DYNAMIC LOOP VECTORIZATION FOR EXECUTING OPENCL KERNELS ON CPUS

BY

IZZAT EL HAJJ

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Wen-Mei Hwu

ABSTRACT

Heterogeneous computing platforms are becoming increasingly important in supercomputing. Many systems now integrate CPUs and GPUs cooperating together on a single node. Much effort is invested in tuning GPU-kernels. However, it can be the case that some systems may not have GPUs or the GPUs are busy. Maintaining two versions of the same code for GPUs and CPUs is expensive. For this reason, it would be ideal if one could retarget GPU-optimized kernels to run efficiently on a CPU.

Many efforts have been made to compile OpenCL kernels to run efficiently on CPUs. Such approaches typically involve running work-groups in parallel on different CPU threads, and executing work-items within a work-group in one thread serially via loop-based serialization or in parallel via SIMD vectorization. SIMD vectorization is particularly difficult where control divergence is present. This thesis proposes a technique for transforming divergent loops in OpenCL kernels such that vectorization opportunities can be extracted when possible and memory access patterns can be improved. The transformations presented show promising speedups for kernels that follow GPU programming best practices, and slowdowns for kernels that do not.

To Mama, Baba, Hind, Hammoodi, and Mona
for their unconditional love and support

ACKNOWLEDGMENTS

All praise is due to God.

I would like to thank my advisor, Professor Wen-Mei Hwu, for his guidance and support. He has always been a source of wisdom and insight when things got difficult. His high spirit, dedication, and passion for what he does have been an inspiration for me to become a better person.

I would like to thank the IMPACT research group for all their help and companionship. I would especially like to thank Hee-Seok Kim for his mentorship and patience, and Marie-Pierre Lassiva-Moulin for always creating a positive environment.

I would like to thank my parents for their unconditional love and support, for their dedication to help me succeed, and for being such great role models. My gratitude to them is beyond what I could express in words.

I would like to thank my sister Hind for being a great teacher and best-friend, and my brother Mohammad for being himself. I would like to thank my wife Mona for all the sacrifices she has made and the care she has provided. Finally, I would like to thank my roommates Nabil Hirzallah and Mohammad Usama Zahid for their brotherhood, and Dr. Ahmed Taha for being a father away from home.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AoS	Array-of-Structures
CUDA	Compute Unified Device Architecture
DVEC	Dynamic Vectorization
DVEC-S	Dynamic Sub-Vectorization
GPU	Graphics Precessing Unit
MxPA	Multicore Cross-Platform Architecture
OpenCL	Open Computing Language
SIMD	Single Instruction Multiple Data
SoA	Structure-of-Arrays
SVEC	Static Vectorization (referring to MxPA's vectorization technique)

CHAPTER 1

INTRODUCTION

Computing systems are moving toward heterogeneity with accelerators such as Graphics Processing Units (GPUs) and Intel Xeon-Phi's playing an increasingly important role in supercomputing. Heterogeneity provides the advantage of specialization where each device has its own class of computing patterns that it performs best, and the programmer orchestrates matching computing tasks to various kinds of devices.

Because devices in a heterogeneous system have different properties, they require different programming models to best utilize their characteristics and maximize their performance. For example, a program running on a heterogeneous platform with CPUs and GPUs working together is typically composed of a heterogeneous codebase such as a mixture of C/C++ modules targeting the CPUs and OpenCL or CUDA modules targeting the GPUs.

The problem with such an arrangement is that a computation pattern becomes bound to a particular hardware device. For example, if a program contains matrix multiplication code in OpenCL to run on the GPU, it will be slowed down or run into trouble if the GPU is busy or if the code needs to be run on a system without GPUs. One solution for this problem is to maintain two versions of the module in the codebase: one C/C++ version for the CPU and another OpenCL/CUDA version for the GPU. However, code maintenance is an expensive operation and ideally one would want to maintain a single version only. For this reason, it would be desirable if we could compile to multiple platforms from a single piece of source code and still obtain reasonable performance.

The work presented in this thesis is part of an OpenCL compiler and runtime system that aims to retarget GPU-optimized OpenCL kernels to CPUs. Various attempts have been made to compile OpenCL kernels for CPUs. Work-groups are commonly distributed among various CPU threads and run in parallel which is enabled by the fact that they are independent

and do not synchronize. However, the different approaches have dealt with work-items within a single work-group in different ways such as user-level threads, loop serialization, or SIMD vectorization. The approach that is built upon in this thesis adopts the vectorization approach.

Work-items in a work-group can easily be vectorized when they are convergent because it is known at compile time that they will all execute so the vectorization code can statically be generated. However, static vectorization is not possible for regions where the work-items diverge. The focus of this thesis is on a method for vectorization of divergent loops. Various transformation techniques are proposed and evaluated which seek to maximize CPU performance by: (1) extracting vectorizable statements from the divergent loops where possible, and (2) preserving the intended memory access order for better locality. In doing so, these loops are executed on the CPU in a way that best mimics the GPU execution model thereby maximizing efficiency and improving performance.

CHAPTER 2

BACKGROUND

2.1 Overview of OpenCL and GPUs

Multiple GPU programming languages exist with varying terminology. OpenCL terminology will be used throughout this document because the framework in which the work was done uses OpenCL. The OpenCL GPU programming model is as follows. A sequential thread running on a host invokes an OpenCL function (called a kernel) that runs on a device. The kernel is run in an NDRange which is subdivided into multiple work-groups which can run independently and in parallel. The work-groups are further subdivided into multiple work-items that can also run in parallel. Work-items within the same work-group can synchronize with one another, and they can share data with one another via a fast scratchpad memory called local memory. Work-items in different work-groups cannot synchronize with one another, and can only share data via the global device memory.

A GPU is composed of multiple compute units also called streaming multiprocessors which operate in parallel and have a common L2 cache and device memory. Each compute unit has multiple sequential cores which operate in parallel and have a common L1 cache and local memory. When an OpenCL kernel runs on a GPU, work-groups are assigned to different compute units and execute on those compute units until completion. Work-items in a work-group execute on the different cores of the compute unit. There could be more work-groups than compute units and more work-items than cores, in which case the hardware needs to manage sharing and scheduling of resources. This architectural model is diagrammed in Figure 2.1.

While writing GPU kernels may be easy, a great deal of tuning is required to make them run efficiently. Oftentimes, performance optimizations are geared toward maximizing resource utilization. Two such optimizations are

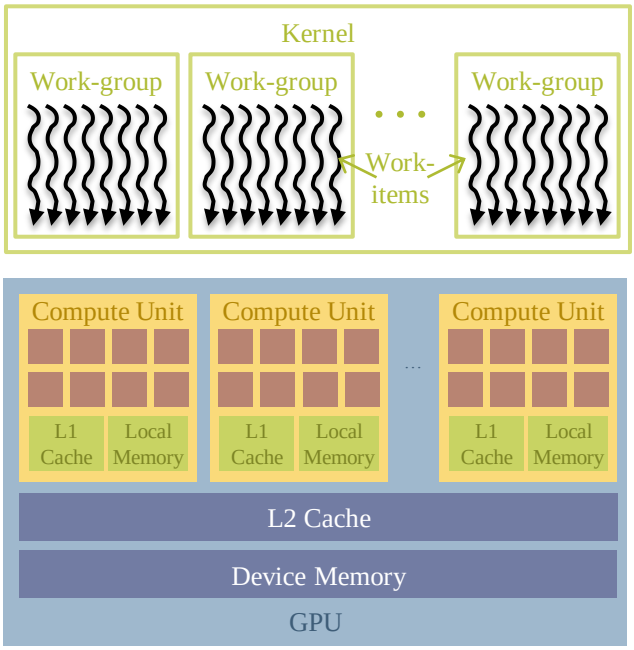


Figure 2.1: OpenCL and GPU architecture overview.

avoiding control divergence and coalescing memory accesses which maximize efficiency of the functional units and memory subsystem respectively. These two optimizations are important considerations for the transformations proposed in this thesis.

2.2 Control Divergence

On current GPU hardware, a work-group is divided into multiple subsections called wavefronts or warps, and the work-items of a wavefront are executed together in SIMD on the cores of the compute unit. The typical size of a wavefront is 32 (NVIDIA) or 64 (AMD) work-items. When work-items in the same wavefront encounter a branch where different work-items take different paths, the execution is said to exhibit control divergence. Because the work-items operate in lock-step on the SIMD hardware, all work-items end up taking all execution paths together with the inactive work-items on each path predicated out. Such behavior is unfavorable because it underutilizes the execution units. For maximal resource utilization, programmers are encouraged to avoid control divergence and write their code in such a way where work-items in the same wavefront always take the same path [1].

