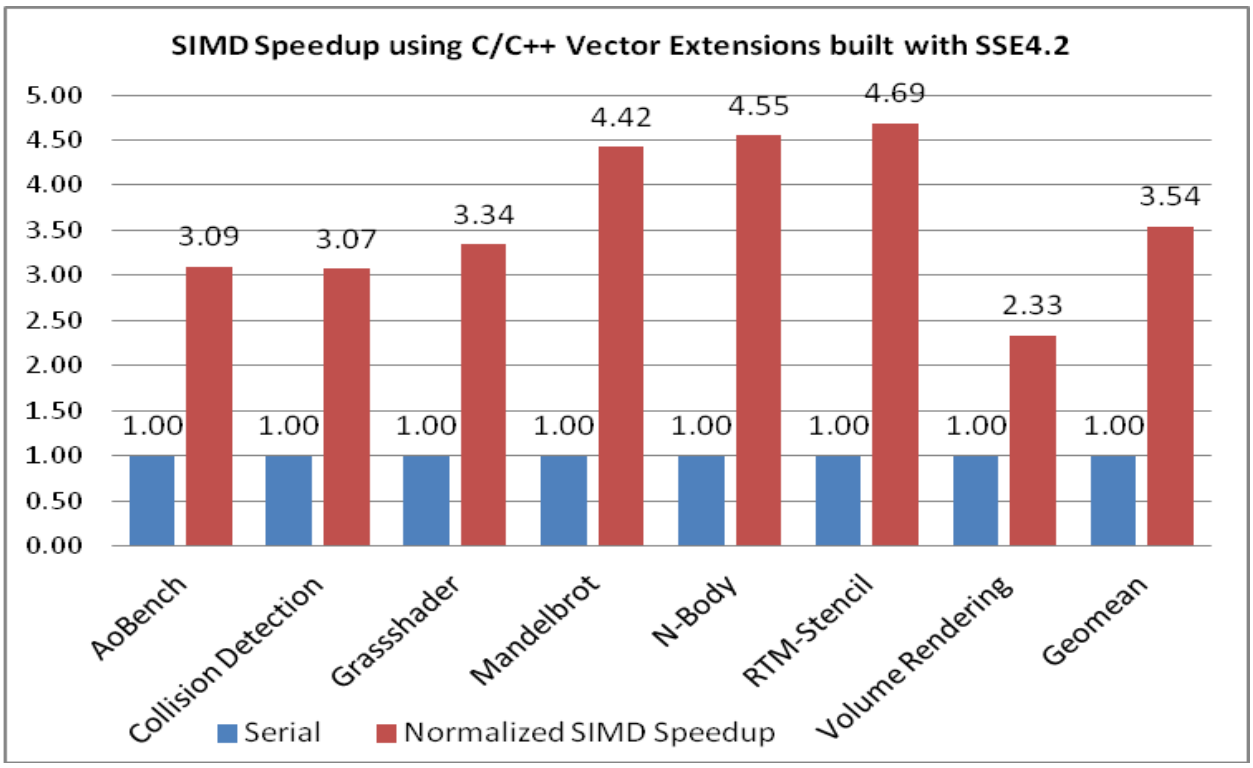
Author: Robert Geva, robert.geva@intel.com

Introduction

Many financial applications have high performance computing characteristics. The well-known examples include Trinomial trees, Monte Carlo simulation, LIBOR market model, etc. These applications can benefit from parallelization. Many of those are currently deployed using multi-processing. Multi-processing has many advantages. An advantage over multi-threading is that it is a safer model, and is less error prone, e.g. with respect to data races. A potential disadvantage in multi-core deployment is that it tends to require more memory, as each process may require an input set. As the size of memory is not keeping up with the increase in core count, the limitation of multi-processing are becoming evident. Multi-processing, multi-threading and the tradeoff between them are relatively well understood. However, CPUs provides parallel resources at multiple levels. Threading can utilize the parallelism between cores, hyper threads and sockets. In addition, each core provides parallelism at the vector level. The AVX technology provides 256 bit wide operations and allows SIMD parallelism with 8 single precision operations in parallel. Taking advantage of vector level parallelism is a part of parallel computing, and in general, equally important. This brief introduction presents recent developments in vector programming, in an attempt to bring vector programming support to the same level as parallel programming, where the algorithms is expressed explicitly, rather than relying on compiler optimizations which may or may not occur, and at the C/C++ level, rather than at low level intrinsics.

