Function/Kernel Vectorization via Loop Vectorizer

Matt Masten, Evgeniy Tyurin, Konstantina Mitropoulou, Eric Garcia and Hideki Saito
Intel Corporation
firstname.lastname@intel.com

***Abstract*—Currently, there are three vectorizers in the LLVM trunk: Loop Vectorizer, SLP Vectorizer, and Load-Store Vectorizer. There is a need for vectorizing functions/kernels: 1) Function calls are an integral part of programming real world application code and we cannot always rely on fully inlining them. When a function call is made from a vectorized context such as vectorized loop or vectorized function, if there are no vectorized callees available, the call has to be made to a scalar callee, one vector element at a time. At the programming model level, OpenMP declare simd is a standardized syntax to address this problem. LLVM needs a vectorizer to properly vectorize OpenMP declare simd functions. 2) Also, in the GPGPU programming model, such as OpenCL, work-item (thread) parallelism is not expressed with a loop; it is implicit in the execution of the kernels. In order to exploit SIMD parallelism at this top-level (thread-level), we need to start from vectorizing the kernel.**

One of the obvious ways to vectorize functions/kernels is to add a fourth vectorizer that specifically deals with function vectorization. In this paper, we argue that such a naive approach will lead us to sub-optimal performance and/or higher maintenance burden. Instead, we present a technique to take advantages of the current functionalities and future improvements of Loop Vectorizer in order to vectorize functions and kernels.

I. Introduction

Vectorization has been a widely adopted technology for achieving high computational throughput on high performance processors. In the early days, deeply pipelined vector execution units dominated the implementation of vector parallelism (e.g., [29], [37]). Modern microprocessors, however, implement vector parallelism via SIMD execution units [3], [12], [17]. A SIMD unit executes a single operation over multiple data elements at the same time. For example a 4-wide vector `add` operation executes four scalar `adds` in parallel. As the processors implement wider and wider SIMD units [3], [12], [17], importance of vectorzing more parts of the application programs steadily grew over the last decade, especially in the areas of High Performance Computing (HPC).

Traditionally, vectorization has been applied to loops. Loop Vectorization has been widely studied in the literature over several decades (e.g., [7], [8], [28], [38], [42], [47]). It works by converting multiple consecutive iterations of a scalar loop into a single iteration of a vectorized loop body (Figure 1).

Function call sites have always been a bottle-neck for Loop Vectorization. A naive solution is to keep the call site scalar; a call is made to each vector element, one at a time. This usually involves the overhead of moving data from the vector registers into the scalar registers and vice versa, and provides no speed up from the callee side, since the callee code is still scalar.

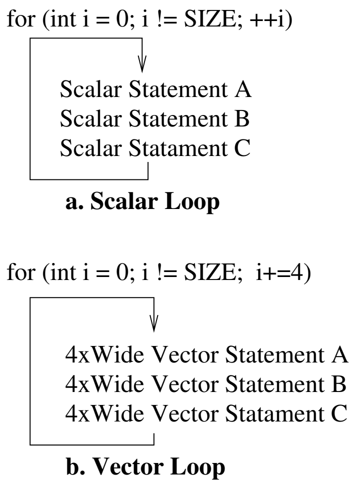

for (int i = 0; i != SIZE; ++i)
Scalar Statement A
Scalar Statement B
Scalar Statament C
a. Scalar Loop
for (int i = 0; i != SIZE; i+=4)
4xWide Vector Statement A
4xWide Vector Statement B
4xWide Vector Statament C
b. Vector Loop

Fig. 1: Conceptual representation of Loop Vectorization with a Vector Factor = 4.

A commonly applied alternative is aggressively inlining the called function into the loop and vectorize the inlined function body as part of the loop body. This is a powerful and often high-performing strategy, but it is not always feasible, due to reasons such as code size, compile time, recursive calls, and callee availability only in OBJ/ASM forms. Therefore, another solution, to actually deal with vectorizing a call, is still needed.

For a long time, many compiler vendors have been supplying vectorized math library functions that can be invoked from the vectorized caller context. In LLVM, VECLIB implements such a functionality, and Intel's SVML (Intel Short Vector Math Library [2]) can be invoked through it. This mechanism provided excellent speedups, but limited only to functions available in libm and other popular and well-understood standard libraries. Intel's Cilk(TM)Plus [1] SIMD enabled functions pioneered the programming model support for explicit vectorization of user-functions and later became a foundation for the `OpenMP declare simd` functions. Both of them define the vector interface for the caller and the callee such that the vectorized calls can be made even if the caller and the callee belong to different compilation modules.

Currently, LLVM trunk supports Loop Vectorization, SLP Vectorization, and Load-Store Vectorization. The LoopVectorize pass historically has been focusing on vectorizing inner-most loops. There is an ongoing infrastructure update to enable vectorization of outer level loops [5]. The SLPVectorize pass deals with vector parallelism across multiple instructions of the same kind and grows the groups of SIMDizable instructions using use-def chains. The LoadStoreVectorize pass merges loads/stores to/from sequential memory addresses into vector loads/stores, mainly targeting GPUs.

In LLVM, the Loop Vectorizer already has most of the information needed to map the scalar function name to an appropriately mangled vector function name for the given calling context. Thus, a relatively simple extension is all we need to implement the caller side support. SLP Vectorizer and Loop Vectorizer should be able to share the most of the mechanism. In this paper, we discuss the caller side support in the context of implementing `OpenMP declare simd` functionality.

