Multi-tier Dynamic Vectorization for Translating GPU Optimizations into CPU Performance

Hee-Seok Kim, Izzat El Hajj, John A Stratton and Wen-Mei W. Hwu
{kim868, elhajj2, stratton, w-hwu}@illinois.edu

IMPACT Technical Report
IMPACT-14-01
University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
Feb 05, 2014
Revision: September 6, 2015

Abstract

Developing high performance GPU code is labor intensive. Ideally, developers could recoup high GPU development costs by generating high-performance programs for CPUs and other architectures from the same source code. However, current OpenCL compilers for non-GPUs do not fully exploit optimizations in well-tuned GPU codes.

To address this problem, we develop an OpenCL implementation that efficiently exploits GPU optimizations on multicore CPUs. Our implementation translates SIMT parallelism into SIMD vectorization and SIMT coalescing into cache-efficient access patterns. These translations are especially challenging when control divergence is present. Our system addresses divergence through a multi-tier vectorization approach based on dynamic convergence checking.

The proposed approach outperforms existing industry implementations achieving geometric mean speedups of 2.26× and 1.09× over AMD's and Intel's OpenCL implementations respectively.

I. Introduction

Modern supercomputers are becoming heterogeneous computing systems equipped with CPUs and accelerators (such as GPUs or Xeon Phis). Each type of device comes with different characteristics that require significant tuning efforts for applications to run on the device efficiently. One fundamental challenge with targeting heterogeneous platforms is maintaining multiple source codes optimized for different platforms.

OpenCL [15] provides a unified interface for different device architectures. The abstract computing model of OpenCL is intended to be architecture neutral enabling functional equivalence across architectures. Due to the close proximity between OpenCL's abstract hardware model and the GPU architectures, executing OpenCL code on GPUs is relatively straightforward. However, compiling OpenCL code for other architectures CPUs in particular requires more involved compiler support.

To efficiently compile GPU kernels for CPUs, a good understanding of how these kernels are optimized is required. High performance GPU kernels require efficient use of the memory subsystem and execution units. In order to improve utilization of the memory subsystem, memory accesses must be coalesced to improve effective memory bandwidth by exploiting spatial locality. Moreover, convergent control flow maximizes utilization of the SIMD execution units which exploit the parallelism from the SIMT execution model to improve instruction throughput. Maximizing effective memory bandwidth and instruction throughput are universal optimization targets, and it would be ideal if such characteristics could be preserved when retargeting optimized GPU kernels to a CPU.

Previous work [22], [10] has approached this problem by serializing work-items of a work-group within barrier separated regions using thread-loops or user-level threads. Such approaches capture program correctness but fail to preserve locality optimizations and SIMD vectorization opportunities. Other approaches [14] have improved on work-item serialization by doing what is equivalent to strip-mining the thread-loops to benefit from SIMD vectorization on the CPU, and using software predication techniques such as masking [13] to handle control divergence. These approaches, however, incur unnecessary masking overhead when control flow is actually convergent.

To overcome these obstacles, we present the design, implementation, and evaluation of the key efficiency-preserving techniques in our OpenCL implementation. To preserve memory access efficiency, loop interchange and vectorization allow work-items in a work-group to maintain the lock-step wave-front-style execution expected by GPU programmers. A multi-tier vectorization technique is used to enable situations where work-items in a work-group encounter divergent if statements or have divergent loop trip counts. The multi-tiered design allows the generated code to maintain its efficiency in the presence of potentially divergent control flow and gradually lose efficiency when divergence arises at run-time. For the experiments in this paper, our compiler generates output code in vectorized C that is well-suited for modern vendor compilers, elevating the final performance noticeably.

In this paper, we make the following contributions:

- We design and implement a compiler that generates efficient vectorized code that emulates the GPU execution behavior and thus enables the transfer of memory access and SIMD execution optimizations from GPUs to CPUs.
- We employ a multi-tiered vectorization technique that can efficiently implement the desired execution behavior even in the presence of potentially divergent conditionals and loops.
- We present real-hardware measurements that demonstrate that the output code outperforms leading vendor OpenCL compilers and related academic compilers.
- We analyze the important use cases that benefit from the proposed technique as well as situations where kernels do not benefit, using popular benchmark suites.

The rest of this paper is organized as follows. Section II outlines previous approaches that have been taken to execute GPU code on CPUs. Section III details the improvements our approach makes on previous approaches and optimizations we perform. Section IV presents and discusses our experimental results. Section V covers some more related work and section VI concludes.

II. Previous Approaches

The problem of compiling GPU kernels for CPU architectures has previously been addressed in various research and industry tools. In this section, we go over a few well-known implementations highlighting their design principles and discussing some performance implications.

A. MCUDA

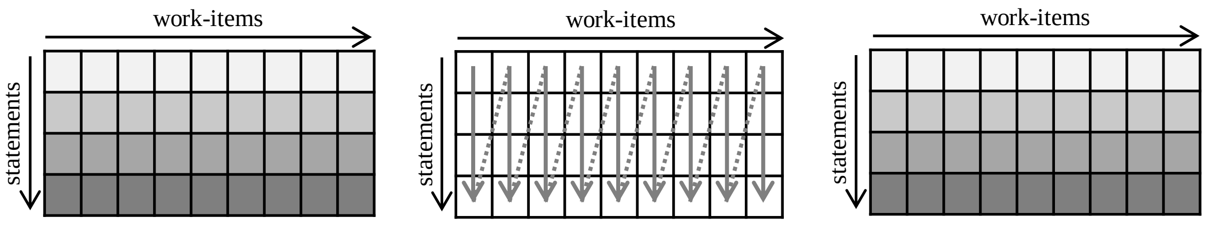
MCUDA [22] is a source-to-source translator from CUDA for GPU architectures to multi-threaded C for multicore CPU architectures. It serializes the work of a thread block within a single CPU thread and parallelizes the work of the kernel at thread block granularity. The work of a thread block is serialized by wrapping regions of CUDA statements with a loop (called a *thread-loop*) that iterates to execute those statements for all threads in the block. The thread blocks are divided among available CPU threads and executed until completion.

The semantics of barrier synchronization are conserved in MCUDA by forcing thread-loops to not include any barriers. The kernel is first divided into regions of code whose boundaries are delineated by barrier synchronizations, and with every control structure containing a barrier synchronization also being a region boundary. The regions are then transformed separately and each is wrapped in a thread-loop. This forces the work for all threads to be completed for one region before the next region begins executing. All thread-dependent variables that span multiple regions must undergo scalar expansion so that their values can be preserved across regions for all threads.

The MCUDA work provides foundation for this work. While MCUDA preserves the functional equivalence and barrier synchronization semantics of GPU kernels, it does not attempt to preserve other aspects of the execution model on which some optimizations are founded.

