High-Level Hardware Feature Extraction for GPU Performance Prediction of Stencils

Anonymous Author(s)

Abstract

High-level functional programming abstractions have started to show promising results for HPC (High-Performance Computing). Approaches such as Lift, Futhark or Delite have shown it is possible to have both, high-level abstractions and performance, even for HPC workloads such as stencils. In addition, these high-level functional abstractions can also be used to represent programs, and their optimized variants, within the compiler itself. However, such high-level approaches rely heavily on the compiler to optimize programs which is notoriously hard when targeting GPUs.

Compilers either use hand-crafted heuristics to direct the optimizations or iterative compilation to search the optimization space. The first approach has fast compile times, however, it is not performance-portable across different devices and requires a lot of human effort to build the heuristics. Iterative compilation, on the other hand, has the ability to search the optimization space automatically and adapts to different devices. However, this process is often very time consuming as thousands of variants have to be evaluated. Performance models based on statistical techniques have been proposed to speedup the optimization space exploration. However, they rely on low-level hardware features, in the form of performance counters or low-level static code features.

Using the Lift framework, this paper demonstrates how low-level, GPU-specific features are extractable directly from a high-level functional representation. The Lift IR (Intermediate Representation) is in fact a very suitable choice since all optimization choices are exposed at the IR level. This paper shows how to extract low-level features such as number of unique cache lines accessed per warp, which is crucial for building accurate GPU performance models. Using this approach, we are able to speedup the exploration of the space by a factor $2000x$ on an AMD GPU and $450x$ on Nvidia on average across many stencil applications.

GPGPU'20, February 23, 2020, San Diego, CA, USA
2020.

1 Introduction

Recent years have witnessed the emergence of high-level approaches for high-performance computing such as Accelerate [21], Futhark [9], Delite [4], Lift [34] and AnyDSL [18]. They enable programmers to write hardware-agnostic code while putting the burden on the compiler to extract performance. Tuning a compiler is very laborious and time-consuming, especially when considering accelerators such as GPUs (Graphics Processing Units) and this process has to be repeated for every new hardware generation.

Lift proposes to use rewriting [33] to solve this problem. Rewriting for compiler optimizations is an approach first proposed in 2001 in the Haskell compiler [29]. Lift's rewrite rules attempt to define the set of all possible algorithmic and, crucially, hardware-specific optimizations. Rewrite rules liberate compiler writers from having to implement hard-coded optimizations and make it easy to extend the compiler. Optimizations are simply implemented as rules and a generic rewriting engine explores the space automatically.

However, this approach results in a large optimization space. The optimization process takes a few hours for stencils on GPUs [8], even when using an efficient auto-tuner [1]. In response, this paper develops an automatic performance model predicting the best optimized program variant using static features from the high-level Lift IR. This removes the necessity for compiling and running programs which accounts for the majority of the exploration time.

The use of performance modeling for GPUs is not novel [10, 11, 25, 26, 37]. However, to the best of our knowledge, this is the first paper to show how information about low-level GPU-specific features is extractable from a high-level functional IR. This paper demonstrates that a high-level IR is amenable to the extraction of low-level information useful for predicting performance using high-level semantic information. It also shows how cache locality information is extractable at this level. This relies on the use of the rich information stored in the Lift type system together with the ability to reason about array indices in a symbolic manner.

Using the extracted features, a performance predictor is built using machine-learning. This leads to a highly accurate model for the stencil domain, an important class of high-performance code. The model achieves a correlation of 0.8 and 0.9 on GPUs from Nvidia and AMD, respectively. Using the model to search the space requires less than 5 runs in the majority of the cases to achieve performance within 90% of the best available. In comparison, a random search requires 100s of runs in the majority of the cases.

To summarize, the paper makes three contributions:

- It shows how low-level GPU hardware features are extracted from a high-level functional IR;
- It presents a simple machine-learning model that predicts program performance;
- It shows that the model is able to drastically reduce exploration time of the optimization space.

The rest of this paper is organized as follows: Section 2 motivates this work while Section 3 presents background information about OpenCL and Lift. Section 4 explains how low-level hardware

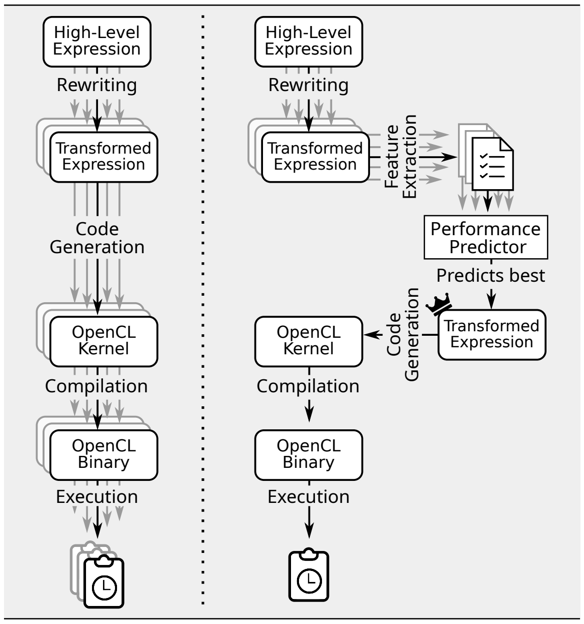


Figure 1. LIFT compilation and exploration. a) The current approach compiles and executes all transformed expressions. b) The new strategy ranks the transformed expressions with a model and only compile and execute the best ones.

features are extracted from the high-level LIFT IR and Section 5 presents the performance model. Section 6 analyses the features and the model performance while Section 7 shows that the model is able to speedup drastically the optimization space exploration. Finally, Section 8 discusses related work and Section 9 concludes.

2 Motivation

Current LIFT Exploration LIFT [33] explores the GPU optimization space using rewrite rules. Figure 1a presents an overview. First, a high-level expression representing the program is used as an input to the compiler. This generic high-level expression does not encode any optimizations. Then, the rewriting takes place and the LIFT exploration module applies rewrite rules to search the space randomly. This results in a set of *transformed* expressions where optimizations have been applied and parallelism has been mapped.

The transformed expressions are then fed into the LIFT code generator which produces OpenCL kernels. These kernels are compiled with the vendor-provided OpenCL compiler into binaries. Finally, all binaries are executed, the performance is recorded and the best found kernel is reported.

