Input Space Splitting for OpenCL

Simon Moll Johannes Doerfert Sebastian Hack
Saarland University, Germany
{moll, doerfert, hack}@cs.uni-saarland.de

Abstract

The performance of OpenCL programs suffers from memory and control flow divergence. Therefore, OpenCL compilers employ static analyses to identify non-divergent control flow and memory accesses in order to produce faster code. However, divergence is often input-dependent, hence can be observed for some, but not all inputs. In these cases, vectorizing compilers have to generate slow code because divergence can occur at run time.

In this paper, we use a polyhedral abstraction to partition the input space of an OpenCL kernel. For each partition, divergence analysis produces more precise results i.e., it can classify more code parts as non-divergent. Consequently, specializing the kernel for the input space partitions allows for generating better SIMD code because of less divergence.

We implemented our technique in an OpenCL driver for the AVX instruction set and evaluate it on a range of OpenCL benchmarks. We observe speed ups of up to $9\times$ for irregular kernels over a state-of-the-art vectorizing OpenCL driver.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Optimization

General Terms Algorithms, Performance

Keywords Polyhedral Representation, OpenCL, SPMD, Vectorization, Divergence

1. Introduction

For OpenCL on modern vector CPUs, control flow and memory access divergence is a major performance obstacle [25]. A branch is divergent if not all (active) vector lanes either take it or do not take it. In such a case, the compiler has to convert control flow to data flow and replace branching with predication or blending. The program has to execute both directions of the branch and reconverge later. As a consequence, more instructions are executed than in the non-divergent case and SIMD utilization deteriorates. A memory access is divergent if not all active vector lanes access memory either uniformly (all access the same address) or in a consecutive manner. As a consequence, the compiler has to resort to a complex and less efficient scatter or gather instruction for a single vector memory access. For the hardware this means that multiple cache lines have to be transferred instead of only one.

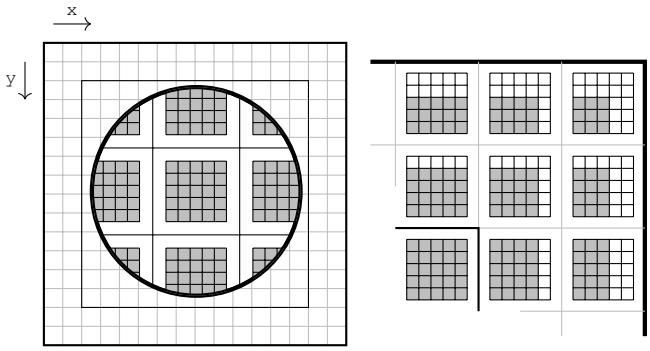


Figure 1. Divergence in the convolution kernel. In the image center, all stencil elements are applied regardless of the pixel position (lhs). This is not true for the boundary, however (rhs). Whether a stencil element is applied, varies between neighboring pixels. This causes divergence in the `i` and `j` loops when the kernel is vectorized.

Typically, not all branches and memory accesses of a kernel are divergent. To statically detect non-divergent code, vectorizing compilers apply *divergence analysis*. Being a static program analysis, divergence analysis has to be sound for *all* possible inputs. Many kernels however only show divergence for *some* of their inputs but not all of them. For example, consider the 2D stencil kernel in Figure 2. When this kernel is applied to a 2D grid (see Figure 1), only the instances on the grid's border exhibit divergence in the loop counters. For the border work items of the image, the loop iterates less often. For the large majority of the instances the loop trip counts are uniform.

However, static divergence analysis has to classify the loop branches as divergent due to these "corner cases". Consequently, the compiler generates code that accommodates divergence. This does not only involve vectorizing the loop trip count but also complex mask manipulation code to track which instances still iterate the loop and which have already left it. This code is significantly slower than a variant that handles the border and another variant—in which the loop trip counter is uniform—for the inner part of the image. This variant for the inner part of the image is shown in Figure 3 in a vectorized version. Because there is no divergence in the image center, all control flow is preserved.

In this paper, we present an approach to create these variants *automatically.* To this end, we represent OpenCL kernels in the polyhedral model. We express non-divergence (for both, control flow and memory accesses) using linear constraints on the iteration domain and memory access functions of the kernel. Based on this representation we generate variants of the kernel that exhibit less divergence than the original kernel. In turn, for these variants more efficient vector code can be generated. Furthermore, we use polyhedral techniques to generate a dispatcher that selects at

runtime—based on the parameters of the kernel—an appropriate variant of the kernel.

In summary, this paper makes the following contributions:

- We introduce a polyhedral representation of OpenCL kernels. We show how to model the OpenCL execution model and API interactions in the polyhedral model and present overapproximation techniques to represent kernels that are not naturally representable in the polyhedral model (Section 5).
- We present a novel technique to specialize OpenCL kernels based on architecture-specific properties like non-divergence. To this end, we represent these properties as linear constraints in a polyhedral representation of the kernel (Section 3.2).
- We show how the polyhedral representation can be used to automatically synthesize a dispatcher that selects an appropriate variant at runtime (Section 4.1).
- We demonstrate the efficacy of our approach on an OpenCL implementation for the AVX SIMD instruction set. The implementation optimizes for regular control flow and memory access patterns. We show that our technique can automatically create and select variants of kernels. We observe speed ups up to $9\times$ over a state of the art OpenCL driver (Section 6).

```
__kernel
void Convolution2D(int *output, int *input,
                   float *mask, int width,
                   int height, int maskWidth,
                   int maskHeight) {
  int x = get_global_id(0);
  int y = get_global_id(1);

  int vstep = (maskWidth - 1) / 2;
  int hstep = (maskHeight - 1) / 2;

  int left   = MAX(0, x - vstep);
  int right  = MIN(width - 1, x + vstep);
  int top    = MAX(0, y-hstep);
  int bottom = MIN(height - 1, y + hstep);

  float sumFX = 0;
  for (int i = left; i <= right; ++i) {
    for (int j = top; j <= bottom; ++j) {
      int maskIndex = (j - (y - hstep)) * maskWidth
                    + (i - (x - vstep));
      int index = j * width + i;

      sumFX += input[index] * mask[maskIndex];
    }
  }
  output[y * width + x] = (int)(sumFX + 0.5);
}
```

Figure 2. Convolution 2D kernel.

2. Background

In this work, we use the polyhedral model as intermediate representation to optimize the vectorization of OpenCL kernels.