B. Twin Peaks

Twin Peaks [10] is an OpenCL stack for CPUs developed at AMD, which takes a different approach from MCUDA in mapping blocks and threads (a.k.a. work-groups and work-items in OpenCL) to the CPU. Like MCUDA, it divides work-groups among CPU threads and executes an entire work-group within a single CPU thread until completion. However, it differs in that it uses user-level threads within each work-group for the execution of work-items. Barrier synchronizations are implemented by suspending user-level threads which have reached the barrier until all



time goes from lighter to darker

```
statement1;
for(k=0; k<100; ++k) {
  statement2;
}
statement3;
```

(a) Original OpenCL kernel

time goes in direction of arrows

```
for(x=0; x<N; ++x){
  statement1;
  for(k=0; k<100; ++k) {
    statement2;
  }
  statement3;
}
```

(b) C kernel with thread-loops

time goes from lighter to darker

```
statement1(0:N);
for(k=0; k<100; ++k) {
  statement2(0:N);
}
statement3(0:N);
```

(c) C kernel with vectorization

Fig. 1. Execution pattern of thread-loops compared to static vectorization for convergent code. N denotes the number of work-items. The notation statement(0:N) means that statement is executed for N work-items starting with work-item 0, and that vector expressions are used to execute it. This notation will be used throughout.

remaining work-items for the same work-group have reached the barrier. This barrier implementation is much less performance efficient than MCUDA's. The advantage of Twin Peaks is that it does not rely on compiler techniques, but rather it moves the work-item scheduling into the runtime system. Runtime work-item scheduling allows for incorporating runtime information for more informed scheduling as well as reuse of off-the-shelf compilers and debugging tools. However, the user-level threading approach makes it more difficult to incorporate the techniques we present in this paper.

C. Karrenberg et al.

Karrenberg et al. [14], [13] present techniques for running OpenCL code on CPUs using SIMD vectorization. Their compiler divides the code at barrier synchronizations and vectorizes each region with the size of the hardware SIMD width (W). They execute an entire region for W work-items before moving on to the next W. This approach is similar to the MCUDA approach, except that the thread-loop is strip-mined. They also use software predication techniques to handle situations where control divergence arises. The main differences between this technique and ours will be discussed in the related work section after our technique is presented.

D. Intel

Detailed information about the implementation of Intel's OpenCL stack [12] is not disclosed to the public. Our experiments indicate that there seems to be some similarity with MCUDA in how it maps work-groups and work-items to the CPU execution units. However, details about code generation and compilation techniques are not well known. In this paper, we compare the performance of our implementation against the Intel OpenCL stack as a black box.

III. Proposed Approach

As with previous approaches, our proposed OpenCL implementation divides work among CPU threads at work-group granularity. However, it differs from previous approaches in the treatment of work-items within the same work-group. Our approach vectorizes kernel statements across the entire work-group when code regions are convergent. When code is divergent, it uses a multi-tiered approach which tests for convergence at runtime and selects between different execution paths accordingly. This section highlights the key techniques that our implementation adopts and provides technical details about our code generation strategies.

A. Problems with Serialization

Consider the pseudo OpenCL kernel code in Figure 1(a). Executing on a GPU, statement1 is executed simultaneously for a number of work-items, followed by statement2, etc. When this code is translated to C using a thread-loop or user-level threads, the resulting behavior is illustrated in Figure 1(b). While the code in Figure 1(b) captures the correctness of the program, it does not utilize hardware resources on the CPU in the most efficient way. First, the SIMD vectorization opportunities are lost because the parallelism in the SIMT execution model are not exploited. Second, the order of memory accesses that the programmer hopes for changes. For example, the programmer tends to place the stride-one access pattern across memory accesses from nearby work-items executing the same statement, in order to maximize locality. When the code in Figure 1(b) executes many statements for a single work-item before moving on to the next, this locality could be lost resulting in poor cache behavior.

B. Static Vectorization of Convergent Control Flow

Our first transformation technique improves on serialization-based approaches by using vector expressions to execute a statement for all work-items in a work-group before moving on to the next statement. This can be done whenever the compiler can determine that all work-items are convergent. The impact of this transformation is illustrated in Figure 1(c). For statement2 inside the loop, the vectorization has the effect of a complete loop-interchange because the thread-loop is removed from around the convergent loop and implicitly brought inside around the statement.

The example in the figure shows the treatment of a one-dimensional work-group for simplicity. In the case of multi-dimensional work-groups, the vectorization is performed in one dimension only while the remaining dimensions are serialized with a thread-loop. Our implementation can vectorize in any dimension. Moreover, the thread-loops of the non-vectorized dimensions are also brought inside the loop in the case of statement2 so the loop-interchange remains complete.

1) Invariance and Divergence Analysis: One drawback of the vectorization approach is that the number of variables undergoing scalar expansion increases. With thread-loops, variables having live ranges that are contained in the thread-loop region do not need to be expanded. In other words, if a variable is only ever defined and used in statements 1, 2, and 3 in Figure 1(b), multiple versions of it need not be created. However, with vectorization, these variables must be expanded to allow all the work-items to make progress simultaneously. Like MCUDA [22], we alleviate this problem via selective replication where only variables that vary across work-items are expanded, thereby reducing memory usage. We also eliminate redundant work because the non-expanded variable only needs to be computed once, whereas using thread-loops it would need to be redundantly computed in every loop iteration.

An invariance analysis [14] is used to determine which variables are variant and need to be expanded and which variables are invariant. The invariance analysis also informs the divergence analysis which classifies conditional control structures as divergent if their conditions are variant.

2) Stride Analysis and Extrapolation: The objective of stride analysis [14] is to determine whether the values of an expanded variable are at unit strides from each other. By knowing that the values have unit stride, it is sufficient to load the value for the first work-item and extrapolate the rest instead of loading all values. Such an optimization reduces memory accesses and eliminates inefficiency due to indirection in gather and scatter operations.

Consider the code example in Figure 2(a). In this code, idx is a variant unit-stride variable. After scalar expansion and vectorization, the transformed version of the second line without stride analysis and optimization is shown in Figure 2(b). However, if the stride analysis can detect that idx has constant stride, the compiler can instead generate the line of code shown in Figure 2(c). This stride-based extrapolation has two advantages. First, it reduces the number of memory accesses to the idx array to just one. Second, it removes the gather operation from the access to elements of arr and replaces it with a simple vector load which can be handled more gracefully. In practice, unit-stride memory accesses are quite common and this optimization proves to be very useful. Stride-based extrapolation can also be easily extended to any constant-stride variables as well.