2.3 Memory Coalescing

When a GPU kernel contains a load instruction, multiple loads are issues simultaneously – one for each work-item in the wavefront. These loads cannot all be processed simultaneously by the memory subsystem which results in a serialization of memory accesses. However, in the case where adjacent work-items load data from adjacent memory locations, loads to the same cacheline can be consolidated into a single load which maximizes the efficiency of the memory hierarchy and improves performance. This access pattern is referred to as memory coalescing and is considered a good programming practice [1].

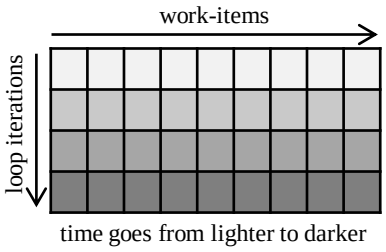
CHAPTER 3

PREVIOUS WORK

The technique proposed in this thesis is an improvement on the work done in MxPA [2] which itself is an improvement on MCUDA [3]. In both MxPA and MCUDA, parallelization on CPU threads is done at work-group granularity. In other words, work-groups are divided across multiple CPU threads and work-items within a work-group are performed within the same thread. This arrangement is the most natural because work-groups do not cooperate and can execute in any order so they present an ideal unit of work for parallelization. On the other hand, CPU parallelization at the work-item granularity would incur too much overhead because CPU threads are much heavier weight than GPU work-items.

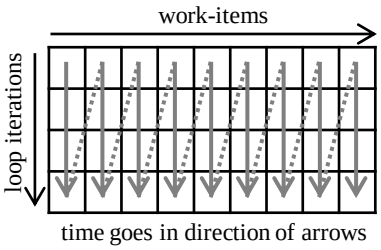
MxPA improves on MCUDA in the treatment of work-items within a work-group. In MCUDA, work-items are serialized via a thread-loop. The kernel must be divided into regions around synchronization barriers and each region must be serialized separately so that the senchronization semantic is preserved. On the other hand, in MxPA, the work-items within a work-group are vectorized if the code region is convergent. A static analysis is employed to determine whether the code region is convergent or not. A code region is convergent if it is inside a conditional control structure that has a work-item-independent condition. If a region is convergent, work-item-dependent statements are vectorized (for convenience of representation, Intel CEAN [4] is used for the vectorization). If not, then the compiler falls back onto the thread-loop approach for handling divergent regions.

Consider the example code in Figure 3.1(a) where the condition cond is work-item-independent making the loop convergent. When executed on a GPU, the execution order is such that the first iteration is executed for a bundle of work-items, followed by the second iteration, and so on. When serialization is applied in MCUDA, the statement execution order changes to that shown in Figure 3.1(b). When vectorization is applied in MxPA, it



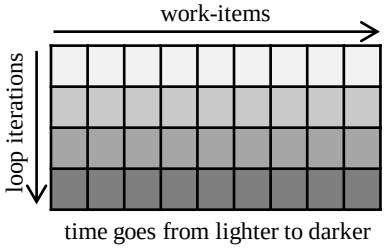

```
while(cond) {
  body;
}
```

(a) Original OpenCL kernel




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

(b) C kernel with thread-loops (MCUDA approach)




```
while(cond) {
  body(0:N);
}
```

(c) C kernel with vectorization (MxPA approach)

Figure 3.1: Behavior of convergent loops in previous approaches. The expression cond is independent of the work-item id making the loop convergent. The variable i in (b) represents the work-item id for which the code will be executed. The notation body(0:N) in (c) indicates that body is executed for N work-items starting from 0 using vector instructions.

becomes as shown in Figure 3.1(c) which is identical to the order assumed in 3.1(a). The advantage of vectorization is that it improves performance by extracting more parallelism from the program. It also matches the order in which the iterations of the work-items are executed which is better if the programmer was optimizing for data locality. As mentioned earlier, the programmer will tend to have adjacent work-items access adjacent memory locations when executing a load to achieve better memory coalescing. In such cases, the pattern in 3.1(b) is likely to thrash the cache, whereas that in 3.1(c) is likely to achieve better performance.

The vectorization of convergent regions, as opposed to serialization, is beneficial because it leverages the work-item parallelism to utilize the CPU's SIMD execution units for better performance. Moreover, it corrects the order of memory access to that which is assumed by the programmer which results in a better memory access pattern if the programmer had memory coalescing in mind when writing the program.

However, this technique is limited when the loop is divergent (i.e. cond is dependent on the work-item index). The reason is that not all work-items are in the loop all the time so the statement cannot be vectorized as shown in Figure 3.1(c). Therefore the technique must fall back on the serialization approach if the divergence property of the loop is not known statically. In this thesis, we show how to overcome this limitation.

CHAPTER 4

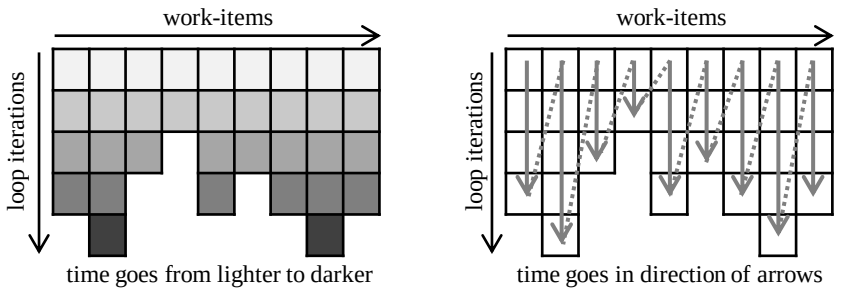
VECTORIZATION OF DIVERGENT LOOPS

Although a loop may seem divergent when analyzed statically, it is often the case that the loop converges dynamically for some iterations. This can happen in computations with boundary conditions where the input data size is not a perfect multiple of the work-group or NDRange size so work-items must iterate over multiple elements. It can also happen in generally divergent loops where each work-item has a different loop bound, but the loop is still convergent at least for the first few iterations. In such cases, it can be a waste of vectorization opportunities as well as potentially better memory access patterns to serialize the work-items.

Consider the example code in Figure 4.1(a) where cond is work-item dependent. On the GPU, the iterations will be executed as shown in the figure because GPUs have hardware support for masking out inactive work-items. However, a regular vectorization of body will not work on the CPU because body does not always execute for all work-items. For this reason, the previous approach handles this case by serializing the loop as shown in Figure 4.1(b).

4.1 Dynamic Loop Vectorization

Continuing with the example in Chapter 3, the intuition for the approach to be proposed is based on the observation that although the loop diverges toward the latter part of the execution, all work-items are indeed active for the first two iterations. This means that vectorization could be performed for those iterations. Unfortunately, it is difficult if not impossible to determine statically how many iterations converge. For this reason, we propose a dynamic checking technique where the transformed code checks on every iteration whether the loop converges, vectorizing if so and serializing otherwise. The transformed code and resulting execution pattern looks like that




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

(a) Original OpenCL kernel

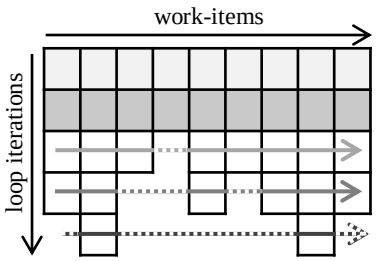
```
for(i=0; i<N; ++i){
  while(cond) {
    body;
  }
}
```

(b) C kernel with thread-loops (MCUDA and MxPA)

Figure 4.1: Behavior of divergent loops in previous approaches. The expression cond is dependent on the work-item id making the loop divergent. The variable i in (b) represents the work-item id for which the code will be executed.

in Figure 4.2.

First, the condition is evaluated for all work-items and assigned to a pred vector (line 01). Next, the total number of active work-items is computed (line 02), and then the loop begins (line 03). At the beginning of every loop iteration, the number of active work-items is checked (line 04). If all work-items are active, a vectorized version of the body and next condition is executed (lines 05-06). Otherwise, a serialized version of the body and next condition is executed (lines 08-13). At the end, the total number of work-items in the next iteration is computed (line 15) and the next iteration begins. Iterations end when all work-items have dropped out. Note that even in the cases when a loop iteration across work-items is not vectorized, the serialization happens horizontally across work-items as opposed to vertically across iterations. Such change in iteration order is beneficial for the memory access pattern because the programmer is expected to access adjacent memory locations across multiple work-items on the same iteration.




```
01  pred[0:N] = cond(0:N)
02  numTrue = sum(pred[0:N])
03  while(numTrue > 0) {
04    if(numTrue == N) {
05      body(0:N);
06      pred[0:N] = cond(0:N);
07    } else {
08      for(i=0; i<N; ++i){
09        if(pred[i]) {
10          body;
11          pred[i] = cond(i);
12        }
13      }
14    }
15    numTrue = sum(pred[0:N])
16  }
```

Figure 4.2: Dynamic loop vectorization execution pattern and pseudocode. The array pred contains a predicate variable for each work-item indicating whether the work-item is active in each iteration. The function sum counts the number of true values in the array. The remaining variables and notations are as in Figures 3.1 and 4.1.

