Locality-Centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures

Hee-Seok Kim[1], Izzat El Hajj[1], John Stratton[2],
Steven Lumetta[1] and Wen-Mei Hwu[1]
[1]University of Illinois at Urbana-Champaign, {kim868,elhajj2,lumetta,w-hwu}@illinois.edu
[2]Colgate University, jstratton@colgate.edu

Abstract

With heterogeneous computing on the rise, executing programs efficiently on different devices from a single source code has become increasingly important. OpenCL, having a bulk-synchronous programming model, has been proposed as a framework for writing such performance-portable programs. Execution order of work-items in a program is unconstrained except at barrier synchronization events, giving some freedom to an implementation when scheduling work-items between synchronization points.

Many OpenCL (and CUDA) compilers have been designed for targeting multicore CPU architectures. However, scheduling work-items in prior work has been done with primary focus on correctness and vectorization. To the best of our knowledge, no existing implementations consider the impact of work-item scheduling on data locality.

We propose an OpenCL compiler that performs data-locality-centric work-item scheduling. By analyzing the memory addresses accessed in loops within a kernel, our technique can make better decisions on how to schedule work-items to construct better memory access patterns, thereby improving performance. Our approach achieves geomean speedups of 3.32× over AMD's and 1.71× over Intel's implementations on Parboil and Rodinia benchmarks.

1. Introduction

Modern computing systems are transitioning to heterogeneous platforms equipped with both 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 code versions optimized for different platforms. Ideally, programmers should write their code using a single programming model, and the compiler would handle transforming the program to run optimally on the target architecture.

OpenCL [21] provides a unified interface for diverse device architectures. The abstract computing model of OpenCL is intended to be architecture neutral, enabling functional equivalence across architectures. The bulk-synchronous programming model assumes an abstract device architecture composed of multiple *compute units*, each consisting of multiple *processing elements*. The program is organized into multiple *work-items* which are grouped into *work-groups*. Work-items within a work-group execute on a single compute unit and can synchronize and share memory. Work-groups execute independently on different compute units and do not synchronize with each other. It is largely the vendor's responsibility to map the abstract computing model to the physical execution entities.

Many previous works have proposed compilation techniques for OpenCL kernels to CPUs [13, 15, 18, 23, 33, 36, 37]. All these works schedule work-groups by executing them in distinct parallel CPU threads. This scheduling approach is most intuitive because no barrier synchronization takes place between work-groups which makes them ideal units of SPMD parallelism on the CPU.

Since work-items within a work-group can synchronize with each other, they are executed within the same CPU thread to avoid the high synchronization overhead that exists across CPU threads. A kernel is typically divided into bulk-synchronous code *regions* by synchronization barriers. Scheduling work-items to execute a code region is done by wrapping a region with a *work-item loop* [15, 18, 23, 33, 36] or via user-level threads [13]. Some works strip-mine the work-item loop to benefit from the CPU's SIMD execution units. Strip-mining is done either by using explicit vector instructions [18, 33] or by annotating the loops for a later compilation phase [15].

Scheduling work-items by wrapping entire code regions with work-item loops captures program correctness and optimizes for instruction throughput in the case of strip-mining. However, existing approaches do not consider the impact of such scheduling on data locality.

In this work, we propose a code analysis and work-item scheduling technique that is locality-centric. By analyzing the memory addresses accessed in loops in a kernel, our technique can make better decisions on how to statically schedule work-items to construct better memory access patterns, thereby improving performance. The proposed scheduling technique can also be used in conjunction with vectorization to achieve higher performance than scheduling for vectorization alone.

In this paper, we make the following contributions:

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00

- We show that existing work-item scheduling techniques in the literature achieve suboptimal locality results because they do not consider the impact of work-item scheduling on the memory access pattern.
- We propose a method based on a robust compiler analysis of memory accesses to statically select a better work-item schedule in the presence of loops.
- We implement our technique in a real OpenCL compiler and evaluate it on 18 benchmarks from the Parboil [38] and Rodinia [5] benchmarks suites.
- We demonstrate that our technique selects a better schedule for all benchmarks, for improved cache and TLB performance. Overall, we show that locality-centric scheduling achieves geometric mean speedups of 3.32× over AMD's and 1.71× over Intel's OpenCL implementations based on real hardware measurements.

The rest of this paper is organized as follows. Section 2 discusses existing OpenCL and CUDA compilers for CPUs, highlighting the common trends and the distinguishing features of each. Section 3 provides the motivation for our work and details our proposed approach and its implementation. Section 4 evaluates the impact of our approach on data locality and performance, benchmarking it against existing OpenCL compilers from the industry. Finally, Section 5 outlines the related work and Section 6 concludes.

2. Background and Existing Approaches

OpenCL (and CUDA) compilers for CPUs have been proposed in both academia and industry, such as MCUDA [36, 37], SnuCL [23], pocl [15], Karrenberg's [18], AMD's Twin Peaks [13], and Intel's OpenCL compiler [33]. These compilers use different approaches to handle work-groups, work-items, and synchronization to execute OpenCL kernels on CPUs correctly and efficiently.

The OpenCL programming model is a bulk-synchronous model where multiple work-items execute the same set of kernel instructions on different sets of data. The work-items are divided into work-groups such that work-items within a work-group can synchronize using barrier instructions while work-items in different work-groups cannot. Figure 1 is a dependence graph that shows the immediate dependencies between dynamic instructions in OpenCL kernels. The graph is conservative because the arrows indicate all dependencies that *may* exist between instructions for some program. The absence of an arrow between instructions indicates independence that is guaranteed by the programming model.

One observation is that work-groups are completely independent because there is no path connecting work-items in different work-groups. This property allows all work-groups to run concurrently, independently, and in any order. All existing works handle work-groups by scheduling them in distinct CPU threads. Thus, CPU threads do not need to synchronize until kernel completion, which is convenient because synchronization across threads on a CPU is expensive.