C. Dynamic Vectorization of Divergent Control Flow

The vectorization approach in the previous section works for convergent control flow, when all work-items in the same work-group can be statically determined to always take the same control flow path. However, a problem

```
idx = get_local_id(0);
var = arr[idx];
```

(a)

```
idx[0:N] = ...
var[0:N] = arr[idx[0:N]];
```

(b)

```
idx[0:N] = ...
var[0:N] = arr[idx[0]:N];
```

(c)

Fig. 2. Stride-based extrapolation optimization example. The notation var[i:N] means that N consecutive values are loaded from the array var starting with that at index i. (a) Input OpenCL code snippet (b) Loading data is indirect memory access (c) Loading data is vectorized

arises in the presence of divergent control flow. The simplest way to take care of divergence is to simply serialize the work-items using a thread-loop whenever a divergent control structure is encountered.

It is often the case, however, that although convergence cannot be proven statically, work-items within a work-group do in fact take the same path frequently. For this reason, we distinguish between static and dynamic divergence. A conditional control structure is *statically convergent* if the compiler can prove it is convergent, otherwise it is *statically divergent*. A control structure is *dynamically convergent* if the work-items actually converge while executing it at runtime, otherwise it is *dynamically divergent*.

There are many situations where code can be statically divergent but dynamically convergent. Examples include boundary checks where only the boundary work-items are dynamically divergent, or loops where the number of iterations vary but all work-items are active for at least the first few iterations. In such cases, it is sub-optimal not to vectorize execution of the work-items, however it is not feasible to do so at compile time. To handle these situations, we generate code that checks if work-items dynamically converge, executing a vectorized version of the code if yes, and a serialized version otherwise. We refer to this technique as *dynamic vectorization*.

Condition blocks such as that shown in Figure 3(a) are translated as shown in Figure 3(b). First, the condition is computed and its value is stored in a predicate array (line 1). Next, the predicate array is reduced to get the total number of active work-items (line 2). If this number is equal to the number of work-items in the work-group or to zero, this means that the work-group is convergent so the then- or else-statements are allowed to execute respective vectorized code (lines 3-4 and 12-13). Otherwise, the work-group is divergent and the execution of work-items must be serialized (lines 6-10 and 15-19). Note that even in the case of serialization, the then-statements of all work-items are first executed followed by the else-statements, which better captures the would-be execution order on a GPU in case coalesced memory accesses are present.

Loop blocks such as that in Figure 4(a) would be translated as shown in Figure 4(b) if a thread-loop-based serialization were used. Loops exaggerate the problem of serialization because a work-item must complete all its iterations before the next work-item starts. Our dynamic vectorization transformation on loops restores the order of (work-item, iteration) pairs as shown in Figure 4(c). The execution goes as follows. First, the loop condition is checked for all work-items to determine the number of work-items executing the loop (lines 1-2). The loop iterates until all work-items have dropped out (line 3). Inside the loop, the body and next condition evaluation are vectorized if all work-items are active (lines 4-6) and serialized otherwise (lines 8-13). Finally, the number of active work-items in the next iteration is recomputed (line 15). Note that even in the case when loop iterations are serialized, the transformation may still be beneficial because it corrects the order of the (work-item, iteration) pairs. In other words, even when the divergent loop cannot be vectorized, it is still interchanged with the thread-loop.

D. Multi-tier Dynamic Vectorization

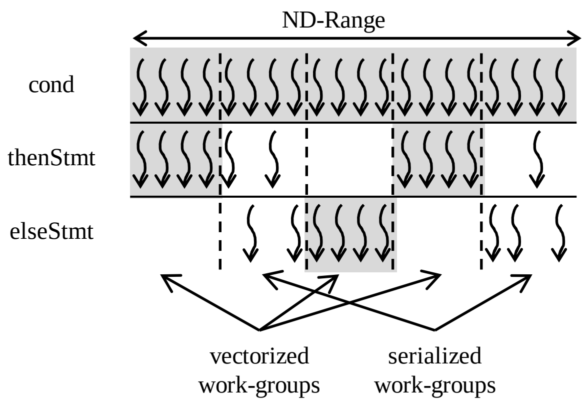
When OpenCL programmers think about avoiding control divergence, they often think about it in the context of wavefronts (commonly known as warps in CUDA terms) not work-groups. This is because no performance penalty is incurred if two work-items in the same work-group diverge as long as those work-items are not in the same wavefront. For this reason, there are some workloads where the work-items in a work-group diverge, but

```
if(cond) {
  thenStmt;
} else {
  elseStmt;
}
```

(a) Original OpenCL kernel

```
pred[0:N] = cond(0:N);
numTrue = reduce(pred[0:N]);
if(numTrue == N) {
  thenStmt(0:N);      // vectorized
} else if(numTrue > 0) {
  for(x=0; x<N; ++x){
    if(pred[x]) {
      thenStmt;       // serialized
    }
  }
}
if(numTrue == 0) {
  elseStmt(0:N);      // vectorized
} else if(numTrue < N) {
  for(x=0; x<N; ++x){
    if(!pred[x]) {
      elseStmt;       // serialized
    }
  }
}
```

(b) C kernel with dynamic vectorization



(c) Dynamically vectorized execution

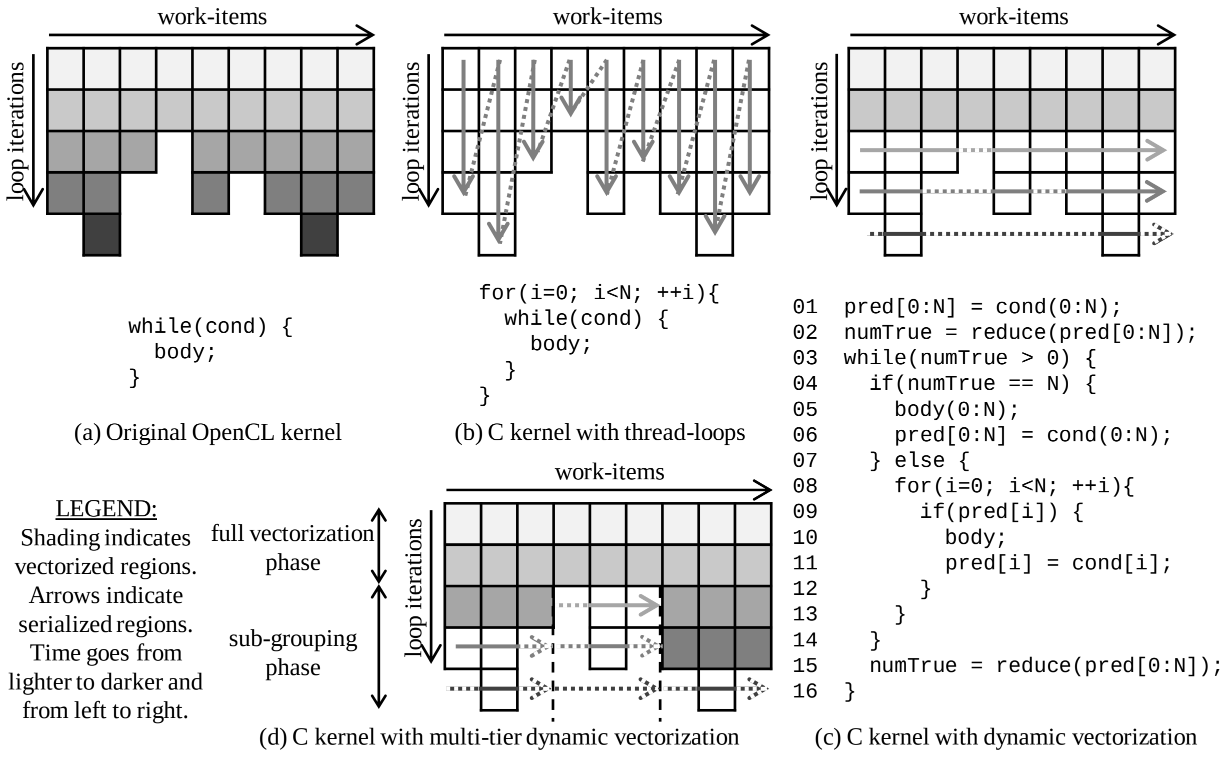
Fig. 3. Handling divergent conditionals. In this example, cond is variant, and the generated code checks for convergence at runtime and selects between vectorized and serialized code versions. The notation cond(0:N) means that the expression is evaluated for N work-items starting at work-item 0, and the result is generated into a vector. This notation will be used throughout.