The need for vectorizing a function is not limited to the callee functions of Loop Vectorization. In the GPGPU programming model such as OpenCL, parallelism at the top level is represented by the parallel invocation of a kernel. In order to exploit SIMD parallelism at the top level (thread/kernel level), it is imperative to vectorize the kernel itself. In this paper, we will also explain the great similarity between `OpenMP declare simd` vectorization and OpenCL kernel vectorization and how the facility built for the former is extended to support the latter.

The naive approach would be to create one vectorizer for functions and another one for OpenCL kernels. In this paper, however, we will explain the great similarity between OpenMP declare simd vectorization and OpenCL kernel vectorization and how the facility built for the former is extended to support the latter. Furthermore, we will also point out that function and kernel vectorization are very similar to loop vectorization.

The contributions of this paper are:

- We present a new architecture for function vectorization without introducing yet another vectorization pass.
- We present VecClone, a pre-processing pass right before the Loop Vectorizer, which transforms the function vectorization problem into a Loop Vectorization problem.
- We extend VecClone to handle OpenCL kernel vectorization.

The rest of the paper is organized as follows: Section II introduces the proposed VecClone-based function vectorization approach while Section III describes it in greater detail. Section IV presents the vectorization of OpenCL kernels using VecClone, and Section V describes how all components are put together and presents the overall architecture. Finally, Section VI presents an overview of the related work, and Section VII concludes the paper.

II. A New Approach toward Function Vectorization

A. Function Vectorization

As already explained in Section I, call-sites are a bottleneck for the Loop Vectorizer. Given that function inlining is not always possible, we need to seek an alternative. Function Vectorization aims at solving this problem, by enabling seamless vectorization across function calls. It enables vectorization of both the caller and callee sites, as shown in Figure 2. Both caller and callee get widened, which allows the loop of the caller site to be fully vectorized.

```
#pragma omp declare simd simdlen(4) uniform(a) linear(k)
float dowork(float *a, int k) {
        a[k] = a[k] + 9.8f;
        return a[k];
}

float a[4096];
int main() {
    int k;
    #pragma omp simd
    for (k = 0; k < 4096; k++){
         a[k] = k * 0.5;
         a[k] = dowork(a, k);
    }
}
```

(a) Before Function Vectorization. The function to be vectorized is marked with `#pragma omp declare simd`.

```
<VL x float> vec_dowork(float *a, int k) {
        a[k:k+VL−1] = a[k:k+VL−1] + 9.8f;
        return a[k:k+VL−1];
}

float a[4096];
int main() {
    int k;
    for (k = 0; k < 4096; k+=VL){
         a[k:k+VL−1] = {k, k+1, k+2, ..., k+VL−1} * 0.5;
         a[k:k+VL−1] = vec_dowork(a, k);
    }
}
```

(b) After Function Vectorization, a copy(clone) of the function dowork is created, the loop body in the main function is widened and the scalar call to dowork is replaced by the vectorized call.

Fig. 2: High-level Function Vectorization Example.

B. Implementation Strategy

After formulating the function vectorization problem, it is relatively straightforward to think about implementing it as a new vectorizer specifically aimed at vectorizing functions. In fact, different kinds of vectorizers, such as LoopVectorize, SLPVectorize, LoadStoreVectorize, and now-retired BB-Vectorize passes have been built in the LLVM along that idea. That approach has an obvious advantages of isolating the problems/solutions and building modular blocks of code specifically aimed at solving the given problem, however, the downside includes each of them being implemented as monolithic optimizers and has very little reusability among them.

If vectorization had stayed as a simple widening of the incoming context, the drawbacks would not have been a major problem. However, as the importance of vectorization is rediscovered, more and more optimizations are invented and added to them, and as a result we started to observe more similarities amongst them than their differences. For example, LoopVectorize can benefit from SLP-awareness and vice versa.

The world is quickly shifting from exploiting one kind of vector parallelism to optimally coordinating multiple levels and kinds of vector parallelism. If we are to newly design a purpose-built function vectorizer, we are going against the big paradigm shift we are facing with.

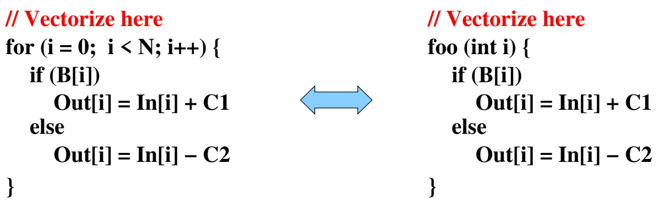
C. Function Vectorization versus Loop Vectorization

Function Vectorization has been studied in engineering, academic, and standardization angles [4], [19], [20], [41]. The Function Vectorizer, as described by [19], consists of the following phases:

1) Preparatory transformations which ensure that each loop will have one incoming edge and one back-edge.
2) Vectorization analysis (e.g. they identify which values to broadcast, which gep operations should be replicated.)
3) Mask generation.
4) Select generation.
5) CFG linearization wherever it is possible.
6) Instruction vectorization.

These steps are very similar to those performed by the Loop Vectorizer. Therefore, instead of creating a new vectorizer from scratch, we can re-use the Loop Vectorizer. This can be accomplished by massaging the function body accordingly. In this way, we will be able to get the function vectorized with the Loop Vectorizer.