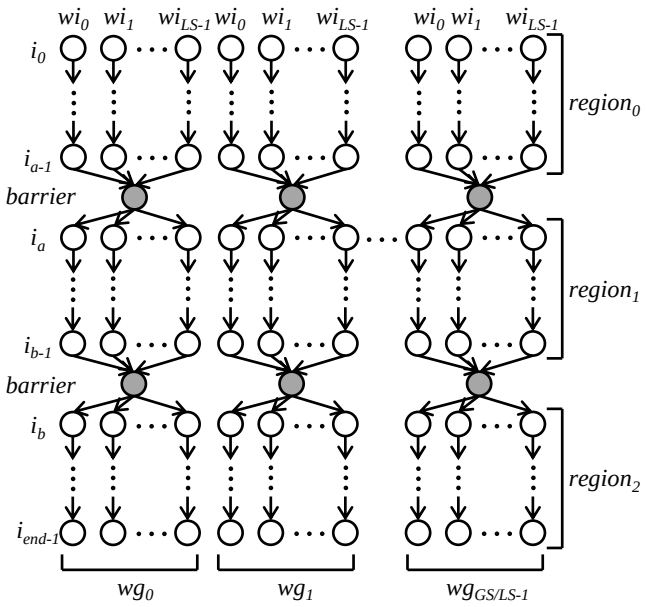


Figure 1: Immediate dependencies between dynamic instructions or instruction blocks (i) in OpenCL kernels. Each work-group (wg) contains local size (LS) work-items (wi). Work-items in different work-groups execute independently until completion. Work-items in the same work group synchronize at barriers. Barriers divide the program into code regions within which work-items execute independently. Work-item independence within regions provides great flexibility for work-item scheduling.

Another observation is that dependencies between work-items within a work-group are only introduced by barrier instructions. Therefore barriers divide the kernel into *regions* such that work-items within a region execute independently. Existing approaches employ a *region formation* algorithm to divide up the kernel into barrier-separated regions. The remaining problem to be tackled is the scheduling of work-items within a region.

AMD's Twin Peaks [13] schedules work-items with user-level threads which has the advantage of moving work-item scheduling into the runtime instead of relying on compiler techniques. The remaining approaches [15, 18, 23, 33, 36, 37] perform the scheduling at compile time via compiler-generated *work-item loops*: the compiler inserts loops around each region that iterate over all work-items in a work-group.

The advantage of using work-item loops as opposed to user-level threads is that it enables compiler optimizations, which are important for performance. One such optimization is selective replication [36]. In the presence of barriers, OpenCL variables need to be replicated for each work-item so that all work-items could run concurrently. How-

ever, replication is unnecessary for uniform variables (variables having the same value for all work-items) and variables whose lifetime is confined to a region. With work-item loops, replication is done via scalar expansion of variables into arrays. It can therefore be selectively avoided by keeping candidate variables scalar. With user-level threads, the work-item context is replicated as a whole, and therefore cannot be selectively avoided for individual variables.

Another important optimization is strip-mining of work-item loops to benefit from SIMD vectorization for maximizing instruction throughput. One approach [15] does so by annotating the work-item loops using LLVM parallel loop annotations. Other approaches [18, 33] do so explicitly using vector instructions. Situations where control divergence arises could be handled via dynamic convergence checking [40] and/or control-flow to data-flow conversion [29].

Scheduling of work-items within a region can have a large impact on data locality because it directly impacts the order of memory accesses. On GPUs, scheduling is dictated by the hardware. The GPU notion of warps (or wavefronts) enforces that a sub-group of work-items in a work-group will execute the same instruction before moving on to the next. Moreover, the warp scheduler controls the execution of work-items across warps. Since the programmer has little control over instruction scheduling, it becomes incumbent on the programmer to adapt their code and data structures to the anticipated hardware scheduling policy for better data locality. Such adaptations are the subject of many GPU optimizations such as data layout transformation, memory coalescing, and dynamic tiling [4, 39, 41].

On the other hand, the CPU hardware is not actively involved with the scheduling of work-item instructions within a region, leaving instruction scheduling up to the compiler and runtime. This allows the compiler to adapt its scheduling to the code to achieve the best memory access pattern, alleviating the programmer's burden to optimize for data locality. Prior approaches only consider correctness and instruction throughput when scheduling work-item instructions. Ours is the first approach to consider the impact of work-item scheduling on data locality.

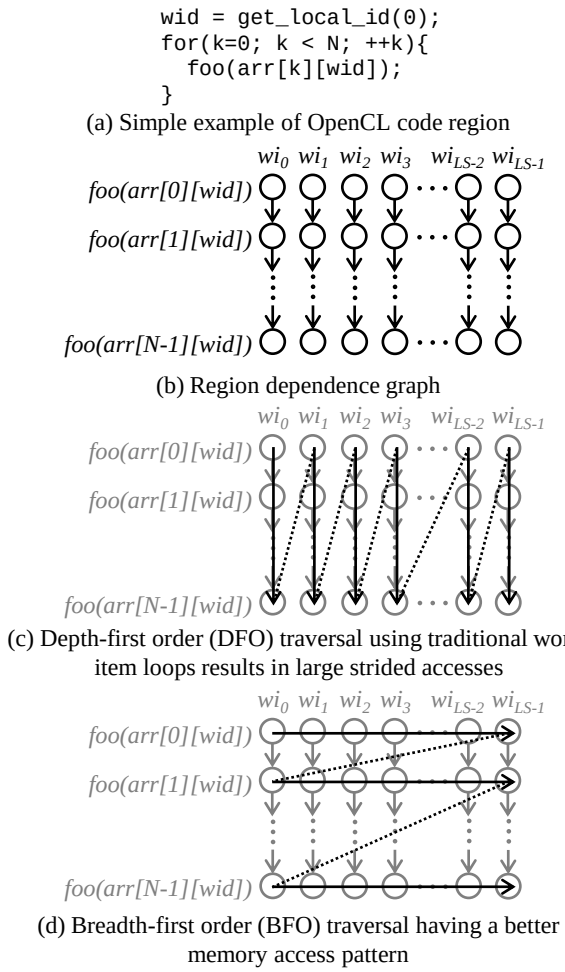
3. Proposed Approach

Traditional work-item scheduling is not always good for locality. An alternative schedule is suggested which performs better for applications having particular classes of memory accesses. A selection algorithm is proposed that picks the schedule likely to result in better locality based on a static analysis of the memory access patterns. Finally, the implementation details of the alternative schedule and the selection mechanism are discussed.

3.1 A Motivating Example

The example in Figure 2 demonstrates the effect of work-item scheduling on locality. Figure 2(b) depicts the dependence graph of the code in Figure 2(a), where each white circle represents a dynamic instruction block from a single loop iteration. If the traditional work-item scheduling is used to execute this region, each work-item will execute the region to completion before the next work-item begins, as shown in Figure 2(c). Such a traversal is clearly suboptimal because it results in a sequence of memory loads having a large stride. A better traversal of loads can be achieved by scheduling the work-items as shown in Figure 2(d). Such a traversal results in the largest number of unit stride accesses.