Vector Programming

In order to achieve high performance, applications have to utilize the parallel resources in the HW. Many practitioners associate parallel programming with utilization of multiple cores. The need to also utilize the HW vector units is a less well understood part of parallel programming. However, data is showing that for data parallel problem, the use of vector HW and vector programming adds a multiplicative performance improvement of 3-4X both to serial code and to parallel code, even for high levels of parallelism. The chart below shows performance speed up of a few workloads when vectorized compared to their scalar equivalent. The data was measured by Xinmin Tian and published at IPDPS 2012.



The C and C++ languages do not provide Standard programming languages good support for vector programming. Recently, based upon Intel's work, the OpenMP® 4.0 standard supports vector programing in the form of #pragma omp simd. Here we will use Intel's #pragma simd based syntax which predates the one supported by OpenMP® 4.0, however, the two sets of capabilities are isomorphic and equivalent. Prior to the introduction of these pragma, which provide explicit vector programming at the C/C++ level, there were two common ways in which programmers utilized the vector units:

1. Use intrinsics. These are very low level mechanisms, almost as low level as assembly language. The code is not portable, and needs to be rewritten as the HW ISA extends and add new instructions, such as move from SSE2 to SSE4.
2. Write scalar code that does not express vector semantics, and rely on auto vectorization to generate vector code by way of optimization. Data is showing that this approach is very limited in capability and fails to vectorize code which can be vectorized. Analysis shows that the root cause is the semantic gap and specifics of C/C++ language rules which prevents the compiler from proving legality of vectorization. Moreover, auto vectorization is limited to simple inner loops. Language based vector programming applies more generaly, including outer loops, loops with function calls, loop with complex control flow, etc.

A vector loop

The syntax of #pragma simd allows placement of the pragma ahead of a for (;;) loop. As is the case with the OpenMP pragma for parallel programming, the pragma changes the semantics of the loop from sequential execution to vector execution. The new semantics allow (and do not force) an order of evaluation that allows expression from iteration n+1 of the loop to execute ahead of expressions from iteration n.

We use the relation of "sequenced before" used in the C++11 specification to specify the semantics of vector loops. The order of evaluation of expressions inside vector loops is defined as:

0. If X is sequenced before Y in the body of a vector loop, for each iteration i, then X_i is sequenced before Y_i.
1. For every expression X and Y evaluated as part of a vector loop, if X is sequenced before Y and i < j then X_i is sequenced before Y_j

In addition, if the programmer uses the additional, optional syntax and limit the maximal allowed chunk size, then:

2. If X and Y are expressions in the loop and X is sequenced before Y in the loop body, then Y_i is sequenced before X_{i+c}.

The language rules allow the placement of #pragma simd only ahead of countable for loop. A loop is countable if the trip count can be determined in run time ahead of the execution of the loop. Specifically, there needs to be a comparison expression and an increment expression (unlike the general case where these are allowed to be empty). The increment expression needs to increment a variable (the loop control variable, LCV) which is used in the compare expression. The other expression in the comparison is the loop limit and it should not change within the loop body. The LCV should not be modified elsewhere in the loop body. There should not be side exits out of the loop, such as break, return or throwing an exception.

The pragma allows clauses. A list of capabilities is below:

Capability	**Syntax**	**Meaning**
Vector loop	#pragma simd for (; ;) {}	Vector order of evaluation; potential remainder loop.