4.2 Dynamic Loop Sub-Vectorization

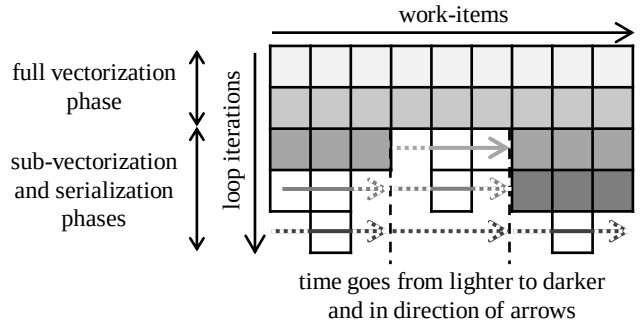
While the technique in Section 4.1 can successfully extract vectorization opportunities from divergent loops and rearrange the loop iteration order for a better memory access pattern, the vectorization can no longer be done when the first work-item in the work-group drops out. This can be improved by performing vectorization at a smaller granularity when the vectorization across the entire work-group fails. This technique is called dynamic sub-vectorization.

Continuing with the same example, Figure 4.3 shows the impact of sub-vectorization on the execution patterns. Previously, after the first two iterations ended, the remaining iterations were completely serialized. However, in this example, by dividing up the work-group into multiple sub-groups and checking each sub-group for convergence separately, more vectorization opportunities can be extracted. This results in better utilization of the SIMD execution units at the expense of additional checking overhead. The additional logic for sub-vectorization involves looping over each sub-group (line 08), computing the number of active work-items in that sub-group only (lines 09-10), vectorizing the sub-group if all work-items are active (lines 11-14), and serializing the sub-group otherwise (lines 15-22). The final overall decision-making flow for the transformation is diagrammed in Figure 4.4.

4.3 Handling Irregular Control Flow

The presence of irregular control flow makes the transformations a bit more tricky. Goto statements are not handled in this approach. On the other hand, break and continue statements must be treated differently based on whether the execution is in the full vectorization, sub-vectorization, or serialization phases. Table 4.1 summarizes how breaks and continues are handled in each of the execution phases.

When a break statement is executed in the full vectorization body, it implies that all work-items want to break out of the loop. Therefore it can simply be kept as a break statement without any changes. When a break statement is encountered in the sub-vectorization phase, it means that only the work-items in the sub-group want to terminate. Therefore keeping the




```
01  pred[0:N] = cond(0:N)
02  numTrue = sum(pred[0:N])
03  while(numTrue > 0) {
04    if(numTrue == N) {
05      body(0:N);
06      pred[0:N] = cond(0:N);
07    } else {
08      for(subStart=0; subStart<N; subStart+=32) {
09        subSize = min(32, N-subStart);
10        numSubTrue = reduce(pred[subStart:subSize]);
11        if(numSubTrue == subSize) {
12          body(subStart:subSize);
13          pred[subStart:subSize] =
14            cond(subStart:subSize);
15        } else {
16          for(i=subStart; i<subStart+subSize; ++i){
17            if(pred[i]) {
18              body;
19              pred[i] = cond(i);
20            }
21          }
22        }
23      }
24    }
25    numTrue = sum(pred[0:N])
26  }
```

Figure 4.3: Sub-vectorization execution pattern and pseudocode. The variable subStart indicates the id of the first work-item in the sub-group. The variable subsize indicates the size of the sub-group. The remaining variables and notations are as in Figures 3.1, 4.1, and 4.2.

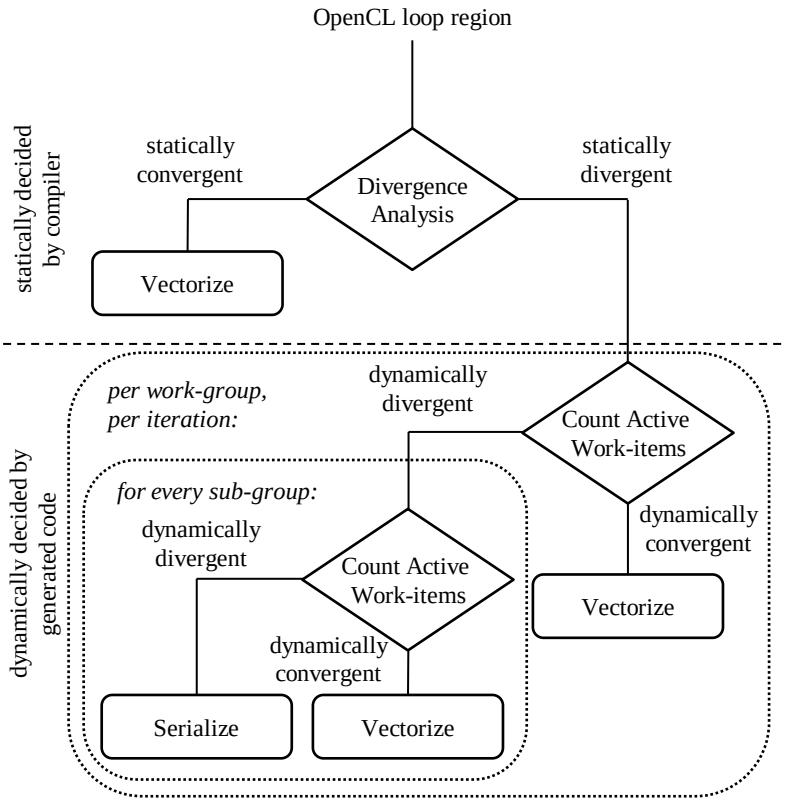


Figure 4.4: Overall decision-making flow of dynamic loop vectorization.

Table 4.1: Handling break and continue statements.

OpenCL	`break;`	`continue;`
Full Vectorization	`break;`	`pred[0:N] = cond(0:N);` `continue;`
Sub-vectorization	`pred[subStart:subSize] = 0;` `continue;`	`pred[subStart:subSize] = cond(subStart:subSize);` `continue;`
Serialization	`pred[i] = 0;` `continue;`	`pred[i] = cond(i);` `continue;`

break is incorrect because it will break out of the loop over sub-groups preventing succeeding work-items from executing. Instead, the break needs to be replaced with a continue to ensure subsequent sub-groups execute as well. To ensure that the sub-group that is breaking does not execute on the next iteration, all the predicates of the sub-group must be set to false. Therefore, a break statement is replaced with an assignment of all sub-group predicates to false followed by a continue statement. When a break statement is encountered in the serialization phase, it implies that only that work-item wants to break. For the same reasoning as the sub-groups, the break must be replaced with an assignment to that work-item's predicate to false followed by a continue statement.

When a continue statement is encountered in any of the three phases, it remains a continue. However, before the continue statement is executed, the next iteration condition must be evaluated for the continuing work-items. Therefore a condition evaluation of the entire work-group, the sub-group, or the single work-item must be inserted before the continue statement in the full vectorization, sub-vectorization, and serialization phases respectively.

4.4 Handling Nested Loops

When two divergent loops are nested, the dynamic loop vectorization transformations get more complicated. Every work-item that is inactive in the outer loop continues to be inactive in the inner loop. The inner loop must inherit the divergence information of the outer loop and use it as a starting point for computing its own divergence information. The inner loop, however, must not overwrite the divergence information because once a work-item becomes inactive in the inner loop, it must remain active in the outer loop to finish the execution.

