Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach

Philip Ginsbach
The University of Edinburgh
philip.ginsbach@ed.ac.uk

Toomas Remmelg
The University of Edinburgh
toomas.remmelg@ed.ac.uk

Michel Steuwer
University of Glasgow
michel.steuwer@glasgow.ac.uk

Bruno Bodin
The University of Edinburgh
bbodin@ed.ac.uk

Christophe Dubach
The University of Edinburgh
christophe.dubach@ed.ac.uk

Michael F. P. O'Boyle
The University of Edinburgh
mob@ed.ac.uk

Abstract

Heterogeneous accelerators often disappoint. They provide the prospect of great performance, but only deliver it when using vendor specific optimized libraries or domain specific languages. This requires considerable legacy code modifications, hindering the adoption of heterogeneous computing.

This paper develops a novel approach to automatically detect opportunities for accelerator exploitation. We focus on calculations that are well supported by established APIs: sparse and dense linear algebra, stencil codes and generalized reductions and histograms. We call them *idioms* and use a custom constraint-based Idiom Description Language (IDL) to discover them within user code. Detected idioms are then mapped to BLAS libraries, cuSPARSE and clSPARSE and two DSLs: Halide and Lift.

We implemented the approach in LLVM and evaluated it on the NAS and Parboil sequential C/C++ benchmarks, where we detect 60 idiom instances. In those cases where idioms are a significant part of the sequential execution time, we generate code that achieves 1.26× to over 20× speedup on integrated and external GPUs.

CCS Concepts • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Domain specific languages**;

ACM Reference Format:
Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/http://dx.doi.org/10.1145/3173162.3173182

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/http://dx.doi.org/10.1145/3173162.3173182

1 Introduction

Heterogeneous accelerators provide the potential for great performance. However, achieving that potential is difficult. General purpose languages such as OpenCL [37] provide portability, but the achieved performance often disappoints [30]. This shortfall has led vendors to deliver specialized libraries to bridge the gap [2]. Alternatively, domain specific languages (DSLs) [16, 46] have been proposed, attempting to deliver both portability and performance [42].

Hardware becomes increasingly heterogeneous, (*e.g.* TPU [26]). This means library or DSL based programming is likely to become far more common and future programmers are expected to target those APIs.

However, there are problems with this trend. Firstly, users have to learn multiple specialized DSLs and vendor-specific libraries. Secondly, users have to restructure and rewrite their applications to use them. Having to learn and understand several new APIs and then rewrite existing applications is a severe impediment to the wide-spread efficient exploitation of heterogeneous hardware. Ideally, we would like a mechanism that automatically maps existing code to heterogeneous hardware using the appropriate APIs without user effort.

Our approach is based on detecting specific structures or *idioms* in user code that correspond to the functionality of existing APIs for heterogeneous acceleration. We focus on idioms that are well supported by existing libraries and DSLs. These are likely to be both relevant to existing code bases and have efficient heterogeneous implementations. We consider sparse and dense linear algebra, stencils and generalized reductions and histograms.

At the heart of our approach is the ability to describe each idiom in a concise Idiom Description Language (IDL). After the user's C/C++ program has been compiled down to LLVM IR, our tool reads in an IDL program and translates into a set of constraints. These are passed to a fast solver to search the user's program, detecting all idiom instances.

Once detected, the idioms are mechanically translated into the appropriate DSL or replaced with a library call. This optimized code is then linked into the original program. We currently target the libraries cuSparse, clSparse, cuBLAS, clBLAS for sparse and dense linear algebra and target the DSL Halide [42] for stencil computations. We also target Lift [48] - a data parallel language that supports generalized reductions as well as stencils and linear algebra. This allows the freedom to target many APIs for the same idiom and pick the implementation that best suits the target platform.

New idioms can be easily added thanks to the flexibility of IDL. This provides a powerful means of determining whether a new heterogeneous API matches existing code without touching the core compiler. The idioms addressed in this paper can be expressed in less than 500 lines of IDL code. Our approach is also highly robust, it has been applied to the entire NAS and Parboil benchmark suites and is evaluated on three platforms.

We present a novel approach that:

- Defines a programming language for specifying code idioms, the Idiom Description Language (IDL)
- Implements common idioms in IDL to automatically discover opportunities for accelerator exploitation
- Efficiently translates and maps the detected idioms to APIs for heterogeneous systems

While there has been much research in using constraints for program analysis [35], there is little prior work in its use for idiom detection. In [17], constraints are used for detecting reductions, but this is tightly coupled to a specialized code generation phase for small-scale multi-core systems.

The work most similar in approach concerns discovery of stencil computation and mapping to the Halide DSL. Helium [32] recovers stencils from image-processing binaries. This requires large scale dynamic analysis of binary traces and replacing them with Halide calls. This is significantly extended in [28] which detects stencils in FORTRAN. In this work the focus is on inferring post invariants based on syntax guided synthesis in translation to Halide. However, it uses a narrow approach to selecting code snippets and relies on well structured FORTRAN with occasional user annotations. Our approach is distinct in that we use an external programming language to describe the idioms we are interested in. This allows an unbounded set of idioms to be considered across arbitrary programs and is not restricted to stencils.

To summarize, this paper presents an automatic approach that discovers idioms in legacy code and maps them to heterogeneous platforms via libraries and DSLs. We apply it to 21 C/C++ programs from the NAS and Parboil benchmark suites and demonstrate that it detects more reductions, stencils, matrix multiplications and sparse matrix-vector computations than existing schemes. For the idioms that dominate execution time, we generate code and evaluate on 3 platforms: a multi-core CPU, an integrated and an external GPU.

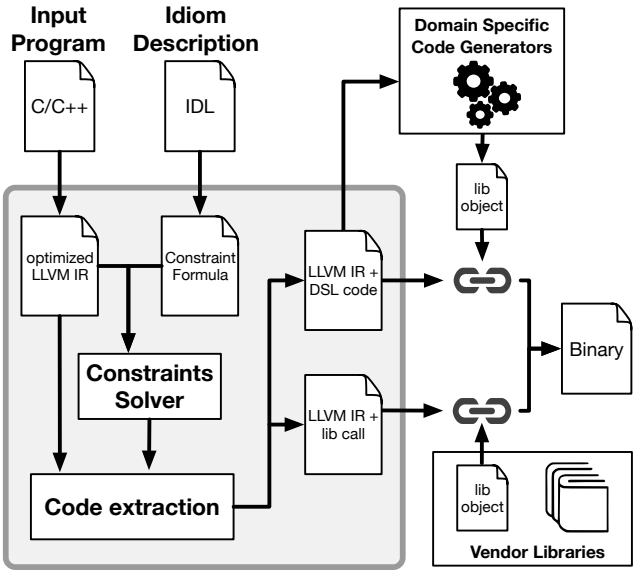

Figure 1. Workflow of our system

Overall we detect 60 idioms. In 10 programs these dominate sequential execution time and are worth exploiting. This results in speedups ranging from 1.26× to over 20×.

2 Overview

Our approach is automatic and has been implemented inside the LLVM compiler infrastructure. It takes arbitrary sequential C/C++ programs as input. Using the clang compiler, the input source code is compiled into a Single Static Assignment (SSA) intermediate representation. We then search this representation for particular idioms which are replaced with calls to specific APIs. Finally, the code generated by the LLVM compiler and the output of the idiom specific code generators/libraries are linked together into a binary, producing an optimized program. LLVM was chosen as it is the best supported SSA-based compiler; the methodology could easily be transferred to other infrastructures such as gcc.