2.1 Polyhedral Model

The polyhedral model is a mathematical program representation that precisely represents the control and memory effects of dynamic statement instances in parametric Static Control Parts (*SCoPs*) [4]. It is used as an intermediate representation by polyhedral loop optimizers such as Polly [12], Graphite [30], Pluto [5] or PoCC [31]. Conceptually, the polyhedral model describes statement instances as well as memory access functions as parametric polyhedra. These are internally represented as sets of quasi-affine inequalities of parameters (statically unknown but dynamically invariant values)

```
// [..] Definition of vstep, hstep, top, bottom, y
int x  = <x position of first vector lane>;

float8 sumFX = (float8) 0.0;
for (int i = x - vstep; i < x + vstep; ++i) {
  for(int j = top ; j <= bottom; ++j) {
    int maskIndex =
          j - (y - hstep)) * maskWidth
        + (i - (x - vstep));
    int index     = j * width + i;

    sumFX += input[index:index+7]
          * (float8) mask[maskIndex];
  }
}
```

Figure 3. Vectorized Convolution2D kernel for the image center. The vectorized version exploits the regular structure of the i and j loops and the memory acccesses.

and loop iteration variables. We will use $\mathbb{P}$ to denote the set of all SCoP parameters and consequently $\mathbb{IV}$ for the iteration variables. To build a polyhedral representation of a program part it has to fulfill the requirements of a *SCoP*:

Static Control Parts (SCoPs) Polyhedral optimizers analyze program regions known as Static Control Parts (*SCoPs*). Classically, SCoPs are restricted to loop nests that satisfy the following conditions:

Control Flow inside a SCoP is representable by affine functions in the parameters of the SCoP as well as loop iteration variables surrounding the branch conditions.

Memory Accesses have access functions that can be described as affine functions in the parameters of the SCoP as well as loop iteration variables surrounding the access. Furthermore, there cannot be aliasing between different base pointers.

Function Calls are only allowed if the called function is pure, e.g., common math functions like `sqrt`.

However, in Section 5 we will describe approximations for these restrictions that allow to represent general OpenCL kernels.

Polyhedral Representation In the polyhedral model a program is defined by the statements it contains. Each statement in turn by an iteration space and the memory accesses it contains. Both are represented with parametric integer polyhedra denoted as $\mathbb{Z}_{\mathbb{P}}$. For a statement Q surrounded by n loops, we denote the iteration space, or domain, with $\mathcal{I}_Q \in \mathbb{Z}^n_{\mathbb{P}}$ and the set of access functions in Q with $\mathbb{F}_Q \subset \mathbb{Z}^n_{\mathbb{P}} \to \mathbb{Z}_{\mathbb{P}}$.

Intuitively, $\mathcal{I}_Q$ describes all dynamic statement instances of Q. Each instance is identified as an iteration vector $\vec{i}_Q \in \mathbb{Z}^n_{\mathbb{P}}$ that contains all loop iteration variables (from the outer to innermost) surrounding Q. We will use the notation i_d to subscript dimension d in an iteration vector $\vec{i}$. We will denote an increment in this iteration variable by the vector $\vec{v}_d$.

An access function $\mathcal{F} \in \mathbb{F}_Q$ maps dynamic statement instances to the accessed memory location. To simplify our representation and to be consistent with our approach, we omit the access base pointer in the definition of $\mathcal{F}$. While this is important for polyhedral optimizers that compute and use the dependences between statement instances, we do not use it in this work.

Vectorization in the Polyhedral Representation The polyhedral representation models scalar instances of statements. Only when a statement is vectorized later on, instances are grouped together to form a vector instruction. To this end, we define the *vector width* w as the number of scalar instances that will be mapped to one vector instruction.

3. Optimization

Our optimization relies on two principal techniques that both operate on the polyhedral representation of the kernel: *Control Flow Simplification* and *Domain Splitting*.

3.1 Control Flow Simplification

Polly's code generator [13] allows to hoist conditionals based on their domain. In the simplest form this corresponds to versioning as illustrated in Figure 4. Our optimizer applies this versioning on every dimension of the input in order to simplify the control in the vectorized part as much as possible. As the code duplication might become an optimization criterion or obstacle at some point, we allow the user to limit the versioning to some dimensions.

```
for (unsigned d0 = 0; d0 < size0; d0++)
  for (unsigned d1 = 0; d1 < size1; d1++)
    if (d0 < p)
      S: ...
    else
      P: ...
```

(a) A parametric branch condition inside an affine loop nest.

```
for (unsigned d0 = 0; d0 < p; d0++)
  for (unsigned d1 = 0; d1 < size1; d1++)
    S: ...
for (unsigned d0 = p; d0 < size0; d0++)
  for (unsigned d1 = 0; d1 < size1; d1++)
    P: ...
```

(b) Simplified control flow after the condition is hoisted.

Figure 4. Control flow simplifications using code versioning.

3.2 Domain Splitting

Domain Splitting is concerned with creating multiple versions of static program statements such that most of them result in more efficient vector code. Domain splitting was pioneered in the context of polyhedral schedule optimizations [11].

In the case of our optimization, the partition is defined by a binary *splitting predicate* that splits the input space into two partitions. Consequently, the statement is duplicated: One of the two statements will execute for all instances for which the predicate holds. The other statement is split off and will execute for the remaining part. Consider the simple kernel in Figure 5a.

```
__kernel
void foo(float * A, float * B) {
  int x = get_local_id(0);
R: A[x] = B[max(5, x)]; // irregular access to B
}
```

(a) A simple kernel with an irregular memory access.

```
__kernel
void foo(float * A, float * B) {
  int x = get_local_id(0);
  if (5 <= x) {
R_true: A[x] = B[x]; // consecutive accesses
  } else {
R_false: A[x] = B[5]; // remaining accesses
  }
}
```

(b) The same kernel after splitting the iteration domain of statement R with the *splitting predicate* `x >= 5`. The split yields two versions, Statement R_{true} where all memory accesses are consecutive and Statement R_{false} for the remainder of the domain.

Figure 5. Domain splitting.

In Statement R, the memory access to the array `B` is irregular and will yield slow code when the kernel is vectorized. Statement R has the iteration domain

$$R(x) : 0 \leq x < \texttt{get\_local\_size(0)} \ .$$

However, the *splitting predicate* $5 \leq x$ defines a subset of the iteration domain of R for which the memory access is consecutive. We partition the iteration domain of Statement R into two parts. One part characterizes the iterations for which the predicate holds. The other is split off and contains all remaining iterations. We split Statement R into copies R_{true} and R_{false} and assign to each of them one of the two domain parts of R.

$$R_{true}(x) : 5 \leq x < \texttt{get\_local\_size(0)}$$
$$R_{false}(x) : 0 \leq x < \mathit{min}\big(5, \texttt{get\_local\_size(0)}\big)$$

Figure 5b shows the result of the transformation. If the kernel is vectorized, Statement R_{true} exhibits a fast consecutive access. The remainder of the work items will use the memory access of Statement R_{false}, which has become uniform as a side effect.