The technique presented in this paper focuses on regions containing loops because loops are the source of the longest running regions having working sets large enough such that locality is a major concern. Among the 30 Parboil and Rodinia benchmarks, 18 of them have loops within kernel regions. These 18 benchmarks will be used to evaluate our approach. The main question is whether to schedule a work-item to execute an entire region before the next work-item begins (the approach taken by existing compilers), or to schedule all work-items to execute the same loop iteration



(a) Simple example of OpenCL code region

(b) Region dependence graph

(c) Depth-first order (DFO) traversal using traditional work-item loops results in large strided accesses

(d) Breadth-first order (BFO) traversal having a better memory access pattern

Figure 2: A motivating example demonstrating the impact of scheduling on the memory access pattern.

		Work-item Stride		
		0 (W0)	**1** (W1)	**Other** (WX)
Loop Iteration Stride	**0** (L0)	-	DFO	DFO
	1 (L1)	BFO	-	DFO
	Other (LX)	BFO	BFO	-

Table 1: Classification of memory accesses and scheduling decision preferred by each class (if any).

before moving on to the next (the alternative approach shown in Figure 2(d)). These two scheduling techniques are denoted as *depth first order (DFO)* and *breadth first order (BFO)* respectively, based on how they traverse the dependence graph.

There is no single approach that fits all. Section 4 shows that out of 18 benchmarks, DFO does better for 5 while BFO does better for 13. DFO is well suited for capturing locality among the memory accesses within each work-item, whereas BFO will expose collective memory locality across work-items. The better schedule depends on the memory accesses dominating the loop body.

3.2 Selecting a Schedule

3.2.1 Classifying Memory Accesses

Locality-centric (LC) scheduling selects between DFO and BFO based on which technique is predicted to have better locality. The first step is to classify the memory operations inside the loop at compile time. The classification we use is summarized in Table 1. This classification is based on two dimensions: loop iteration stride and work-item stride. For each of these dimensions, memory accesses are classified as stride zero, stride one, or other.

Stride zero (i.e., invariant) means that the memory access index is the same for all loop iterations or all work-items in a work-group respectively. Stride one means that the memory access index increases by one for consecutive loop iterations or consecutive work-items respectively. Other means that the memory access index is neither invariant nor stride one. Throughout this paper, these access types are abbreviated as shown in Table 1 where 'W' means work-item, 'L' means loop iteration, '0' means stride zero, '1' means stride one, and 'X' means other.

A class of memory operations favors the schedule resulting in a smaller memory access stride. If a memory access had a smaller stride with respect to the loop index, then it is best traversed when a work-item runs deeply to finish executing a loop before the next work-item begins. Therefore the memory access prefers DFO. If a memory access had a smaller stride with respect to the work-item id, then it is best traversed when a loop iteration executes broadly across work-items before the next iteration begins. Therefore the memory access prefers BFO. Illustrations of how reuse is maximized this way are shown in Table 2.

This paper focuses on stride-zero (invariant) and stride-one memory accesses because these are most common in practice and sufficient for the proof of concept. However, the same approach can be generalized to any non-unit stride value which is left for future work. To classify the memory accesses, multiple analyses are needed such as loop-invariance analysis, loop index analysis, work-item uniformity analysis, and stride analysis. These analyses are individually solved problems in the literature [8, 14]. However, the stride analysis is augmented for more flexible classification.

3.2.2 Approximate Stride Analysis

A precise stride analysis is not necessary for making scheduling decisions because the stride information is used to inform an optimization decision that does not impact correctness. For this reason, an approximate stride analysis is used to more aggressively classify the stride. The approximate stride analysis differs from the precise stride analysis in the treatment of three operators:

- Modulus: In the statement $a = b\%N$ where b is stride one and N is arbitrary, a precise stride analysis labels a as unknown. However, a in practice is stride one for the most part. Thus, the result of a modulus on a stride one variable is approximated as stride one.
- Division: In the statement $a = b/N$ where b is stride one and N is arbitrary, the value of a in consecutive threads differs by either 0 or 1. Used as an array index, a will result in a memory pattern that is at least as good as a stride one pattern. For this reason, a can be approximated as stride one.
- Select/Phi functions: In the statement $a = (b > 0)?b : 0$ where b is stride one, the precise stride of a is unknown. However, the footprint created by a as an index is comparable to that of the worse of the two select/phi parameters (in this case, b). The decision is modeled as a semilattice (stride-zero $\rightarrow$ stride-one $\rightarrow$ unknown) such that the return value of a select or phi operator is classified according to the meet of its operands' classifications.

These three operators are quite common in OpenCL code when doing index calculations and boundary conditions, which renders the approximate stride analysis quite useful.

3.2.3 Deciding on a Schedule

Once the classification of each memory access is performed, the number of memory accesses favoring each schedule is tallied, and the schedule with a greater number of tallies is chosen. In the case of a tie, DFO is selected being the current state-of-the-art, and also to avoid the overhead of performing BFO, particularly in divergent contexts (discussed in Section 3.4).

The decision for the preferred schedule is made on a per-loop basis. Therefore, different loops in the same re-

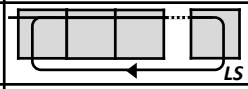
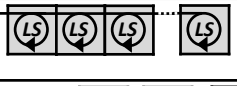
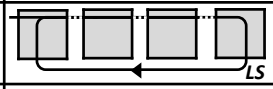
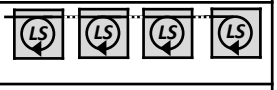
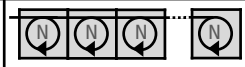
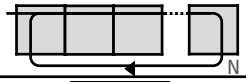
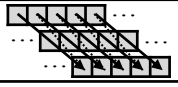
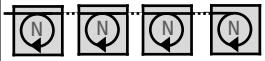
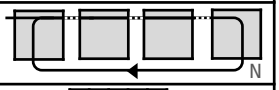
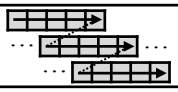
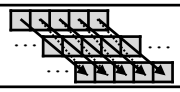
Memory Access Class	Simple Code Example	DFO Memory Access Pattern	DFO vs. BFO	BFO Memory Access Pattern
W0L1	`for(k=0; k<N; ++k) foo(arr[k]);`		BFO: better temporal locality →	
W0LX	`for(k=0; k<N; ++k) foo(arr[f(k)]);`		BFO: better temporal locality →	
W1L0	`for(k=0; k<N; ++k) foo(arr[wid]);`		← DFO: better temporal locality	
W1LX	`for(k=0; k<N; ++k) foo(arr[f(k)+wid]);`		BFO: better spatial locality →	
WXL0	`for(k=0; k<N; ++k) foo(arr[f(wid)]);`		← DFO: better temporal locality	
WXL1	`for(k=0; k<N; ++k) foo(arr[f(wid)+k]);`		← DFO: better spatial locality	

Table 2: Comparison of scheduling policies for non-neutral memory access classes. In the access pattern illustrations, shaded boxes represent a memory locations and the arrow represents the access order. A loop inside a box means the location is accessed repetitively. A loop spanning multiple boxes means that the consecutive locations are accessed sequentially then the same sequence is repeated. LS is the local size of the work-group and N is the number of loop iterations.

gion could receive different schedules. Moreover, in the case where decisions in loop nests cannot be simultaneously granted, priority is given to the inner loops. Therefore, in the case where a DFO loop contains a BFO loop, the outer DFO loop will be scheduled with BFO to enable the inner BFO loop to be scheduled correctly. The algorithm for performing this scheduling is discussed in the next section.