2.1 Compiler Flow

The structure of our approach is described in more detail in Figure 1. Our compiler takes two programs as inputs: the first is the user's program source code, the second describes the idioms we wish to detect using our idiom description language (section 3). The same idioms, of course, can be detected across many user programs, so the IDL program does not have to change from one run to the next.

The program source code is compiled to optimized LLVM IR code while the idiom description is translated into constraints and represented internally as a C++ object. The C++ representation of the constraints and the user program LLVM IR code are then passed as inputs to a backtracking solver [17], which detects all cases where the idioms can be found in the LLVR IR.

```
1 Constraint FactorizationOpportunity
2 ( {sum} is add instruction and
3   {left_addend} is first argument of {sum} and
4   {left_addend} is mul instruction and
5   {right_addend} is second augment of {sum} and
6   {right_addend} is mul instruction and
7   ( {factor} is first argument of {left_addend} or
8     {factor} is second argument of {left_addend}) and
9   ( {factor} is first argument of {right_addend} or
10    {factor} is second argument of {right_addend}))
11 End
```

Figure 2. IDL formulation of (x*y)+(x*z) pattern

The recognized idioms as well as the LLVM IR code are then passed on to the transformation phase of our system. The sections of code that correspond to computational idioms are extracted and reformulated for the appropriate heterogeneous APIs. For library APIs this means replacing the code covered by the idiom with a library call. For domain specific language interfaces, things are a little more involved. As before, we first extract the code associated with the idiom and replace it with a function call. This extracted code is now translated into the appropriate DSL and then passed on to the external DSL compiler which optimizes and generates code. The generated code is then linked with the object code from the main program.

Determining the best heterogeneous APIs to use for a given platform and the best idioms to exploit will become a major issue as the number of idioms and APIs grows. Currently, in this paper, we just try all applicable libraries and DSLs and pick the best executing code. Determining the best option is future work.

2.2 IDL Example

At the core of our approach is IDL, which is described in section 3. A fundamental part of its design is the ability to detect complex idioms. Here we first focus on a simple example to show how IDL works. Consider the standard factorizing optimization that applies the algebraic rule of distributivity

$$(x * y) + (x * z) = x * (y + z)$$

to simplify calculations by reducing the number of multiplications in an expression. The established way of implementing such an optimization is to hard code a detection compiler pass. In LLVM, this is 47 lines of code inside the `instcombine` pass.

Using IDL, we can formulate this in only a few lines of an easily understandable program (Figure 2). For this simple example, the underlying constraint problem is immediately visible: There are four variables `sum`, `left_addend`, `right_addend`, `factor` and nine individual constraints that are combined with boolean operators.

From the formulation in Figure 2, the IDL compiler builds a representation of the underlying constraint problem that is passed to a constraint solver. For a given section of user code, this solver returns the set of factorization opportunities, each containing four entries `sum`, `left_addend`, `right_addend`, `factor` that refer to values inside the user code. Figure 3 shows a simple example. The incoming C code is translated to optimized LLVM IR. The solver then finds a single solution to the constraint problem.

Original C code:

```
1 int example(int a, int b, int c) {
2     int d = a;
3     return (a*b) + (c*d);
4 }
```

Resulting LLVM IR:

```
1 define i32 @example(i32 %a, i32 %b, i32 %c) {
2   %1 = mul i32 %a, %b
3   %2 = mul i32 %c, %a
4   %3 = add i32 %1, %2
5   ret i32 %3
6 }
```

Detected factorization opportunities:

```
1 { "sum" :          %3,
2   "left_addend" :  %1,
3   "right_addend" : %2,
4   "factor" :       %a }
```

Figure 3. Demonstration of simple idiom detection

In this case, the variable `sum` is matched to the value `%3`, an add instruction, while the variables `left_addend` and `right_addend` match the left and right operands `%1` and `%2` of this instruction.

Lines 7 and 8 of Figure 2 say that `factor` can either be the first argument OR the second argument of `left_addend`. As the `left_addend` is `%1`, then `factor` can be either `%a` or `%b`. Similarly lines 9 and 10 of Figure 2 say that `factor` can be be the first argument OR the second argument of the `right_addend`. As the `right_addend` is `%1`, then `factor` can be either `%c` or `%a`. The two disjunctions in Lines 7-10 are connected by AND, so they must both hold.

$$\begin{aligned}&((factor = a) \vee (factor = b))\\ \wedge&((factor = c) \vee (factor = a))\\ &\implies factor = a\end{aligned}$$

The only value of `factor` that satisfies this condition is `factor = %a`. Therefore, the solution at the bottom of Figure 3 is the only factorization opportunity in the code.

```
1 for (j = 0; j < m; j++) {
2   d = 0.0;
3   for (k = rowstr[j]; k < rowstr[j+1]; k++)
4     d = d + a[k]*z[colidx[k]];
5   r[j] = d; }
```

```
1  ; <label>:2:
2    %j = phi i64 [ %j_next, %12 ], [ 0, %1 ]
3    %j_cond = icmp slt i64 %j, %m
4    br i1 %j_cond, label %3, label %13
5  ; <label>:3:
6  %4 = getelementptr i32, i32* %rowstr, i64 %j
7    %5 = load i32, i32* %4
8    %j_next = add nuw nsw i64 %j, 1
9    %6 = getelementptr i32, i32* %rowstr, i64 %j_next
10   %7 = load i32, i32* %6
11   %k_begin = sext i32 %5 to i64
12   %k_end = sext i32 %7 to i64
13   br label %8
14 ; <label>:8:
15   %k = phi i64 [ %k_next, %9 ], [ %k_begin, %dnext ]
16   %d = phi double [ 0.0, %3 ], [ %d_next, %9 ]
17   %k_cond = icmp slt i64 %iv, %k_end
18   br i1 %k_cond, label %9, label %12
19 ; <label>:9:
20   %a_addr = getelementptr double, double* %a, i64 %k
21   %a_load = load double, double* %a_addr
22   %cix_addr = getelementptr i32, i32* %colidx, i64 %k
23   %cix_load = load i32, i32* %cix_addr
24   %10 = sext i32 %cix_load to i64
25   %z_addr = getelementptr double, double* %z, i64 %10
26   %z_load = load double, double* %z_addr
27   %11 = fmul double %a_load, %z_load
28   %d_next = fadd double %d, %11
29   %k_next = add nsw i64 %k, 1
30   br label %8
31 ; <label>:12:
32   %r_addr = getelementptr double, double* %r, i64 %j
33   store double %d, double* %r_addr
34   br label %2
```

Figure 4. Sparse linear algebra in C and LLVM IR

Idiom detection with IDL program in Figure 12

Variable Name	Assigned IR Value
iterator	%j
inner.iter_begin	%k_begin
inner.iter_end	%k_end
inner.iterator	%k
idx_read.value	%cix_load
indir_read.value	%a_load
seq_read.value	%z
output.address	%r_addr
iter_begin	0
iter_end	%**m**
idx_read.base_pointer	%**colidx**
seq_read.base_pointer	%**a**
indir_read.base_pointer	%**z**
…	…

Figure 5. Constraint solution for sparse mv

Code generation: insert arguments, replace code

```
1 cusparseDcsrmv(context,
2     CUSPARSE_OPERATION_NON_TRANSPOSE, m, n,
3     rowstr[m+1]-rowstr[0], &gpu_1, descr, gpu_a,
4     gpu_rowstr, gpu_colidx, gpu_z, &gpu_0, gpu_r);
```

Figure 6. Generated function call to cuSPARSE

2.3 Sparse Linear Algebra in IDL

Although the previous example illustrated how constraints can be applied to program analysis, we want to detect much more complex idioms and map them to existing APIs.

