Document number N3520
Date 2013-01-11
robert.geva@intel.com

Critical sections in vector loops

A solution to a problem with vector loops raised in the Portland meeting

Robert Geva

In the WG21 meeting in Portland in October 2012, I presented "parallel loops and vector loops". The biggest concern raised at that presentation was regarding the proposal that critical sections would be disallowed inside vector loops. It is worthwhile to emphasize that the occurrence of the problem is unlikely, and the way in which this concern was raised was not in any way specific to critical sections and vector loops. The reason for the concern was that any exclusion of one language construct inside another creates a context which programmers have to be aware of, and that is error prone.

Why is there a problem?

The order of evaluation of expressions in vector loops differs from that of serial loops and that of parallel loops. Iterations of a parallel loop can execute in any order, and it is therefore acceptable for the iteration corresponding to i==4 to be fully evaluated before any expression in the iteration corresponding to i==2 starts. This order is not permissible in vector loops. As the proposal captures existing practices, it specifies a possible order of evaluation of the expressions in vector loops. For example, a vector loop with 4 iterations and the expressions A, B and C can execute as:

```
A0;A1;A2;A3
B0;B1;B2;B3;
C0;C1;C2;C3;
```

This order of evaluation then allows the compiler to group the 4 instances of A, which are now consecutive, and generate a single vector instruction for them instead of 4 scalar instructions.

It is also important to note that existing practices expect the vector order of evaluation to be allowed but not mandated. There are multiple downsides to mandating the vector order of evaluation, such as significant loss of performance when the consecutive expressions are not vectorizeable and may cause generation of temporaries.

Another inherent characteristic of vector loops is that vectorized iterations cannot make forward progress independent of each other. Therefore, if vectorized iterations of a loop are trying to acquire a lock, then if the first iteration gets the lock, then the second iteration cannot get it. Yet the first iteration cannot make forward progress and release the lock either, and the result is a deadlock.

There are two concepts that are part of the proposal for vector loops, which will help make the proposed solution to the above problem obvious to understand. These concepts are quoted here because they help understand the solution to the specific concern raised in the meeting. Their inclusion in the standard are not require for the purpose of the proposed solution.

Elemental Functions

Elemental functions are a language construct that can be applied to functions that may be called from vector loops. When called from a vector loop, the body of an elemental function executes as if it was part of the loop body. The function is invoked once per vector loop iteration, and the body of the function executes a vector chuck of times per invocation instead of only once. A chunk of arguments can be packed together (for example, inside a HW vector register) and be passed to the single invocation of the function. By construct, if there is a call to a function inside a vector loop which is not an elemental function, then no vector code for that function exists. The function will be called once per iteration.

To illustrate, consider a loop with a function call, compiled for a CPU with 4-element-wide vector registers:

```
float f(int);
float a[4], b[4], c[4];
simd_for (int i = 0; i < 4; ++i) {
    a[i] += b[i];
    c[i] = f(i);
    a[i] *= c[i];
}
```

If the function f is an elemental function then 4 sets of arguments are collected and passed together, the function is invoked once and returns a vector of results:

```
float4 vec_f(int4);
float4 t1 = *(float4 *)b;
*(float4 *)a += t1;
float4 t2 = vec_f((float4)(0, 1, 2, 3));
*(float4 *)c = t2;
*(float4 *)a *= t2;
```

whereas if the function f is a “regular”, not elemental function, then it is invoked 4 times, and the results have to be packed together:

```
float4 t1 = *(float4 *)b;
*(float4 *)a += t1;
float t3 = f(0);
float t4 = f(1);
float t5 = f(2);
```

```
float t6 = f(3);
float4 t2 = (float4)(t3, t4, t5, t6);
*(float4 *)c = t2;
*(float4 *)a *= t2;
```

Obviously, operations that acquire and release locks are "regular", not elemental function calls. In the context of the problem and solution described here, the thing to note is the order of evaluation of "regular" (non elemental) function calls within a vector loop – their argument passing and the collection of their return values.

In-order blocks

A part of the proposal for vector loops is to allow the programmer to constrain a section of the loop to execute in scalar order (i.e. the same order the expression would evaluate in existing loops) while allowing the rest of the loop to execute in vector order. Use cases for such a construct include the ability to use a critical section, appending nodes to a linked list and others. The semantics of an in-order block are, obviously, that the expressions within the block are evaluated in the order as specified in the C++11 spec, not in the vector order that is allowed by the surrounding vector loop.

The Original, Restrictive Solution

The intent in the proposal as presented in October is simply to forbid (i.e., make ill-formed or undefined behavior) the use of critical sections in vector loops. Note that this restriction is not as severe as some may have feared from my presentation in Portland. A non-elemental function call could acquire a lock, for example, provided that it also releases the lock before returning to the loop. Such a function would be valid, since it does not impose an illegal synchronization requirement across loop iterations. Even acquiring and not releasing a lock would valid, provided no other iteration attempts to acquire the same lock. The only problem scenario is when a vector loop invokes a function that tries to acquire a lock, and return to the loop without releasing it, and then the subsequent iteration is trying to acquire the same lock.

The Non-restrictive Solution

In order to not create a special context to exclude critical sections from vector loops, the solution is simply to specify the region of the loop from the first call to a non-elemental function to the last call to a non-elemental function as an in-order block. With that order of evaluation, the first iteration will be able to acquire a lock, continue processing and release the lock before the second iteration will attempt to acquire the lock, and therefore the deadlock scenario does not occur. The advantage of this solution is that it simplifies the standard. The cost is that some optimization opportunities could be lost.