3.3 Implementing the Schedule

Our compiler is implemented as an AST-based transformation from OpenCL code to regular C code. The C code can then be passed to a vendor or third-party C compiler to be compiled for the underlying CPU architecture. Figure 3 shows an example of how a simple region containing a convergent loop is transformed. The OpenCL kernel in Figure 3(a) can be scheduled with DFO or BFO as shown in Figures 3(b) and 3(c) respectively. DFO scheduling is implemented by simply wrapping the entire code region with a work-item loop which is equivalent to what existing works [15, 23, 36] do. BFO scheduling is more complicated because it needs to divide the region into *subregions* and wrap each subregion with a work-item loop separately.

The pseudocode for *subregion formation* is shown in Subroutines 1 to 4. The process starts in Subroutine 1, and is divided into two phases: (1) marking subregion boundaries within a region, and (2) creating subregions between those boundaries. Subroutine 2 marks the boundary statements in a compound statement individually. The compound statement is marked as containing a boundary if one of its statements is marked as a boundary. Individual statements are marked

```
i = get_local_id(0);
if(foo()) {
  bar(i);
  while(baz()) {
    qux(i);
  }
}
```

(a) OpenCL simple code region with non-divergent loop

```
for(wid = 0 to LS-1) {
  i = wid;
  if(foo()) {
    bar(i);
    while(baz()){
      qux(i);
    }
  }
}
```

(b) DFO scheduling

```
for(wid = 0 to LS-1) {
  i[wid] = wid;
}
if(foo()) {
  for(wid = 0 to LS-1) {
    bar(i[wid]);
  }
  while(baz()){
    for(wid = 0 to LS-1) {
      qux(i[wid]);
    }
  }
}
```

(c) BFO scheduling

Figure 3: Example of scheduling a non-divergent loop nested in a non-divergent if-statement. Here, foo and baz are work-item independent.

as boundaries in Subroutine 3. A loop is marked as a boundary if it were selected for BFO scheduling, or if its body contains a boundary. An if-statement is marked as a boundary if either its then- or else-statements contain a boundary. Uniform statements are also marked as boundaries, but we omitted this detail and its associated discussion for simplicity and brevity. All other statements cannot be boundaries.

After subregion boundaries are marked, Subroutine 4 constructs the subregions between the boundaries. It starts with an empty subregion and iterates through every state-

Subroutine 1 subRegionFormation(*Region*)

```
SubRegionList = { }
markBoundaries(Region)
createSubRegions(Region, SubRegionList)
return SubRegionList
```

Subroutine 2 markBoundaries(*CompundStmt*)

```
CompundStmt.containsBoundary = false
for every statement S in CompoundStmt do
  markStmtIfBoundary(S)
  CompundStmt. containsBoundary |= S.isBoundary
```

Subroutine 3 markStmtIfBoundary(*S*)

```
switch typeof S:
  case LOOP:
    markBoundaries(S.body)
    S.isBoundary = S.isBFOLoop ∨ S.body.containsBoundary
  case IF:
    markBoundaries(S.then) ; markBoundaries(S.else)
    S.isBoundary = S.then.containsBoundary
                   ∨ S.else.containsBoundary
  default:
    S.isBoundary = false
```

Subroutine 4 createSubRegions(*CompoundStmt, SubRegionList*)

```
SubRegion = { }
for every statement S in CompoundStmt do
  if (not S.isBoundary) then
    SubRegion.add(S)
  else // if a boundary is reached
    SubRegionList.add(SubRegion)
    switch typeof S:
      case IF:
        createSubRegions(S.then, SubRegionList)
        createSubRegions(S.else, SubRegionList)
      case LOOP:
        createSubRegions(S.body, SubRegionList)
    SubRegion = { }
SubRegionList.add(SubRegion)
```

ment, adding it to the subregion until a boundary is reached. Once a boundary is reached, the subregion is completed and added to the list. The boundary is handled by processing its body recursively depending on whether it is an if-statement or loop. The subregion finally resumes after the region boundary is processed and continues creating subregions in the same manner until the end is reached.

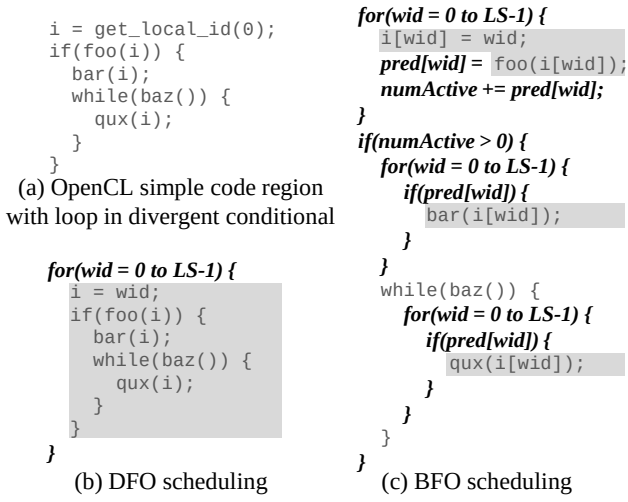
Consider the example in Figure 3(a) where the while loop is selected for BFO scheduling. In this case, the subregion formation algorithm must transform the code to that shown in Figure 3(c). In the first phase, Subroutines 2 and 3 iterate through the statements to mark subregion boundaries. The while loop is marked as a boundary because it is a BFO loop. The if-statement is marked as a boundary because its then-statement contains a boundary (namely, the loop). In the second phase, Subroutine 4 is invoked on the region. The initialization of i is made into a subregion bounded by the if-statement (which was marked as a boundary). The subregion formation is then recursively invoked on the then-statement. Inside the then-statement, bar is made into a subregion bounded by the loops (which was marked as a boundary). The subregion formation is then recursively invoked on the loop body. Inside the loop body, qux is made into a subregion terminated by the end of the loop. Finally, each subregion is wrapped with a work-item loop and non-uniform variables such as i are expanded.