The equivalence between loop vectorization and function vectorization is shown in Figure 3. Given the loop on the left-hand side that gets vectorized at `// Vectorize here`, we can extract the loop body and place it into a function `foo()` (right-hand side) and vectorize at the function level. Similarly, given a function marked to be vectorized as shown on the right-hand side, we can wrap a loop around the function body and vectorize at the loop-level.



```
// Vectorize here
for (i = 0;  i < N; i++) {
  if (B[i])
    Out[i] = In[i] + C1
  else
    Out[i] = In[i] – C2
}
```

```
// Vectorize here
foo (int i) {
  if (B[i])
    Out[i] = In[i] + C1
  else
    Out[i] = In[i] – C2
}
```

Fig. 3: Equivalence between Loop Vectorization (left-hand side) and function vectorization (right-hand side) [38].

D. The New Architecture for Function Vectorization (The VecClone pass)

To leverage the Loop Vectorizer for enabling Function Vectorization, we need to perform a set of preparatory transformations, which we call VecClone. VecClone creates a new copy (clone) of the function and emits a for-loop around its body. This loop triggers the Loop Vectorizer which widens the code. At the caller site, the original function call is replaced by a call to the new cloned (vectorized) function. In this way, the VecClone-based architecture succeeds in vectorizing both the caller and callee sites. Overall, this architecture transforms the function vectorization problem into a loop vectorization problem. This makes the maintenance of this design more manageable and less error-prone than the alternative of a dedicated function vectorizer. The VecClone pass is placed right before the loop vectorizer, as shown in Figure 4.

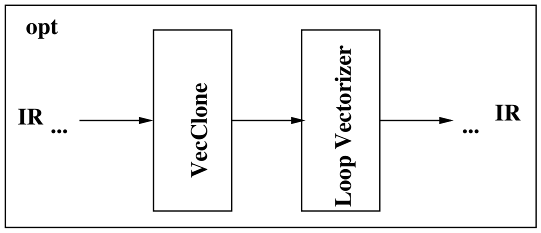


Fig. 4: The compilation pipeline with VecClone.

Moreover, the VecClone approach can be extended to implement vectorization of OpenCL kernels, without the need for yet another vectorization pass. The OpenCL programming model implements a Single Instruction Multiple Thread (SIMT) model which, similarly to OpenMP, describes the parallelism in an explicit way. In OpenCL we need vectorize the kernels, which are similar to regular functions. There are, however, several OpenCL-specific technical differences, which need special treatment. Section III describes how VecClone transforms the code and Section IV presents how VecClone can be re-used to vectorize OpenCL kernels.

III. VecClone

As a first step in VecClone, the user needs to annotate the function that should be vectorized with a `#pragma omp declare simd` (Figure 5a). The VecClone architecture will go through the following steps to vectorize the code:

1) Initially, the front-end parses the `pragma omp declare simd` clauses, in order to generate a unique mangled name that describes the vector factor, the ISA etc., as in the function name of Figure 5b. The complete details are described in Section III-A (Phase 1).
2) Next, the VecClone pass creates a clone of the original function (with the mangled name) and emits a loop around the function body and annotates it with a `#pragma omp simdlen(VL)`, as shown in Figure 5b. This is presented in Section III-B (Phase 2).
3) Now, it is time to vectorize the body of the cloned function. The for-loop that we emitted in VecClone pass triggers the Loop Vectorizer. This is shown in Section III-C (Phase 3).
4) Finally, in the Loop Vectorizer, the original function call is replaced with a call to the clone. In addition, the function's vector parameters and the return value of the cloned (non-void) function call are widened. This is described in Section III-D (Phase 4).

A. Phase 1: Create vector-variant attributes

When a user marks the function that he wants to vectorize with a `#pragma omp declare simd` (as it is shown in

```
#pragma omp declare simd simdlen(4) uniform(a) linear(k)
float dowork(float *a, int k) {
        a[k] = a[k] + 9.8f;
        return a[k];
}
```

(a) The user marks the function that he wants to vectorize with a `#pragma omp declare simd`.

```
define __stdcall <4 x 32> @_ZGVbN4ul_dowork(f32* %a, i32 %k) #0 {
   #pragma omp simdlen(4)
   for (int %t = %k; %t < %k + 4; %t++)
        %a[%t] = %a[%t] + 9.8f;
   vec_load xmm0, %a[%k : 4]
   return xmm0;
}
```

(b) VecClone creates a clone ("_ZGVb4Nul_dowork") of the original function. Next, it wraps the body of the cloned function in a loop, it widens the function arguments and the return value and it emits the `pragma`.

```
define __stdcall <4 x 32> @_ZGVbN4ul_dowork(f32* %a, i32 %k) #0 {
   vec_load xmm1, %a[%k : 4]
   xmm0 = vec_add xmm1, [9.8f, 9.8f, 9.8f, 9.8f]
   store %a[%k : 4], xmm0
   return xmm0;
}
```

(c) The Loop Vectorizer widens the cloned function ("_ZGVb4Nul_dowork") and the loop of the main function.

Fig. 5: The transformation of the callee site.

```
float a[4096];
int main() {
    int k;
    #pragma omp simd
    for (k = 0; k < 4096; k++){
         a[k] = k * 0.5;
         a[k] = dowork(a, k);
    }
}
```

(a) Original code of the main function.

```
float a[4096];
int main () {
    int k;
    vectorized_for (k=0; k<4096; k+=VL) {
         a[k:VL] = {k, k+1, k+2, k+VL−1} * 0.5;
         a[k:VL] = _zGVb4Nul_dowork(a, k);
    }
}
```

(b) Instead of the original function, the cloned function is called.