The way to handle nested loops is explained using a simple code example shown in Figure 4.5. The condition computation for the outer loop is done in the same way as before (lines 01-02). The first part of the loop denoted by A is transformed on its own by using the same convergence checking methods as before (lines 04-08). Upon encountering the divergent inner loop, the condition computation must be predicated by the predicate of the outer loop (line 09). That is because only work-items which are active inside the outer

```
while(cond1) {
  A
  while(cond2) {
    B
  }
  C
}
```

(a) Nested OpenCL Divergent Loop

```
01  pred1[0:N] = cond1(0:N);
02  numTrue1 = reduce(pred1[0:N]);
03  while(numTrue1 > 0) {
04    if(numTrue1 == N) {
05      vectorize A
06    } else {
07      sub-vectorize or serialize A
08    }
09    pred2[0:N] = pred1[0:N] && cond2(0:N);
10    numTrue2 = reduce(pred2[0:N])
11    while(numTrue2 > 0) {
12      if(numTrue2 == N) {
13        vectorize B and cond2
14      } else {
15        sub-vectorize or serialize B and cond2
16      }
17      numTrue2 = reduce(pred2[0:N])
18    }
19    if(numTrue1 == N) {
20      vectorize C and cond1
21    } else {
22      sub-vectorize or serialize C and cond1
23    }
24    numTrue1 = reduce(pred1[0:N])
25  }
```

(b) Dynamic Vectorization of Nested Loop

Figure 4.5: Handling nested loops.

loop must evaluate the condition. The inactive work-items will inherit the 0 predicate from the outer loop which will short circuit the condition check. Once the inner loop is entered, its body denoted by B is dynamically vectorized using the same methods as before (lines 12-16). Finally, the remaining part of the outer loop denoted by C is also dynamically vectorized.

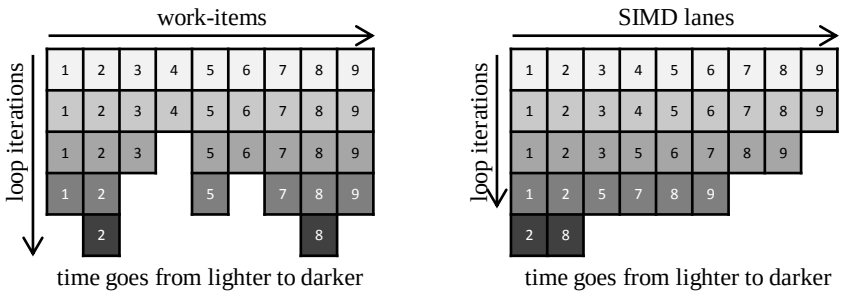
One important observation is that for this transformation to work, the loop must be divided into multiple regions and each region is transformed separately. All of region A is executed, then all of B, and then all of C. This means that the divergent loops begin to act as synchronization points in the generated code. This must be taken into consideration in the region formation algorithm which previously only considered explicit barrier synchronization points. Another observation is that the condition evaluation must take place in the last region inside the loop. For example, in Figure 4.5, cond1 is only evaluated in region C (line 22). It can also be treated as its own region at the end of the loop and dynamically vectorized on its own.

4.5 Dynamic Work-Item Compaction

Dynamic work-item compaction is an experimental technique that was attempted without success. This technique is described in this section but no results will be shown because the technique was abandoned before completion.

Dynamic work-item compaction attempts to compact the active work-items into adjacent SIMD lanes so that they can be vectorized. To do so, an additional level of indirection is necessary on every memory load to convert from the SIMD-lane index to the work-item index. The advantage of this technique is that it enables vectorized computation all the time, at the expense of some checking overhead as well as additional intermediate memory loads.

An example of how the technique works is shown in Figure 4.6. First, an intermediate list active is generated which contains a contiguous list of all the active work-items in the work-group (lines 01-06). The number of active work-items is also tracked. Next, the loop begins and iterates until all work-items are no longer active (line 07). The body of the loop is executed as a vector operation, however all references to the thread index are now replaced




```
01  numTrue = 0;
02  for(i=0; i<N; ++i) {
03    if(cond(i)) {
04      active[numTrue++] = i;
05    }
06  }
07  while(numTrue > 0) {
08    body(active[0:numTrue]);
09    oldNumTrue = numTrue;
10    numTrue = 0;
11    for(i=0; i<oldNumTrue; ++i) {
12      if(cond(active[i])) {
13        active[numTrue++] = i;
14      }
15    }
16  }
```

Figure 4.6: Work-item compaction execution pattern and pseudocode. The numbers in the figure indicate the id of the work-item for which the loop iteration is being executed. The array active contains a contiguous list of the work-items which are still executing the loop. The remaining variables and notations are as in the previous figures.

with an access to the active array which indexes just the active threads (line 08). Next, a new active list is generated with the work-items that are to be active in the next iteration (lines 09-15).

Although this technique is very promising in the amount of vectorization it can achieve, the increase in memory accesses due to the additional level of indirection to access the active array proved to be too high. It is particularly expensive when the thread index is used to access an array because it transforms a vector access such as arr[0:N] to a scatter or gather operation such as arr[active[0:N]] which is serialized by the compiler. An implementation of several relevant benchmarks using this technique showed significant degradation in locality and performance. For this reason, the technique was abandoned before maturity and no thorough analysis of its impact on all

benchmarks was made.

CHAPTER 5

A WORKING EXAMPLE

This chapter goes through a simple example of sparse matrix-vector multiplication (SpMV) using the JDS format to show how the transformations proposed apply to a real application. The JDS format is illustrated in Figure 5.1 with the corresponding variables from the code indicated. The following is an OpenCL version of SpMV from the Parboil benchmark suite [5].

```
01  int ix = get_global_id(0);
02  float sum = 0.0f;
03  int bound = sh_zcnt_int[ix/32];
04  for(int k = 0; k < bound; k++) {
05    int j = jds_ptr_int[k] + ix;
06    int in = d_index[j];
07    float d = d_data[j];
08    float t = x_vec[in];
09    sum += d*t;
10  }
11  dst_vector[d_perm[ix]] = sum;
```

The loop (line 04) in this kernel is statically divergent. The reason is that the loop condition (k < bound) is work-item dependent because bound is work-item dependent. However, by design of the JDS format, the loop has a great amount of convergence. Because rows are sorted according to the number of non-zeros, neighboring work-items will tend to process a similar number of elements. This makes the loop a good candidate for dynamic vectorization. The rest of this section goes through how this code is transformed.

The variables ix, sum, and bound are all work-item dependent so they must be expanded to have one version for each work-item so that the work-item executions can continue in parallel. The variable assignments (lines 01-03) become like the following where sx is the number of work-items in the work-group.

```
int ix[sx];     ix[:] = get_global_id0[:];
float sum[sx];  sum[:] = 0;
int bound [sx]; bound[0:sx] = sh_zcnt_int[ix[0:sx]/32];
```

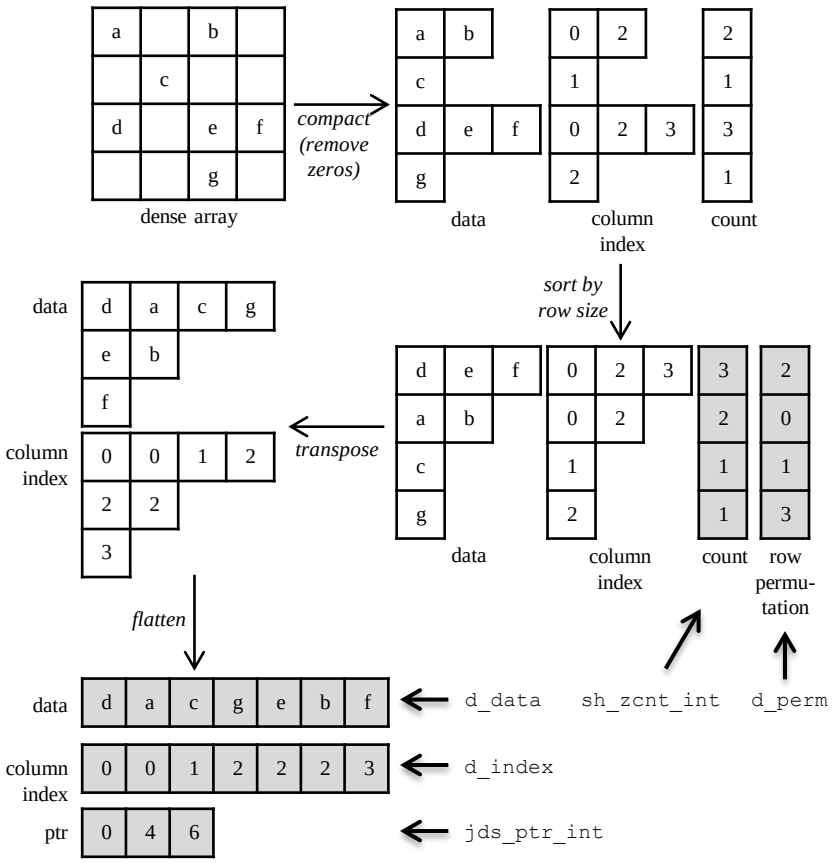


Figure 5.1: JDS format.