One could interpret BFO scheduling as selectively introducing barrier synchronizations inside loops to force work-items to synchronize after every iteration so that they do not get ahead of each other in accessing memory. This is analogous to the dynamic tiling optimization [4] on GPUs where the programmer introduces synchronizations inside loops which are not necessary for correctness but enhance performance by preventing work-items from getting too far ahead of each other, thereby improving temporal and spatial locality.

Another way one could interpret BFO scheduling is taking the traditional DFO-scheduled code and optimizing it with a series of scalar expansions, loop distributions, and loop interchanges. However, there are multiple reasons why it is not always feasible to pass DFO-scheduled code to another compiler for automatic transformation into BFO code. First, a traditional compiler attempting to perform such an optimization would have to first conservatively prove that the loops are interchangeable. However, it cannot always be determined that there are no loop-carried dependencies across work-item loop iterations, especially when indirect references obfuscate the loop-dependence analysis. On the other hand, a compiler with direct access to the OpenCL kernel has that guarantee from the programming model, so it can make stronger assumptions without complicated loop-dependence analyses. Second, the presence of control divergence makes a simple loop interchange infeasible and requires much more complex transformations. For these reasons, BFO scheduling can much more effectively be performed when work-item loops are inserted, rather than being outsourced to loop-manipulating optimization passes by an underlying compiler.

3.4 Handling Control Divergence

Control divergence arises when not all work-items take the same execution path. In a schedule which only uses DFO, this is not an issue. Region boundaries are by definition points of synchronization in the program. Since all work-items must be active at synchronization points, it is safe to assume that work-items are always convergent at the entry and exit points of a region. For this reason, a loop over all work-items can be inserted around the entire region without any concern about some work-items not being active.

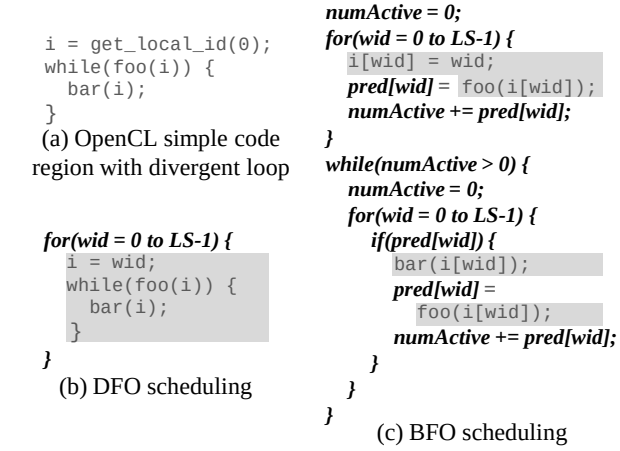


(a) OpenCL simple code region with loop in divergent conditional

(b) DFO scheduling

(c) BFO scheduling

Figure 4: Example of scheduling a non-divergent loop in a divergent context using predicated work-item loops.

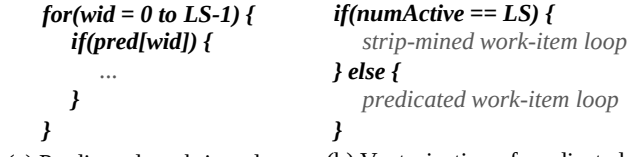


(a) OpenCL simple code region with divergent loop

(b) DFO scheduling

(c) BFO scheduling

Figure 5: Example of scheduling a divergent loop.



(a) Predicated work-item loop

(b) Vectorization of predicated work-item loop

Figure 6: Vectorization based on runtime convergence checking.

On the other hand, not all work-items are guaranteed to be active at the entry and exit points of a subregion because a subregion could be within the body of a divergent conditional or loop. Therefore, wrapping subregions with a work-item loop is not sufficient for BFO scheduling. Instead, control divergence is handled by introducing a predicate array that tracks which work-items are active. Before the subregion is executed for a particular work-item, the predicate array must be checked for whether the work item is active. The combination of the work-item loop with the predicate check is denoted as a *predicated work-item loop*.

Control divergence can be introduced whenever there is work-item dependent control flow due to conditionals or loops. Figure 4(a) illustrates the case where a loop is guarded with a divergent conditional. DFO scheduling is done by simply wrapping the entire region with a work-item loop as shown in Figure 4(b). However, to perform BFO scheduling, the condition evaluation must be stored and used for executing the subregions inside the conditional via predicated work-item loops as shown in Figure 4(c).

Figure 5(a) illustrates the case where the loop itself is divergent because the loop condition is dependent on the work-item id. The DFO code still follows the same strategy as shown in Figure 5(b). The BFO code is as shown in Figure 5(c). In addition to storing a predicate array and using predicated work-item loops to wrap subregions, the total number of active work-items is also maintained at all iterations to know when the loop must terminate.

Predicated work-item loops are difficult to vectorize because of the loop-dependent conditional surrounding the body of the loop. For this reason, our tool statically generates two versions of the code and selects between them dynamically based on a runtime divergence check. The first version uses a regular strip-mined work-item loop that is selected when all work-items in the work-group are active. The second version is a serial predicated work-item loop that we fall back on when not all work-items are active. The resulting code is shown in Figure 6. A similar technique is employed in [11, 20, 22, 40]. In some cases, vectorization can be improved using control flow to data flow conversion techniques such as those employed in [18] and [33]. However, vectorization is not the main focus of this paper.

4. Evaluation

The performance of the proposed compiler with locality-centric scheduling is evaluated in this section. We show that locality-centric scheduling is able to consistently select the schedule having fewer data cache misses. We also compare our implementation with other OpenCL implementations from the industry to demonstrate the overall performance of our technique.

4.1 Experimental Setup

The proposed compilation approach is implemented as an extension of the Clang compiler framework. An AST-level source-to-source translator takes OpenCL code and emits C code. Vectorization is performed by annotating loops with SIMD pragmas. The final machine binary is assembled using the Intel C Compiler (ICC) of version 14.0.1. The same compiler is used for building all benchmarks.

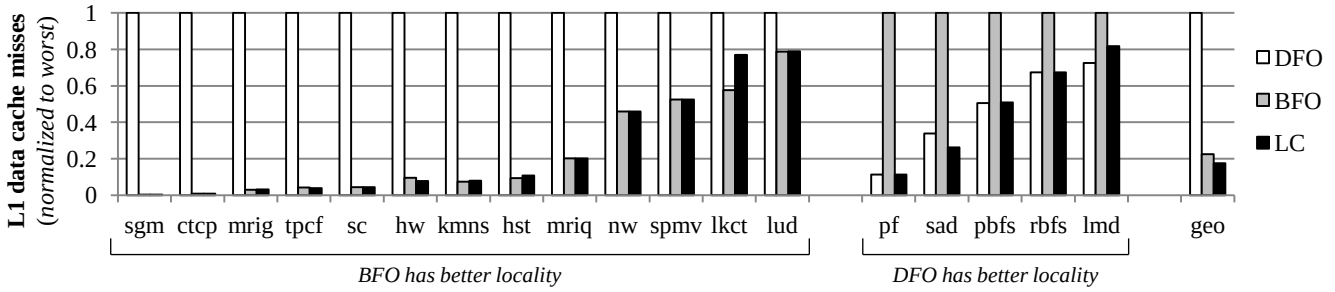


Figure 7: Locality comparison of DFO, BFO, and locality-centric (LC) scheduling. Results are normalized to the worst performing schedule. LC has geomean reduction in L1 data cache misses of 5.72× and 1.29× over DFO and BFO respectively.