Fig. 6: The transformation of the caller site.

Figure 5a), the compiler will generate at least one vectorized version of the `dowork()` function.

The user provides the compiler with information such as the vector length, whether the arguments are uniform, linear or vector etc. through the `pragma` clauses. The front-end analyzes these clauses and annotates the function with this information in the form of vector-variant attributes. This is done by encoding the given information into a mangled name using Intel's Vector ABI [43]. Alternatively, the name mangling can be done by any other vector ABI e.g. ARM's Vector ABI [44]. According to these Vector ABIs, the mangled name should have the following format: `_ZGV + <isa> + <mask> + <vlen> + <vparameters> + '_'`.

In the example of Figure 5a (that uses Intel's ABI), the front-end will create a "_ZGVbN4ul_" mangled name where "_ZGV" is the prefix of the vector function name, "b" indicates the xmm ISA class of the target processor (assuming that we vectorize for 128-bits), "N" indicates that this is an un-masked version ("M" indicates that the function is vectorized with a mask), "4" is the vector length (assuming that we vectorize here for 4 elements/work items), "u" indicates that the first parameter has the uniform property, "l" indicates that the second parameter has linear property. Once the mangled name is created, it is added as an attribute to the function. Later on, VecClone reads the function attribute and it transforms the code accordingly for the Loop Vectorizer.

Also note that multiple vectorized versions of the original function might be needed. For example, the function might be vectorized for different vector lengths, depending on the vector length used on the caller site. In this case, we should generate one vectorized version of the function for each of these vector lengths. Therefore, multiple mangled names will be created and consequently, multiple vector-variant attributes will be added in the function. Similar to C++ overloaded functions, uniqueness of multiple vector-variants is represented by unique mangled names.

B. Phase 2: VecClone Pass

Listing 1 describes the main steps of VecClone. Initially, VecClone determines which functions to clone and transform based on the existence of vector variant attributes (listing 1 line 4). The code transformation begins for each vector variant (listing 1 line 11) by creating a copy of the original function (listing 1 line 16). The name of the clone is based on the following format:

mangled name + "_" + *original function's name*

The mangled name is taken from the variant which is passed as a parameter to CloneFunction (listing 1 line 16). For the example of Figure 5b, the name of the cloned function will be `_ZGVb4Nul_dowork`.

Next, the loop around the function body is inserted (listing 1 line 20). To do that, the control-flow is split after the entry block (listing 1 line 20) and before the return block (listing 1 line 21). Then, a new latch is created (listing 1 line 22) and the back-edge is emitted (listing 1 line 23).

After emitting the loop around the function body, the next step is to widen the function parameters and the return values (listing 1 line 26). For this reason, a new vector `alloca` instruction is created for all the parameters and the return values. Next, the uses of the widened parameters are updated (listing 1 line 28). This is done by replacing the initial store to the old `alloca` with a vector one. The uses of the old

Listing 1: The main functions of VecClone pass.

```
1    runOnModule{Module &M}{
2
3      std::map<Function*, std::vector<StringRef>> FunctionsToVectorize;
4      getFunctionsToVectorize(M, FunctionsToVectorize);
5
6      for (elem : FunctionsToVectorize) {
7
8        Function& F = *(elem->first);
9        std::vector<StringRef> Variants = elem->second;
10
11       for (unsigned i=0; i< Variants.size(); i++) {
12
13         VectorVariant Variant(Variants[i]);
14
15         // Create a new clone of the function.
16         Function *Clone = CloneFunction(Func);
17
18         // Emit the loop around the body of the cloned function. The trip count of the loop
19         // is the Vector Length.
20         BasicBlock *LoopBlock = splitEntryIntoLoop(Clone, EntryBlock, Variant);
21         BasicBlock *ReturnBlock = splitLoopIntoReturn(Clone, ReturnBlock);
22         BasicBlock *LoopExitBlock = createLoopExit(Clone, ReturnBlock);
23         PHINode *Phi = createPhiAndBackedgeForLoop(Clone, EntryBlock, LoopBlock, LoopExitBlock, Variant
               ↪.getVectorLength());
24
25         // Expand scalar parameters to vector ones and update their uses in the function.
26         Instruction *ExpandedReturn = expandVectorParametersAndReturn(Clone, Variant, EntryBlock,
               ↪LoopBlock, ReturnBlock);
27
28         updateScalarMemRefsWithVector(Clone, F, EntryBlock, ReturnBlock, Phi);
29
30         // Update any linear variables with the appropriate stride.
31         updateLinearReferences(Clone, F, Variant, Phi);
32
33         // Remove the old scalar instructions associated with the return and
34         // replace with packing instructions.
35         updateReturnBlockInstructions(Clone, ReturnBlock, ExpandedReturn);
36
37         // Remove the old scalar allocas associated with vector parameters.
38         removeScalarAllocasForVectorParams();
39
40         // If this is the masked vector variant, insert the mask condition and if/else blocks.
41         if (Variant.isMasked())
42            insertSplitForMaskedVariant(Clone, LoopBlock, LoopExitBlock, Mask, Phi);
43
44         // Insert the basic blocks that mark the beginning/end of the SIMD loop.
45         insertDirectiveIntrisics(M, Clone, F, Variant, EntryBlock, LoopExitBlock, ReturnBlock);
46       }
47     }
```

`alloca` within the loop will be replaced with a `GEP` using the old `alloca`'s address along with the proper loop index.