The rows in the JDS format are sorted by number of non-zeros so that adjacent work-items would process similarly sized rows, thereby having similar loads which reduces control divergence. The transposition ensures that adjacent threads access adjacent memory locations which results in memory coalescing. An additional optimization not shown in the figure is that rows can be padded with zeros such that rows in the same wavefront all have the same number of elements which reduces control divergence. As a result, a bound only needs to be stored for each wavefront instead of each row, which is why the index to sh_zcnt_int is divided by 32 (wavefront size) in the code.

The next variable to be dealt with is the loop index k. Although k is work-item dependent, the dependence analysis is intelligent enough to find out that it does not need to be expanded. The details of the dependence analysis are beyond the scope of this thesis. Therefore the initialization in the loop header remains:

```
int k = 0;
```

Next, the loop condition must be evaluated for the first iteration and stored in a predicate array. The condition check is vectorized as follows where p0 is the predicate array.

```
unsigned int p0[sx]; p0[:] = k < bound[:];
```

After that, the number of active work-items (or threads) is calculated by summing up the predicate array using the CEAN function __sec_reduce_add and the loop begins to iterate until all work-items become inactive.

```
unsigned int numActiveThreads0 = __sec_reduce_add(p0[:]);
while(numActiveThreads0 > 0) {
  ...
}
```

Inside of the loop, the local variables are declared. The variables j, in, d, and t are all expanded because they are all work-item dependent.

```
int j[sx];
int in[sx];
float d[sx];
float t[sx];
```

After variable declarations, the number of active work-items is checked. If it is equal to the number of work-items in a work-group, the body of the work group (lines 05-09) and condition of the next iteration are vectorized for the entire work-group.

```
if (numActiveThreads0 == sx) {
  j[:] = jds_ptr_int[k] + ix[:] ;
  in[:] = d_index[j[0]:sx];
  d[:] = d_data[j[0]:sx];
  t[:] = x_vec[in[0:sx]];
  sum[:] += d[:]*t[:] ;
  p0[:] = k < bound[:];
} else {
  ...
}
```

Notice how instead of using j[0:sx] to express the values of j, the transformed code uses j[0]:sx. This optimization saves on memory accesses by taking advantage of the fact that it knows j is stride 1. In other words, it knows

that j[n + 1] = j[n] + 1 for n ranging from 0 to sx-1, so it can replace all instances of j[0 + k] with j[0]+k for k in bounds, and only need to load j[0]. While executing d_index[j[0:sx]] results in a gather operation serialized by the compiler, d_index[j[0]:sx] will result in a vectorized load which is much more efficient. The details of this optimization are outside the scope of this work. If the work-items are not all active, the execution next goes into the sub-vectorization mode and loops over each individual sub-group. Here, bsx is the sub-group size and bx is the id of the first work-item in the sub-group.

```
const unsigned int bsx = 32;
for (unsigned int bx = 0; bx < sx; bx += bsx) {
  ...
}
```

For each sub-group, the code first counts the number of active work-items in the sub-group. The code is actually more complicated when sx is not a multiple of bsx, but we ignore that case in this example to keep the code simple.

```
unsigned int numActiveThreadsInTile = __sec_reduce_add((p0[bx:bsx]));
```

If all work-items in the sub-group are active, then the sub-group is vectorized.

```
if (numActiveThreadsInTile == bsx) {
  j[bx:bsx] = jds_ptr_int[k] + ix[bx:bsx] ;
  in[bx:bsx] = d_index[j[bx]:bsx];
  d[bx:bsx] = d_data[j[bx]:bsx];
  t[bx:bsx] = x_vec[in[bx:bsx]];
  sum[bx:bsx] += d[bx:bsx]*t[bx:bsx] ;
  p0[bx:bsx] = k < bound[bx:bsx];
} else {
  ...
}
```

Otherwise it will be serialized with a check on every predicate.

```
for (unsigned int x = bx; x < (bx + bsx); x++) {
  if ((p0[x])) {
    j[x] = jds_ptr_int[k] + ix[x];
    in[x] = d_index[j[x]];
    d[x] = d_data[j[x]];
    t[x] = x_vec[in[x]];
    sum[x] += d[x]*t[x];
    p0[x] = k < bound[x];
  }
}
```

The number of active work-items is recomputed for the next iteration.

```
numActiveThreads0 = __sec_reduce_add(p0[:]);
```

Finally, once the loop is over, the last statement (line 11) is executed.

```
dst_vector[d_perm[ix[0]:_sx_]] = sum[0:_sx_];
```

The fully transformed code for this example is shown in Appendix A.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Experimental Setup

The transformations proposed in this thesis were implemented as part of the MxPA [2] codebase. The Intel C Compiler (ICC) 13.13 is used for compiling the output C+CEAN code to final machine code.

The evaluation hardware includes an Intel i7-3820 processor and 16G of DDR3 DRAM with dual channel configuration. The system was running a 64-bit Ubuntu 12.04. The evaluation kernels were extracted from two benchmark suites to evaluate the performance of each implementation: Parboil 2.5 [5] and Rodinia 2.4 [6]. Only kernels that contained divergent loops which dominated performance were used in the evaluation because other kernels are not relevant to the transformations proposed.

Throughout this section, results are shown for three incremental versions of the transformation:

1. SVEC: which is the original static vectorization of convergent regions in [2]. Divergent regions always serialized. This version is used as a baseline.

2. DVEC: which is the dynamic vectorization of divergent loops using just the all-or-none approach for vectorizing work-items in a work-group.

3. DVEC-S: which is the dynamic vectorization of divergent loops including sub-vectorization attempts before falling back on serial code.

The evaluation will consist of an analysis of the impact of the proposed technique on dynamic instruction count, data locality, and overall execution time.

6.2 Impact on Dynamic Instruction Count

The vectorization technique proposed in this thesis is expected to decrease the dynamic instruction count for kernels with loops that are convergent dynamically under the assumption that programmers will aim to minimize control divergence when optimizing their kernels. However, if kernels are not well optimized and contain loops that have a high amount of control divergence, then the transformation is expected to make the dynamic instruction count worse because it incurs a large amount of unnecessary overhead for dynamic convergence checking then falls back to the serialization approach.

The results in Figure 6.1 show the dynamic instruction count for each technique normalized to the baseline SVEC implementation. The results in Figure 6.2 show the dynamic convergence rate of the loops in each kernel. The convergence rates for heartwall could not be obtained because the kernel was too large to instrument.

It is evident from the results that the kernels having an improved instruction count due to vectorization (spmv and lavaMD) both have high convergence rates and respectively take the full vectorization and sub-vectorization paths most of the time. This demonstrates the effectiveness of the dynamic vectorization approach and improving performance when vectorization opportunities are present despite the checking overhead.

On the other hand, the two kernels having low convergence rate (mri-gridding and bfs-rodinia) both show the most significant slowdown. Because these kernels fall back on the serial version most of the time, they incur the extra cost of dynamic convergence checking but do not benefit from vectorization to offset that cost. It would be ideal if one could predict the likelihood that a loop would dynamically converge or diverge and back off from applying dynamic vectorization entirely to avoid the unnecessary cost. However, it is very difficult to make such a prediction statically. One possible remedy to this problem would be to dynamically track the behavior of a loop. If the loop has been diverging most of the time during execution, then at some threshold the transformed code could stop attempting to vectorize it and run the serial version from the start.

The remaining five benchmarks (bfs-parboil, histo, sad, tpacf, and particlefilter) are anomalous in that they all show good dynamic convergence rates but still show degradation in dynamic instruction count. These bench-