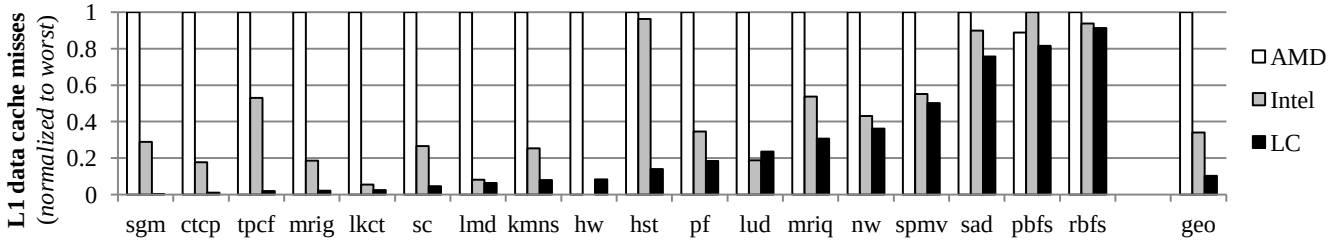


Figure 8: Locality comparison of AMD, Intel, and LC compilation approaches. Results are normalized to the worst performing tool. LC has geomean reduction in L1 data cache misses of 9.81× and 3.35× over AMD and Intel respectively.

The evaluation platform consists of an Intel i7-3820 processor running at 3.6GHz, having 4 cores with hyperthreading enabled. The memory hierarchy includes 32KB L1 private data caches, 10MB shared last-level cache, and 16GB of DDR3 DRAM with dual channel configuration. The system is running 64-bit Debian Jessie distribution. A PMU-based performance monitoring library, perfmon2 [12], is used for collecting performance counters throughout.

The industry implementations we compare against are AMD's [13] and Intel's [33] OpenCL compilers. The driver versions used are 1445.5 and 1.2.0.8 respectively, which are the latest versions at the time this paper was submitted.

Throughout the experiments section, data is normalized against the approach scoring highest for the metric under study as opposed to a common baseline. The reason for doing so is that if a single baseline is taken, the values for locality and speedup could span three to four orders of magnitude (0.01× to 10×) which is difficult to plot on a single axis. This normalization methodology seemed to make the graph more readable and make better use of the space than log plots.

4.2 Benchmarks

Eighteen benchmarks from the Parboil [38] and Rodinia [5] benchmark suites were selected for evaluation. The benchmarks selected are those having loops which are completely contained within a code region such that our technique is applicable. The remaining benchmarks are not relevant because they either do not contain loops within regions, or the loops have short constant trip counts such that they disappear after unrolling.

Benchmark	Abbreviation	Benchmark	Abbreviation
cutcp	ctcp	nw	nw
heartwall	hw	parboil's bfs	pbfs
histo	hst	particlefilter	pf
kmeans	kmns	rodinia's bfs	rbfs
lavaMD	lmd	sad	sad
leukocyte	lkct	sgemm	sgm
lud	lud	spmv	spmv
mri-gridding	mrig	streamcluster	sc
mri-q	mriq	tpacf	tpcf

Table 3: Evaluated benchmarks from Parboil and Rodinia benchmark suites with abbreviations used.

Table 3 lists the benchmarks evaluated and the abbreviations used throughout this section for each. Each benchmark is executed ten times for evaluation of the average execution time and associated performance counters. Three benchmarks (histo, leukocyte, and mri-gridding) have device functions in the dominant loops. These functions are manually inlined to focus the comparison with AMD and Intel on locality, since their compilers seem to inline device functions while our framework does not currently support that.

4.3 Impact of Scheduling on Locality

Figure 7 compares the number of L1 data cache misses (lower is better) of DFO, BFO, and LC scheduling. The values for each benchmark are normalized to the policy having the highest (worst) number of misses. The benchmarks are categorized according to the schedule (DFO or BFO) having

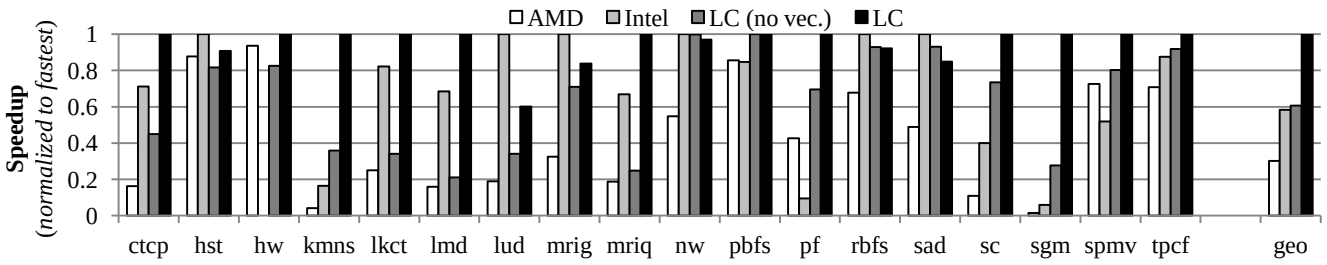


Figure 9: Performance comparison with AMD and Intel. Results are normalized to the faster tool. LC achieves a geomean speedup of 3.32× and 1.71× over AMD and Intel respectively.

better performance and sorted in increasing order of LC's relative performance.

The graph shows that 13 benchmarks have better locality with BFO scheduling and 5 are better with DFO. Moreover, it shows that LC scheduling consistently selects the correct schedule, achieving geomean reductions in L1 data cache misses of 5.72× and 1.29× over DFO and BFO respectively.

The conclusions drawn from this experiment are:

- Current state-of-the-art work-item scheduling techniques (i.e., DFO) yields suboptimal data locality behavior.
- A single scheduling technique (whether DFO or BFO) will not always result in the best locality, thereby necessitating that scheduling be locality-aware.
- Our locality-centric scheduling is successful at choosing the schedule resulting in better locality.