This process is time consuming as it produces a large number of kernels (1,000 for this paper). In addition, every LIFT generated kernel is executable with a different number of threads leading up to 10,000 kernel executions.

Time breakdown Figure 2 shows the percentages of the time spent in the different stages of the current LIFT compilation and exploration. Unsurprisingly, the last part of LIFT's workflow, the kernel execution, requires by far the most time (up to 90%). For this paper, executing all kernels for a single application, including the

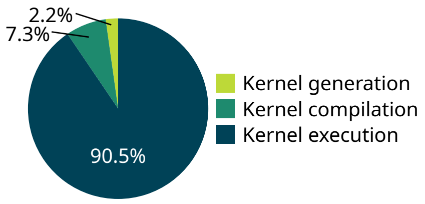


Figure 2. Time breakdown for the LIFT exploration process. *Kernel generation* includes time to rewrite and compile LIFT expressions to OpenCL kernels. *Kernel compilation* is the vendor-provided OpenCL compiler time. *Kernel execution* is the time required to execute all generated kernels.

exploration of thread configurations took up to 41 minutes while all kernels were generated in less than a minute, which is about 2% of the overall time.

Using a Performance Predictor for Exploration The major bottleneck for exploration is clearly the OpenCL compilation and execution time of the generated kernels, which represent 98% of total time. This paper addresses this bottleneck by using a trained performance predictor directly on the transformed LIFT expression. Figure 1b shows how the exploration strategy is modified with a performance model.

Once the transformed expressions have been produced, the idea is to extract features that are informative about performance. These features are fed into a predictive model which almost instantaneously ranks the transformed expressions. Then, the transformed expression with the fastest predicted performance is selected, the corresponding kernel generated, compiled and finally executed.

While this approach seems very simple, the challenges are twofold. First, we need to identify features that are informative about performance, such as memory access patterns. Then, they need to be extracted from the high-level functional LIFT IR. As we will see, the LIFT IR encodes all the required information to calculate low-level GPU-specific features. The next section gives background information about the LIFT IR while Section 4 will discuss feature extraction.

3 Background

This section introduces OpenCL, the existing LIFT IR and the rewrite systems used to produce efficient OpenCL kernels.

3.1 OpenCL

An OpenCL program (*kernel*) is executed by multiple threads (*work-items*) organized in *work-groups*, providing a two-level thread hierarchy. Both work-items and work-groups are organized in three dimensional grids identified by unique IDs. On GPUs, multiple work-groups are executable on a core and work-items are scheduled in groups of 32 for Nvidia (warp) or 64 for AMD (wavefront). It is generally desirable to start a large thread number to reach maximum *occupancy*.

OpenCL provides a three-level memory hierarchy: *Global memory* is accessible by all work-items and throughput is maximized when threads in the same warp/wavefront access the same cache line (coalesced accesses). Work-items of the same group communicate via a fast shared *local memory* and each work-item has its own *private memory*.

3.2 Lift IR

Lift [33, 34] is a functional language based on lambda calculus, offering a small set of reusable primitives. It is a compiler-internal data-parallel intermediate language and is compiled to high-performance OpenCL code. Lift's distinguishes algorithmic primitives which express *what* to compute, from OpenCL-specific primitives which express *how* to compute by explicitly mapping computations to the OpenCL programming model. Lift's type system supports scalar types (*e.g.* `int`, `float`), tuple-types (denoted as $U \times T$) and array-types (denoted as $[T]_n$) where the array size n is part of the type.

Algorithmic Primitives Lift provides well-known functional primitives defined on arrays as listed below:

$$\begin{aligned}
\textbf{\textit{map}} &: (f : T \to U,\ in : [T]_n) \to [U]_n \\
\textbf{\textit{reduce}} &: (init : U,\ f : (U, T) \to U,\ in : [T]_n) \to [U]_1 \\
\textbf{\textit{zip}} &: (in1 : [T]_n,\ in2 : [U]_n) \to [T \times U]_n \\
\textbf{\textit{iterate}} &: (in : [T]_n,\ f : [T]_n \to [T]_n,\ m : \texttt{int}) \to [T]_n \\
\textbf{\textit{split}} &: (m : \texttt{int},\ in : [T]_n) \to [[T]_m]_{n/m} \\
\textbf{\textit{join}} &: (in : [[T]_m]_n) \to [T]_{m \times n} \\
\textbf{\textit{slide}} &: (size : \texttt{int},\ step : \texttt{int},\ in : [T]_n) \to [[T]_{\text{size}}]_{\frac{n-\text{size}+\text{step}}{\text{step}}} \\
\textbf{\textit{pad}} &: (l : \texttt{int},\ r : \texttt{int},\ h : (i : \texttt{int},\ \text{len} : \texttt{int}) \to \texttt{int}, \\
&\quad\ in : [T]_n) \to [T]_{l+n+r} \\
\textbf{\textit{at}} &: (i : Cst,\ in : [T]_n) \to T \\
\textbf{\textit{get}} &: (i : Cst,\ in : T_1 \times T_2 \times \ldots) \to T_i \\
\textbf{\textit{array}} &: (n : \texttt{int},\ f : (i : \texttt{int},\ n : \texttt{int}) \to T) \to [T]_n \\
\textbf{\textit{userFun}} &: (s1 : ScalarT,\ s2 : ScalarT', \ldots) \to ScalarU
\end{aligned}$$

Lift supports the definition of arbitrary scalar-based sequential OpenCL-C function called *userFun*. These are directly embedded in the generated OpenCL code.

OpenCL-specific primitives Lift's OpenCL specific primitives expose OpenCL's thread and memory hierarchy. These primitives are used to explicitly dictate how to perform the computation expressed with the algorithmic primitives.

Parallelism is exposed via specialized variations of *map*: $mapGlobal_d$, $mapWorkgroup_d$, $mapLocal_d$ and *mapSeq*. These primitives directly correspond to OpenCL's thread hierarchy. The computation specified within a OpenCL-specific *map* is performed by it's particular level and dimension $d \in \{0, 1, 2\}$ of the thread hierarchy, or executed sequentially by a single thread (*mapSeq*). OpenCL's memory hierarchy is exposed via *toGlobal(f)*, *toLocal(f)* and *toPrivate(f)*, which specify where the output of the function f is stored in memory.