The C code in Figure 4 shows the performance bottleneck of the NAS *Conjugate Gradient (GC)* benchmark, as well as the corresponding LLVM IR code. It implements a standard operation from sparse linear algebra, namely a multiplication of a sparse matrix in Compressed Sparse Row (CSR) format with a dense vector.

This code contains several features that make it unsuitable for most established compiler optimizations: The iteration domain of the nested loop is memory dependent (line 3) and there is indirect memory access (line 4). This makes the iteration domain of the loop nest non-polyhedral and the access structure to memory non-affine. Under these conditions, simple data dependence models, but also sophisticated analysis based on the polyhedral model, would fail.

We can express this idiom in IDL (section 4, Figure 12). The IR code, together with the IDL program, is fed into a constraint solver, which outputs a constraint solution as shown in Figure 5. We can see that different parts of the IR have been assigned to IDL variables.

Figure 6 shows how this solution is used to generate a call to a cuSPARSE procedure. The solution variables are inserted into the `cusparseDcsrmv` code template as function arguments. The original code is then cut out and replaced with this function call. The cuSPARSE library is then linked with the object code produced by the LLVM compiler, resulting in a speedup of 17× on a GPU as described in section 8.

Central to our approach is the ability to detect idioms. In the next section we introduce a powerful description language that is capable of expressing a wide class of idioms that are suitable for acceleration by heterogeneous hardware.

3 Idiom Description Language

Any detection method needs to be robust and work on real code. It should work in the presence of complex language features, such as the standard library containers, operator overloading and class hierarchies in C++, as well as the myriad different ways users can write the same, common algorithms.

This rules out a syntactic approach. To allow robust detection of complex idioms, we devised IDL, a domain specific constraint language that operates on the SSA based LLVM IR. In IDL, idioms are specified in a modular fashion, exploiting standard compiler primitives such as types and data and control flow analysis.

IDL was developed with the aim of enabling analysis routines that are too complex to directly implement by hand. However, it is still targeted at compiler experts. Writing and debugging IDL code is challenging, but the modularity mechanisms make it very suitable for unit testing. The full syntax specification of IDL in BNF notation is shown in Figure 7.

```
specification ::= Constraint ⟨s⟩ ⟨constraint⟩ End
   constraint ::= ⟨atomic⟩ | ⟨grouping⟩ | ⟨collect⟩ | ⟨rename⟩ | ⟨rebase⟩ | '(' ⟨constraint⟩ ')'
     grouping ::= ⟨conjunction⟩ | ⟨disjunction⟩ | ⟨inheritance⟩ | ⟨forall⟩ | ⟨forsome⟩ | ⟨forone⟩ | ⟨if⟩
  conjunction ::= '(' ⟨constraint⟩ and ⟨constraint⟩ {and ⟨constraint⟩} ')'
   disjunction ::= '(' ⟨constraint⟩ or ⟨constraint⟩ {or ⟨constraint⟩} ')'
  inheritance ::= inherits ⟨s⟩ ['(' ⟨s⟩ '=' ⟨calculation⟩ {',' ⟨s⟩ '=' ⟨calculation⟩} ')']
        forall ::= ⟨constraint⟩ for all ⟨s⟩ '=' ⟨calculation⟩ '..' ⟨calculation⟩
      forsome ::= ⟨constraint⟩ for some ⟨s⟩ '=' ⟨calculation⟩ '..' ⟨calculation⟩
        forone ::= ⟨constraint⟩ for ⟨s⟩ '=' ⟨calculation⟩
            if ::= if ⟨calculation⟩ '=' ⟨calculation⟩ then ⟨constraint⟩ else ⟨constraint⟩ endif
      rename ::= ⟨grouping⟩ with ⟨var⟩ as ⟨var⟩ and ⟨var⟩ as ⟨var⟩
      rebase ::= ⟨grouping⟩ [with ⟨var⟩ as ⟨var⟩ and ⟨var⟩ as ⟨var⟩] at ⟨var⟩
     collect ::= collect ⟨s⟩ ⟨n⟩ ⟨constraint⟩
      atomic ::= ⟨var⟩ is (integer | float | pointer) [constant zero]
               | ⟨var⟩ is (unused | a constant | a compile time value | an argument | an instruction)
               | ⟨var⟩ is ( store | load | return | branch | add | sub | mul | fadd | fsub | fmul | fdiv
                          | select | gep | icmp ) instruction
               | ⟨var⟩ is [not] the same as ⟨var⟩
               | ⟨var⟩ has (data flow | control flow | control dominance | dependence edge) to ⟨var⟩
               | ⟨var⟩ is (first | second | third | fourth) argument of ⟨var⟩
               | ⟨var⟩ reaches phi node ⟨var⟩ from ⟨var⟩
               | ⟨var⟩ [does not] [strictly] [(data flow | control flow)] dominates ⟨var⟩
               | all [(data | control)] flow from ⟨var⟩ to ⟨var⟩ passes through ⟨var⟩
               | all flow from ⟨varlist⟩ to ⟨varlist⟩ is killed by ⟨varlist⟩
   varsingle ::= ⟨s⟩ | ⟨varsingle⟩ '.' ⟨s⟩ | ⟨varsingle⟩ '[' ⟨calculation⟩ ']'
   varmulti ::= ⟨varsingle⟩ | ⟨varmulti⟩ '[' ⟨calculation⟩ '..' ⟨calculation⟩ ']'
     varlist ::= '{' ⟨varmulti⟩ ',' {⟨varmulti⟩ ','} ⟨varmulti⟩ '}'
         var ::= '{' ⟨varsingle⟩ '}'
 calculation ::= ⟨s⟩ | ⟨n⟩ | ⟨calculation⟩ ('+' | '-') (⟨s⟩ | ⟨n⟩)
```

Figure 7. BNF notation of IDL syntax

Terminals The symbols ⟨**s**⟩ and ⟨**n**⟩ in the grammar correspond to arbitrary strings and positive integer literals respectively, the ⟨`specification`⟩ top level construct of the language binds an idiom definition to a name. The significant part of the language specification is everything covered by ⟨`constraint`⟩.

Atomic Constraints All idiom definitions are eventually built up by combining atomic constraints. These correspond to basic boolean predicates that may hold for one or more values in the IR. The atomic constraints describe standard properties within the IR. Control flow in our model is evaluated on the granularity of instructions. This is to reduce the size of the language, there is no notion of basic blocks. For phi nodes, the incoming basic blocks are identified with their terminating branch instruction.

Higher Level Constructs Atomic constraints can be combined with many higher level language constructs. The semantics of ⟨`conjunction`⟩ and ⟨`disjunction`⟩ correspond to AND, OR. The ⟨`inheritance`⟩ inserts another idiom description into the current one. Idiom definitions can be parameterized in a way that is inspired by C++ templates with integers, allowing more concise descriptions. The ⟨`if`⟩ constraint has the standard meaning.

The ⟨`forall`⟩ and ⟨`forsome`⟩ constructs provide range based versions of conjunction and disjunction. The contained constraint formula is duplicated for each index in the provided range and the contained variable names are modified according to the index (i.e. if the index occurs in a variable name, it is substituted with the current iteration value). The duplicated formulas are then combined with conjunctions or disjunctions respectively.