4.4 Locality Comparison with Industry Implementations

Figure 8 compares the number of L1 data cache misses (lower is better) of AMD, Intel, and LC. The values for each benchmark are normalized to the approach having the highest (worst) number of misses. The benchmarks are sorted in increasing order of LC's relative performance.

The graph shows that our locality-centric scheduling achieves locality results which are consistently better or as good as that of AMD's and Intel's implementations. The missing hw datapoint for Intel is because Intel's compiler crashed when compiling this benchmark. LC scheduling was able to achieve geomean reductions in L1 data cache misses of 9.81× over AMD and 3.35× over Intel.

The conclusions drawn from this experiment are:

- Industry implementations of current state-of-the-art work-item scheduling yield suboptimal data locality behavior.
- Our locality-centric scheduling achieves better data locality on average than current industry implementations.

On a side note, we observe that AMD's locality results are significantly worse than Intel's and LC's, even for cases where DFO is better for locality (pf is a clear example of this behavior). This is due to the overhead of replicating variables for all work-items even when variables are uniform. This fundamental limitation in AMD's user-level threads technique has already been discussed in Section 2. Another note is that when LC selects DFO, it performs slightly better than Intel in many cases (again, pf is a clear example). That is because the schedule Intel uses is not a pure DFO (as will be mentioned in Section 5). Intel uses narrow vectors of work-items to execute a region which can be seen as a hybrid between DFO and BFO skewed much towards DFO.

4.5 Performance Comparison with Industry Implementations

Figure 9 compares the relative performance (inverse of time, higher is better) of AMD, Intel, and LC. Since AMD does not seem to vectorize across work-items while Intel does, we include performance results for a vectorized and non-vectorized version of LC to isolate the impact of locality for fair comparison with both. The values for each benchmark are normalized to the best performing tool. The benchmarks are sorted in alphabetical order.

The graph shows that our OpenCL implementation for CPUs with locality-centric scheduling achieves significant speedups over AMD and Intel. LC outperforms AMD in most benchmarks with a geomean speedup of 3.32×. Non-vectorized LC still outperforms AMD with geomean speedup 2.01×, despite the fact that it only achieves geomean 0.80× the number of instructions. This result further confirms that LC's improvement over AMD is due to the improved locality, and not just vectorization or instruction efficiency.

In comparison to Intel, LC achieves a geomean speedup of 1.71×. As for instruction count, LC achieves a geomean of 0.98× the number of instructions of Intel. This result demonstrates that LC's improvement over Intel is due to locality, not instruction efficiency. Moreover, LC with vectorization turned off is still able to match Intel's implementation with vectorization turned on, achieving a geomean speedup of 1.04×. This result reflects the importance of considering locality, and not just instruction throughput, when scheduling work-items.

Table 4 summarizes the comparison between our approach and the other industry implementations for L1 data cache misses, speedup, and other metrics.

The conclusion drawn from this experiment is:

Metric	LC/AMD	LC/Intel
Speedup	3.32x	1.71x
L1 Data Cache Misses	0.10x	0.30x
Data TLB Misses	0.26x	0.33x
LLC Misses	0.92x	0.77x

Table 4: Geomeans summarizing the comparison of locality-centric scheduling with industry implementations.

- Our OpenCL implementation with locality-centric scheduling meets industry performance standards and outperforms state-of-the-art industry implementations.

5. Related Work

Compilers such as MCUDA [36, 37], SnuCL [23] and Jo et al.'s work [16] use scalar work-item loops to serialize execution of work-items in a work-group. The technique is called *work-item coalescing* by Lee et al. [25] and it is the same as the DFO scheduling discussed in this paper. AMD's implementation [13] employs user-level threads for serializing work-items which has an effect similar to scalar work-item loops, hence DFO scheduling. The disadvantage of AMD's approach is that it does not allow compile-time optimizations across work-items such as scalarization of uniform variables. Intel [33], Karrenberg et al. [17, 18], and Kerr et al. (for CUDA) [20] use explicitly strip-mined work-item loops, emitting vectorized code to exploit SIMD instructions when executing adjacent work-items. On the other hand, pocl [15] uses annotated work-item loops, outsourcing the strip-mining to an existing loop vectorizer. Strip-mined work-item loops achieve a memory access ordering that has slightly more breadth than a pure DFO schedule (as much as the vector width), but not nearly as much breadth as a BFO schedule for medium and larger work-group sizes. Our work is the first to: (1) implement and demonstrate the advantage of BFO scheduling, and (2) use a selective rather than fixed schedule to optimize for locality.

The quality of vectorization is largely degraded in the presence of control divergence because predication techniques [17, 18, 29, 33, 35] need to be employed. Kerr et al. [20] and Timnat et al. [40] address this problem by branching to predication-free code when all work-items are active at runtime, and falling back on predicated or serialized code otherwise. Selecting between multiple code versions based on a runtime divergence check is parallel to our work on vectorizing BFO scheduled code. Implicit vectorization using loop annotations and a third party vectorizer is adopted by pocl [15] and also studied in Jo et al.[16]. Although our approach also uses loop annotations to vectorize, it differs in that the dynamic divergence check results in loops which are more easy to vectorize by the underlying compiler.

Ocelot [10] and Kerr's work [20] execute CUDA kernels on CPUs and other architectures as well. Lee et al. [25] propose an OpenCL platform targeting Cell, with particular interest in data memory management and coherency. The pocl implementation [15] has support for ARM, MIPS and other architectures as it relies on LLVM for target code generation. All of these works either implement work-item loops (with vectorization for some) or rely on a third party OpenCL implementation such as AMD and Intel in order to execute work-items on scalar computing units. Therefore, they are bound to a fixed scheduling policy (DFO in practice).

Seo et al. [34] have proposed a profiling-based approach for choosing work-group sizes in pursuit of working set and load balancing optimizations. Their approach shows that nesting of multi-dimensional work-item loops can be shuffled for locality. However, it does not consider optimizing kernel loops and work-item loops as a whole which is what locality-centric scheduling does. Lee et al. [26] have provided a performance analysis of CPU OpenCL stacks in varying work-group sizes, thread coarsening factors, and memory access patterns. Shen et al. [35] have compared performance between CPU and GPU OpenCL stacks from using a same source code. However, these works do not identify nor analyze the performance implications of work-item scheduling. An interesting observation in [10] as well as [35] is that altered memory access pattern compared to a corresponding GPU execution degrades performance significantly on CPUs. This is largely due to the suboptimally placed work-item loops and subsequent memory access ordering by their respective implementations, which does not reveal any concern for locality and motivates our approach. Our approach generates locality friendly code so that data layout transformation as suggested in [35] is not required. Instead of changing the data layout to match the execution order, we change the execution order to match the data layout so that the same data structures can be used for both the CPUs and GPUs alike.