Then, the linear variables are updated with the appropriate stride (listing 1 line 31). Before each use of the parameter, a `mul`/`add` sequence is inserted. For linear pointer parameters, the stride calculation is just a `mul` instruction using the loop induction variable and the stride value on the parameter. This `mul` instruction is then used as the index of the `GEP` that will be inserted before the next use of the parameter. The users of the parameters are also updated with the new calculation involving the stride.

Next, we update the scalar instructions that are related to the old scalar `return`. These instructions are erased and they are replaced with vector ones (listing 1 line 35). Then, we clean up the code from any remaining scalar `allocas` associated with the vector parameters (listing 1 line 38).

If it is needed to generate a masked variant, then VecClone will add the mask condition and `if`/`else` blocks (listing 1 line 41). The mask condition is a compare instruction that checks whether the mask bit is on Figure 7.

Finally, the `#pragma omp simdlen()` is emitted before the for-loop as it is shown in Figure 5b. In this way, the vector length is communicated to the Loop Vectorizer and the SIMD loop region [45] is created. Now, the code is ready for the Loop Vectorizer.

C. Phase 3: Loop Vectorization of Cloned Function (Callee Site)

The Loop Vectorizer should do two things: 1. widen the cloned function (callee site), 2. widen the arguments and the return value of the call of the cloned function at the caller

```
define __stdcall <4 x 32> @_ZGVbM4ul_dowork(f32* %a,
                                     i32 %k, <VL x int> mask) #0 {
  #pragma omp simdlen(4)
  for (int %t = %k; %t < %k + 4; %t++) {
     if (mask[%t] != 0)
        %a[%t] = %a[%t] + 9.8f;
  }
  vec_load xmm0, %a[%k : 4]
  return xmm0;
}
```

Fig. 7: Variant with mask condition.

site. The Loop Vectorizer is a function pass and it processes each function separately. In the case of the cloned function, the Loop Vectorizer widens the loop according to the information it gets from the OpenMP `#pragma` (Figure 5c).

D. Phase 4: Vectorizing the Caller Site

So far, we have shown how to vectorize the function itself (the `dowork()` function of Figure 5a). Next, we are going to show what happens at the caller site: The Loop Vectorizer will start to vectorize each instruction of the k-loop of the main function in Figure 6a. Once it reaches the call to the `dowork()`, the Loop Vectorizer checks if the function has vector-variant attributes. Next, it analyzes the function's parameters in the loop in order to determine whether the parameters have the following properties:

- uniform
- linear + stride
- vector
- aligned
- called inside a conditional branch or not.

Based on these properties, the signature of the vectorized call is generated. In the example of Figure 6a, the signature is "_ZGVbN4ul_" because:

- "a" is uniform because it is the base address of the array "a",
- "k" is linear, as "k" is the induction variable with stride=1,
- SIMD "dowork" is called unconditionally in the candidate k loop,
- assume that it is compiled for SSE4.1 with the vector length VL=4.

Next, we check if this signature "_ZGVbN4ul_" exists in the vector-variant attributes. If yes, then the suitable vectorized version is found. Then, the call to the original function is replaced with the call to the matched vectorized version. In the example of Figure 6a, the `dowork()` is replaced with `_ZGVb4Nul_dowork` call. Finally, the Loop Vectorizer widens the parameters and the return value of the `_ZGVb4Nul_dowork` call.

IV. Extension to Support OpenCL Kernels

The OpenCL programming model partitions the code into host and kernel code. The kernel code is not contained in a loop-nest. Instead, it explicitly operates on an index space

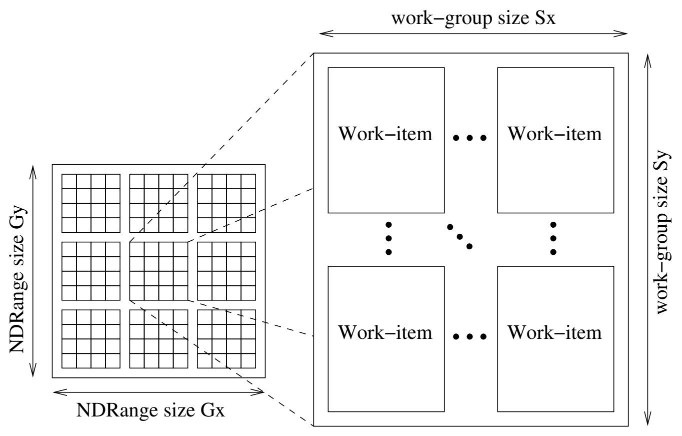
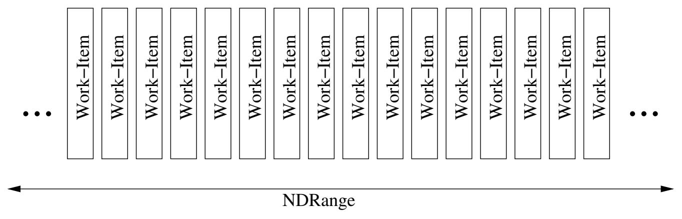
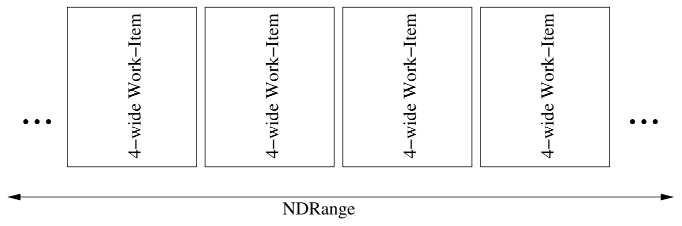


Fig. 8: An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs [6].