To allow modularity, complementing the inheritance feature, there are two mechanisms to change the variable names in the contained constraint specification. With ⟨`rename`⟩, the translation of variable names is done with a simple dictionary, where every variable that is not explicitly mentioned in the dictionary remains unchanged. The ⟨`rebase`⟩ has the same behaviour for variables in the dictionary, but for every other variable, a prefix is added to the variable name.

The ⟨`collect`⟩ construct is more powerful. It is used to capture all possible solutions of a given constraint for expressions that require the logical ∀ quantifier. For example, it can be used to collect all affine array accesses in a loop.

4 Specification of Idioms in IDL

With the definition of IDL, we can now specify idioms. The complete set of idioms used in this paper comprises of ≈500 lines of code. Due to space restrictions, we first show a simple constraint that we rely on – single entry, single exit regions – and then describe the top level constraints for each idiom.

4.1 Building Blocks

Before any algorithmic idiom can be specified, we need some basic control flow constructs. The most fundamental is the single entry single exit region (SESE) [25] which is used amongst other things to determine loop bodies. A SESE region is a part of code spanned by two instructions *A* and *B* such that *A* dominates *B*, *B* postdominates *A* and every cycle containing either *A* or *B* also contains the other. It is defined in Figure 9.

Using simple building blocks such as SESE, we can define more complex control structures such as loops and important memory access patterns such as matrix reads. From this we build powerful idiom definitions that capture complex computational patterns that can include arbitrary control flow.

4.2 Full Idiom Definition

The generalized matrix multiplication idiom is described in Figure 10. The control flow is captured by three nested for loops. Inside these loops, the memory access is characterized by three matrix accesses, each with a different subset of the loop iterators. The corresponding `MatrixRead` and `MatrixWrite` idioms model generic access to matrices allowing strides, transposed matrices etc. The actual computation is encapsulated by the `DotProductLoop` idiom. This also contains the linear combination with factors `alpha` and `beta` that is part of the generalized matrix multiplication.

Figure 11 shows the generalized histogram idiom. It is contained in a for loop and the basic memory access pattern is a read-modify-write to a bin array. This memory access can be conditional as long as the condition is well behaved, which is guaranteed by the later `KernelFunction` idiom. The histogram uses input data that is read from input arrays using the loop iterator as a base index (that can be strided, offset etc.). Finally there are two well behaved kernel functions in a histogram, one to compute the access index and one to compute the updated value.

The sparse matrix vector multiplication defined in Figure 12 is different to the other idioms in that the control flow of the skeleton of the idiom does not consist of perfectly nested for loops. Instead, the iteration space of the inner loop is read from an array using the `ReadRange` idiom. The actual computation that SPMV performs is a dot product and thus it uses the same `DotProductLoop` idiom as `GEMM` but the memory access pattern is different, with indirect memory access in `indir_read`.

Figure 13 shows the basic stencil idiom. Stencils consist of a loop nest with a multidimensional memory access to store the updated cell value. This updated value is computed with a kernel function using a number of values that are constraint by the `StencilRead` idiom, which specifies multidimensional array access with only constant offsets in all dimensions.

The scalar reduction idiom is specified in Figure 14. We can see that its structure is similar to the histogram idiom, but instead of a read-modify-write memory access it operates on an induction variable that is implemented with the `InductionVar` idiom.

4.3 Not Syntactic Pattern Matching

The idiom descriptions may at first appear to be shallow syntactic pattern matching. In fact, because it operates on the IR level, it can detect idioms that are written in superficially distinct style but are semantically equivalent. For example, there are two syntactically distinct programs in Figure 8, which in fact are both implementations of general matrix multiplication. The IDL in Figure 10 discovers they are both instances of GEMM and they can both be replaced with an API call to GEMM.

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm+nn*ldc] =
        C[mm+nn*ldc] * beta + alpha * c;
  }
}
```

```
for(int i = 0; i < 1000; i++)
    for(int j = 0; j < 1000; j++) {
        M3[i][j] = 0.0f;
        for(int k = 0; k < 1000; k++)
           M3[i][j]+=M1[i][k]*M2[k][j]; }
```

Figure 8. Two matching instances of GEMM

There are limitations to this semantic matching. In particular, the use of low level optimizations that circumvent the usual IR representation, *e.g.* SIMD compiler intrinsics, would distort the algorithms beyond recognition by our system. In practice, this is rarely encountered.

```
Constraint SESE
( {precursor} is branch instruction and
  {precursor} has control flow to {begin} and
  {end} is branch instruction and
  {end} has control flow to {successor} and
  {begin} control flow dominates {end} and
  {end} control flow post dominates {begin} and
  {precursor} strictly control flow dominates
      {begin} and
  {successor} strictly control flow post dominates
      {end} and
  all control flow from {begin} to {precursor}
      passes through {end} and
  all control flow from {successor} to {end}
      passes through {begin})
End
```

Figure 9. IDL specification of SESE region

```
Constraint GEMM
( inherits ForNest(N=3) and
  inherits MatrixStore
      with {iterator[0]} as {col}
       and {iterator[1]} as {row}
       and {begin} as {begin} at {output} and
  inherits MatrixRead
      with {iterator[0]} as {col}
       and {iterator[2]} as {row}
       and {begin} as {begin} at {input1} and
  inherits MatrixRead
      with {iterator[1]} as {col}
       and {iterator[2]} as {row}
       and {begin} as {begin} at {input2} and
  inherits DotProductLoop
      with {loop[2]}         as {loop}
       and {input1.value}    as {src1}
       and {input2.value}    as {src2}
       and {output.address} as {update_address})
End
```

Figure 10. IDL specification of GEMM

```
Constraint Histogram
( inherits For and
  inherits ConditionalReadModifyWrite
      with {indexkernel.output} as {address}
       and {kernel.output} as {value} and
  collect i
  ( inherits VectorRead
       with {read_value[i]} as {value}
        and {iterator} as {idx}
        and {begin} as {begin} at {read[i]}) and
  inherits Concat
      with {read_value}   as {in1}
       and {old_value}    as {in2}
       and {kernel.input} as {out} and
  inherits KernelFunction
      with {begin} as {outer}
       and {body.begin} as {inner} at {kernel} and
  inherits KernelFunction
      with {read_value} as {input}
       and {begin} as {outer}
       and {body.begin} as {inner} at {indexkernel})
End
```

Figure 11. IDL specification of generalized histogram

```
Constraint SPMV
( inherits For and
  inherits VectorStore
      with {iterator} as {idx}
       and {begin} as {begin} at {output} and
  inherits ReadRange
      with {iterator} as {idx}
       and {inner.iter_begin} as {range_begin}
       and {inner.iter_end}   as {range_end} and
  inherits For at {inner} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {idx_read} and
  inherits VectorRead
      with {idx_read.value} as {idx}
       and {begin} as {begin} at {indir_read} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {seq_read} and
  inherits DotProductLoop
      with {inner}             as {loop}
       and {indir_read.value} as {src1}
       and {seq_read.value}   as {src2}
       and {output.address}   as {update_address})
End
```

Figure 12. IDL specification of SPMV