In the following, we introduce *splitting predicates* for control flow and memory access patterns as they are used in our optimization. The splitting predicates are parametric in the vectorized dimension d.

3.2.1 Full Tile Splitting

When a loop is to be vectorized in the polyhedral model, it is first strip-mined with the vector width w. The loop iterating over the strip elements, the strip loop, describes a *vector tile*. Figure 6a shows the result of strip mining a simple loop iterating from 0 to $n-1$.

For loops with a trip count that is not a multiple of the vector width, strip mining produces a remainder loop that executes the remaining instances. Full Tile splitting partitions the iteration domain of the vector tile into a *full* part that contains an instance for each vector lane and a *partial* remainder. This makes sure that the vector tile is in fact full and the remaining iterations can run scalar as shown in Figure 6b.

```
for (int i = 0; i < n; i += 8) {
  for (int j = i; j < min(i + 8,n); ++j) {
    R: ...
  }
}
```

(a) Naively strip mined loop. Applied to the inner loop, the vectorizer will have to account for cases where n is not a multiple of the vector width 8, i.e. the vector tile is *partial*.

```
for (i = 0; i + 7 < n; i += 8) {
  for (int j = i; j < i + 8; ++j) {
    S: ...
  }
}
for (;i < n; ++i) {
  T: ...
}
```

(b) With full tile splitting. When Statement S is vectorized, no predication is necessary because the trip count of the inner loop always equals the vector width, i.e. the vector tile is *full*. The remaining instances, now belonging to Statement T, remain scalar.

Figure 6. The effect of full tile splitting on a strip mined loop.

We formally define the full tile predicate as

$$\mathit{Full}_n(d) := (i_d - (i_d \bmod w)) + w - 1 < n \ .$$

We require i_d to range from 0 to $n-1$ where n is a supplied upper bound on the iteration dimension. This requirement is met by the OpenCL work item loops in our model for which n is a parameter of the polyhedral representation. Figure 7 gives an intuitive understanding of the predicate.

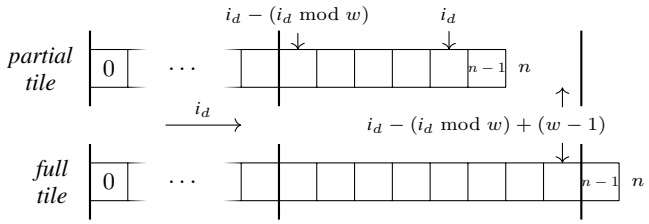


Figure 7. An iteration i_d only belongs to a full tile, if the last iteration of its vector tile executes as well. Consequently, the vector tile in the upper part is partial whereas the vector tile below is full. The predicate is shown for a vector width of 8.

3.2.2 Vector Memory Access Patterns

```
__kernel void replicate(float * A, float * B) {
  A[local_id(1) * width + local_id(0)] = B[local_id(1)];
}
```

Figure 8. Memory access patterns in an OpenCL kernel. The write access to `A` is *consecutive* in the first dimension. The read access to `B` is *consecutive* in the second dimension and *uniform* in the first.

When a scalar memory instruction is vectorized, it operates on a vector of addresses instead of a single memory address. In the worst case, these addresses are unrelated. Thus, the compiler has to emit slow scatter/gather instructions or scalar memory accesses along with vector pack/unpack instructions. Slow memory accesses due to *memory-access divergence* are a major cause of poor performance in vectorized programs.

However, compilers generate fast vector accesses if they can prove that the address vector has certain properties which we define below. Figure 8 shows a simple data transfer kernel which features the memory access patterns captured by our splitting predicates.

Let $\mathcal{F} \in \mathbb{F}_Q$ denote a memory access function of statement Q. We model vector access patterns as parametric in the access function $\mathcal{F}$ and the vectorized dimension d.

Consecutive In a *consecutive* address vector, the values referenced by consecutive address components are laid out contiguously in memory without any gaps between them. A consecutive memory access is vectorized by transferring an entire vector register to or from the memory pointed to by the first address vector component. This requires the accessed addresses to be offset by a multiple of the vector component size from the first address.

$$Cons_{\mathcal{F}}(d) := \mathcal{F}(\vec{i}_Q + \vec{v}_d) = \mathcal{F}(\vec{i}_Q) + 1$$

Uniform In an *uniform* address vector, all components point to the same address. The compiler leaves the memory access scalar and in case of a load operation broadcasts the loaded value into the components of a fresh vector register.

$$Uni_{\mathcal{F}}(d) := \mathcal{F}(\vec{i}_Q + \vec{v}_d) = \mathcal{F}(\vec{i}_Q)$$

We refer to a memory access that is neither *uniform* nor *consecutive* as divergent.

3.2.3 How to split?

At this point, the question remains how kernel statements should be split given a set of *splitting predicates*.

Domain splitting is a trade-off between code size and optimization potential in the split-off regions. Splitting results in new statements and hence in more code. On the other hand, splitting-off a part of the domain that shows a high potential for optimizations (many statements become uniform or consecutive) is beneficial as our experiments show. However, a part with high potential that is executed only for a small part of the iteration domain will have a negligible effect on the overall run time.

We present a heuristic solution to make appropriate splitting decisions in Section 4.

4. Heuristics

In order to optimize OpenCL kernels for vectorization, we identify parts of the work space that can be executed using vector memory instructions. Hence, we identify and optimize the subdomains with non-divergent memory accesses as well as non-divergent control flow. However, at compile time the size of these subdomains are not known. Furthermore, multidimensional kernels allow for different dimensions to vectorize over.

As we cannot statically determine for which dimension vectorization will give the best result we instantiate the kernel for each possible vectorization choice. In each kernel copy, called a *subkernel*, we will then strip-mine and vectorize one dimension. Afterwards, we can split the iteration domain of the subkernel according to the subdomain constraints for the vectorized dimension only. While e.g., the output code size might grow linear in the number of kernel dimensions, the subkernel approach can be restricted to a given set of dimensions to bound the code size. In any case it offers various advantages over the single kernel version, including:

Vectorization Dimension The kernel chooses an appropriately vectorized subkernel at runtime given parameter assignments and the work sizes.