(a) The execution of work-items in NDRange.



(b) Work-items after vectorization.

Fig. 9: OpenCL work-item execution before and after vectorization.

called NDRange (Figure 8). An NDRange is an N-dimensional index space, where N is one, two or three.

Each kernel instance (work-item) gets its own index in the NDRange, called global ID. The work-items are organized into work-groups (Figure 8). The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. All work-items are allowed to execute in any order (there is no explicit ordering among them).

We can vectorize the kernel code by conceptually merging multiple work-items into a single wider work-item (Figure 9). In this way, fewer work-group loop iterations are needed to complete the program execution. To achieve this, we can either develop a dedicated vectorizer, or we can make use of VecClone's function vectorization support. This works because an OpenCL kernel is a function, and as such it can be vectorized by VecClone with minor OpenCL-specific modifications.

A. *Vectorizing OpenCL Kernels with VecClone*

OpenCL kernel vectorization is essentially function vectorization with some addition functionality. OpenCL kernels differ from *OpenMP declare simd* functions in the following key aspects. In OpenCL:

1) all kernel arguments are uniform,
2) there are no return values,
3) there are OpenCL-language specifics, for example `get_global_id()`, `get_local_id()` and other intrinsic functions,
4) each kernel instance can be indexed in a 3-dimensional space (or less).

The first two points can already be handled by the existing VecClone implementation. Regarding point three, the OpenCL language specifics can be easily supported with small modifications of the Loop Vectorizer. For example, the `get_global_id()` calls, are loop invariant and are hoisted outside the loop inserted by VecClone. The last point is where kernel vectorization requires some non-trivial support from the function vectorizer. Since a kernel instance could operate on a 3-Dimensional index space, it is up to the vectorizer to figure out which dimension should be vectorized in order to achieve the best performance. In the simple (and most common) case where the index space is 1-Dimensional, vectorization is trivial (Figure 10).

```
_kernel void
f(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    C[x] = A[x] + B[x]
}
```

(a) Original kernel.

```
_kernel void
_ZGVbN4uuu_f(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    #pragma omp simdlen(4)
    for (int i=0; i<4; i++)
        C[x+i] = A[x+i] + B[x+i];
}
```

(b) After VecClone

```
_kernel void
_ZGVbN4uuu_f(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    C[x+0:3] = A[x+0:3] + B[x+0:3];
}
```

(c) After Loop Vectorizer

Fig. 10: Vectorizing one-dimensional kernel.

Unlike OpenMP Function Vectorization where we only need to parse the OpenMP pragma clauses to get all the information needed for creating the vector-variant attributes, in OpenCL we collect the information from many sources, including the OpenCL run-time and other analysis passes. Next, similarly to OpenMP, VecClone creates a new clone of the kernel to be vectorized. Then, it emits a for-loop around the body of the cloned kernel, and a `pragma omp simd` is inserted on it (Figure 10b). The `get_global_id()` is loop invariant, therefore, it is moved outside the for-loop. Now, the cloned kernel is ready to be processed by the Loop Vectorizer.

If the kernel has more than one dimension, then we should find the best dimension to vectorize. To do this, VecClone will generate one clone for each dimension. In each clone, the induction variable of the for-loop will be added to the dimension that we want to vectorize (Figures 11b, 11c and 11d). For example in Figure 11b we want to vectorize on dimension 'x', therefore we change `C[x]` into `C[x+i]`. Next, the Loop Vectorizer will vectorize each clone. A post-processing pass will compare the vectorization cost of each clone and it will keep the one with the smallest cost. The other clones will be discarded. For simpler cases, a pre-processing pass selecting the optimal dimension to vectorize can be employed. For more complex cases, the above mentioned approach provides more precision to the cost modeling.

In our implementation, kernel vectorization is orthogonal to barrier handling. The barriers are treated by the loop vectorizer as an ordinary function call. Firstly, the kernel is vectorized. Next, the kernel is propagated to the barrier pass which resolves the barriers and optimizes the code. It is important to note that the run-time decides whether the scalar or vectorized version will be executed. If the number of the work items is not evenly divided by the Vector Length, then the vectorized code is skipped and all work items are executed by the scalar code. Otherwise the remaining iteration will never reach the barrier in the remainder loop .

V. Overall Architecture

Up to this point, we have shown how VecClone transforms functions and OpenCL kernels so as to enable vectorization. Figure 13 summarizes the overview of the design. As Figure 13 shows, input can come from either the *OpenMP declare simd* functions (top left) or the OpenCL kernels (bottom left). The VecClone pass will process either of them in a similar way, with the OpenCL side applying some additional language-specific transformations. The output of VecClone is fed into the Loop Vectorizer and its output is forwarded to either the rest of the LLVM pipeline, or the OpenCL post-processing pass that selects the best performing variant.

This design minimizes the replication of functionality across both the OpenMP and the OpenCL paths, and is therefore easier to maintain.