```
Constraint Stencil
( inherits ForNest and
  inherits PermMultidStore
      with {iterator} as {input}
       and {begin} as {begin} at {write} and
  collect i
  ( inherits StencilRead
      with {write.input_index} as {input}
       and {kernel.input[i]} as {value}
       and {begin} as {begin} at {reads[i]}) and
  {kernel.output} is first argument of {write.store} and
  inherits KernelFunction
      with {begin}      as {outer}
       and {body.begin} as {inner} at {kernel})
End
```

Figure 13. IDL specification of simple stencil

```
Constraint Reduction
( inherits For and
  collect i
  ( inherits VectorRead
       with {iterator}         as {idx}
        and {read_value[i]}    as {value}
        and {begin} as {begin} at {read[i]}) and
  inherits InductionVar
      with {old_value}       as {old_ind}
       and {kernel.output}   as {new_ind} and
  {old_value} is not the same as {iterator} and
  inherits Concat
      with {read_value}   as {in1}
       and {old_value}    as {in2}
       and {kernel.input} as {out} and
  inherits KernelFunction
      with {begin}      as {outer}
       and {body.begin} as {inner} at {kernel})
End
```

Figure 14. IDL specification of scalar reductions

4.4 Compilation Process and Implementation

Idiom definitions are compiled to C++ functions that perform idiom recognition on LLVM IR. In a first step, the compiler eliminates ⟨`inheritance`⟩, ⟨`forall`⟩, ⟨`forsome`⟩, ⟨`if`⟩, ⟨`rename`⟩ and ⟨`rebase`⟩. They are replaced with the simpler ⟨`conjunction`⟩ and ⟨`disjunction`⟩ constructs. This also involves removing all parameterizations from the formula and flattening all variable names. Next, variables are collected and ordered to assist constraint solving. The ordering impacts performance, as it determines how well the search space is pruned. For each variable, all the constraints associated with the variable are assembled.

The compiler then emits C++ code which is passed to a generic solver based on [17] to search for idiom instances. This solver is based on standard backtracking. As shown in the results section, this increases compilation time, but the overhead is modest. [1]

5 Targeted Heterogeneous APIs

After idiom detection, we must transform the user program to exploit the relevant API. Two types of heterogeneous APIs are currently targeted: libraries and domain specific languages with their optimizing compilers.

5.1 Domain Specific Libraries

Libraries provide narrow interfaces but are often highly optimized. For example, the cuBLAS library is only suitable for a limited set of dense linear algebra operations and only works on Nvidia GPUs, but its implementation provides outstanding performance. For sparse linear algebra we use the vendor provided cuSPARSE, clSPARSE, and MKL libraries. For dense BLAS routines cuBLAS, clBLAS, CLBlast, and MKL are used.

5.2 Domain Specific Code Generators

Domain Specific Languages provide wider interfaces than libraries and allow problems to be expressed as composition of dedicated language constructs. An optimizing compiler then specializes the program for the target hardware. We currently support Halide and Lift as domain specific code generators.

Halide [42] is a language and optimizing compiler targeted at image processing applications. Optimized code is generated for CPUs as well as GPUs. Halide separates the functional description of the problem from the description of the implementation which is called a *schedule*. This allows retargeting of Halide programs to different platforms. We translate some of the stencil idioms and linear algebra idioms into Halide. Stencils involving control flow in their computations are not easily expressible in Halide.

[1] Our impementation of IDL is available as open source on https://github.com/asplos18ginsbach.

Lift [22, 47, 48] is an optimizing code generator based on rewrite rules. The Lift language consists of functional parallel patterns such as *map* and *reduce* which express a range of parallel applications. For this work we translate stencil idioms, complex reductions and linear algebra idioms to Lift.

6 Translating Computational Idioms

This section describes how the detected idioms are mapped to the previously described library APIs domain specific languages. The two types of APIs (library interfaces and domain specific languages) are treated individually.

6.1 Library

For library call interfaces, the original code is removed and an appropriate function call is inserted. The solution that is generated by the solver using the IDL program contains both the IR instructions to remove as well as the arguments that are to be used for the function call.

For example, in the case of the `GEMM` program that was shown in Figure 10, the original code is removed by deleting the IR instruction at `output.store_instr` explicitly, which captures the store instruction of the `MatrixStore` subprogram. The remaining cleanup is left to the standard dead code elimination pass. The arguments that specify the matrix dimensions are taken from `ForNest` in combination with the stride and offset determined by `MatrixRead` and `MatrixWrite`.

The mapping of solution variables to the arguments of the generated function call is implemented individually for each backend, as we have no way to describe it in IDL itself. Once the code is replaced, LLVM continues with code generation as usual.

6.2 DSL

For domain specific languages, the situation is a bit more involved. Reduction, histogram and stencil idioms are higher order functions that contain a kernel function or reduction operator that has to be represented for the DSL.

For each individual combination of idiom and DSL there is a parameterized skeleton program. This skeleton is then specialized for the appropriate data types and numeric parameters as well as the kernel function or reduction operator.

Numerical parameters are picked from the constraint solution in the same way that was described previously for library call interfaces. Also from the constraint solution, we have the loop body that contains the kernel function or reduction operator, as well as the input values and the result value used. We use this information to cut out the kernel function that is then used to generate code appropriate for the DSL backends:

Lift expects stencil kernels or reduction operators to be sequential C code with a specific function interface that is used internally by Lift when generating OpenCL code. We therefore implemented a rudimentary LLVM IR to C backend for generating this function.

Halide is a language embedded in C++, it requires a syntax tree of the kernel functions built using a class hierarchy.