3.3 Rewriting

Lift encodes optimizations as semantics-preserving rewrite rules. These rules are used to transform a high-level expression written using the algorithmic primitives into a transformed expression in which parallelism and memory is explicitly exploited. Similar to Lift's primitives, rewrite rules are also categorized into algorithmic or OpenCL-specific rules. Algorithmic rules such as the divide-and-conquer rule:

$$map(f) \to join \circ map(map(f)) \circ split(n)$$

```
stencil(arg: [float]_N) =
  map(reduce(+,0), slide(3,1, pad(1,1, 0, arg)))
```

Listing 1. 1D 3pt-stencil example in Lift.

```
transformedStencil(arg: [float]_N) =
  mapWrg(tile =>
    mapLcl(toGlobal(reduce(+,0)), slide(3,1,
      mapLcl(toLocal(id, tile))))
  )(slide(18,16, pad(1,1, 0, arg)))
```

Listing 2. 1D transformed 3pt-stencil example in Lift.

Type	Feature
Parallelism	global size (dimensions 0, 1 and 2) local size (dimensions 0, 1 and 2)
Memory	amount of local memory allocated global stores per thread global loads per thread local stores per thread local loads per thread average cache lines per access per warp
Control Flow & Synchronisation	barriers per thread if statements per thread for loop bodies executed per thread

Table 1. List of extracted features

create a space of possible algorithmic implementations for the same expression. OpenCL-specific rules such as:

$$map(f) \to mapGlobal_0(f)$$

map expressions to the OpenCL's programming model.

3.4 Example

Listing 1 shows a 1D 3-point stencil expressed in Lift [8]. *pad* is applied adding one element (`0`) to the left and right of the input array `arg` to implement a simple boundary handling. *slide* creates overlapping neighborhoods of three elements which are summed up using *map* and *reduce*.

Applying rewrite rules leads to Listing 2, where overlapped tiling has been applied. Every tile is processed by a work-group (*mapWrg*) loading all elements to local memory and computing the output using its work-items before storing it in global memory. From this expression high-performance OpenCL code is generated as shown in [8].

4 Feature Extraction

This paper proposes a performance model that predicts the performance of transformed Lift expressions on GPUs in order to identify the best variant. The model relies on static features extracted from the high-level Lift IR. Although the features are extracted at a high-level, they capture information about low-level hardware features. They broadly fall into three categories as seen in Table 1.

4.1 Parallelism

For a fixed input size, the number of threads launched influences how much parallelism versus sequential work is performed. We include both global and local thread counts across the three thread dimensions as features. Local thread count affects how large each work-group will be, which may affect data reuse or the number of concurrent groups.

4.2 Memory

This section covers the features related to the amount of memory allocated, number of accesses and access patterns.

4.2.1 Local memory usage

One of the important factors that determines performance on a GPU is occupancy. Occupancy is typically maximized when multiple work-group execute concurrently. More concurrent work-groups typically translates to more threads executing concurrently, which ultimately helps hiding memory latency.

The number of work-groups that execute simultaneously on a core depends on the amount of resources used by each work-group. One important resource is the amount of fast local memory (shared memory) used by the work-group. Therefore, it is crucial to determine this quantity.

Extracting the amount of local memory used in a Lift program is straightforward. The program is traversed once, collecting memory allocation sizes and summing up these numbers.

```
   input  :Lambda expression representing a program
   output :Numbers of different types of memory accesses.
  countAccesses(lambda)
1  totalLoad[local] = 0; totalLoad[global] = 0
2  totalStore[local] = 0; totalStore[global] = 0
3  countAccessesExpr(lambda.body, 1)
4  return {totalLoad,totalStore}
  countAccessesExpr(expr, iterationCount)
5  switch expr do
6    case fc@FunCall
7      foreach arg in fc.args do
8        countAccessesExpr(arg, iterationCount)
9      switch expr.f do
10       case is l@Lambda
           countAccessesExpr(l.body, iterationCount) ;
11       case is t@toPrivate or t@toLocal or toGlobal
12         countAccessesExpr(t.f.body, iterationCount)
13       case is m@MapSeq or m@MapGlb or m@MapLcl or ...
14         n = fc.input(0).length
15         countAccessesExpr(m.body, iterationCount * n)
16       case is it@Iterate
17         countAccessesExpr(it.body, iterationCount * it.count);
18       case is uf@UserFun
19         foreach arg in fc.args do
20           totalLoad[arg.addrsSpace] += iterationCount
21         totalStore[arg.addrsSpace] += iterationCount
22       otherwise do // Nothing to count ;
23   otherwise do // Nothing to count ;
24 return counts
```

Algorithm 1: Pseudo-code for counting the total number of loads/stores for each type of memory.

4.2.2 Number of Memory Accesses

Performance is largely affected by the amount and type of memory operations. Applications that exhibit large amount of data re-usage will benefit from exploiting the fast local memory. The program can simply reuse the data in local memory several times, reducing the number of global memory accesses, resulting in increased performance.

```
1 example(arg_0: [float]_N, arg_1: [float]_N) =
2   mapWrg(x =>
3     mapLcl(toGlobal(multByTwo), mapLcl(toLocal(add)), x)
4   )(split(64, zip(arg_0, arg_1)))
```

Listing 3. Example for memory access count extraction.

Algorithm The Lift code generator only produces loads and stores to memory when a user function is called. Therefore, counting the number of loads and stores boils down to counting how often each user function is called. As can be seen in Algorithm 1, a depth-first traversal is performed on the IR while keeping track of the number of times the body of patterns generating loops is executed. Once a user-function is reached, the feature extractor simply updates the total number of loads and stores. In addition to this, the extractor keeps track of the type of memory being accessed, local or global, using the *toLocal* and *toGlobal* patterns. The information about the address space is encoded directly into the IR and is populated by another pass that runs prior to feature extraction. The number of global/local loads and stores is then normalized by the number of total threads.

Example Consider the program in Listing 3. The algorithm starts with the top-level lambda and soon encounters the *mapWrg* primitive. At this point in the algorithm, line 14, n will be $N/64$ (the length of the outer dimension of the input after the *split*). The algorithm calls recursively *countAccessesExpr* with $N/64$ as the *iterationCount*. When visiting either of the *mapLcl* in line 3 of Listing 3, n will this time be 64 (the length of the inner dimension of the input after the *split*).