VI. Related Work

Traditionally, High Performance Computing (HPC) has made heavy use of vector-powered machines ([29], [37]) over several decades and experimental vector machines. Short SIMD vector ISAs are widely supported by general purpose

```
_kernel void
f(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    size_t z = get_global_id(2);
    C[x] = A[y] + B[z];
}
```

(a) 3-dimensional kernel.

```
_kernel void
_ZGVbN4uuu_f_x(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    size_t z = get_global_id(2);
    #pragma omp simdlen(4)
    for (int i=0; i<4; i++)
        C[x+i] = A[y] + B[z];
}
```

(b) X-dimension variant

```
_kernel void
_ZGVbN4uuu_f_y(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    size_t z = get_global_id(2);
    #pragma omp simdlen(4)
    for (int i=0; i<4; i++)
        C[x] = A[y+i] + B[z];
}
```

(c) Y-dimension variant

```
_kernel void
_ZGVbN4uuu_f_z(_global int *A, _global int *B,_global int *C) {
    size_t x = get_global_id(0);
    size_t y = get_global_id(1);
    size_t z = get_global_id(2);
    #pragma omp simdlen(4)
    for (int i=0; i<4; i++)
        C[x] = A[y] + B[z+i];
}
```

(d) Z-dimension variant

Fig. 11: For each dimension, VecClone will create one variant.

```
i = get_global_id()
A[i] = A[i]+c1
barrier()
B[i] = B[i] + c2
```

(a) Before vectorization.

```
i = get_global_id()
for (k = 0; k < VF; k++) {
    A[i+k] = A[i+k] + c1
    barrier()
    B[i+k] = B[i+k] + c2
}
```

(b) After VecClone.

```
i = get_global_id()
A[i:i+k] = A[i:i+k] + c1
barrier()
B[i:i+k] = B[i:i+k] + c2
```

(c) After vectorization.

Fig. 12: Vectorization of kernels with barriers.

CPUs [17]. Graphics processors (GPUs) [23] also implement vector datapaths for achieving high throughput.

High performance compilers implement auto-vectorization techniques for automatically converting scalar code to vector code. Auto-vectorization is similar to explicit vectorization, but it also includes the additional task of proving the legality of the transformation. There are two main types of auto-vectorization techniques: Loop Vectorization and SLP Vectorization.

Loop auto-vectorization has been studied over several decades. Many fundamental problems of Loop Vectorization have been addressed by early work on the Parallel Fortran Converter [7], [8] and others [14], [21], [46]. Since then, several improvements have been proposed for the loop auto-vectorization algorithms [10], [15], [27], [28], [35], [38], [39], [42]. The effectiveness of Loop Vectorizing compilers has been studied by Maleki et al. [25].

Control-flow linearization is important for Loop/Function Vectorization. Moll et al. [26] present a novel if-conversion technique which linearizes only the divergent branches. In this way, they manage to considerably reduce the overhead over the traditional if-conversion techniques [9], [13], [18], [30].

SLP Vectorization is an alternative auto-vectorization approach, which does not operate on loops, but rather on straight-line code. An early algorithm was introduced by Larsen and Amarasinghe [22]. Both GCC and LLVM implement SLP auto-vectorization, with both of them implementing a variant of the algorithm proposed by Rosen et al. [36]. Several improvements have been proposed that further improve its coverage and the generated code's performance [11], [16], [24], [31]–[34], [40], [48], [49].

Function Vectorization has been the focus of [4], [19], [20], [41]. Karrenberg et.al. [19] proposes a whole function vectorization algorithm, as a separate compiler pass, which includes a set of control-flow transformations. Our VecClone-based scheme re-uses the Loop Vectorizer to perform Function Vectorization. In [20], they extend the whole function vectorization approach to vectorize OpenCL kernels. In this work we show how VecClone can be re-used for vectorizing OpenCL kerenls as well.

VII. Concluding Remarks

In this paper, we argued that LLVM should not add a fourth vectorizer that is purposely built to vectorize functions/kernels. We presented an alternative approach to accomplish the same or better results at a lower overall development/maintenance cost. Our approach takes advantage of existing functionality and future improvements of Loop Vectorizer (e.g. outer-loop vectorization), by transforming the function vectoriza-

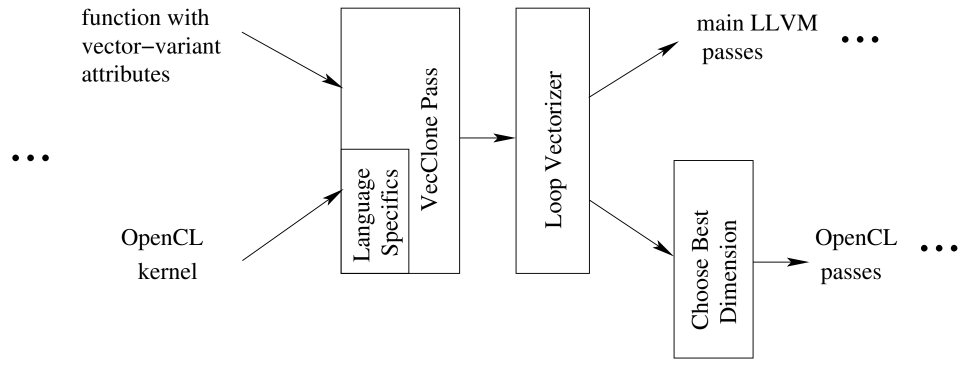


Fig. 13: Overall architecture.

tion problem into a loop vectorization problem. We have implemented the proposed technique to vectorize `OpenMP declare simd` functions and we have also extended it to vectorize OpenCL kernels. We believe that the same idea can be extended to vectorize functions/kernels outside of OpenMP and OpenCL, without the complexity of developing, improving, and maintaining a separate function vectorizer.