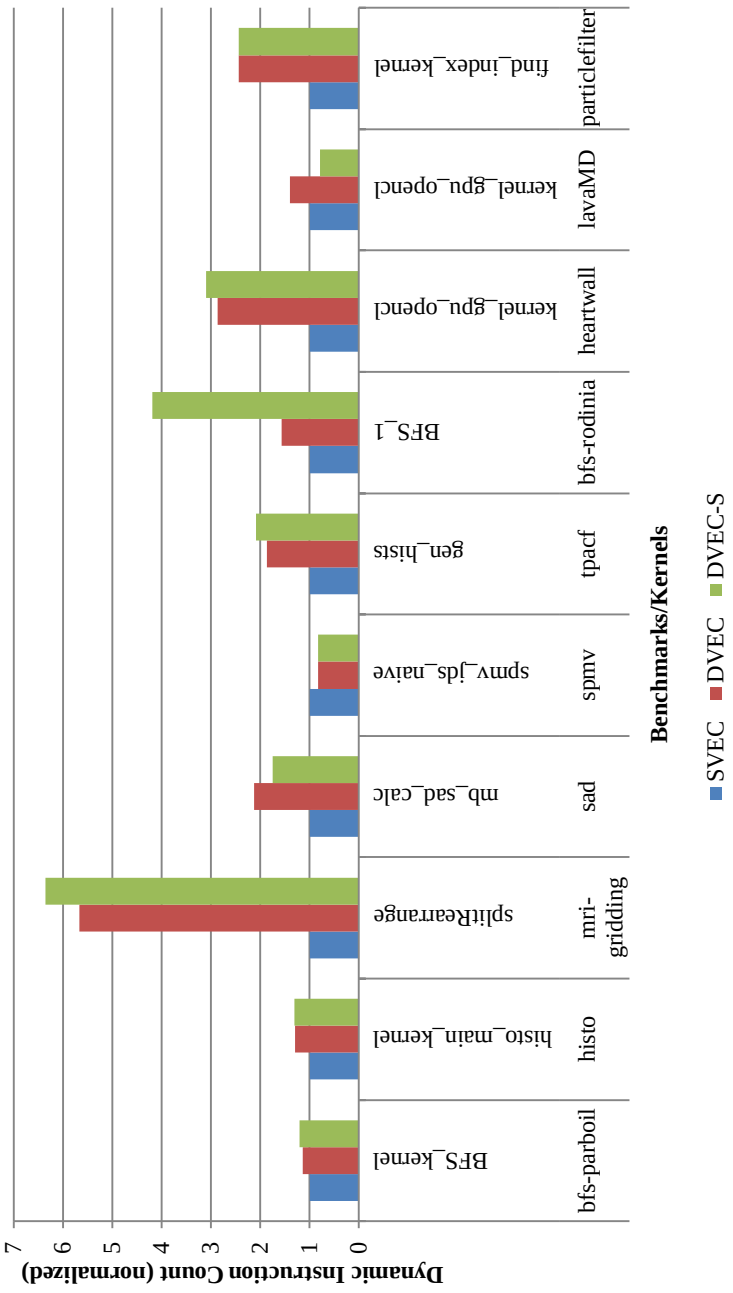


Figure 6.1: Dynamic instruction count of each kernel normalized to the baseline previous approach.

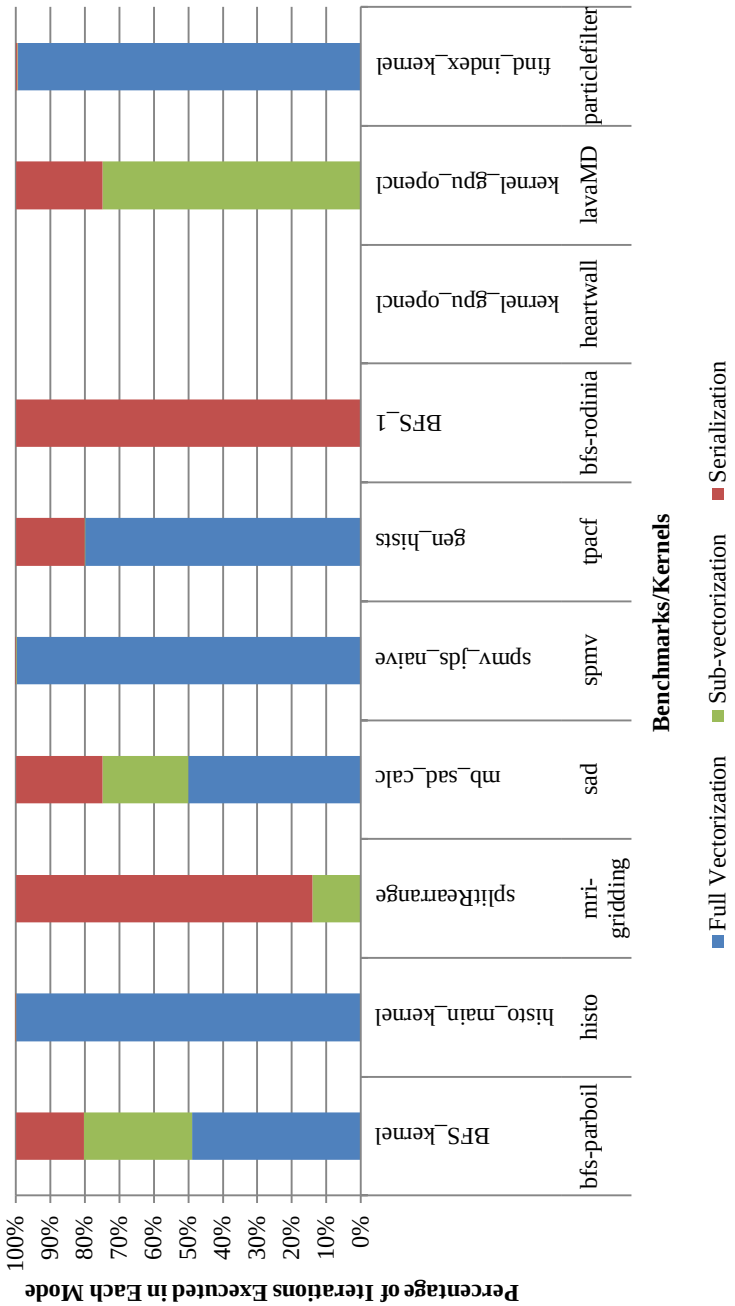


Figure 6.2: Dynamic convergence rates of each kernel measured by the percentage of loop iterations executed in each of the vectorization/serialization modes.

marks fall into two categories. The first category is the benchmarks where the loop body is very small such that there is not enough work done in the vectorized loop to amortize the checking overhead. The benchmarks that suffer from this problem are tpacf and particlefilter. A remedy for this problem would be to apply a cost function that would evaluate the amount of work a loop performs and avoid vectorizing it if it is too little. The second category is the benchmarks where the loop body contains code that cannot be vectorized by the compiler. In other words, after the dynamic checking is performed and the vectorization or sub-vectorization versions are selected, the execution gets serialized anyways and the potential performance benefit is lost. The benchmarks falling into this category are bfs-parboil, histo, and sad. The reason these benchmarks cannot be vectorized is the presence of calls to device/intrinsic functions which get serialized by the compiler. One remedy for this problem is to inline the device functions before vectorizing the code so that the device functions can be vectorized as well. In the case where a function or intrinsic cannot be inlined and vectorized such as an atomic operation, this can be included in the cost function suggested earlier which statically decides whether it should attempt to dynamically vectorize the loop or not.

6.3 Impact on Data Locality

The vectorization technique proposed in this thesis is expected to improve the memory access pattern and decrease the number of data cache misses for well-optimized kernels with loops where work-items access adjacent memory locations on the same iteration. However, if kernels are not well optimized and memory access is arbitrary, then the transformation is expected to increase the number of cache misses because the extra variables needed for dynamic convergence checking increase the memory footprint as a whole.

The results in Figure 6.3 show the L1 data cache load misses for each technique normalized to the baseline SVEC implementation. Three benchmarks show a significant improvement in data locality: histo, spmv, and heartwall. Code inspection of these three benchmarks reveals that all three are optimized for memory coalescing where adjacent work-items access adjacent elements.