sub-groups of work-items are convergent. In these situations, it is wasteful to serialize the entire work-group and forgo potential vectorization opportunities.

To address this issue, our transformation sub-groups work-items and performs the vectorization at sub-group granularity whenever a work-group is found to be dynamically divergent. The result is a multi-tiered approach that tries to fully vectorized execution of the work-group until it can no longer do so, then falls back onto vectorized execution of each sub-group, or serializing it if it can't. Figure 4(d) illustrates an example of how execution will flow for a divergent loop when multi-tier dynamic vectorization is employed.

E. Nested Control Structures

If a divergent conditional control structure is nested inside another divergent conditional control structure, the child inherits the predicate of its parent because all work-items inactive in the parent must remain inactive in the



```
while(cond) {
  body;
}
```

(a) Original OpenCL kernel

```
for(i=0; i<N; ++i){
  while(cond) {
    body;
  }
}
```

(b) C kernel with thread-loops

LEGEND:
Shading indicates vectorized regions. Arrows indicate serialized regions. Time goes from lighter to darker and from left to right.

(d) C kernel with multi-tier dynamic vectorization

```
pred[0:N] = cond(0:N);
numTrue = reduce(pred[0:N]);
while(numTrue > 0) {
  if(numTrue == N) {
    body(0:N);
    pred[0:N] = cond(0:N);
  } else {
    for(i=0; i<N; ++i){
      if(pred[i]) {
        body;
        pred[i] = cond[i];
      }
    }
  }
  numTrue = reduce(pred[0:N]);
}
```

(c) C kernel with dynamic vectorization

Fig. 4. Handling divergent loops. In this example, cond is variant, and the divergent loop is interchanged with the thread-loop. The generated code checks for convergence at runtime and selects between vectorized and serialized code versions. Notations are similar to those in the previous figure.

child. Next, the work-items that were active in the parent all evaluate the child's condition, and those which evaluate the condition to false must turn off their predicate flag. However, the newly deactivated work-items must know to become active again when the child is finished. For this reason, the parent's predicate must not be forgotten and needs to be pushed on a stack then popped when the child has terminated.

```
if(cond0) {
  if(cond1) {
    thenStmt;
  } else {
    elseStmt;
  }
}
```

(a)

```
1                p0[0:N] = cond0(0:N)
2                p1[0:N] = p0[0:N]
3  <p0>          p1[0:N] = cond1(0:N)
4  <p1>          thenStmt(0:N)
5  <p0 & !p1>    elseStmt(0:N)
```

(b)

Fig. 5. Nesting divergent control structures. In this figure, cond1 and cond2 are variant. The notation $\langle p \rangle$ means that the statement marked by that notation is dynamically vectorized based on the predicate p.

The example in Figure 5(a) shows two nested divergent conditionals. The transformed version of the code is shown in Figure 5(b). First, the child's predicate is cloned from the parent's predicate (line 2). Next, the child's condition is evaluated for only the work-items active in the parent (line 3). The code generation to dynamically vectorize the then-statement of the conditional (line 4) is done as in Figure 3 lines 2-20 using the new predicate. However, the code generation for the else-statement differs. This is because a zero value in p1 *could* mean that

Name	Description	Treatment of convergence	Treatment of divergence
TL	Modified MCUDA (thread-loop transformation)	Serialize	Serialize
SVEC	Static vectorization of convergent control flow	Vectorize	Serialize
DVEC-N	Dynamic vectorization of divergent control flow (naïve)	Vectorize	Vectorize work group if converges, else vectorize sub-group if converges, else serialize
DVEC	Dynamic vectorization of divergent control flow (transformation selection)	Vectorize	Code analysis to statically select whether to apply dynamic vectorization or to serialize

TABLE I
ABBREVIATIONS USED FOR INCREMENTAL VERSIONS OF OUR IMPLEMENTATION

the work-item evaluated the cond1 to false, but it could also mean that the work-item was not active in the parent to begin with (i.e. cond0 was false). For this reason, the parent's predicate must also be included in the dynamic convergence check, and the statement must only be executed if p1 is false and p0 is true (line 5).

F. Region Formation

In the thread-loop-based transformation in [22], the code is divided into multiple regions and each region is transformed separately. Every barrier synchronization is treated as a region boundary and every control structure containing a barrier synchronization is also treated as a region boundary. Since work-items execute an entire region before moving on to the next, this ensures that barrier synchronizations are handled correctly.

The approach we take in this paper necessitates an augmentation to this region formation algorithm. First, eliminating redundant computation of invariant variables necessitates that all work-items finish their computations before those variables are updated. Therefore invariant computations also become synchronization points in the code. Moreover, the dynamic vectorization technique necessitates that all work-items evaluate the condition of a divergent control structure before any can enter it, in order to select what version to execute. Therefore divergent control structures become synchronization points as well.

The region formation algorithm must now consider two new criteria for dividing regions: statement invariance and control structure divergence. In other words, in the new algorithm, changes in the statement invariance property must be treated as region boundaries, divergent control structures must be treated as region boundaries, and every control structure containing either of these situations must also be treated as region boundaries.

G. Transformation Selection

The performance gained from using dynamic vectorization may sometimes not be worth the housekeeping overhead incurred. This usually happens when the amount of work and memory access performed inside the control structure is not significant enough. It also happens when no convergent work-item sequences can be extracted from the divergence pattern and therefore the two vectorization tiers are never actually used. For this reason, we employ a transformation selection analysis which statically decides whether to dynamically vectorize a divergent control structure or just keep it serial. The analysis uses information such as the complexity of the control structure's body and the number of operations and memory accesses it contains to decide whether or not dynamic vectorization should be applied.