```
1  float mult(float x, float y) { return x*y; }
2  float add(float x, float y) { return x+y; }
3
4  gemm_in_lift(A, B, C, alpha, beta) {
5   map(fun(a_row, c_row) {
6    map(fun(b_col, c) {
7     map(fun(ab){ add(mult(alpha, ab), mult(beta, c))},
8      reduce(add, 0.0f, map(mult, zip(a_row, b_col))))
9    }, zip(transpose(B), c_row))
10  }, zip(A, C))
11 }
```

Figure 15. Example of matrix multiplication in Lift.

After code for the DSLs is generated, it is passed to the DSL code generator. Figure 15 shows an example of the Lift code generated for GEMM (`gemm_in_lift`). It performs a dot product (expressed in line 8 using the Lift skeletons `zip`, `map`, and `reduce`) for each row of matrix A (`a_row`) and column of matrix B (`b_col`). This code is compiled by Lift into optimized OpenCL code.

Finally, we again replace the idiom code in the user's code with a call to the code generated by the DSL and continue once again with LLVM code generation.

6.3 Aliasing

Since idiom detection works statically, we are unable to fully rule out aliasing of pointers, which can make transformations unsound. For dense linear algebra this is easily solved with some basic run time checks for non-overlapping memory. However, for sparse linear algebra this is not as straightforward and in corner cases our approach is unsound. In practice this did not cause problems on any of the benchmark programs, however this means that optimizations based on these techniques will have to provide appropriate feedback to the programmer.

7 Experimental Setup

Benchmarks We applied our approach to all of the sequential C/C++ versions of the NAS Parallel Benchmarks. We use the SNU NPB implementation by the Seoul National University, containing the original 8 NAS benchmarks plus two of the newer unstructured components UA and DC. We also evaluated our approach on all Parboil benchmarks, giving 21 programs in total.

Platform and Evaluation We use an AMD A10-7850K APU with a multi-core CPU and an integrated Radeon R7 GPU on the same die using driver version 1912.5, as well as an Nvidia GTX Titan X as an external GPU using driver version 375.66. We report the median runtime of 10 executions for each program.

Alternative detection approaches There are no easily available compilers to compare against that perform idiom detection. Instead, we consider two well known parallelizing compilers and examine whether they detect idioms as part of their parallelization approach. As their goal is parallelization and not idiom detection, this should be borne in mind in the results section.

Polly [14] is an LLVM based polyhedral compiler capable of finding parallel loops and reductions in static control flow (SCoP) parts of programs. This allows comparison against another approach that uses the same compiler infrastructure. We gathered the SCoPs that Polly detected with the options `-O3 -mllvm -polly -mllvm -polly-export` and manually inspected the reported SCoPs for stencil like parallel loops and reduction operations. When Polly captured such a loop as a SCoP, we counted it as an idiom detection, although Polly itself has no concept of idioms. This gives an optimistic estimate as to what idiom coverage a polyhedral based approach can achieve.

The Intel C++ Compiler (ICC) is a mature industry strength compiler that provides a detection mechanism for parallelizing reduction idioms based on data dependency analysis. We use the `-parallel -qopt-report` command line options and checked in the optimization report files whether the corresponding loop is considered parallelizable.

8 Results

We first evaluate how often our approach is able to detect idioms and its compile time cost. We then investigate the runtime coverage of the idioms to see where exploitation might be beneficial. Where runtime coverage is substantial, we report speedups compared to the sequential C code and compare the performance of each of the targets APIs. We also compare against the handwritten OpenMP and OpenCL implementations that are included with the benchmark suites as reference implementations.

8.1 Idiom Detection

Table 1 shows the number of idioms found by our approach, Polly, and ICC. Polly finds 3 scalar reductions and 6 stencils while ICC which just considers scalar reduction finds 28. Polly is unable to perform idiom specific optimizations on GEMM. Other approaches do not detect any histograms or sparse matrix operations, because such code involves indirect and thus non-affine memory accesses. This fundamentally contradicts assumptions that these tools rely on and is not merely an implementation artifact. Our IDL approach detects 60 idioms overall with the compile time cost shown in figure Table 2. On average, the compilation time is increased by 82%, which can be reduced further by optimizing the solver.

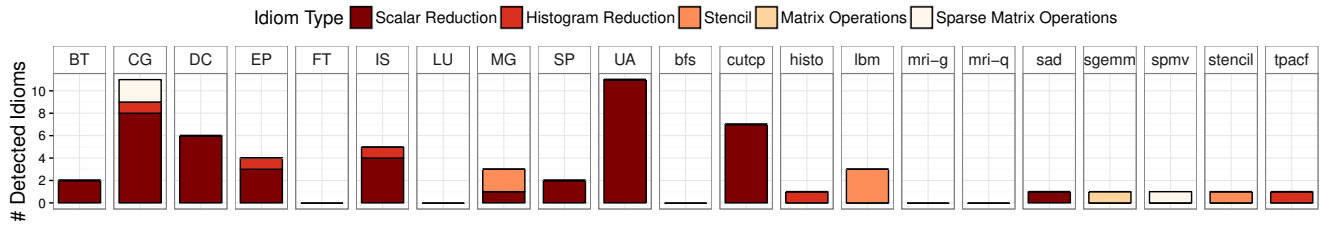

Figure 16. The different computational idioms found in all benchmarks.

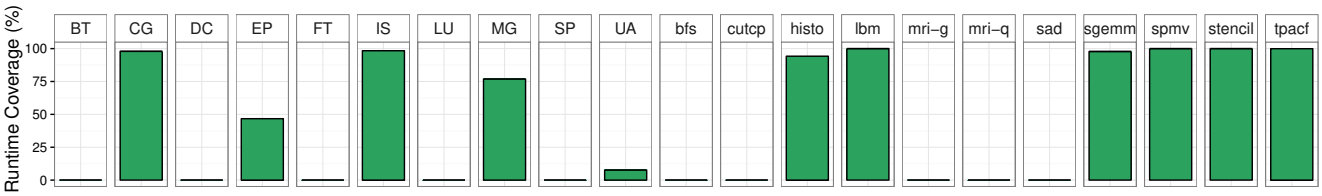

Figure 17. Runtime coverage of the detected idioms in all benchmarks.

	Scalar Reduction	Histogram Reduction	Stencil	Matrix Op.	Sparse Matrix Op.
Polly	3	—	5	—	—
ICC	28	—	—	—	—
IDL	45	5	6	1	3

Table 1. Idioms detected by IDL, ICC, Polly

Figure 16 shows the different idioms detected across the benchmarks. We detect both scalar and histogram reductions as well as stencils, dense matrix operations and sparse matrix-vector multiplication. While Polly and ICC are only capable of detecting simple scalar reductions we are able to detect histogram reductions, *e.g.* in the *histo* benchmark as well. For stencils, Polly detects two in *lbm* and *stencil* while our approach detects all the stencils in *lbm*, *stencil* and *MG*. Unlike any existing approach, we detect sparse matrix-vector operations in *C*G and *s*pmv as well as dense matrix operations in *sgemm*. It is worth repeating, however, that both Polly and ICC are parallelizing compilers, not idiom recognition tools.

8.2 Runtime Coverage

To determine if the detected idioms are actually important, Figure 17 shows the percentage of time spent in the detected computational idiom. This data shows that either the detected idioms have a low runtime contribution or they dominate almost the entire execution. *EP* is the only exception where about 50% of the runtime is spent inside a detected histogram reduction. We focus on the 10 programs which spend a significant amount of time in the detected idioms, as only these can reasonably expect a performance gain using our approach.

	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA	bfs	cutcp
without IDL	1.9	0.5	1.0	0.3	0.6	0.3	1.9	0.8	1.6	2.7	0.4	0.4
with IDL	4.0	0.8	1.6	0.6	1.2	0.5	3.9	4.5	3.2	7.3	0.5	0.6
overhead in %	116	77	57	77	93	62	103	484	97	169	30	65

	histo	lbm	mri-g	mri-q	sad	sgemm	spmv	stencil	tpacf
without IDL	0.2	0.3	0.2	0.2	0.4	0.6	0.3	0.2	0.2
with IDL	0.2	0.6	0.4	0.3	0.6	0.7	0.7	0.2	0.4
overhead in %	35	87	100	52	58	24	115	36	54

Table 2. Compile time cost in seconds

8.3 Performance Results

Speedup vs. Sequential Figure 18 shows the end-to-end speedup obtained by accelerating idioms with heterogeneous APIs on a CPU, an integrated GPU, and an external GPU. All results include data transfer overhead to and from the GPUs. Here the best performing API is shown; Table 3 provides detailed results for all APIs.

For five benchmarks we obtain moderate speedups from 1.26× for *histo* up to 4.5× for *IS*. All of these benchmarks (besides *MG*) have a scalar or histogram reduction as their performance bottleneck and are, therefore, not computationally expensive. Interestingly, we can see that different hardware is beneficial for different benchmarks: for *tpcaf* the CPU is the best platform, beating the GPU for which the data transfer time dominates; for *MG* and *histo* the integrated GPU strikes the right balance between computational power while avoiding the movement of data to the external GPU; and, finally, for *EP* and *IS* the data transfer to the GPU pays off exploiting the high GPU internal memory bandwidth.