Induction variable	#pragma simd
	linear(x:s1,y:s2,z)
	for (; ;)
		Lane n+1 has the value of s + value at lane n. Increment has to execute in all iterations, the increment shall be loop invariant.
		If a step is not specified the default step is a unit stride.

reduction	#pragma simd
	reduction(+:sum)
	sum += a[i];
		Designate a variable as a target of reduction operation. The reduction is computed using partial sums and the value is undefined during the loop.

Uniform variables	Declare the object outside the loop	Same object (and value) in all vector lanes.
Private variables	Declare the object inside the loop	Each vector lane has a distinct implementation of the object

Limit the chunk size	#pragma simd
	vectorlength(N)
	for (; ;)
		Vector order allowed only across K consecutive iterations and not more

Elemental functions	__declspec(vector [claues]) return_type fund_name (args) { body }	The function can be compiled independently, as if it is a part of the body of a vector loop.

A SIMD enabled function

One key construct needed for writing a vector loop is the ability to make function calls that will execute as if they were part of the body of the loop. The ability to designate function as “SIMD enabled functions” allows modular programming, including the ability to separate the work in the loop into functions and the ability to develop libraries. These, without losing performance across the interface. The language provides syntax to designate a function as an elemental function and qualify arguments as varying, uniform or linear. When compiling an elemental function, the compiler generates a vector version, in which every formal parameter is extended to a short vector of parameters, and the function returns a short vector of values instead of a single value. The code inside the function is then vectorized across a chunk of consecutive calls to the function. When a SIMD enabled function is called from a vector loop, a chunk of calls is replaced by a single call. The following chart shows the capabilities of SIMD enabled functions:

Capability	**Syntax**	**Meaning**
A SIMD enabled	_declspec(vector[clauses]) for Windows.	Vector order of evaluation across a chunk of consecutive calls to the function is

function	__attribute__(vector[clauses]) for Linux	allowed; potential remainder loop. Note: the function can still be invoked in a scalar context.
A Uniform argument	__declspec(vector(uniform(arg1))	The value of arg1 is the same across the ch function.
A Linear argument	__declspec(vector(linear(arg2:incr))	The chunk of values of arg2 are linear incre
Chunk size	__declspec(vector(vectorlength(N))	The number of consecutive invocations of t be grouped into a single invocation. Editori this may create linkage.
Multiple versions	__declspec(vector[clauses]) __declspec(vector[clauses])	Multiple versions of the function are generated, each corresponding to a different set of clauses.

Outer loops: Note that this syntax allows vectorization of outer loop, as is shown in the example below:

```
#pragma simd
for (i=0; i<n; i++) {

   complex<float> c = a[i];
   complex<float> z = c;
   int j = 0;
   while ((j < 255)  && (abs(z)< limit)) {
    z = z*z + c;
    j++;
   };
   color[i] = j;
 }
```

The meaning of vectorization of an outer loop is that every vector lane execute an independent instance of the inner loop.

Data in vector loops:

In general, a variable declared inside the loop is private to each iteration, and a variable declared outside of the loop is uniform, and is the same object across all iterations. An implication is that if different iterations within a chunk attempt to assign a value to a uniform variable, the behavior is undefined. The language provides

two important exceptions to this rule, for cases that involve variables declared outside of the loop and are modified within the iterations: inductions and reductions. The syntax to designate a variable as a linear induction variable is linear (var_name: stride) where stride is optional and defaults to 1. The syntax to designate a reduction is reduction (op : var). An induction variable must be incremented (or decremented) by a fixed amount in each loop iteration. The value can be used inside the loop. A reduction can modify the variable by a different amount in each iteration (and it is also possible for some of the iterations to not modify) and the value is undefined during the loop. It is only available after the loop. An illustration follows:

```
float sum = 0.0f;
float *p = a;
int step = 4;
#pragma simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
    sum += *p;
    p += step;
}
```

Simd enabled functions:

Below is a simple example of a simd enabled function:

```
__declspec (vector)
double option_price_call_black_scholes(
                  double S,       // spot (underlying) price
                  double K,       // strike (exercise) price,
                  double r,       // interest rate
                  double sigma,   // volatility
                  double time)    // time to maturity
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

The compiler generates at least two implementations for a SIMD enabled function. One is a scalar function that can be invoked in a scalar context, and behaves in the same way as if it didn't have the __decspec(vector) designation. The second is a vector variant. The vector variant received a short vector (in a vector register) of arguments, returns a short vector of return values, and carries out as many evaluations of the function in a single invocation as the number of arguments it received. The compiler then may vectorized the code across a few consecutive invocations of the function. In the example above, the compiler can send two sets of arguments, 64 bits each, in the 128 bit registers of the SSE4.2 ISA and receive two return values in an XMM register. It then can vectorized across each two consecutive invocation of the function. The invocation then can use a vector loop:

```
#pragma simd
for (int i=0; i<NUM_OPTIONS; i++) {
      call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
```

Use of argument qualifiers:

The ability to designate a function argument as a uniform or a linear argument can gain significant performance improvement. The author of the SIMD enabled function may be able to determine that their function can gain performance from these designations and use them accordingly. The main use case is for memory access, and makes the difference between the very efficient vector load / store operations and the slower gather / scatter operations.

Illustration:

```
__declspec(vector)
void
vec_add ( float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

Recall that the vector version of the function takes a short (2, 4 or 8, depending on the size of the HW registers in the ABI of the platform to which the function is compiled) vector or r pointers, and the same number of op1, op2 pointers and i values. In general, there is no relationship among the multiple values of the r pointers, and among the op1 pointers and the op2 pointers. Therefore, the collection of memory accesses that results from op1[i] are unrelated to each other. The implementation can either use a gather operation or multiple scalar loads and merge the values. Similarly, the assignment into r[i] would either be a scatter operation or a split of the value of the vector addition into multiple scalar values and a sequence of scalar stores. This implementation would be correct for any combination of input values. However, the most prevalent use of this kind of interface is when the pointers are base addresses of arrays, and the i is a loop iteration variable. In that case, the implementation can greatly benefit from the knowledge of these properties of the arguments, and can use vector loads for the set of op1[i] operations, and a vector store for the r[i] operation. The programmer can inform the compiler of these relationships using the syntax:

```
__declspec (vector (uniform (r,op1, op2) linear(i))
void
vec_add(float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

However, the author of the function may not be sure that all invocation of the function will use base addresses as arguments. What if some invocation do use base address, but others use pointers with no particular relationship among them? Consider the two calling sequences below:

```
#pragma simd
for (int i = 0; i<N; ++i) {
    vec_add(a,b,c,i);
}
```

```
#pragma simd
for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[[i]],c[x3[[i]],i);
}
```

The first caller can benefit from designating the arguments as uniform / linear. However, if that is the way the function is declared, then it becomes incorrect to use the implementation that uses vector loads and stores for the second call site. In that case, the compiler will have to avoid using the vector version of the SIMD enabled function, and instead will have to invoke the scalar version, processing one operation at a time. The author of the SIMD enable function can satisfy both ways of invoking the function by providing two declspec directives, one for each envisioned case, as follows:

```
__declspec( vector )
__declspec (vector (uniform (r,op1, op2) linear(i))
void
vec_add(float *r, float *op1, float *op2, int i)
{
    r[i] = op1[i] + op2[i];
}
```

In this case, the compiler generates at least 3 translations of the function. One scalar, one to be used with uniform and linear arguments, and one to be used in other vector invocation. The compiler matches the right version of the function when compiling the code that invokes the function.

Summary

This paper presented a brief introduction into vector programming as a subset of parallel programming. Current data shows that financial applications such as trinomial trees, Monte Carlo simulation, LIBOR market model and the like can significantly benefit from parallelism in general and exploding vector execution in particular. These codes can be effectively and productively parallelized and vectorized using many parallelization techniques such as OpenMP, TBB and similar choices, and vectorized using the techniques described here. Language support for vector programming will continue to evolve as the HW presents new capabilities and more importantly, as users present new challenges and algorithms that can benefit from more syntax. Intel is looking forward to such interactions, and to help solve real programming challenges.