References

[1] Intel Cilk Plus. https://www.cilkplus.org/.

[2] Intel Short Vector Math Library. https://software.intel.com/en-us/node/523613.

[3] Intel Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/en-us/node/523613.

[4] OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[5] RFC: Extending LV to vectorize outerloops. http://lists.llvm.org/pipermail/llvm-dev/2016-September/105057.html.

[6] The OpenCL Specification. https://www.khronos.org/registry/OpenCL/.

[7] J. R. Allen and K. Kennedy. PFC: A program to convert fortran to parallel form. Technical Report 82-6, Department of Mathematical Sciences, Rice University, 1982.

[8] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *Tranactions on Programming Languages and Systems (TOPLAS)*, 9(4), 1987.

[9] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189. ACM, 1983.

[10] A. Anderson, A. Malik, and D. Gregg. Automatic vectorization of interleaved data revisited. *ACM Trans. Archit. Code Optim.*, 12(4):50:1–50:25, Dec. 2015.

[11] R. Barik, J. Zhao, and V. Sarkar. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2010.

[12] I. Corportation. IA-32 Intel Architecture Software Developers Manual. *Intel Corportation*, 2018.

[13] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 320–329. IEEE, 2011.

[14] J. Davies, C. Huson, T. Macke, B. Leasure, and M. Wolfe. The KAP/S-1- an advanced source-to-source vectorizer for the S-1 Mark IIa supercomputer. In *Proceedings of the International Conference on Parallel Processing*, 1986.

[15] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[16] J. Huh and J. Tuck. Improving the effectiveness of searching for isomorphic chains in superword level parallelism. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 718–729, New York, NY, USA, 2017. ACM.

[17] Intel Corporation. IA-32 Architectures Optimization Reference Manual, 2007.

[18] R. Karrenberg. Automatic simd vectorization of ssa-based control flow graphs. 2014.

[19] R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 141–150. IEEE Computer Society, 2011.

[20] R. Karrenberg and S. Hack. Improving performance of OpenCL on CPUs. In *International Conference on Compiler Construction*, pages 1–20. Springer, 2012.

[21] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Symposium on Principles of Programming Languages*, 1981.

[22] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), 2008.

[24] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[25] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[26] S. Moll and S. Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 543–556. ACM, 2018.

[27] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[28] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[29] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8), 1992.

[30] M. Pharr and W. R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

[31] V. Porpodas. SuperGraph-SLP Auto-Vectorization. In *2017 International Conference on Parallel Architecture and Compilation (PACT)*, pages 330–342. IEEE, 2017.

[32] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2015.

[33] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 163–174, New York, NY, USA, 2018. ACM.

[34] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. VW-SLP: Auto-Vectorization with Adaptive Vector Width. In *2018 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2018.

[35] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[36] I. Rosen, D. Nuzman, and A. Zaks. Loop-aware SLP in GCC. In *GCC Developers Summit*, 2007.

[37] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1), 1978.

[38] H. Saito et al. Extending LoopVectorizer Towards Supporting OpenMP4.5 SIMD and Outer Loop Auto-Vectorization. https://llvm.org/devmtg/2016-11/Slides/Saito-NextLevelLLVMLoopVectorizer.pdf, 2015.

[39] H. Saito, S. Preis, N. Panchenko, and X. Tian. Reducing the functionality gap between auto-vectorization and explicit vectorization. In N. Maruyama, B. R. de Supinski, and M. Wahib, editors, *OpenMP: Memory, Devices, and Tasks*, pages 173–186, Cham, 2016. Springer International Publishing.

[40] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[41] X. Tian, S. S. Kozhukhov, S. V. Preis, R. Y. Geva, K. A. Pyjov, H. Saito, M. B. Girkar, A. G. Kasov, and N. V. Panchenko. Vectorization of Scalar Functions Including Vectorization Annotations and Vectorized Function Signatures Matching. Patent US9015688 B2.

[42] X. Tian, H. Saito, M. Girkar, S. V. Preis, S. S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 2349–2358, Washington, DC, USA, 2012. IEEE Computer Society.

[43] X. Tian, H. Saito, S. Kozhukhov, K. B. Smith, R. Geva, M. Girkar, and S. V. Preis. Vector Function Application Binary Interface. https://www.cilkplus.org/sites/default/files/open_specifications/Intel-ABI-Vector-Function-2012-v0.9.5.pdf, 2015.

[44] X. Tian, H. Saito, S. Kozhukhov, K. B. Smith, R. Geva, M. Girkar, and S. V. Preis. Vector Function Application Binary Interface Specification for AArch64. https://developer.arm.com/products/software-development-tools/hpc/arm-compiler-for-hpc/vector-function-abi, 2018.

[45] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. Garcia. LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, pages 4:1–4:11, New York, NY, USA, 2017. ACM.

[46] M. Wolfe. Vector optimization vs. vectorization. In *Supercomputing*. Springer, 1988.

[47] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[48] H. Zhou and J. Xue. A compiler approach for exploiting partial SIMD parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[49] H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 59–69. ACM, 2016.