When the add function is visited, global loads is updated twice, since the add function has two inputs (the tuple is automatically unboxed). Since at this point, the iterationCount is $N/64 * 64 = N$, the total number of global loads is $N * 2$ and the total number of local stores is N. When the multByTwo function is visited, local reads and global store are both updated once, resulting in N local loads and N global stores.

4.2.3 Memory Access Patterns

The way a program accesses memory has a profound impact on performance. GPUs coalesce several memory requests into a single one when threads in the same warp/wavefront access a single cache line (typically 128 bytes). It is, therefore, important to extract information about memory access patterns for building an accurate performance predictor.

General Algorithm To determine the total number of cache line reads, the feature extractor recursively traverses the IR, keeping track of the iteration count. When a memory access is encountered, it determines the number of unique cache lines accessed by the warp as follows. First, it generates the actual index expression using the existing mechanism of the Lift compiler [34]. If the expression contains no thread id, it means all the threads are accessing the same cache line.

When the expression contains a thread id, a new index expression is generated for each thread in the warp by adding a constant to its id (threads in a warp have consecutive ids). Let's denote the original array index expressed as a function of the thread id as $access(tid)$. Given n, the number of threads in a warp, the set of array indices

```
example(in: [float]_N) = mapGlb(mapSeq(f), split(n, in))
```

Listing 4. Example for extracting memory access patterns.

accessed by the warp is:

$$\{access(tid+0), access(tid+1), \cdots, access(tid+n-1)\}$$

This list of indices expresses the different addresses accessed by a warp. Given the cache line size s (expressed as a multiple of data size), we compute the list of cache lines accessed:

$$\{access(tid+0)/s, access(tid+1)/s, \cdots, access(tid+n-1)/s\}$$

Finally, we can subtract the elements in the list with each other to identify which ones are equal (when the subtraction results in 0) and count the number of unique accesses.

Implementation details The approach explained above is conceptually correct, however, it relies on having the ability to simplify symbolically arithmetic expressions. While the Lift arithmetic simplifier supports a significant set of simplifications, it is not powerful enough to deal with some simplifications. In such cases the feature extractor might fail to recognize identical accesses. The following paragraphs explain a few workarounds used inside the feature extractor.

The first issue we encountered, is the difficulty in calculating the set of unique cache lines by subtraction. Conceptually, one could take the first access $access(tid+0)/s$, subtract every other accesses by it and hope that the algebraic simplifier would be able to return 0 in case where two accesses are identical. Simplifying expressions as simple as

$$(tid+0)/s - (tid+1)/s$$

which is 0 when $s > 1$, is far from trivial given that / represents the integer division.

To overcome this challenge, we modify our approach slightly and add an extra step. Before dividing by s, we first calculate all the relative array accesses as an offset of the first access by simple subtraction. The intuition behind this two-fold. First, it is much easier to simplify a subtraction if it does not contain terms with integer division. Secondly, we only care about the distances between the accesses rather than their absolute location, therefore, we will still be able to identify the number of unique cache line accessed.

So if the original accesses are

$$\{tid+0, tid+1, \cdots\}$$

they become

$$\{(tid+0)-(tid+0), (tid+1)-(tid+0), \cdots\}$$

which simplifies trivially to $\{0, 1, \cdots\}$. Then, we perform the division as before, which leads to $\{0/s, 1/s, \cdots\}$ which trivially simplifies to $\{0, 0, \cdots\}$. Now it is much easier to identify the unique cache lines.

Example Consider the example program from Listing 4. The array index being read for the argument of f is `i + n * gl_id` where i is the iteration variable of the *mapSeq* and `gl_id` the global thread id. Depending on the split factor n, a different number of cache lines will be accessed by a warp. With a split factor of $n = 1$, a single cache line would be accessed since the accesses within a warp are consecutive. However, if the split factor is larger than the warp size, then each warp will be touching a different cache line.

```
stencil(input: [float]_N) =
  MapGlb(ReduceSeq(+, 0.0f),
    Slide(3, 1,
      Pad(1, 1, Clamp, input)))
```

Listing 5. Example for a simple stencil program.

With a cache line of 32 words, 32 threads per warp and 1 word for float, the cache line indices within a warp are:

$$\{(i+n*gl\_id), (i+n*(gl\_id+1)), \cdots, (i+n*(gl\_id+31))\}$$

Using the trick presented earlier, we can express all indices as an offset from the first one:

$$\begin{gathered}\{(i+n*gl\_id)-(i+n*gl\_id),\\ i+n*(gl\_id+1)-(i+n*gl\_id),\\ \cdots,\\ i+n*(gl\_id+31))-(i+n*gl\_id)\}\end{gathered}$$

which simplifies trivially to: $\{0, n, \cdots, n*31\}$. Now dividing by the cache line size, we obtain $\{0, n/32, \cdots, n*31/32\}$.

If the split factor n is 1, this results in 32 zeros, meaning all the thread in the warp access a single cache line. When the split factor $n = 4$, this will results in the following list: $\{0, 0, 0, 0, 1, 1, 1, 1, \cdots, 7, 7, 7, 7\}$. Since it has 8 unique values, the warp touches 8 cache lines for this memory access.

4.3 Control Flow and Synchronisation

Another important factor that often limits performance for GPUs is control flow and synchronization. *if-then-else* and *for loop* statements produce branching instructions which is notoriously bad for GPU performance. Similarly, barriers are detrimental to performance since execution is alted until all threads have reached the barrier. For this reason, the feature extractor determines the total number of *if-then-else*, *for loops* and barriers produced by the code generator.

Algorithm This is similar to the algorithm used to count the number of memory operations. It traverses the IR recursively, keeping track of the number of times each function is executed. Whenever a pattern that might produce a loop (*e.g. iterate*, *mapLocal*, *reduceSeq*) is encountered, it checks whether a loop will be emitted and update a global loop counters, taking into account the current iteration count.

The algorithm also detects special cases where loops might not be emitted. There are two cases to consider. First, when a *mapSeq* iterates over an array of size 1, it is clear that a loop is not required. The second case is more subtle and involves *mapLocal*, *mapWrg* or *mapGlobal*. If the size of the input array is smaller than the number of local threads, workgroups or global threads, respectively, the code generator will emit a if-then-else statement instead of a loop since the loop can at most be executed once per thread or workgroup.