The overall decision making flow for our transformations is shown in Figure 6. Each code region formed by the region formation is transformed separately. If the region is statically convergent, it is unconditionally vectorized. If it is statically divergent, it is analyzed for complexity. If transformation selection finds that the region has a handful computation load, it will serialize the region with a thread-loop, otherwise it will emit the multi-tier dynamic vectorization code. At runtime, the multi-tier dynamic vectorization code will perform vectorized execution if the region is dynamically convergent. Otherwise, it will partition it into sub-groups and check each sub-group separately. If the sub-group is dynamically convergent, it will take vectorized execution path, otherwise its execution will be serialized with a thread-loop.

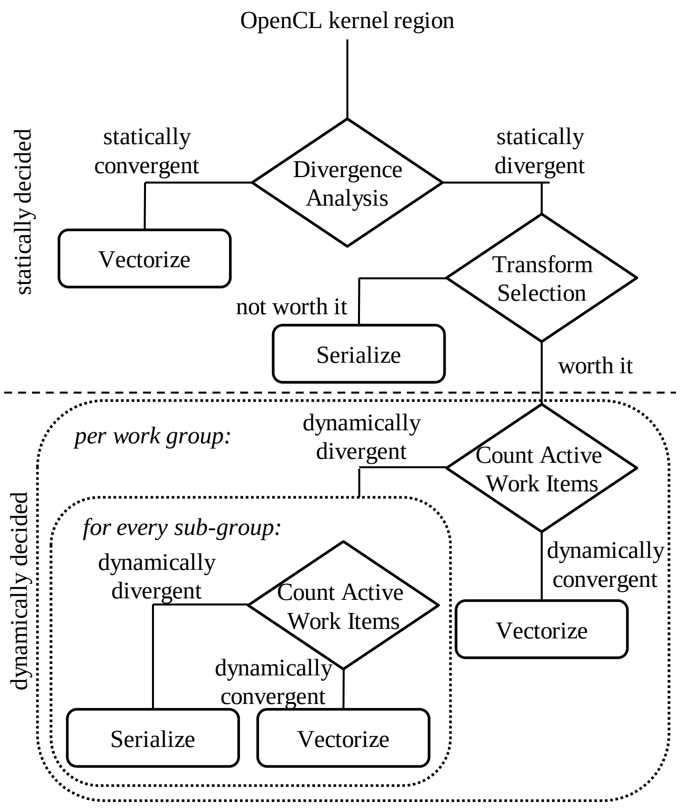


Fig. 6. Overall decision making flow for multi-tier dynamic vectorization

IV. Experimental Results and Evaluation

A. Experiment Setup

The proposed OpenCL stack is implemented as an extension of the Clang [16] compiler framework. Our tool is an AST-based source-to-source translator which transforms OpenCL code to multithreaded C code that uses vector operations. The compilation of the output code to final machine code is done offline using the Intel C Compiler (ICC) 13.1.3. C Extensions for Array Notations (CEAN) [11] is exploited as a convenient representation for communicating vector operations to the ICC.

The evaluation platform consists of an Intel i7-3820 processor running at 3.6GHz and 16G of DDR3 DRAM with dual channel configuration, running 64-bit Ubuntu 12.04. Two benchmark suites were used to evaluate the performance of each implementation: Parboil 2.5 [21] and Rodinia 2.4 [5].

We have modified MCUDA to work on OpenCL programs and use it as a reference for showing the incremental improvement of each of the techniques we apply. We then compare the performance of our final version to that of the two industry OpenCL implementations from AMD of version 1214.3 and Intel of Build 76921. Table I lists the incremental versions of our tool which we evaluate, and designates abbreviations for each that will be used from here on.

B. Incremental Evaluation of Proposed Techniques

Figure 7 compares the incremental performance impact of our OpenCL implementation starting from the modified MCUDA version (TL) for all thirty benchmarks from the Parboil [22] and Rodinia [5] suites. It shows the relative speedup of each version normalized to that of the best performing version. Similarly, Figures 8 and 9 show the dynamic instruction count and number of L1 cache misses respectively, normalized to the version with the largest value for the metric. The results are based on ten runs per benchmark per implementation.

1) Evaluation of Static Vectorization: Static vectorization of convergent regions (SVEC) shows substantial performance improvement over TL for a large number of benchmarks. Sixteen benchmarks show a speedup greater

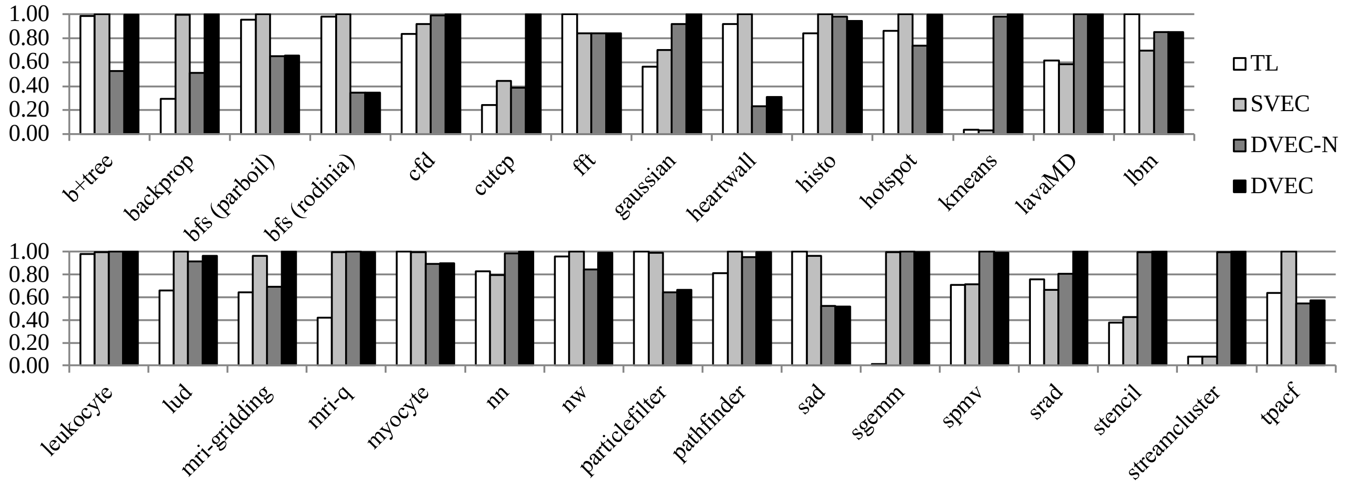


Fig. 7. Relative speedup (higher is faster) of incremental versions for Parboil and Rodinia benchmarks. Fastest is 1.0. The geometric mean speedups of each incremental version over to TL are 1.34×, 1.41×, and 1.60× for SVEC, DVEC-N, and DVEC respectively.