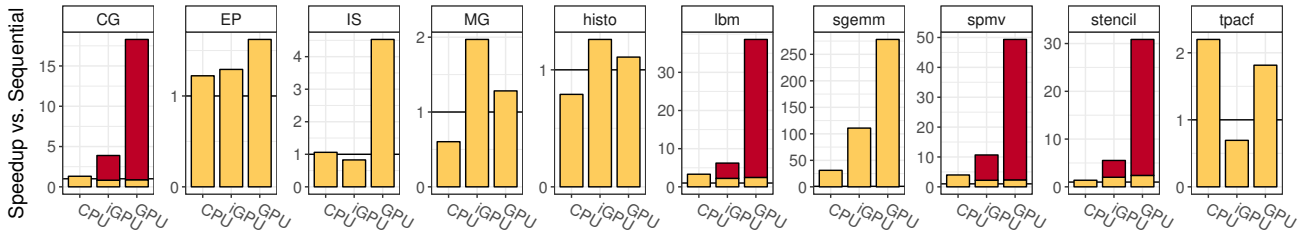

Figure 18. Speedup compared to the sequential C program. Results for the best performing heterogeneous API on each device are shown. The red bars indicate a manual runtime optimization for avoiding unnecessary data transfers.

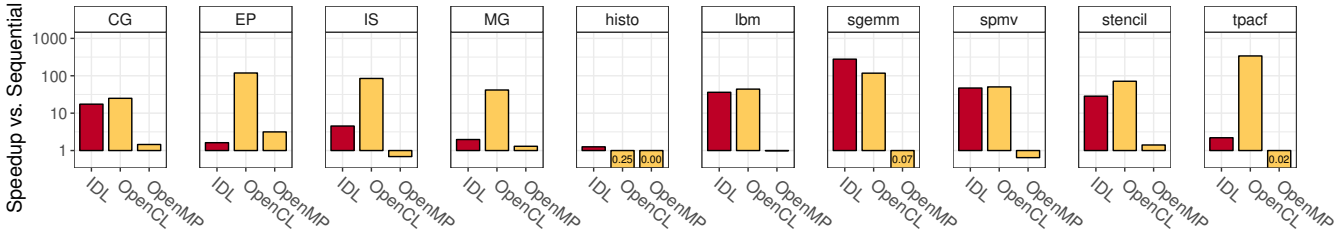

Figure 19. Speedup of our constraints based approach (executed on the best hardware and highlighted in red) compared to handwritten parallel OpenCL (executed on the GPU) and OpenMP (executed on the CPU) implementations.

	CPU						iGPU					GPU			
	MKL	libSPMV	Halide	clBLAS	CLBlast	Lift	clSPARSE	libSPMV	clBLAS	CLBlast	Lift	cuSPARSE	libSPMV	cuBLAS	Lift
CG	**1504.21**	–	–	–	–	–	**644.02**	–	–	–	–	**113.51**	–	–	–
EP	–	–	–	–	–	**32762.50**	–	–	–	–	**30983.40**	–	–	–	**24680.70**
IS	–	–	**426.95**	–	–	1765.61	–	–	–	–	**547.28**	–	–	–	**99.95**
MG	–	–	–	–	–	**4699.63**	–	–	–	–	**1439.58**	–	–	–	**2211.56**
histo	–	–	–	–	–	**27.42**	–	–	–	–	**17.20**	–	–	–	**19.54**
lbm	–	–	–	–	–	**6457.93**	–	–	–	–	**5335.09**	–	–	–	**590.60**
sgemm	**53.50**	–	–	1661.75	660.44	1339.15	–	–	**14.73**	19.03	15.04	–	–	**5.99**	7.87
spmv	–	**218.17**	–	–	–	–	–	**102.233**	–	–	–	–	**18.437**	–	–
stencil	–	–	**5760.81**	–	–	21951.80	–	–	–	–	**2261.48**	–	–	–	**279.38**
tpacf	–	–	–	–	–	**19276.40**	–	–	–	–	**61111.90**	–	–	–	**23358.20**

Table 3. Detailed performance results for each heterogeneous API used in milliseconds. Fastest implementations for each benchmark and target hardware are highlighted in bold.

These results emphasize the significance of heterogeneous code generation flexibility.

For five of the benchmarks we achieve significantly higher performance gains, from 17× for *CG* and up to over 275× for *sgemm*. These benchmarks are computationally expensive and the external GPU is always the fastest architecture by a considerable margin.

The red highlighting in the plot indicates an important runtime optimization: redundant data transfers for the iterative *CG*, *lbm*, *spmv* and *stencil* benchmarks. All of these benchmarks execute computations inside a for loop and do not require access to the data on the CPU between iterations. We manually applied a straightforward lazy copying technique by flagging memory objects to avoid redundant transfers, similar to [24]. As can be seen this runtime optimization is crucial for achieving high performance for these benchmarks.

API performance comparison Table 3 provides a breakdown of the performance of each API on each program and platform. Not all APIs target all platforms, *e.g.* cuSPARSE only targets NVIDIA GPUs and in the case of Halide, the current version that we have access to failed to generate valid GPU code for any of the benchmarks we tried. The best performing API is highlighted in bold in the table entries. The *spmv* benchmark uses an unusual sparse matrix format,

so that we implemented a custom library libSPMV for this benchmark.

On the multicore CPUs, the Intel MKL library gives the best linear algebra performance, outperforming the other libraries and Lift. Halide achieves good performance for the NPB *IS* and Parboil *stencil* benchmarks on the CPU, outperforming Lift due to its more advanced vectorization capabilities. In the programs where scalar reductions dominate, Lift performs well. On the iGPU, clBLAS provides a better matrix-multiplication implementation than either CLBlast or Lift. On the external GPUs, the libraries provide better linear algebra implementations, while Lift performs well on stencils and reductions.

Speedup vs. Parallel Handwritten Implementations Figure 19 shows the performance of our approach compared to hand-written reference OpenMP and OpenCL implementations. For some of the benchmarks, the parallel versions are significantly modified using different algorithms beyond the domain of automation. We can see that for benchmarks where the handwritten implementation does not make algorithmic changes (*CG*, *histo*, *lbm*, *sgemm*, *spmv*, *stencil*), we achieve comparable – or better – performance. For four benchmarks (*EP*, *IS*, *MG*, and *tpacf*) it is beneficial to parallelize the entire application – which is beyond the scope of this paper. Future work will examine outer loop parallelism as an idiom to exploit.

For the *sgemm* and *stencil* benchmarks we improved the baseline implementation provided by the benchmarks as these had extremely poor performance. A simple interchange of two loops improved performance by almost 20 times.

Summary We detect 60 idioms across the benchmark suites and are able to achieve significant performance improvements for those benchmarks where idioms dominate execution time by targeting different heterogeneous APIs.

9 Related and Future Work

Domain specific Languages DSLs have received much attention in recent years, ranging from SPIRAL [38], a DSL for Fast Fourier Transforms, over Lift [22, 47, 48] to UFL [1], a DSL for partial differential equations. Stencils in particular have received much attention [22, 33], the best known of which is Halide [42]. DSLs to exploit complex reductions are less studied. In [44] they introduce a type of DSL via annotations that allow expression of complex reductions based on the Platform-Neutral Compute Intermediate Language [4]. In the case of matrix multiplication, this is a well specified idiom supported by specific libraries [2, 23, 36].

Generation of Performance Portable Code for Heterogeneous Hardware Recent research has highlighted the challenges of generating code that performs well on different heterogeneous hardware architectures. PetaBricks [39] is one of the first languages to address this performance portability challenge by encoding algorithmic choices which are then empirically evaluated and automatically taken by the compiler. Similarly [34] explores automatic selection of code variants using machine learning. In a similar spirit, Lift [47] uses rewrite rules to explore optimization choices automatically.

Functional Code Generation Approaches There exist multiple functional approaches for generating code for heterogeneous hardware. Accelerate is a Haskell embedded domain specific language aimed at generating efficient GPU code [10, 31]. Recently, Nvidia introduced NOVA [12], a new functional language targeted at code generation for GPUs, and Copperhead [8], a data parallel language embedded in Python. Delite [7, 9] is a system that enables the creation of domain-specific languages using functional parallel patterns and targets multi-core CPUs or GPUs. In contrast, to these approaches, we require no rewriting of legacy programs.

Idiom Detection Idiom based optimization [40] has fallen out of fashion, with more systematic approaches based on SSA [29] and polyhedral representations [6]. They were largely based on syntactic pattern matching and not robust in the presence of complex control and dataflow. More recently, [3] describes a compiler based parallelization approach for heterogeneous computing, based on an idiomatic intermediate representation called KIR. It is not clear how such an approach would work on general C/C++ programs.

Stencils Stencil detection has been driven by the introduction of DSLs such as Halide. Helium [32] tackles the challenging task of detecting stencils in binary code. It relies on dynamic analysis and cannot easily be extended to other idioms. Another closely related paper is [28], which detects stencils in FORTRAN by the verified lifting of code segments to a representation that can be mapped to Halide DSL. It uses syntax guided synthesis to verify translation with added constrains to ensure that it can be mapped to Halide. It however requires nested loops without conditionals in well behaved FORTRAN and in some cases requires user annotations.