To determine the number of barriers, the algorithm looks at *mapLcl* as OpenCL only has barriers inside workgroups. The Lift code generator detects unnecessary barriers [34] and tags the call to *mapLcl* when it is not required. Therefore, we run this barrier elimination pass before feature extraction and use this information to ignore the *mapLcl* which have been marked as not requiring a barrier.

```
kernel void stencil (float* in, float* out, int N){
  float acc;
  for (int gid=global_id(); gid<N; gid+=global_size()) {
    acc = 0.0f;
    for (int i = 0; i < 3; i += 1) {
      int pos = gid - 1 + i;
      acc += in[( (pos >= 0) ? (
                  (pos < N)  ? pos : (N - 1) ) : 0 )]; }
    out[gid] = acc; }}
```

Listing 6. OpenCL-ish code generated for a simple stencil.

4.4 Use of High-Level Semantic Information

Another practical issue has to do with the *pad* pattern which is used to implement boundary conditions in stencil programs. Listing 5 shows a simple stencil program applying a clamping boundary condition which simply repeats the outermost value in case of out-of-bounds accesses. Listing 6 shows the generated pseudo-OpenCL code for this program. The *pad* pattern introduces a lot of ternary operators `?:` which check that every memory access is in bound. This operator makes it harder for the simplifier to subtract memory accesses with each other to identify unique cache lines.

To overcome this, we exploit the high-level semantic information available: the padded data is rarely accessed and most accesses are in bound. The feature extractor focuses on the common case by simply ignoring the ternary operator and calculate the index for the common case. Identifying the common case by statically analyzing the OpenCL code is much harder even for this simple example. We would have to predict the common case for two ternary operators whos predicates depends on two opaque function calls (`global_id` and `global_size`) to the OpenCL library.

4.5 Summary

This section has shown how low-level GPU-specific features are extracted from the Lift IR. Memory-related, control flow and synchronization features, are extracted using information about the length of arrays from the type. We have seen how the fine-grained memory feature related to cache lines accesses is computed using the power of the Lift symbolic arithmetic expressions. The next section explains how we build a simple performance model using these features.

5 Performance Model

Having seen how hardware-specific information is extracted from the high-level IR, we now focus on the performance model. It is based on k-Nearest Neighbors (kNN), which makes prediction based on the distance between programs in the feature space. Intuitively, Lift programs that exhibit similar features are likely to have similar performance.

5.1 Output Variable

The prediction output is throughput normalized by the maximum achievable per input/program. This is to ensure that performance is comparable across programs since different programs might exhibit different number of operations.

5.2 Principal Component Analysis

Given that a kNN model works best with a small number of features, we use PCA (Principal Component Analysis) to reduce the dimensionality of the feature space. Prior to applying PCA, the features are centered and reduced with a mean of 0 and a standard deviation of 1. This step is necessary since our features have very different ranges of values. PCA is then applied and we retain the principal components that explain 95% of the variance. In effect, this compresses the feature space by removing redundant features.

5.3 K-Nearest Neighbors Model

A k-nearest neighbors model makes a prediction of a new data point by finding the k closest points to it, using Euclidean distance and averaging their responses to make a prediction. In our case, the distance metric is determined by how close the feature vectors are from one another.

The kNN model does not require any special training. The execution time of rewritten Lift expressions, together with their features, are simply collected and added into a database. When predicting a newly unseen Lift program, we simply look up the k closest neighbors and average their prediction to form a new prediction. In our experiment, we used $k = 5$.

5.4 Making Predictions

To make a prediction about new programs, we first collect data points from a group of training programs. For each program, we conduct an exploration of their optimization space and store the features and corresponding performance. Given a new program, we proceed as follows:

1. For each rewritten program:
   a. The features are extracted, normalized and projected based on the PCA calculated from the training data;
   b. The model predicts the performance using the average of the k-nearest neighbors.
2. The different rewritten programs are sorted based on the prediction.
3. The fastest predicted rewritten program is generated, compiled and executed.

6 Experimental Setup

Platform The setup consists of two GPUs, an NVIDIA Titan Black and an AMD Radeon R9 295X2. The Nvidia platform uses driver version 367.35 and OpenCL 1.2 (CUDA 8.0.0). The AMD platform uses OpenCL 2.0 AMD-APP (1598.5).

Benchmarks and Space We use the 2D stencil benchmarks from [8] listed in Table 2. All experiments are performed using single floating point with matrix sizes from 512^2 to 8192^2.

Model evaluation The performance model is evaluated using leave-one-out cross-validation, the standard machine learning methodology. When evaluating performance on a given benchmark, the traning data consists of all the data collected from all benchmarks, except the one being tested.

7 Feature and Model Analysis

Before looking at how the performance model is used to speedup the optimization space exploration, we first perform an analysis of the features and evaluate the model accuracy.

7.1 Features Analysis

We use the redundancy metric to analyze which features are the most informative about performance:

$$R = \frac{I(X,Y)}{H(X) + H(Y)}$$

Benchmark	Points	Points Used	# grids
Stencil2D	9	9	1
SRAD1	9	5	1
SRAD2	9	3	2
Hotspot2D	9	5	2
Gradient	9	5	1
Jacobi2D 5 pt	9	5	1
Jacobi2D 9 pt	9	9	1
Gaussian	25	25	1

Table 2. Stencil benchmarks used in the evaluation.

The redundancy metric normalizes the mutual information by the sum of the entropy of the two variables. This ensures that different features can be compared with one another. In our case, we are interested in comparing each feature with the output we wish to predict: performance. A higher value between a certain feature and the output indicates that the feature is useful for performance prediction.

Figure 4 shows the normalized mutual information between features and performance. As expected, one of the most important features is the average number of cache lines accessed per warp. This feature, which represents locality, is extremely important for stencil benchmarks.

The next most important feature for both machines is the globalSize in dimension 1. This feature is directly related to the number of threads that execute and, therefore, the amount of parallel work performed. It is also used to determine if the kernels are launched using a 2D or 1D iteration space (in the 1D case, the globalSize1 will be 1). Then, comes the number of global stores, followed closely by the number of global loads. This basically corresponds to the number of memory accesses performed into the slow global memory.

For both platforms, barriers and control flow (for loops) do seem to have only a medium impact on performance, whereas the number of if-statements does not seem very relevant at all. Focusing on the least important features, the number of local loads does not seem to affect performance much. We conjecture that, since local memory is anyway very fast, having fewer or more local loads might not make much of a difference in terms of performance, especially compared to the number of global memory operation.