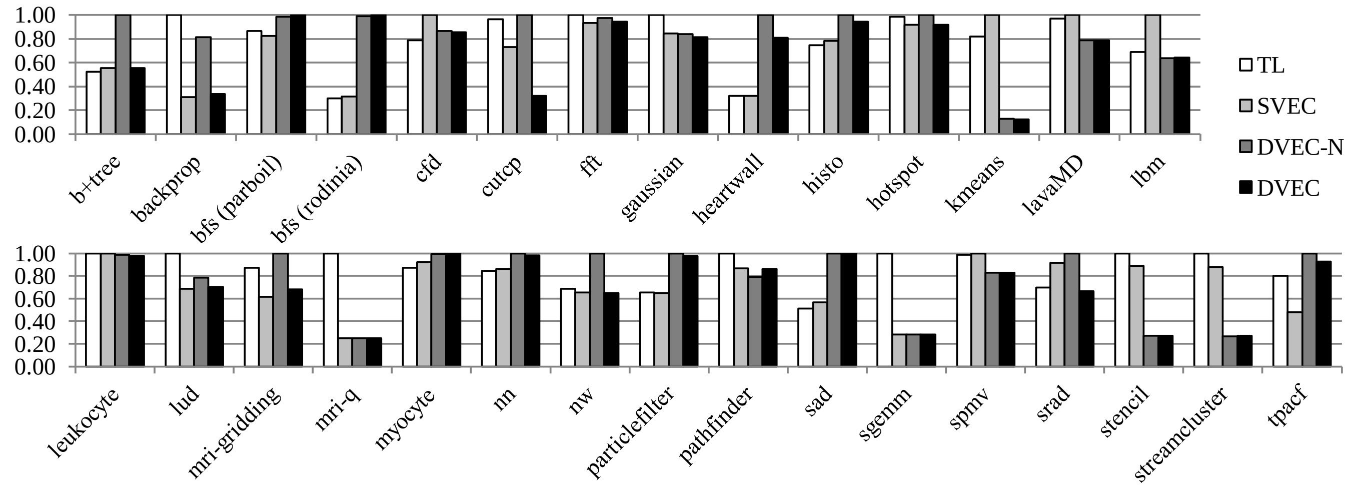


Fig. 8. Dynamic instruction count (lower is better) of incremental versions for Parboil and Rodinia benchmarks. Worst is 1.0.

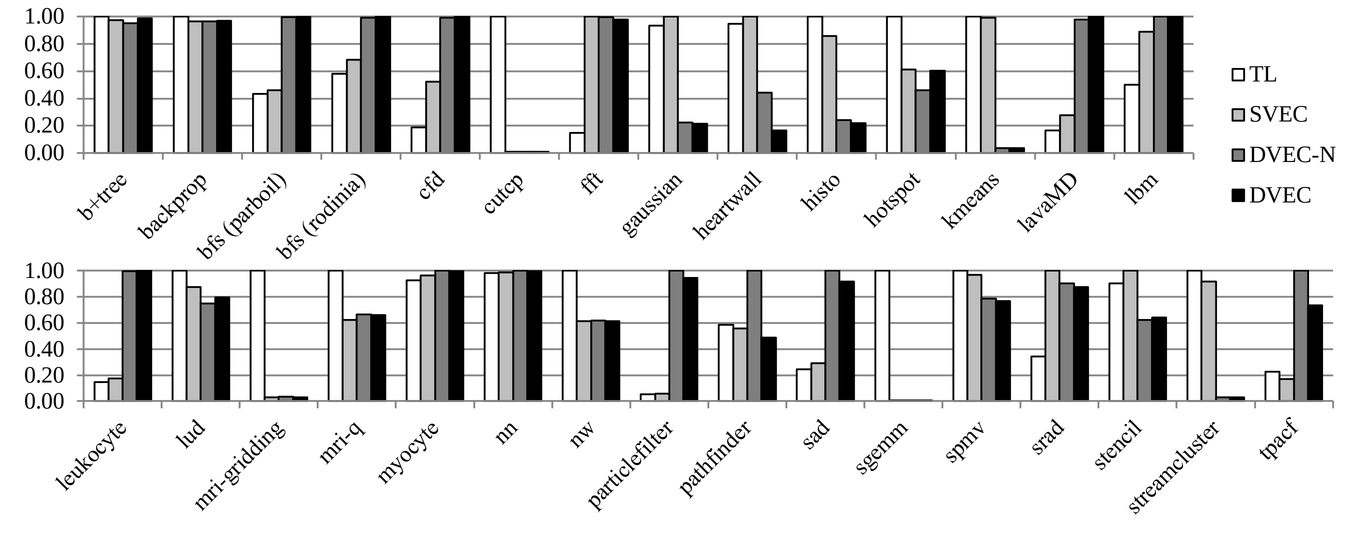


Fig. 9. L1 cache misses (lower is better) of incremental versions for Parboil and Rodinia benchmarks. Worst is 1.0.

than 3%, twelve of them being greater than 10%. Seven benchmarks are affected by less than 3%. Only seven benchmarks show slowdowns greater than 3% with just four greater than 10%. The geometric mean speedup of SVEC over TL is 1.34×.

The success of SVEC is mainly due to two factors: improved data locality and reduced dynamic instruction count due to vectorization. The graphs in Figures 8 and 9 show a clear correlation between speedup, reduced dynamic instruction count, and reduced L1 cache misses.

Out of the sixteen benchmarks showing speedup, eleven show improvement in both instruction count and L1 misses, or at least an improvement in one with little impact on the other. These benchmarks include backprop, cutcp, lud, mri-gridding, mri-q, nw, and sgemm which are all dominated by convergent control flow and most have coalesced memory accesses. They also include histo, hotspot, pathfinder, and tpacf which all have a reasonable portion of convergence and most also have coalesced memory accesses. The remaining five show some degradation in one of the metrics, but still show speedup overall. Reasons why SVEC could hurt instruction count or locality in a few cases will be detailed shortly.

The seven benchmarks that are not affected by the transformation are all dominated by divergent regions, therefore there is little difference between the code generated by TL and SVEC for these cases. These benchmarks include b+tree, bfs (Rodinia), leukocyte, myocyte, particlefilter, spmv, and streamcluster. The remaining seven benchmarks showing performance degradation all show substantial degradation in instruction count, L1 misses, or both.

There are several reasons why SVEC sometimes hurts instruction count or locality. First, SVEC's region formation divides the kernel into finer grain regions than TL because it has more criteria for creating region boundaries. Having finer regions increases the number of variables with live ranges spanning multiple regions that need scalar expansion. This is good when most regions are convergent because it enables vectorization, however when most regions are divergent (and not dynamically vectorized) this is just a waste. Benchmarks that suffer most from this phenomenon are kmeans, lavaMD, lbm, nn, sad, and srad. Second, TL performs better with benchmarks accessing data in array-of-structures (AoS) format. That is because a thread-loop accesses each element of a structure before moving on to the next structure, whereas vectorized kernels access the same element for all structures before moving on to the next element. Thus, vectorization will result in a strided access pattern which has poorer cache behavior. The benchmarks using AoS data layout are cfd, fft and lbm. We note that according to [19], [2], it is considered better practice to use structure-of-arrays (SoA) format in GPU kernels as opposed to AoS.