Reductions Discovering and exploiting scalar reductions in programs has been studied for many years based on dependence analysis and idiom detection [15, 41, 50]. Alongside this data dependence based approach, there has also been a large body of work exploring mapping of reductions in a polyhedral setting [27, 45] The treatment of more general reduction operations has received less attention. Work has focused on exploitation rather than discovery [19–21], examining trade-offs in implementation [53] or exploitation of novel hardware [43, 52]. Recent work [17] shows that more complex reductions can be detected, but this is tied to an ad hoc non-portable code generation phase.

Polyhedral Approaches Polyhedral compilers [5, 51] perform advanced loop optimizations and have been used for the generation of fast GPU kernels. More recently, extensions to the polyhedral framework have been proposed, allowing it to capture reduction computations [11, 18, 49]. Such efforts are described in [14], but they are fragile in the presence of non static control flow.

Future Work Although idioms can be described concisely with IDL, we currently have to implement a separate translation scheme for each API. While much of the translation code is common, it would be preferable to have an API description language similar to IDL that allows automatic generation of API translators. This would allow rapid evaluation of different APIs for the same idiom.

As the number of APIs and idioms grows, a profitability heuristic will be needed to determine the best API to use for each program and platform. Machine learning approaches are an obvious starting point as they easily adapt to changing targets.

This paper restricts its attention to five common idioms. Other idioms such as graph processing can also be described. Given that IDL works on the compiler IR, loop and function parallelism can also be described as idioms. In those cases where user codes do not quite match the platform API and associated idioms, we can apply program transformations to refactor or rejuvenate code to fit.

To be a robust approach to heterogeneous programming, we need to ensure correctness. Syntax guided synthesis is a promising means of verifying the idiom translation.

It would be interesting to see to whether our approach could be used for binary optimization or applied to heavily optimized and complex code bases. Another research direction is investigating explicitly parallel legacy codes.

10 Conclusion

This paper develops a compiler based approach that automatically detects a wide class of idioms supported by libraries or domain specific languages for heterogeneous processors. This approach is based on a constraint based description language that identifies program subsets that adhere to idiom specifications. Once detected, the idioms are mechanically translated into API calls to external libraries or code generated by DSL compilers.

This approach is robust and was evaluated on C/C++ versions of two well known benchmark suites: NAS and Parboil. We detected more stencils, sparse matrix operations and generalized reductions and histograms than existing approaches and generated fast code.

Future work will extend the constraint formulation to consider other common idioms. As the number of idioms detected and of implementations available grows, a smart profitability analysis will be needed and is the subject of future work.

Acknowledgements

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh. Some of the hardware used for this research was donated by the NVIDIA Corporation.

References

[1] Martin S. Alnæs, Anders Logg, Kristian B. Olgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, March 2014.

[2] AMD. clBLAS. https://github.com/clMathLibraries/clBLAS, 2013.

[3] José M. Andión. *Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures*. PhD thesis, University of A Coruña, 2015.

[4] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven Van Haastregt, Alexey Kravets, and Alastair Donaldson. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015.

[5] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[7] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010.

[9] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011.

[10] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[11] Lam Chi-Chung, P Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[12] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716, 2015.

[15] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 135–146, New York, NY, USA, 1994. ACM.

[16] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP TC 2 Working Conference on Domain-Specific Languages*. Springer, 2009.

[17] Philip Ginsbach and Michael F. P. O'Boyle. Discovery and exploitation of general reductions: a constraint based approach. In *CGO*, pages 269–280. ACM, 2017.

[18] Gautam Gupta and Sanjay V Rajopadhye. Simplifying reductions. In *POPL*, volume 6, pages 30–41, 2006.

[19] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 78–87, New York, NY, USA, 2000. ACM.

[20] Eladio Gutiérrez, O Plata, and Emilio L Zapata. Optimization techniques for parallel irregular reductions. *Journal of systems architecture*, 49(3):63–74, 2003.

[21] Eladio Gutiérrez, Oscar Plata, and Emilio L Zapata. An analytical model of locality-based parallel irregular reductions. *Parallel Computing*, 34(3):133–157, 2008.

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *CGO*. ACM, 2018.

[23] Intel. *Math Kernel Library*.

[24] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[25] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[27] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd international conference on Supercomputing*, pages 186–194. ACM, 1989.

[28] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, 2016. ACM.

[29] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[30] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009.

[31] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[32] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM.

[33] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[34] Saurav Muralidharan, Amit Roy, Mary W. Hall, Michael Garland, and Piyush Rai. Architecture-adaptive code variant tuning. In *ASPLOS*, pages 325–338. ACM, 2016.

[35] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[36] Nvidia. *cuBLAS*.

[37] Nvidia. *Nvidia OpenCL Best Practices Guide*, 2011.

[38] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. SPIRAL in Scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences*, GPCE, 2013.

[39] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman P. Amarasinghe. Portable performance on heterogeneous architectures. In *ASPLOS*, pages 431–444. ACM, 2013.

[40] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. *ACM Trans. Program. Lang. Syst.*, 16(3):305–327, May 1994.

[41] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM, 1995.

[42] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[43] Vignesh T Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM international conference on supercomputing*, pages 137–146. ACM, 2010.

[44] Chandan Reddy, Michael Kruse, and Albert Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on GPU. In *PACT*, pages 87–97. ACM, 2016.

[45] Xavier Redon and Paul Feautrier. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*, pages 117–125. ACM, 1994.

[46] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

[47] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*, pages 205–217. ACM, 2015.

[48] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*, pages 74–85. ACM, 2017.

[49] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM.

[50] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*, pages 18–25. ACM, 1996.

[51] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4), 2013.

[52] Huo X., Ravi V., and Agrawal G. Porting irregular reductions on heterogeneous CPU-GPU configurations. In *Proceedings of the 18th IEEE International Conference on High Performance Computing*, 2011.

[53] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1084–1096, 2006.