7.2 Benchmark diversity

Figure 3 shows the features of the best point in the space for our benchmarks. As can be seen, some benchmarks share similarities, which is essential for being able to make prediction. However, we also observe quite a lot of diversity.

7.3 Performance Model Correlation

We analyze correlation between the predicted and actual values to measure the model ability at distinguishing between good and bad points. For all programs, the correlation coefficient is in the range $[0.7 - 0.9]$, with average of 0.9 on Nvidia and 0.8 on AMD, which shows the predictor works

7.4 Summary

This section has shown that the most important features for performance prediction on GPUs are related to memory access pattern, amount of parallelism and number of global memory accesses. The section has also shown that the model's predictions correlate highly with actual performance. The next section shows how the model is used to speedup the optimization space exploration of our benchmarks.

8 Optimization Space Exploration

8.1 Model-based Exploration

This section shows the performance achieved when exploring the space with the predictor. The exploration is conducted by generating 1,000 transformed Lift expression using rewrites and combining them with 10 different thread-counts on average. This leads to 10,000 design point per program/input pair. For each point, we extract the features and use the model to rank them. We then run the design points from highest predicted performance to lowest.

Figure 5 shows the normalized best performance achieved as a function of the number of points evaluated. It also shows the performance achieved using a purely random evaluation order. Using the predictor, it is possible to very quickly achieve 100% of the performance available in the space for all programs. In comparison, the random strategy struggles to reach even 50% of the performance available in some cases after having explored 3% of the whole space.

8.2 Space Exploration Speedups

Figure 6 shows the exploration speedup when using our model compared to random to achieve 90% of the available performance. The speedup is shown in terms of number of samples and total time it takes to run them. A speedup of $10x$ means the performance model needs $10x$ less runs, or $10x$ less time, than random to achieve 90% of the performance.

As can be seen, using the performance model brings large speedup across all programs. When looking at total number of runs required, on Nvidia, the performance model approach requires $35x$ less runs than random. On AMD, there is an even bigger savings, since the model requires $77x$ less runs than random. When it comes to total time, the model based approach is a staggering $450x$ and $2000x$ faster than random for Nvidia and AMD respectively.

8.3 Detailed Results

This last section shows more details per program/input. Figure 7 shows the actual number of runs required to reach 90% of the performance across programs and input. As can be seen, only one run is necessary in the majority of the cases for Nvidia and two for AMD. In contrast, random needs over 60 runs for Nvidia and over 180 for AMD in most cases.

The average number of runs using the model is 3 for Nvidia and 5 for AMD. In comparison, random requires on average 97 runs for Nvidia and 240 for AMD. These results clearly shows that the performance model is working extremely well in the majority of the cases.

Interestingly, there are a couple of outliers programs/input size combination that requires over 30 runs for the model-based approach. In both cases, stencil2d on Nvidia and srad1 on AMD, this is when the largest or smallest input sizes are used. We believe that in such cases, the behavior of these programs probably changes drastically with the input size. For instance, the data might actually fit entirely in the cache for the smallest input size of stencil2d and, therefore, change drastically the behavior of the application for this input size. Since our features have no notion of working-set size, the model might be unable to pick up this change of behavior.

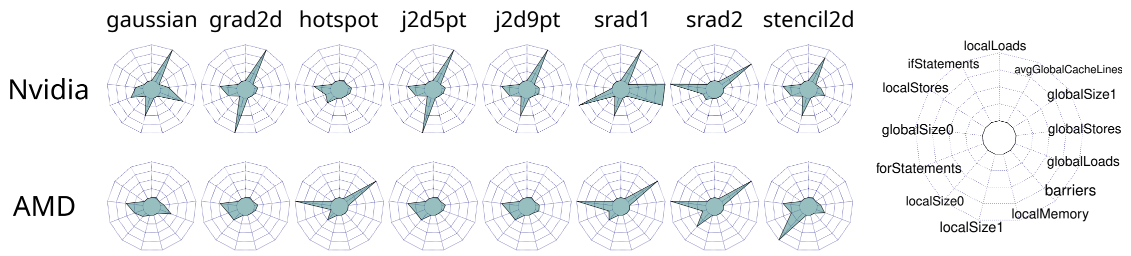
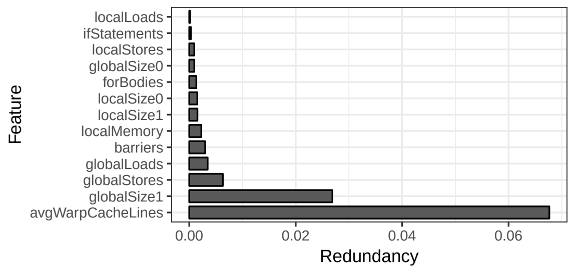
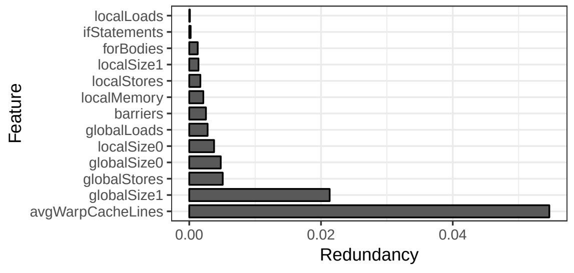


Figure 3. Radar plot of the features for the top points in the space (input sizes 4K).



(a) NVIDIA



(b) AMD

Figure 4. Normalized mutual information (redundancy) between each feature and performance.

However, even in such cases, the model-based exploration is still ahead of random. For stencil2d, the model needs 31 runs while random needs 691, a 21x speedup!

9 Related Work

Auto-Tuning OpenTuner [1] is a framework for domain-specific multi-objective auto-tuners. CLTune [24] is a generic auto-tuner for OpenCL kernels. ATF [30] is a language-independent auto-tuning framework which supports inter-parameter constraints. These auto-tuning approaches attempt to find good implementations using online search which is orthogonal to our approach. In fact, auto-tuners can be easily coupled with a performance predictor.

Analytical Performance Modelling CuMAPz [15] is a compile time analysis tool that helps programmers increase the memory performance of CUDA programs. It estimates the effects of performance-critical memory behaviours, such as data reuse, coalesced accesses, channel skew, bank conflict and branch divergence. GROPHECY [22] uses the MWP-CWP model [11] (Memory Warp Parallelism – Computation Warp Parallelism) to estimate the GPU performance of skeleton-based applications. GPUPerf [32] is an enhanced version of the analytical MWP-CWP model with added metrics and a way of understanding performance bottlenecks. The boat hull model [25] is a modified version of the roofline model that is based on an algorithm classification and produces a roofline model for each class of device.