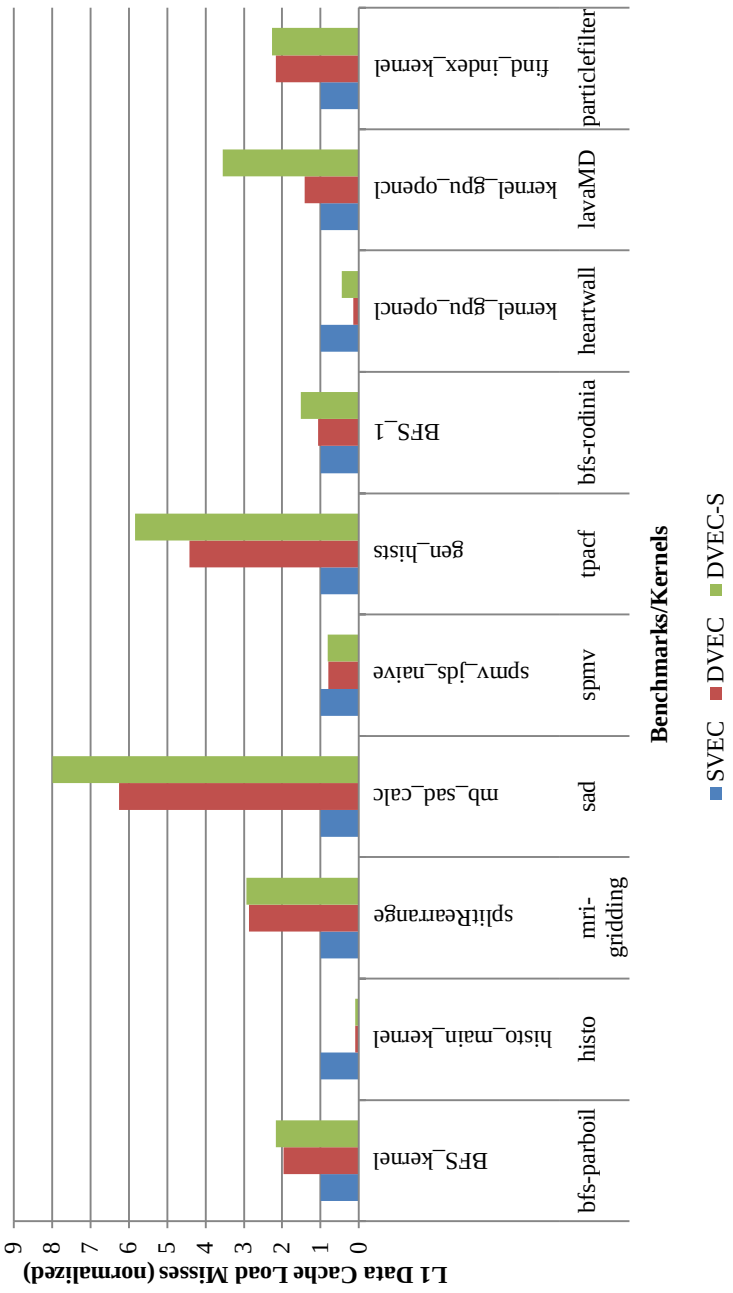


Figure 6.3: L1 cache data load miss rates of each kernel normalized to the baseline previous approach.

The remaining seven kernels all show a degradation in locality. However, it turns out that all these kernels are not well optimized for memory access on the GPU because work-items do not access adjacent elements. In fact, five out of the seven kernels (bfs-parboil, mri-gridding, sad, bfs-rodinia, and particlefilter) have the pattern where each work-item loops over a continuous segment in memory. With such a memory access pattern, serialization is expected to have better performance. Tpacf has an arbitray memory access pattern. LavaMD has a thread coarsening loop which executes only once so the iterations execute in the same order for both techniques and no benefit is gained there. Moreover, the kernel uses an Array-of-Structures (AoS) storage format instead of Structure-of-Arrays (SoA) and accesses all array elements in the loop which is why serialization results in better memory behavior (see Appendix B for explanation). All such access patterns are not consistent with GPU programming best practices. A remedy for these situations could be to perform a static analysis to detect the memory access patterns in the loop. If memory access indices are dependent on the loop index and/or if AoS storage formats are used, then the compiler could statically avoid applying the dynamic vectorization transformation.

6.4 Impact on Overall Execution Time

The results in Figure 6.4 show the speedup of each technique normalized to the baseline SVEC implementation. The benchmarks that show speedup are spmv and lavaMD. The first benefits from improvement in both instruction count and locality whereas the second benefits from improvement in instruction count that is significant enough to compensate for the performance hit to the locality degradation. The benchmark histo shows little change because it experiences an improvement in locality and degradation in instruction count that offset each other. For heartwall, there is an improvement in locality, but the degradation in instruction count results in a net degradation in the execution time. The remaining six benchmarks all show degradation in both metrics, hence degradation in execution time. The reasons for why each benchmark performs the way it does have been described in Sections 6.2 and 6.3. The benchmarks performing well are those that conform to the GPU best practices of minimizing control divergence and coalescing memory ac-

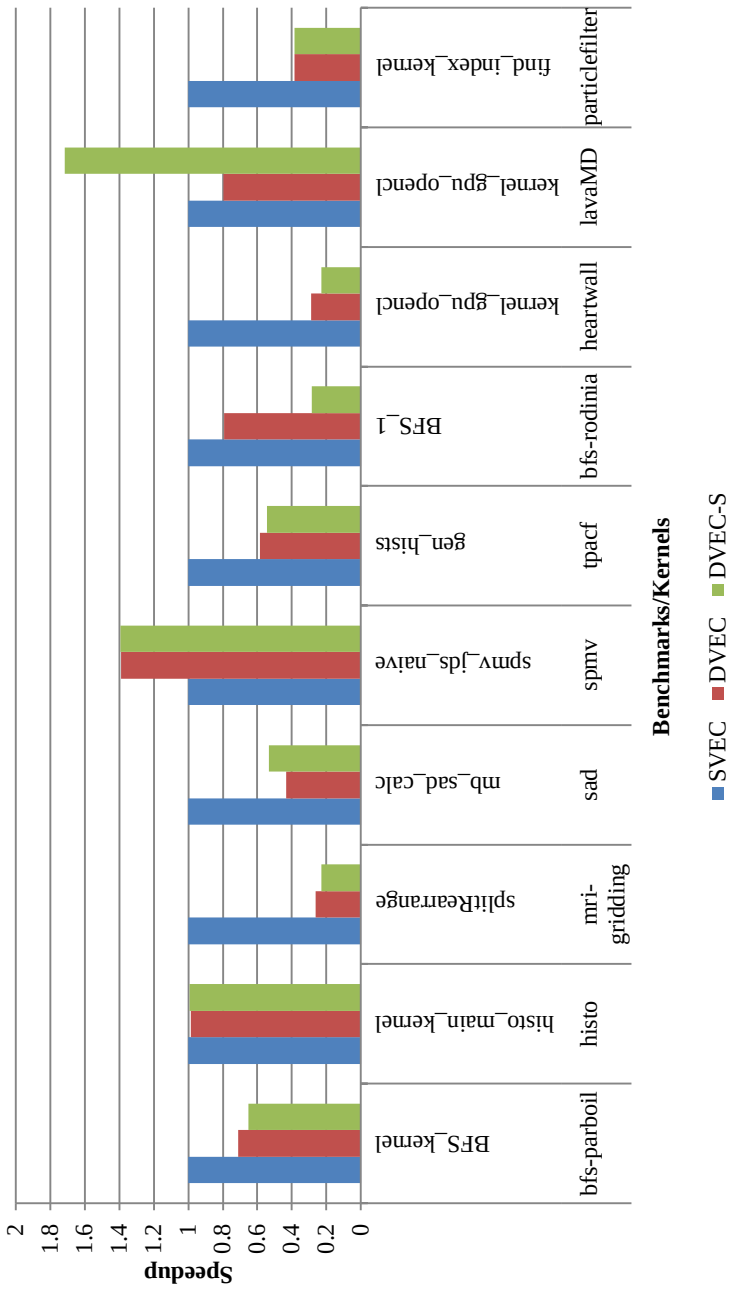


Figure 6.4: Speedup of each kernel over the baseline previous approach due to dynamic vectorization.

cesses. However, it happens that most benchmarks used for evaluation do not exhibit these properties in the divergent loops under study. A summary of the behavior of each benchmark is shown in Table 6.1.

Table 6.1: Summary of the impact of dynamic vectorization.

Benchmark	Impact on Time	Impact on Dynamic Instruction Count	Divergence or Vectorization Pattern	Impact on Data Locality	Memory Access Pattern
bfs-parboil	-	-	serialization (atomics)	-	sequential and indirect
histo	0	-	serialization (device function)	+	coalesced
mri-gridding	-	-	high rate of divergence	-	sequential
sad	-	-	serialization (intrinsic function)	-	sequential
spmv	+	+	high rate of convergence	+	coalesced
tpacf	-	-	little work in loop	-	arbitrary
bfs-rodinia	-	-	high rate of divergence	-	sequential and indirect
heartwall	-	-	N/A	+	coalesced
lavaMD	+	+	high rate of convergence	-	AoS
particlefilter	-	-	little work in loop	-	sequential

Note: The symbol "+" means that the metric improved, i.e. a decrease in time, decrease in dynamic instruction count, and decrease in L1 cache misses. The symbol "-" means that the metric got worse. The symbol "0" means that there was no change in the metric. The term "serialization" refers to when code in the fully vectorized phase needs to be serialized so the benefit of vectorization is lost. The term "sequential" refers to when the same work-item iterates sequentially over a contiguous chunk of memory instead of there being a coalesced memory access pattern in the kernel.

CHAPTER 7

RELATED WORK

Many OpenCL stacks have been proposed targeting multicore CPUs.

AMD's Twin Peaks [7] is an OpenCL stack provided which handles work-items within a work-group by using user-level threads that context switch at barrier synchronization points. Twin Peaks has the advantage of not relying on compiler techniques for work-item scheduling, but moves it into the runtime system which allows for incorporating runtime information into the scheduling, not to mention reusing of off-the-shelf compilers and debugging tools. Twin Peaks' user-level threading approach makes it more difficult to incorporate the vectorization techniques presented in this thesis because the vectorization is not made explicit.