Polyhedral Representation The subkernel selection logic is directly encoded in the polyhedral representation of the kernel. It benefits from the same control simplification techniques as the kernel code itself.

Static Subkernel Exclusion The driver detects which subkernels are unlikely to benefit from vectorization and strips them from the kernel.

4.1 Runtime Subkernel Selection

For an OpenCL kernel with n work dimensions we will create n subkernels, each will be vectorized in a separate work item dimension. Additionally, we generate one sequential fallback version that is used if our cost heuristic will advise against vectorization, e.g., if the local work group sizes are to small for full vectorization.

The *cost heuristic* yields for each subkernel an expression that computes a score from the parameters of the kernel. This score reflects how suitable a subkernel is for given work sizes and parameter assignments. The kernel dispatches execution at runtime to the subkernel that scores highest.

Cost Heuristic The *cost heuristic* generates a parametric cost function for a subkernel k and its associated vectorized dimension d_k. The cost function defined in Equation 1 accounts for the number of kernel iterations and the memory access patterns in the kernel. We use boxes (in contrast to general polyhedra) to overapproximate the iteration space of each memory access. While there exist more precise methods to compute the volume of an iteration space, such as Barvinok [3], we found that our approach works well in practice.

$$Score_n(k) := \begin{cases} \Sigma_{Q \in k, \mathcal{F} \in \mathbb{F}_Q} & \\ \quad w_{cons} \| Box(Cons_{\mathcal{F}}(d_k)) \| & \text{if } n \geq w \\ \quad + w_{uni} \| Box(Uni_{\mathcal{F}}(d_k)) \| & \\ 0 & otw. \end{cases} \tag{1}$$

The result is a weighted product of the parameters, including the work sizes. The parameter n refers to the upper bound of the

Table 1. The heuristic combination of splitting predicates yields up to 4 version of a statement. Each cell combines a row (control) and column (memory) splitting predicate. The cell content shows how a memory access `A[i]` will be vectorized for the intersection of these splitting predicates.

	Access pattern	
Tile	Consecutive ($\mathcal{I}_k^C$)	Remainder ($\mathcal{I}_k^\top$)
Full	`A[i:i+7]`	scatter/gather
Partial	mask move	masked scatter/gather

iteration dimension k_d. We optimized the weights w_{cons} and w_{uni} empirically. The score reflects the number of iterations of the subkernel and how well it can be vectorized. It defaults to 0, if no full vector tile can be executed for this work size. The scalar fallback code receives a score of 1 to be executed in cases where no subkernel is beneficial.

Static Subkernel Elimination We exploit the regular structure of the cost function to remove at compile time subkernels with detrimental expected performance. Consider the loop nest of Statement S

```
for (int i = 0; i < 2 * a; ++i)
  for (int j = 0; j < i; ++j)
    S(i, j);
```

Listing 1. Statement in a non-trivial loop nest.

in Listing 1. The iteration variables `i` and `j` are both bounded by the parametric interval $[0, \ldots, 2a-1]$. Thus, the bounding box volume of Statement S is $(2a)(2a) = 4a^2$. Subkernel scores are linear combinations of these volumes. Assume there is Subkernel *A* with score $2xz+xy$ and Subkernel *B* with score $xz+xy$. Subkernel *A* will always score greater of equal than *B* and the latter is statically discarded.

As each subkernel vectorizes a distinct work item dimension, this avoids vectorization in cases where it is unlikely to be beneficial.

4.2 Splitting Heuristic

The splitting heuristic determines which statements will be split for which subdomains. Table 1 shows the iteration domains resulting from splitting by this heuristic.

Access Function Splitting We split all statements with a global predicate for consecutive memory accesses. The predicate $\mathcal{I}_k^C$ is defined as the intersection of all non-trivial consecutive domain parts of all memory accesses, or formally

$$\mathcal{I}_k^C := \bigcap_{Q \in k} \bigcap_{\substack{\mathcal{F} \in \mathbb{F}_Q, st \\ Cons_{\mathcal{F}}(d_k) \neq \varnothing}} Cons_{\mathcal{F}}(d_k) \ .$$

In this subdomain $\mathcal{I}_k^C$, all memory accesses are consecutive, if they are ever consecutive at all. Its complement defines the split-off part with

$$\mathcal{I}_k^\top := \mathcal{I}_k \setminus \mathcal{I}_k^C \ .$$

Full Tile Splitting We split all statements that depend on the vectorized work item dimension with the *full tile splitting predicate*. This means, if a memory access was split by access function splitting, its parts may be split again for full tiles. The upper bound n required by full tile splitting predicate is given by the parameter $LocalSize_d$.

Discussion Despite their simplicity, the heuristics lend themselves to identify regions with consecutive accesses in arbitrary kernels operating on affine grids. For example, it successfully identifies the regular interior part of the 2D stencil kernel in Figure 2. However, approaches that split multiple times should account for the volume of the split-off parts to estimate their relevance to the total kernel runtime.

5. Polyhedral Representation of OpenCL Kernels

In this section, we describe how we model an OpenCL kernel in the polyhedral model. As OpenCL programs are not necessarily SCoPs, we first discuss overapproximations that we use to deal with the limitations of the polyhedral model described in Section 2.1. Second, we present how to represent features special to the OpenCL execution model in the polyhedral setting.

5.1 Polyhedral Applicability Issues

Non-Pure Function Calls Non-pure function calls are generally not allowed in polyhedral optimizers. In order to avoid them inlining is usually performed. As the restriction to recursion free OpenCL code is not uncommon, we will assume the same and inline all function calls prior to the modeling.

Non-Affine Memory Accesses In order to allow non-affine memory accesses we overestimate their effect. Hence, a non-affine access to an array element is modeled as an access to the whole array. As we cannot represent the access pattern precisely, we evaluate memory predicates for non-affine accesses conservatively, thus pretend each non-affine access is diverging.

Non-Affine and Non-Static Control Flow Polyhedral optimizers have been extended to allow dynamic control flow as well as non-affine bounds before [4, 10]. They describe how to model and generate code for dynamic control flow in order to exploit more parallelism and data locality. In our context we are interested in conditions on memory accesses and control patterns which can be derived statically and evaluated prior to the execution of the analyzed region. To this end, both techniques could allow us to derive assumptions about arbitrary control, though it might require the use of more sophisticated runtime checks, e.g., inspector loops [9].

```
bool cond(int, int);
void dynamicControl(float *In, float *Out, int N) {
  for (int i = 0; i < N; i++)
    for (int j = 0; cond(i, j); j++)
      Out[i] += In[j];
}
```

Figure 9. Loop nest with one affine and one non-affine loop.