GPU cache models [26] have been built by extending reuse distance theory with parallel execution, memory latency, limited associativity, miss-status holding-registers and warp divergence. COMPASS [17] introduces a language for creating analytical performance models that analyze the amount of floating point and memory operations based on static code features. Coloured petri nets [19] have been proposed for GPGPU performance modelling. Another approach [3] builds an analytical performance model to determine the lower bound on execution time. Low-level GPU ISA solving and assembly microbenchmarking [37] has been used to collect data about architectural features and performance.

Sensitivity Analysis via Abstract Kernel Emulation [10] aims to predict execution time and determine resource bottlenecks for a given Nvidia GPU kernel binary.

Analytical models describe low-level details of the hardware to model performance using a model written by a hardware expert. They typically use low-level kernel representations to make their predictions. In contrast, our approach based on machine-learning is fully automatic and works by extracting features at a much higher level.

Statistical Performance Modelling Early work [7] extracts static code features and uses machine learning to predict the performance of optimization sequences.Principal component analysis, cluster analysis and regression modelling have been used [14] to generate predictive models for GPUs and CPUs. Predictive modelling has also been applied in polyhedral compilation [28] to predict speedups for different combinations of polyhedral transformations from hardware performance counters. Graph-based program characterization [27] has also been used for polyhedral compilation to predict the speedups of optimization sequences. Clustering on similarity of a graph-based intermediate representation has been used [6] to cluster similar programs. Another approach [35] uses machine learning models trained on assembly level features to choose a good combination of transformations for vectorization.

All these approaches use hardware counters, low-level code features, assembly-level features or compiler data structures to predict speedups or optimization sequences. In contrast, our work shows how we can extract features at a much higher-level and still predict performance accurately.

MaSiF [5] uses PCA and kNN to auto-tune skeleton parameters for programs written using TBB and FastFlow. Stargazer [12] uses step-wise linear regression together with cubic splines to estimate the performance of programs on different GPU designs in GPGPU-Sim [2]. Starchart [13] uses random sampling and building

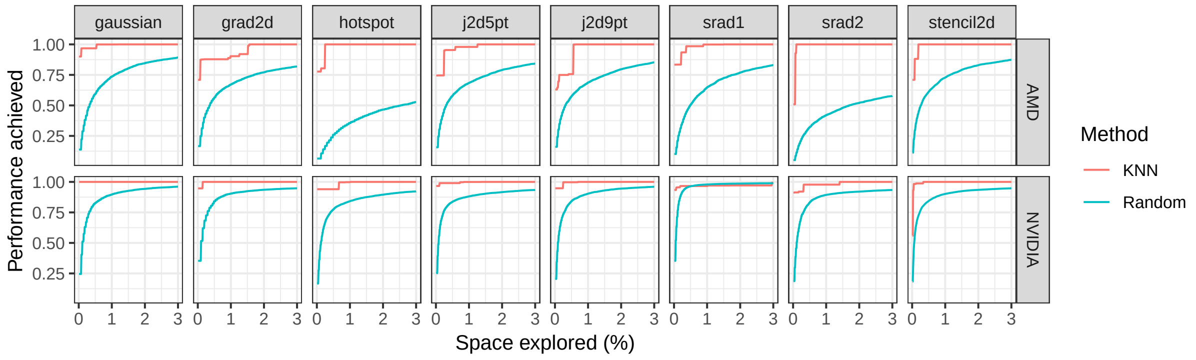
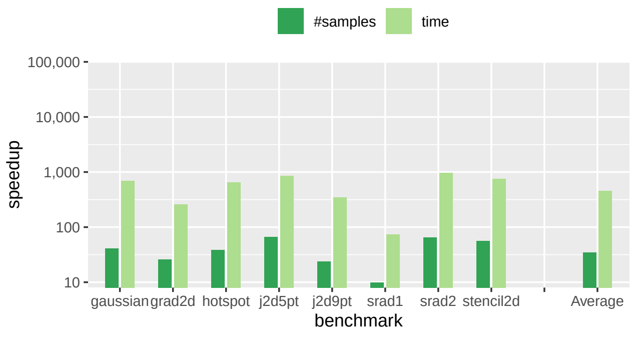
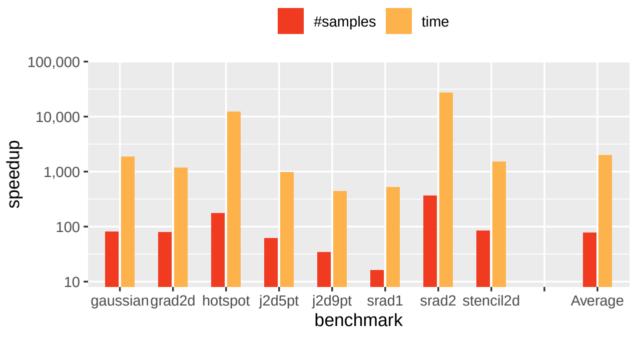


Figure 5. Achieved performance when exploring the space for a 4K input size using a model trained on other programs.



(a) NVIDIA



(b) AMD

Figure 6. Reduction in the number of samples and corresponding time required to explore to reach at least 90% of the available performance (average = geometric mean).

regression trees to divide the whole optimization space into smaller subspaces.

These approaches try to directly predict the effect tunable parameters have on the performance. However, they rely on the fact that the number of parameters is fixed and known in advance. In contrast, our approach predicts the performance independently of the number of parameters in the program.

Artificial Intelligence for Compilers Genetic programming has been used [16] to generate features for predicting loop unrolling factors. Others [23] have proposed ways of generating program features out of simple ones. Features are encoded as numeric relations and new ones are generated by joining existing relations and aggregating them.

Support Vector Machines have also been used in compilers [31]. Machine learning has also been used to automatically learn compiler heuristics. [36] A neural-network cascade [20] is used to determine the amount of thread coarsening to apply to OpenCL programs for different GPUs.

Machine learning models in compilers traditionally use features extracted from a deeper stage in the compilation pipeline. Our work instead extracts them at a considerably higher-level from a functional IR.