Lee et al. [27] have proposed a compiler technique for exploiting scalar units in recent GPU architectures. A similar goal was sought by Collange et al. in [7] where microarchitectural modification was added for detecting uniform and affine vectors at runtime. Magni et al. [28] evaluate the performance of thread coarsening with various parameters, and consider redundancy elimination across coarsened threads an important optimization. Uniform variable analysis [17, 36] and divergence analysis [8] serve similar goals in most OpenCL compilers [15, 17, 33, 36] including ours.

There is a substantial body of literature on loop analyses and optimizations [19, 30]. Polyhedral-model-based approaches [3, 24, 32] have been shown to yield highly optimized loop nests via tiling of affine loop iteration space, although they typically require affine loops and array subscripts on the input code. Theoretically, BFO code can be derived from DFO code through a series of scalar expan-

sions, loop distributions, and selective interchanges of work-item loops and kernel loops. However, this is difficult and sometimes infeasible in practice for the reasons mentioned in Section 3.3. For this reason, BFO-scheduled loops are more effectively generated from OpenCL kernels directly as proposed in this paper.

We do not provide performance comparisons with other CPU parallel programming models such as OpenMP [9] and TBB [31]. Comparisons between OpenCL for CPUs to such models can be found in the literature [1, 26, 35]. These studies found that industry OpenCL compilers (which we outperform) achieve comparable performance to OpenMP and TBB. The comparison of our particular compiler is the subject of future work.

GPU simulators [2, 6] have played a pioneering role in GPU research. They execute OpenCL or CUDA code on CPUs for the purpose of modeling and design studies. However, their goal is not high-performance execution.

6. Conclusion

In this work, we show that the state-of-the-art (depth-first) approach to scheduling work-items in existing OpenCL compilers for CPUs can result in suboptimal memory access patterns for certain workload classes. We present a second (breadth-first) work-item schedule and propose a static analysis and transformation which correctly selects and generates the schedule resulting in better data locality. Our locality centric scheduling results in geomean L1 data cache miss reductions of 9.81× over AMD and 3.35× over Intel, and geomean speedups of 3.32× over AMD and 1.71× over Intel, based on on real hardware measurements. As the memory system becomes increasingly important for performance and energy efficiency in future computing systems, the appropriate selection of work-item schedules will play an even more important role in the future.

Acknowledgments

We thank the entire IMPACT Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is partly supported by the Starnet Center for Future Architecture Research (C-FAR), the Defense Advanced Research Projects Agency (484905-239012-191100), the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), the UIUC CUDA Center of Excellence. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

References

[1] A. Ali, U. Dastgeer, and C. Kessler. Opencl for programming shared memory multicore cpus. In *Proceedings of the 5th Workshop on MULTIPROG*, 2012.

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

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[4] L. Chang, J. A. Stratton, H. Kim, and W. W. Hwu. A Scalable, Numerically Stable, High-performance Tridiagonal Solver Using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 27:1–27:11, 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, IEEE International Symposium on*, pages 44–54, 2009.

[6] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, IEEE International Symposium on*, pages 351–360, Aug. 2010.

[7] S. Collange, D. Defour, and Y. Zhang. Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 46–55, 2010.

[8] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *Parallel Architectures and Compilation Techniques, 2011 International Conference on*, pages 320–329, Oct. 2011.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[10] G. F. Diamos, N. Clark, A. R. Kerr, and S. Yalamanchili. 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*, pages 353–364, 2010.

[11] I. El Hajj. Dynamic loop vectorization for executing OpenCL kernels on CPUs (Master's thesis). 2014.

[12] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer, 2006.

[13] 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*, pages 205–216, 2010.

[14] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, July 2005.

[15] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable OpenCL

implementation, 2014.

[16] G. Jo, W. Jeon, W. Jung, G. Taft, and J. Lee. OpenCL Framework for ARM Processors with NEON Support. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, pages 33–40, 2014.

[17] 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, Apr. 2011.

[18] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction*, pages 1–20, 2012.

[19] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[20] A. Kerr, G. Diamos, and S. Yalamanchili. Dynamic Compilation of Data-parallel Kernels for Vector Processors. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 23–32, 2012.

[21] Khronos OpenCL Working Group and others. The OpenCL Specification. *A. Munshi, Ed*, 2008.

[22] H.-S. Kim, I. El Hajj, J. A. Stratton, and W.-M. W. Hwu. Multi-tier Dynamic Vectorization for Translating GPU Optimizations into CPU Performance. *IMPACT Technical Report*, 2014.

[23] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 341–352, 2012.

[24] M. Kong, R. Veras, K. Stock, F. Franchetti, L. Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 6, pages 127–138, June 2013.

[25] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. Dao, Y. Cho, S. Seo, S. Lee, S. Cho, H. Song, S. Suh, and J. Choi. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 193–204, 2010.

[26] J. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim. OpenCL Performance Evaluation on Modern Multi Core CPUs. In *Parallel and Distributed Processing Symposium Workshops PhD Forum, 2013 IEEE 27th International*, pages 1177–1185, May 2013.

[27] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic. Convergence and scalarization for data-parallel architectures. In *Code Generation and Optimization, 2013 IEEE/ACM International Symposium on*, pages 1–11, Feb 2013.

[28] A. Magni, C. Dubach, and M. F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 11:1–11:11, 2013.

[29] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 138–149, June 1995.

[30] K. S. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[31] C. Pheatt. Intel Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298, Apr. 2008.

[32] B. L. Prism, V. S. Quentin, V. Cedex, and C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, 2004.

[33] N. Rotem. Intel OpenCL Implicit Vectorization Module, 2011.

[34] S. Seo, J. Lee, G. Jo, and J. Lee. Automatic OpenCL work-group size selection for multicore CPUs. In *Parallel Architectures and Compilation Techniques, 2013 22nd International Conference on*, pages 387–397, Sept 2013.

[35] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Traps in OpenCL for CPUs. In *Parallel, Distributed and Network-Based Processing, 2013 21st Euromicro International Conference on*, pages 38–45, Feb. 2013.

[36] J. A. Stratton, S. S. Stone, and W. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In J. N. Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30. 2008.

[37] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 111–119, 2010.

[38] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *IMPACT Technical Report*, 2012.

[39] I. Sung, J. A. Stratton, and W. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 513–522, 2010.

[40] S. Timnat, O. Shacham, and A. Zaks. Predicate Vectors If You Must. In *WPMVP '14: Workshop on Programming Models for SIMD/Vector Processing*, 2014.

[41] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 86–97, 2010.