For our prototype implementation we choose a simpler approximation that does not represent dynamic control flow but over approximates it as one atomic polyhedral statement. Thus, our polyhedral representation for the loop nest in Figure 9 will contain the `i`-loop and the complete `j`-loop will be represented by a single statement. As a result we cannot refer to the inner `j`-loop in the polyhedral representation, thus the access function for the access to `In` cannot be represented as an affine function in the model. Nevertheless, the accesses to `Out` are accurately represented in the model.

5.2 Barrier Elimination

By default the work item instances of an OpenCL kernel execute asynchronously. Synchronization can only be achieved within the same work group and through explicit barrier statements. Work items will suspend execution at the barrier until all instances in the work group have reached it. Only afterwards, work items will resume independent execution. In a well-defined OpenCL kernel, work items belonging to the same work group will always reach the same barrier statement.

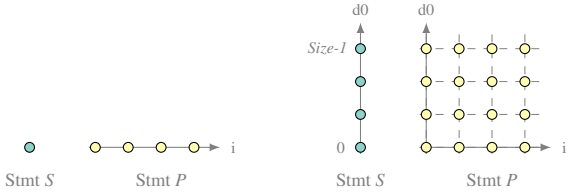
Our OpenCL driver splits kernels at barriers into continuation kernels [15]. Values that live through barriers are stashed in local

```
  __kernel void simple1D(float *In, float *Out, int N) {
S:  float val = In[get_local_id(0)];
    for (int i = 0; i < N; i++)
P:    Out[N * get_local_id(0) + i] += val;
  }
```

(a) Simple OpenCL kernel with one implicit work group dimension.

```
  void simple1D(float *In, float *Out, int N) {
    for (unsigned d0 = 0; d0 < get_local_size(0); d0++) {
S:    float val = In[d0];
      for (int i = 0; i < N; i++)
P:      Out[N * d0 + i] += val;
    }
  }
```

(b) C-version of Figure 10a with an explicit work group dimension.



(c) Polyhedral iteration space of Figure 10a

(d) Polyhedral iteration space of Figure 10b

Figure 10. Simple function as OpenCL kernel (a) and as C version with an explicit work group dimension (b). The polyhedral iteration space representation for both is given in part (c) and (d), respectively.

memory. Different to regular kernels, continuation kernels return the id of the next continuation kernel that shall execute or signal termination. Support code generated by the driver controls the execution of continuation kernels for each work group.

5.3 Modelling OpenCL-specific Features

In order to derive a unified model for both kernel as well as driver code we explicitly add the dimensions over the work groups to the polyhedral representation. These dimensions iterate over the work group, hence they range from 0 to the local work group size for the dimension. Consequently, the local work group id in the kernel can be replaced by the iteration variable of that dimension. Due to this explicit representation we can reuse polyhedral techniques, e.g., to determine the pattern of memory accesses. Additionally, this will allow us to generate the specialized loops for the work group dimensions together with the rest of the kernel code.

Figure 10a shows an one-dimensional OpenCL kernel together with its polyhedral representation in part (c). As we represent the kernel together with the implicit work group dimension, the polyhedral iteration space we generate is one dimensional for statement S and two dimensional for statement P. An illustration of the iteration space is given in part (d) and it corresponds to the C input in part (b).

5.4 OpenCL API Calls

OpenCL kernels use API calls to communicate with the driver at runtime. Information about the sizes of the work or the work groups, as well as to the current offset in both are accessible. In the kernel this information is commonly used to compute offsets for memory accesses but also in loop bounds or conditionals.

As our approach models and materializes the work group dimensions it has to represent these calls. However, all but one are also parameters (or affine combinations thereof) in the polyhedral representation as they do not vary during the exeuction of one work group. The exception are calls to `get_local_id(d)`. The returned value corresponds to the current iteration in dimension d, which in the polyhedral representation translates to the iteration variable $i_d \in \mathbb{IV}$.

Table 2 lists all OpenCL API calls we currently detect and their representation in the polyhedral model.

Table 2. Polyhedral representation of OpenCL API calls. We introduce the additional parameter $GroupOffset_d$ for purposes of the representation. There is no function to query it in the OpenCL language.

OpenCL API Call	Polyhedral Representation
`get_global_size(d)`	$0 \leq GlobalSize_d \in \mathbb{P}$
`get_local_size(d)`	$0 \leq LocalSize_d \in \mathbb{P}$
n/a	$0 \leq GroupOffset_d \in \mathbb{P}$
`get_local_id(d)`	$0 \leq i_d \leq LocalSize_d \in \mathbb{IV}$
`get_global_id(d)`	$i_d + GroupOffset_d$
`get_group_id(d)`	$0 \leq GroupId_d \in \mathbb{P}$

The parameter $GroupOffset_d$ does not correspond to an OpenCL function. It is used to define `get_global_id(d)` without non-affine expressions. It relates to the other parameters by $GroupOffset_d = GroupId_d * LocalSize_d$.

6. Evaluation

6.1 OpenCL Driver

Our experimental OpenCL driver is based on LLVM [19] and its pipeline is illustrated in Figure 11.

The pipeline is activated by a call to the OpenCL API function `clCreateKernel`, which requests an executable OpenCL kernel object. First, the OpenCL source code is translated into LLVM Bitcode. The barrier elimination stage splits the kernel function into a dispatch function and barrier-free continuation kernels (Section 5.2). The dispatch function controls the execution of the continuation kernels. The continuation kernels are passed on to the polyhedral optimizer Polly [12], which translates them into the polyhedral representation (Section 2.1). The optimization applies domain splitting for full tiles and memory access patterns according to the heuristics (Section 4). At this point, the user can provide additional domain knowledge about parameter constraints and the like to the optimization. Afterwards, Polly materializes each continuation kernel as LLVM bitcode. The bodies of strip-mined loops are extracted into separate functions. These functions represent the scalar codelets. The driver vectorizes the codelets with the OpenCL compiler of the Intel CodeBuilder SDK [24][1]. The compiler emits the vectorized kernels in LLVM bitcode. Eventually, the driver inlines all vectorized codelets and compiles them with the builtin JIT compiler of LLVM (MCJIT).