2) Evaluation of Dynamic Vectorization: Dynamic vectorization of divergent regions (DVEC) shows speedup over SVEC greater than 3% for twelve benchmarks, ten of them being greater than 10%. Nine benchmarks are affected by less than 1%. Nine benchmarks show slowdowns greater than 3% with seven greater than 10%. The geometric mean speedup of DVEC is 1.60× and 1.20× over TL and SVEC respectively.

The success of DVEC is also due to improved data locality and reduced dynamic instruction count. The correlation is evident in Figures 8 and 9.

Out of the twelve benchmarks showing speedup, seven show improvement in both instruction count and L1 misses. These benchmarks include cutcp, gaussian, kmeans, spmv, srad, stencil and streamcluster. The remaining five which show some degradation in the metrics but still show speedup overall are cfd, lavaMD, lbm, mri-gridding and nn. Particularly interesting cases are kmeans and streamcluster which show very high speedup. Both these cases are dominated by a convergent loop inside a boundary check which means DVEC will vectorize most of the execution successfully. Other interesting cases are lavaMD and stencil which witness a lot of benefit from the the sub-grouping vectorization tier. Reasons why DVEC could hurt instruction count or locality in a few cases will be detailed shortly.

The nine benchmarks that are not affected by DVEC include fft, mri-q, and sgemm which are all dominated by convergent regions making dynamic vectorization irrelevant. They also include b+tree, backprop, hotspot, nw, and pathfinder, all of which have divergent regions that are very small. They are hurt by naive DVEC (DVEC-N) but transformation selection successfully backs off. The remaining one (leukocyte) is anomalous because it experiences no change in instruction count, substantial increase in cache misses, and no speedup. This benchmark has a divergent region which is dominated by an arctan computation. The reason instruction count and execution time do not benefit from vectorization is that the dominant arctan computation is inside a device function which we treat by serializing. The L1 misses increase because we do more scalar expansion inside the divergent region, but do not benefit from it due to serialization. The degradation in locality has no impact on performance because it is a compute bound kernel. Device functions can easily be handled by inlining them before transformation. Our

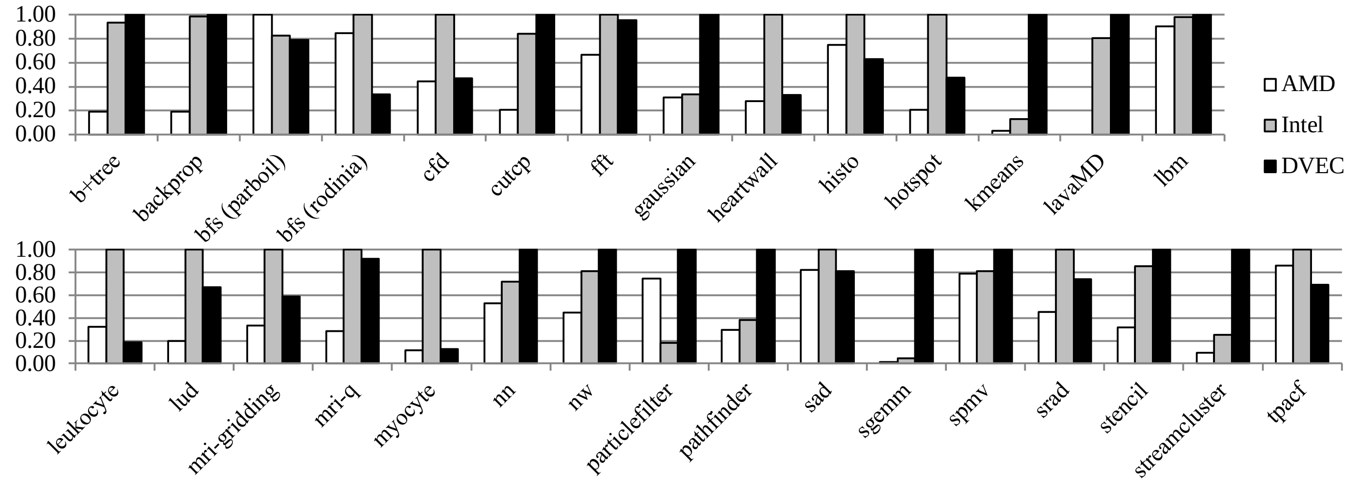


Fig. 10. Relative speedup (higher is faster) of our approach compared with AMD and Intel implementations for Parboil and Rodinia benchmarks. Fastest is 1.0. Intel has a geometric mean speedup of 2.09× over AMD. DVEC has geometric mean speedups of 2.26× and 1.09× over AMD and Intel respectively.

implementation does not currently support that, but it can easily be incorporated.

The remaining nine benchmarks showing performance degradation also show degradation in instruction count, L1 misses, or both. One of them is histo which has the same problem as leukocyte of having a device function that is serialized. As for the others, there are several reasons why DVEC could sometimes hurt instruction count or locality. First, some benchmarks are highly irregular and exhibit a great deal of dynamic control divergence. In such situations, both vectorization tiers are futile and just incur overhead. These benchmarks are bfs (Parboil) and bfs (Rodinia) in which control flow is highly data-dependent, tpacf in which each work-item executes a binary search loop and has a different trajectory, and myocyte which only has two active work-groups each with one active work-item. Second, there are a few cases where a convergent loop wrapped by divergent control flow issues consecutive memory access with respect to the loop index, using work-items higher dimension. When SVEC serializes these cases, the cache behavior is good because the serialized execution of work-item practices better spatial locality. However, when loop iterations are reordered, the contiguous access is lost and locality suffers. The benchmarks that face this problem are bfs (Parboil), bfs (Rodinia), and particlefilter.

It is noteworthy that the two patterns just mentioned are both the antitheses of the GPU best practices DVEC attempts to capture which are control convergence and memory coalescing respectively. In other words, the benchmarks in question all exhibit patterns that perform poorly on GPUs, but rather more appropriate for CPUs. It is therefore expected that DVEC performs poorly on these benchmarks when it attempts to mimic the GPU's execution model. We are reviewing analysis techniques to allow DVEC to screen away such computation patterns.

C. Comparison to Industry Implementations

Figure 10 compares our OpenCL implementation based on the DVEC approach to two widely known industry implementations of OpenCL for CPUs from AMD and Intel for the same benchmarks. It shows the relative speedup of each version normalized to that of the best performing version. The results are based on ten runs per benchmark per implementation.