10 Conclusions

This paper has demonstrated that it is possible to extract low-level hardware-specific features from the Lift high-level functional IR. We have seen how type information, such as array length, is useful for computing certain features. The ability to reason symbolically about array indices also enables the extraction of very fined-grained features such as the number of accessed cache lines per warp. To the best of our knowledge, this is the first time a paper has shown how low-level features can be extracted at such high level, without requiring any profiling or performance counters.

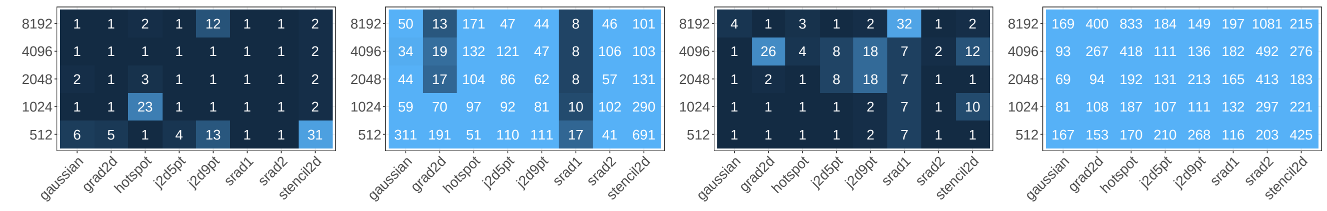
The paper has also demonstrated how a simple performance model is built to make accurate performance predictions about different program variants. Using an Nvidia and AMD GPUs, and stencil applications, we have seen that the model is able to predict points in the space that are within 90% of the best within one or two runs in the majority of the cases. When compared to a random search strategy, the model requires on average 77*x* less runs than random on AMD and 35*x* less on Nvidia which translates to time savings of 2000*x* and 450*x* respectively.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *PACT*. ACM. https://doi.org/10.1145/2628071.2628092

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*. IEEE. https://doi.org/10.1109/ISPASS.2009.4919648

[3] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization Space Pruning Without Regrets. In *CC*. ACM. https://doi.org/10.1145/3033019.3033023



(a) KNN on Nvidia GPU **(b)** Random on Nvidia GPU **(c)** KNN on AMD GPU **(d)** Random on AMD GPU

Figure 7. Number of samples needed to reach 90% of the available performance on for each program/input pair.

[4] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*.

[5] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. 2013. MaSiF: Machine learning guided auto-tuning of parallel skeletons. In *HiPC*. IEEE. https://doi.org/10.1109/HiPC.2013.6799098

[6] John Demme and Simha Sethumadhavan. 2012. Approximate graph clustering for program characterization. *ACM TACO* 8, 4 (2012), 21. https://doi.org/10.1145/2086696.2086700

[7] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, and Olivier Temam. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF*. ACM. https://doi.org/10.1145/1242531.1242553

[8] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *CGO*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[9] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*.

[10] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2018. GPU Code Optimization Using Abstract Kernel Emulation and Sensitivity Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 736–751. https://doi.org/10.1145/3192366.3192397

[11] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *ISCA*. ACM. https://doi.org/10.1145/1555754.1555775

[12] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Stargazer: Automated regression-based GPU design space exploration. In *ISPASS*, Rajeev Balasubramonian and Vijayalakshmi Srinivasan (Eds.). IEEE. https://doi.org/10.1109/ISPASS.2012.6189201

[13] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2013. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *PACT*. IEEE. https://doi.org/10.1109/PACT.2013.6618822

[14] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU Workloads and Systems. In *GPGPU*. ACM. https://doi.org/10.1145/1735688.1735696

[15] Yooseong Kim and Aviral Shrivastava. 2011. CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA. In *DAC*. ACM, 6. https://doi.org/10.1145/2024724.2024754

[16] Hugh Leather, Edwin V. Bonilla, and Michael F. P. O'Boyle. 2009. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *CGO*. IEEE. https://doi.org/10.1109/CGO.2009.21

[17] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. 2015. COMPASS: A Framework for Automated Performance Modeling and Prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 405–414. https://doi.org/10.1145/2751205.2751220

[18] Roland Leissa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-performance Libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (Oct. 2018), 30 pages.

[19] Souley Madougou, Ana Lucia Varbanescu, and Cees de Laat. 2016. Using Colored Petri Nets for GPGPU Performance Modeling. In *CF*. ACM. https://doi.org/10.1145/2903150.2903167

[20] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *PACT*. ACM. https://doi.org/10.1145/2628071.2628087

[21] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM.

[22] Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D. Uram. 2011. GROPHECY: GPU performance projection from CPU code skeletons. In *SC*. ACM. https://doi.org/10.1145/2063384.2063402

[23] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. ACM. https://doi.org/10.1145/1878921.1878951

[24] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *MCSoC*. IEEE. https://doi.org/10.1109/MCSoC.2015.10

[25] Cedric Nugteren and Henk Corporaal. 2012. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *CF*. ACM. https://doi.org/10.1145/2212908.2212937

[26] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri E. Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *HPCA*. IEEE. https://doi.org/10.1109/HPCA.2014.6835955

[27] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *CGO*. ACM. https://doi.org/10.1145/2259016.2259042

[28] Eunjung Park, Louis-Noël Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan. 2011. Predictive modeling in a polyhedral optimization space. In *CGO*. IEEE. https://doi.org/10.1109/CGO.2011.5764680

[29] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. ACM SIGPLAN.

[30] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2017, Bangkok, Thailand, December 18-20, 2017*. 64–71. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.9

[31] Ricardo Nabinger Sanchez, José Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark G. Stoodley. 2011. Using machines to learn method-specific compilation strategies. In *CGO*. IEEE. https://doi.org/10.1109/CGO.2011.5764693

[32] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard W. Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*. ACM. https://doi.org/10.1145/2145816.2145819

[33] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM.

[34] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. http://dl.acm.org/citation.cfm?id=3049841

[35] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using machine learning to improve automatic vectorization. *ACM TACO* 8, 4 (2012), 50. https://doi.org/10.1145/2086696.2086729

[36] Michele Tartara and Stefano Crespi-Reghizzi. 2013. Continuous learning of compiler heuristics. *ACM TACO* 9, 4 (2013), 46. https://doi.org/10.1145/2400682.2400705

[37] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 31–43. https://doi.org/10.1145/3018743.3018755