The Intel OpenCL driver that we compare against is also based on LLVM. It supposedly uses the same vectorizer as the Intel OpenCL compiler of the CodeBuilder SDK. We initially experimented with the Intel C Compiler (icc) for vectorization [17]. We decided against using it, as icc is an independent compiler with its own optimizations and code generation backend. We found that this would have obscured the contribution of our optimization to the runtime performance. The reported results are thus not biased by differences in the employed vectorizer. Instead, they reflect performance gains due to our optimization on top of a production-quality vectorizer.

[1] We use the bitcode-to-source translator axtor [20] to generate OpenCL source code from LLVM bitcode.

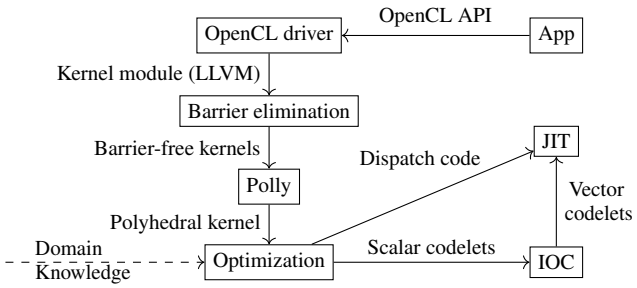


Figure 11. Driver pipeline.

6.2 Benchmarks

We evaluated our optimization on OpenCL benchmarks from the Rodinia 3.0 [6, 7], Parboil [28] and AMD APP SDK v3.0 benchmark suites. The DCT, LU decomposition, 2D convolution, Floyd-Warshall, Fast Walsh Transform, Bitonic sort, Binomial Option and Black-Scholes benchmarks are from the AMD APP SDK v3.0. The cfd, gaussian, kmeans, myocyte, and nn benchmarks from the Rodinia benchmark suite. The mri-q and sgemm benchmarks are from the Parboil benchmark suite.

Benchmark Selection The selection of suitable benchmarks was affected by two main points:

1. Our OpenCL pipeline is a prototype implementation that does not implement the entire OpenCL language. Kernels that use special features, e.g., atomic functions, have been ruled out.
2. Kernels distribute into families with regards to our optimization due to the implicit normalization when they are translated to the polyhedral representation (ref. Section 3.1). The performed optimizations will be the same for each kernel in a family and the results can be expected to be at least similar too. For two major families we choose a representing kernel, namely Bitonic sort and Floyd Warshall, but omitted benchmarks that would simply show the same behaviour e.g., DwtHaar1D from the AMD APP SDK.

6.3 Experimental Setup

All measurements were taken on an Intel© Core™ i7-4810MQ CPU @ 2.80GHz machine running ArchLinux (kernel 4.1.6-1-ck). We measured runtime performance under six settings, all except **intel** were run through our pipeline illustrated in Figure 11.

intel Execution with the Intel OpenCL driver version 1.2.0.57.

scalar No optimizations (incl. vectorization) are performed.

vec only No scoring function is generated and only *full tile splitting* is applied. The driver chooses the first subkernel whose work dimension has at least *vector width* iterations.

split only No scoring function is generated. The driver chooses the first subkernel whose work dimension has at least *vector width* iterations. All splitting optimizations and active control simplification are applied to each subkernel.

score only The driver creates a parametric scoring function for each subkernel. Subkernels with statically lower scores are removed. The scoring functions of the remaining subkernels are evaluated at runtime to decide which subkernel is executed. For each subkernel *full tile splitting* but no other splitting optimizations are applied.

split+score All optimizations presented in this paper are applied. Subkernel selection is the same as for **score only** and subkernel optimizations are the same as for **split only**. Consequently, every remaining subkernel is subjected to our splitting for *full tiles* and *memory access patterns* as well as *active control flow simplification*.

Single-threaded Evaluation All kernels were executed by a single thread[2]. Our optimization improves the per-thread processing time for each work group and is conceptually orthogonal to work group parallelization. Hence, we measure single-threaded execution time to assess the efficacy of our optimization without the non-deterministic effects of multi-threaded execution.

6.4 Runtime Results

In Figure 12 we compare our full optimization pipeline (*split+score*) with *scalar* execution, the *Intel OpenCL* driver as well as plain vectorization (vec only). Complementary, we detail the effects of the different optimizations in Figure 13.

General Discussion In our evaluation we found that both plain vectorization and *split+score* performed in our pipeline are generally better than the *Intel OpenCL* driver. Furthermore, vectorization with any of our proposed optimizations will even increase the average performance. Consequently, the best overall performance was achieved by applying all proposed optimizations, namely *full tile splitting*, *memory access splitting* as well as *active control flow simplification*.

While benchmarks as Floyd Warshall are so regular that plain full tile splitting suffices to achieve the best result, we found that others, e.g. gaussian, require scoring to pick the best work dimension for vectorization. In contrast, the Binomial Option benchmark needs control flow simplifications to perform best. Finally, our running example, Convolution2D, can only be beneficially vectorized if both, full tile splitting as well as memory splitting is performed.

Effect of Input Space Splitting The gap between *vec only* and *split only* reflects performance gains due to input splitting and active control flow simplification. Where this gap is small, the kernel features little divergence and can be vectorized well without splitting. While this holds for most kernels, Convolution2D is one example that mainly benefits from the separation of different input areas. Additionally, active control flow simplification is performed in the *split only* setting. While this improves benchmarks like Binomial Option with many control conditions based on the work group dimensions, it can cause regressions as seen for the nn benchmark. Here, the control flow conditions generated by Polly are too complex in comparison to the actual kernel code.

Effect of Subkernel Scoring Subkernel scoring is the difference between the *vec only* and *score only* settings in Figure 13. It shows generally good results but can cause regressions. This is mainly due to the static elimination of subkernels. Especially for benchmarks with unbalanced work group sizes or a majority of non-affine accesses this can cause problems. A statically less beneficial subkernel might be the only one with a sufficient number of iterations to be vectorized, though it would be eliminated by this scheme statically. Additionally, non-affine access functions will not increase the score. Consequently, a completely non-affine kernel, e.g., Bitonic sort, will be effectively scalarized as the scalar fallback kernel will statically have the highest score.

The best result for subkernel scoring is achieved for gaussian because it is not the first dimension that should be vectorized but the second. However, most other benchmarks are written especially for general OpenCL drivers that vectorize always the first dimension.

[2] The OpenCL command queue was constructed with the flag `CL_QUEUE_THREAD_LOCAL_EXEC_ENABLE_INTEL` to make the Intel OpenCL driver execute in single-thread mode.

As this dimension is implicitly picked by the other schemes too, subkernel scores will not yield a benefit for them.