DVEC outperforms AMD's Twin Peaks at twenty-three benchmarks and matches it at one. One benchmark (lavaMD) crashes using AMD's implementation so it is excluded from the comparison. The remaining five benchmarks in which AMD outperforms DVEC are bfs (Parboil), bfs (Rodinia), histo, leukocyte and tpacf. We previously mentioned that we do not handle histo and leukocyte well because our implementation does not inline device functions, not due to weaknesses in our methodology. We have also detailed in the previous section why the remaining three benchmarks perform better with serialization-based approaches (such as thread-loops or user-level threads) as opposed to vectorization-based approaches. The geometric mean speedup of DVEC over Twin Peaks for all benchmarks is 2.26×.

DVEC outperforms Intel's implementation at fifteen benchmarks and Intel outperforms DVEC at fifteen. Since we do not know what the Intel implementation does, it is difficult to make an in-depth analysis as to why certain benchmarks perform better than others for each approach. The geometric mean speedup of DVEC over Intel for all benchmarks is $1.09\times$.

V. Related Work

Several works have been done in emulating GPU's execution on CPUs. GPU Ocelot [8], Barra [6] and gpgpu-sim [3] are designed to simulate CUDA programs for NVIDIA GPU. While they closely emulate the GPU execution model, however, their greater focus is analyzing and debugging of GPU programs, not performance. MCUDA [22] is one of early approaches in translating CUDA code for CPU performance which we closely reviewed. With the advent of OpenCL, CPU vendors advanced their own implementation of OpenCL with a priority on performance, such as AMD [10] and Intel [12]'s implementations as we discussed. pocl [1] is a community effort to provide a portable compiler framework for OpenCL. Its performance and stability are yet to be evaluated.

Karrenberg et al. [14], [13] present a technique that also uses SIMD vectorization for vectorizing OpenCL code on CPUs, and handle divergent regions with software predication techniques. We highlight the differences between their work and ours due to the particular similarity between them. Their compiler vectorizes the kernel code with the size of the hardware SIMD width (W) instead of the number of work-items in a work-group. They execute an entire region for the W work-items before moving on to the next W. This approach potentially reduces the working set size, however it cannot exploit locality to the fullest in certain situations. For example, in the presence of a convergent loop with coalesced memory accesses (such as that in kmeans, particlefilter, sgemm, and streamcluster), their approach executes the entire loop for W work-items before moving on to the next, whereas ours will execute each iteration for all work-items in the work-group. In other words, we perform the loop interchange more completely bringing the entire thread-loop into the kernel loop. Moreover, in multi-dimensional cases, their approach can only bring in one dimension of the thread-loop into the convergent loop whereas ours can bring in multiple dimensions. This is beneficial for benchmarks like cutcp and sgemm. Finally, our approach handles divergence by generating multiple statically vectorized or serialized versions and selecting them dynamically, whereas their approach has one vectorized version that uses software predication. In situations with high dynamic divergence, their approach is likely more efficient, however in situations with more dynamic convergence, our approach benefits from executing statically vectorized code that escapes the drawbacks of software predication.

Handling control divergence in GPUs has drawn a lot of attention. Micro-architectural improvements over existing architectures have been proposed in works such as Fung et al. [9], Narasiman et al. [18], Brunie et al. [4], Rhu et al. [20] and Vaidya et al. [23]'s. Their main goal is to reduce the number of wasted SIMD lanes via filling them with useful computation, by migrating threads either from intra or inter warp. Compiler level solutions have also been proposed to address this problem. Diamos et al. [7] proposed thread frontier to schedule execution of basic blocks for less degree of divergence.

The idea of predication is well established in computer architecture. Hyperblock, if-conversion, and different levels of hardware support [17] are techniques to efficiently use wide execution units of VLIW architecture in the presence of control flow. Our method is similar to predication but at a courser granularity where a predicate is used for sections instead of individual operations.

VI. Conclusion

In this paper, we have presented an AST-based OpenCL implementation that efficiently executes GPU-optimized kernels on multicore CPUs. It is based on the observation that GPU optimizations aim at maximizing utilization of the memory subsystem and execution units a common optimization target for all architectures. By vectorizing the execution of GPU work-items on the CPU and preserving memory access patterns through loop-interchange, we exploit GPU optimizations to achieve better cache locality and instruction throughput on the CPU.

We use vector expressions to execute work-items in SIMD for code regions which are known statically to be convergent. For regions where convergence is not known statically, we generate multiple versions of the code and select between them at dynamically based on the convergence pattern of the work-items. If all work-items are convergent, the entire work-group is vectorized, otherwise it is partitioned into sub-groups and each sub-group is vectorized or serialized based on the convergence of the work-items it contains. Our approach demonstrates geometric mean speedups of 2.26× and 1.09× over AMD's Twin Peaks and Intel's OpenCL implementation respectively.

Our speedups are highly correlated with improvements in dynamic instruction count and L1 cache misses which demonstrates the effectiveness of our approach at maximizing resource utilization for well-behaved benchmarks. The benchmarks that perform best with our approach are the same benchmarks that abide by GPU programming best practices such as high dynamic convergence rates and coalesced memory access patterns. On the other hand, the benchmarks that perform the poorest are the same ones that violate GPU programming best practices by having high dynamic divergence rates, non-coalesced memory accesses, and data layouts which are less GPU-friendly.

REFERENCES

[1] "pocl - Portable Computing Language." [Online]. Available: http://pocl.sourceforge.net

[2] AMD, "Accelerated Parallel Processing OpenCL Programming Guide," Jul 2012.

[3] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[4] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 49–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337166

[5] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010, pp. 1–11.

[6] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," Aug. 2010, pp. 351–360. [Online]. Available: http://dx.doi.org/10.1109/mascots.2010.43

[7] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "SIMD re-convergence at thread frontiers," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 477–488. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155676

[8] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 353–364. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854318

[9] W. Fung and T. Aamodt, "Thread block compaction for efficient SIMT control flow," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 25–36.

[10] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 205–216. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854302

[11] Intel Corporation, "CEAN Language Extension and Programming Model."

[12] ——, "Intel Opencl SDK 1.2," Sep. 2013.

[13] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, 2011, pp. 141–150.

[14] ——, "Improving Performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction*, ser. CC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28652-0_1

[15] Khronos OpenCL Working Group, "The OpenCL Specification," Nov 2012.

[16] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[17] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. mei W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *IN PROCEEDINGS OF THE 22TH INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, 1995, pp. 138–150.

[18] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 308–317. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155656

[19] NVidia, "Nvidia OpenCL Best Practices Guide," 2009.

[20] M. Rhu and M. Erez, "CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 61–71. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337167

[21] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, D. Lui, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing."

[22] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Languages and Compilers for Parallel Computing," J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30.

[23] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "SIMD divergence optimization through intra-warp compaction," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 368–379. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485954