Intel also has an OpenCL implementation [8] for CPUs and Xeon Phis, but details about their code generation and compilation techniques are not publicly known.

Karrenberg and Hack [9] present an OpenCL compilation technique for CPUs which, like this work, also uses SIMD vectorization for mapping work-items to the CPU. However, their approach is fundamentally different in the way divergent regions are handled. While the approach in this work generates multiple static versions of the divergent loop body and selects between them dynamically, their approach generates a single version only and uses software predication techniques for divergence handling. Their work also differs in the granularity of vectorization. While vectorization of a region in this work is performed at the level of an entire work-group, their work only vectorizes at the granularity of the SIMD width (W). In other words, they execute the entire region for just W work-items before moving on to the next W. Finally, regions in their work are coarser-grain because divergence is not used as a criterion for region formation.

The portable computing language (pocl) [10] uses a similar methodology as MCUDA for translating the kernels except that it does so at the LLVM

IR level. It depends on the LLVM inner loop vectorizer to perform vectorization (if any) by annotating work-item loops using the LLVM parallel loop annotations.

SnuCL [11] is another OpenCL compiler for CPU platforms. Its approach is very similar to MCUDA. The treatment of work-items is very similar to MCUDA's thread-loops. The name given to the technique is work-item coalescing. There does not seem to be an active effort toward any SIMD vectorization.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this thesis, a technique has been proposed for vectorizing divergent loops when retargeting GPU-optimized OpenCL kernels for multicore CPUs. Because actual convergence cannot always be determined statically, the proposed transformation generates multiple static versions of the loop body that are fully vectorized for the entire work-group, partially vectorized for each sub-group, or serialized and selects between these three versions based on a runtime evaluation of the divergence property of the loop. The results show that the technique performs well for kernels that conform to GPU programming best practices such as minimizing control divergence and coalescing memory accesses. On the other hand, kernels that exhibit large amounts of control divergence and poor memory access patterns on the GPU do not translate well to the CPU relative to previous approaches.

There are multiple techniques that could be used to improve the performance of the code generated by the transformations proposed in this thesis. The technique needs to be applied selectively based on whether it is expected to be beneficial or not. This can be done statically via a cost function that ensures that there is enough work being done in the loop to amortize the vectorization overhead, and that the proximity of memory accesses is present across work-items not loop iterations. It can also be done dynamically via a divergence prediction mechanism where the divergence of a loop can be predicted based on its history such that dynamic vectorization of frequently divergent loops can be avoided. Finally, additional support for inlining device functions is expected to avoid serialization of code where vectorization opportunities exist.

APPENDIX A

JDS SPMV TRANSFORMED CODE EXAMPLE

Original OpenCL Code [5]:

```
int ix = get_global_id(0);
float sum = 0.0f;
int bound = sh_zcnt_int[ix/32];
for(int k = 0; k < bound; k++) {
  int j = jds_ptr_int[k] + ix;
  int in = d_index[j];
  float d = d_data[j];
  float t = x_vec[in];
  sum += d*t;
}
dst_vector[d_perm[ix]] = sum;
```

Transformed Code:

```
int ix[sx];     ix[:] = get_global_id0[:];
float sum[sx];  sum[:] = 0;
int bound [sx]; bound[0:sx] = sh_zcnt_int[ix[0:sx]/32];
int k = 0;
unsigned int p0[sx]; p0[:] = k < bound[:];
unsigned int numActiveThreads0 = __sec_reduce_add(p0[:]);
while(numActiveThreads0 > 0) {
  int j[sx];
  int in[sx];
  float d[sx];
  float t[sx];
  if (numActiveThreads0 == sx) {
    j[:] = jds_ptr_int[k] + ix[:] ;
    in[:] = d_index[j[0]:sx];
    d[:] = d_data[j[0]:sx];
    t[:] = x_vec[in[0:sx]];
    sum[:] += d[:]*t[:] ;
    p0[:] = k < bound[:];
  } else {
    const unsigned int bsx = 32;
    for (unsigned int bx = 0; bx < sx; bx += bsx) {
      unsigned int numActiveThreadsInTile = __sec_reduce_add((p0[bx:bsx]));
      if (numActiveThreadsInTile == bsx) {
        j[bx:bsx] = jds_ptr_int[k] + ix[bx:bsx] ;
        in[bx:bsx] = d_index[j[bx]:bsx];
        d[bx:bsx] = d_data[j[bx]:bsx];
        t[bx:bsx] = x_vec[in[bx:bsx]];
        sum[bx:bsx] += d[bx:bsx]*t[bx:bsx] ;
```

```
        p0[bx:bsx] = k < bound[bx:bsx];
      } else {
        for (unsigned int x = bx; x < (bx + bsx); x++) {
          if ((p0[x])) {
            j[x] = jds_ptr_int[k] + ix[x];
            in[x] = d_index[j[x]];
            d[x] = d_data[j[x]];
            t[x] = x_vec[in[x]];
            sum[x] += d[x]*t[x];
            p0[x] = k < bound[x];
          }
        }
      }
    }
  }
  numActiveThreads0 = __sec_reduce_add(p0[:]);
}
dst_vector[d_perm[ix[0]:_sx_]] = sum[0:_sx_];
```

APPENDIX B

ARRAY OF STRUCTURES AND STRUCTURE OF ARRAYS

Figure B.1 shows that the array-of-structures (AoS) storage format performs poorly if the loop is vectorized because it results in a scattered access pattern, whereas it performs well when the loop is serialized because the access pattern is linear. On the other hand, the structure-of-arrays (SoA) format performs well when the loop is vectorized because the accessed memory is contiguous, whereas the access pattern is not as ideal as AoS when the loop is serialized.

Array-of-Structures (AoS) Code

```
struct {
  int x;
  int y;
} s[100];

unsigned i;
for(i = 0; i < 100; ++i) {
  foo(s[i].x);
  bar(s[i].y);
}
```

Structure-of-Arrays (SoA) Code

```
struct {
  int x[100];
  int y[100];
} s;

unsigned i;
for(i = 0; i < 100; ++i) {
  foo(s.x[i]);
  bar(s.y[i]);
}
```

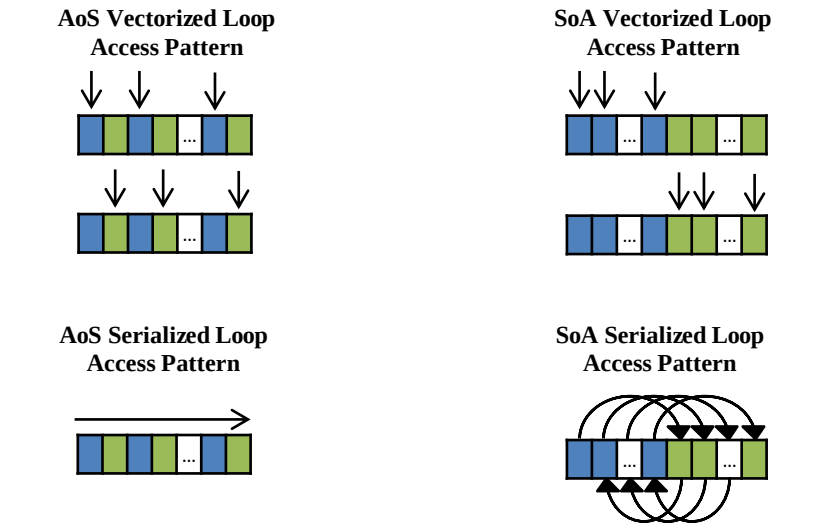


Figure B.1: Array-of-structures and structure-of-arrays.

REFERENCES

[1] NVIDIA, *NVIDIA OpenCL Best Practices Guide*, 1st ed., Aug 2009.

[2] J. A. Stratton, H.-S. Kim, T. B. Jablin, and W. mei W. Hwu, "Performance portability in accelerated parallel kernels." Center for Reliable and High-Performance Computing, Tech. Rep., 2013.

[3] 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.

[4] Intel Corporation, "CEAN language extension and programming model." [Online]. Available: http://software.intel.com/sites/default/files/fd/c5/CEAN_Model.docx

[5] 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," Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, Mar. 2012.

[6] 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.

[7] 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. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854302 pp. 205–216.

[8] Intel, "Intel SDK for OpenCL applications XE 2013 R3." [Online]. Available: http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe

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

[10] "pocl – Portable Computing Language." [Online]. Available: http://pocl.sourceforge.net

[11] 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*. ACM, 2012, pp. 341–352.