Vectorization-only Speed Up The *vec only* setting is already performing better than the Intel OpenCL driver. This may come as a surprise as the Intel OpenCL driver is also based on the LLVM framework and thus applies similar bitcode optimizations. Additionally, our pipeline uses it to vectorize the codelets. However, there are two main factors for this difference in performance:

Control-Flow Simplification In every evaluation setting (except *intel*), we represent all benchmarks in the polyhedral model. Hence, they all undergo polyhedral code generation which might—as a side effect—hoist some conditionals. Therefore, also the kernels in *scalar* may profit from this.

Live Values at Barriers Live value optimization minimizes the amount of memory that has to be allocated for live values that live through OpenCL barrier statements. When variables are live through barriers, their values have to be stashed temporarily to memory. We found out that the Intel OpenCL compiler is using the same barrier elimination approach [15] as we do. Manual inspection of the Binomial Options kernel bitcode shows that the Intel OpenCL driver accesses 72 bytes per work-item. In contrast, for our implementation it is only 12 bytes.

Table 3. Listing of vector codelets with static divergence properties and observed number of executions. Each row refers to a vector codelet of a barrier-free continuation kernel. The codelets are grouped by the continuation kernel they belong to and ordered by execution count.

		Loops		Memory Accesses			
Kernel		*uni*	*div*	*cons*	*uni*	*div*	exec [%]
mri-q PhiMag	0			✓			100.00
mri-q CompQ	0	✓		✓	✓		100.00
DCT	0	✓		✓	✓	✓	100.00
	1	✓		✓	✓	✓	100.00
gauss Fan1	0				✓	✓	100.00
gauss Fan2	0			✓	✓		38.88
	0			✓	✓		35.45
	0			✓	✓		15.93
	0			✓	✓		5.80
	0			✓	✓		2.59
	0			✓	✓		1.34
NN	0			✓		✓	100.00
BinOpt	0			✓			100.00
	1			✓	✓		50.82
	1			✓	✓		48.40
	1			✓			0.78
	2			✓			100.00
	3			✓	✓		100.00
BlackScholes	0			✓			100.00
FloydWarshall	0			✓	✓		100.00
LUCombine	0			✓			74.78
	0			✓			25.22
LUDecompose	0			✓	✓	✓	50.05
	0			✓		✓	49.95
	1			✓		✓	100.00
kmeans kernel	0	✓		✓	✓		100.00
kmeans swap	0	✓		✓		✓	100.00
sgemm	0	✓		✓	✓		100.00
C2D	0	✓		✓	✓		99.59
	0	✓	✓	✓	✓		0.20
	0	✓	✓	✓		✓	0.20
	0	✓	✓	✓	✓		0.02
	0	✓	✓	✓		✓	0.00

6.5 Vector Codelet Statistics

We show relative execution counts of vector codelets along with their divergence properties in Table 3. The table lists all executed codelets grouped by the continuation kernel they belong to (Section 5.2). Some benchmarks execute multiple OpenCL kernels, for example gaussian Elimination consists of the OpenCL kernels *Fan1* and *Fan2*. The number in Column 2 refers to the ID of the continuation kernel. The continuation kernel with ID 0 is always the first that is executed. The table only shows codelets that were executed at least once.

Columns 3 and 4 show whether the codelet has loops with *uniform* or *divergent* control flow. Columns 5 through 7 show whether the memory accesses in the codelet are *consecutive*, *uniform* or *divergent* as of Section 3.2.2. The last column ultimately shows the relative execution count of the vector codelet among all vector codelets of this continuation kernel.

Convolution2D In the case of Convolution2D, we see that a highly regular vector codelet is used for the bulk of the executions. The vector codelet operates in center of the image and thus only features regular control and memory accesses. This leads to the $5.8\times$ speed-up we observed in our runtime experiments, shown in Figure 12.

Myocyte The myocyte benchmark consists of one OpenCL kernel that specifies two completely distinct tasks. The kernel is spawned for just two work groups that each execute one task with only the first work item performing any work[3]. Polly only partially recognizes these guard conditions, strip-mines one work item loop and moves the scalar guard into a vector codelet. This happens both in the *split* and vec only settings. However, the kernel runs with a *local size* of 2 and so the vector code is never executed, which explains the equivalent runtimes for all settings.

7. Related Work

State-of-the-art vectorizers employ divergence analysis [8, 15] to detect optimizable patterns in memory accesses and control flow. Linear variants of divergence analysis have been integrated with polyhedral loop nest optimizers [18, 29] for SIMD code generation. In these polyhedral approaches, vector memory access patterns are made an optimization target in the cost function of the scheduler. Polyhedral-based vectorizers extract codelets that are then subjected to vectorization. Some OpenCL drivers apply loop switching between work dimensions and other kernel loops to optimize memory access patterns [14, 17]. In all prior work that we are aware of, vector patterns are only optimized and detected if they occur for all inputs and loop iterations in the vectorized function or codelet.

We are to our knowledge the first to split the input space with optimization constraints to drive code specialization. In earlier work [11], input space splitting has been used to improve polyhedral schedules by simplifying the dependence structure of the program.

The codelets extracted by other polyhedral approaches must either be acyclic [18] or contain only affine loops [29] due to constraints of the employed codelet vectorizer. In contrast, we leverage the Intel OpenCL vectorizer [24], which handles arbitrary control flow inside the codelet by predication.

Our implementation partitions OpenCL kernels with barriers into barrier-free continuation kernels [15, 27].

Earlier work in the context of SIMD code generation understood statement splitting as insertion of local divergence tests [16, 26]. Local divergence tests are simpler in that they merely evaluate branch predicates for all lanes and require vector masks to test. Our approach models linear divergence tests in the polyhedral represen-

[3] The kernel contains guard conditions of the form `if (get_local_id(0) == 0) {...}`

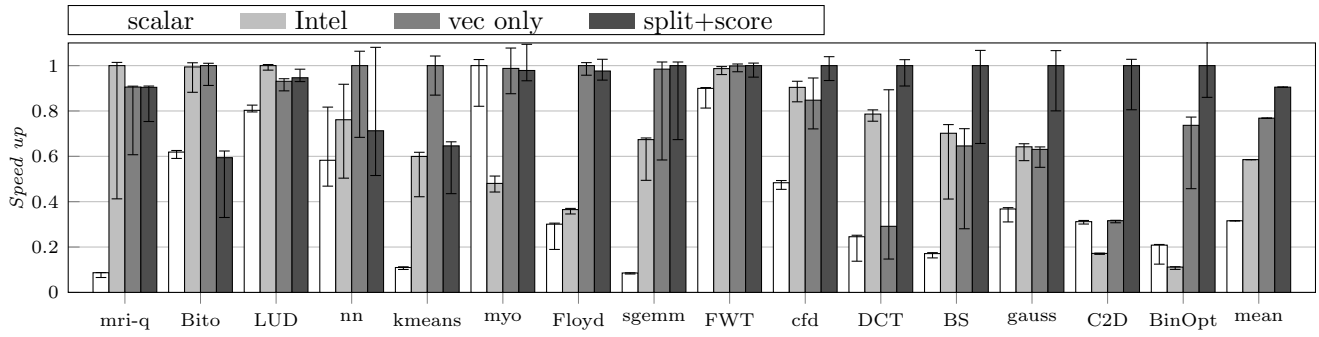


Figure 12. Full optimization setting (opt) compared against Intel, vec only and scalar execution. Median speed ups, normalized to the fastest.

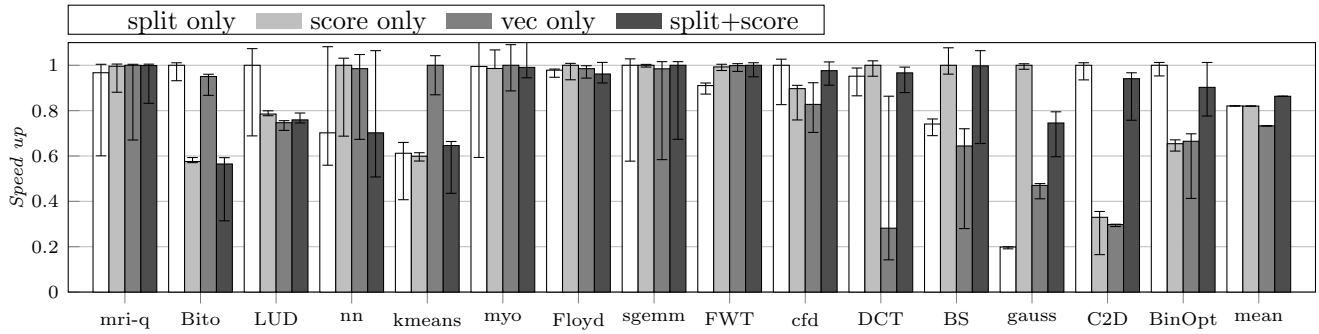


Figure 13. Runtime results, if only splitting (split only) or only subkernel scoring (score only) are applied. Median speed ups, normalized to the fastest.

tation of the kernel. In that representation, the local conditions help to transform the entire kernel function in a holistic way.

Including user-provided domain knowledge into a polyhedral optimization pipeline is, for example, implemented in the pencil language [2].

Runtime variant selection [1, 32], also in a polyhedral setting [23], is an established technique. However, these approaches rely on profiling or user-supplied program variants and heuristics and none of them targets vectorization specifically.

On-the-fly divergence elimination [33] uses data and thread id permutations to reduce control and memory access divergence. Their approach employs a complex runtime system to infer permutations and variant performance at runtime.

In contrast, our approach generates vectorized kernel variants along with the scoring function *automatically* and at *compile time*. Our scoring function could be made more precise using integer point counting [3].

A different approach for device specialization of kernels is auto-tuning [21, 22]. Auto-tuning based approaches profile the kernel runtime under different configurations of performance parameters, such as the work group size, the vectorization width, etc. As auto-tuning specializes the kernel for the entire input space it is as such orthogonal to input space splitting. If the techniques are combined, auto-tuning could, for example, find constant work group sizes that would make the polyhedral kernel representation completely affine.

8. Conclusion & Future Work

In this paper, we have introduced a novel technique to drive code specialization by architecture-specific optimization constraints. The technique specializes kernels for less divergent parts of the input space. To this end, the approach leverages a polyhedral representation of OpenCL kernels.

We have implemented the technique for the AVX SIMD instruction set and evaluated it on a set of OpenCL benchmarks. The results show that our optimization enables effective vectorization where this is not possible for state of the art vectorizers. In particular image processing applications benefit from the handling of boundary conditions.

Our current polyhedral OpenCL kernel representation models a single work group of a barrier-free kernel. Building on these first results, we are improving the representation to include barrier statements and model multiple work groups at once. Future work could investigate applying polyhedral scheduling techniques to OpenCL.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. PLDI '09.

[2] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. van Haastregt, A. Kravets, et al. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. 2015.

[3] A. Barvinok. Lattice points, polyhedra, and complexity. *Geometric Combinatorics, IAS/Park City Mathematics Series*, 13, 2007.

[4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. CC'10/ETAPS'10.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. PLDI '08.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. IISWC '09, .

[7] 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. IISWC '10, .

[8] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. Divergence analysis and optimizations. PACT '11.

[9] R. Das, J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Trans. Comput.*, 44, 1995.

[10] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of while loops. EURO-PAR '95.

[11] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28, 1999.

[12] T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 2012.

[13] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37, 2015.

[14] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43, 2015.

[15] R. Karrenberg and S. Hack. Improving performance of OpenCL on CPUs. CC '12.

[16] A. Kerr, G. Diamos, and S. Yalamanchili. Dynamic compilation of data-parallel kernels for vector processors. CGO '12.

[17] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu. Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures. CGO '15.

[18] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. PLDI '13.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO '04.

[20] S. Moll. Decompilation of LLVM IR, 2011.

[21] N. Moore, M. Leeser, and L. Smith King. Kernel specialization for improved adaptability and performance on graphics processing units (GPUs). PDP '13.

[22] C. Nugteren and V. Codreanu. CLTune: A generic auto-tuner for OpenCL kernels. MCSoC '15, 2015.

[23] B. Pradelle, P. Clauss, and V. Loechner. Adaptive runtime selection of parallel schedules in the polytope model. HPC '11.

[24] N. Rotem. Intel Opencl Implicit Vectorization Module, 2011.

[25] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. PDP '13.

[26] J. Shin, M. W. Hall, and J. Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocessors and Microsystems*, 33, 2009.

[27] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore CPUs. CGO '10.

[28] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, 2012.

[29] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 0, 2009.

[30] T. Vajk, Z. Dávid, M. Asztalos, G. Mezei, and T. Levendovszky. Runtime model validation with parallel object constraint language. MoDeVV '11.

[31] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9, 2013.

[32] M. J. Voss and R. Eigemann. High-level adaptive program optimization with adapt. PPoPP '01.

[33] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. ASPLOS XVI, 2011.